Multitape Alphabet Enlargement and the Linear Speedup Theorem

Formal Proof in Isabelle/HOL

András Z. Salamon Michael Wehar

August 26, 2026

Abstract

Machine-checked formal proof in Isabelle/HOL of the linear speedup theorem for multitape Turing machines, in both its deterministic and nondeterministic forms, via the alphabet-enlargement construction. The theorem is originally due to Hartmanis and Stearns [4]; the textbook proof followed here is by Hopcroft and Ullman [6, Theorems 12.3 and 12.4] and its nondeterministic corollary.

Given a multitape Turing machine M with alphabet Σ_M and a chosen block size c, we build an enlarged machine $M' =$ alphabet_enlarge M. Its alphabet is $\Sigma_M^c \cup \{\square, \triangleright\}$ (with a fresh blank and left-end marker), and it simulates c steps of M at a time, reading its input in blocks of c symbols. A separate wrapping step folds the block-encoder into the machine, so that the textbook statements compare languages over M's original alphabet.

This session (`Multitape_Alphabet_Enlargement`) establishes the following headline theorems:

- *Time bound*: if M accepts w in time $T(n)$, then M' accepts encode_input $bl_M\, w$ in time $2\lceil n/c\rceil + 8\lceil T(n)/c\rceil + 4$ (theorem `alphabet_enlarge_time_explicit`). Both the input length and the running time are divided by the group size c.
- *Language biconditional*: encode_input $bl_M\, w \in L(M') \iff w \in L(M)$ for every well-formed M, with no determinism hypothesis.
- *Determinism-free building block*: for an arbitrary, possibly nondeterministic, well-formed M, the language biconditional and the time bound hold together (`alphabet_enlarge_nae`), with no det_mttm hypothesis. This is the raw (encoded-input) form over the enlarged alphabet; wrapping it in the inline encoder yields the same-alphabet speedup corollaries below, each proved in a deterministic and a nondeterministic version.
- *HU 12.4 (linear-T case)*: for a linear time bound $T(n) \le d_0 n$ and any rational $\varepsilon = 1/q > 0$, there exists a multitape M'' over the original alphabet with $L(M'') = L(M)$ running in time

$(1+\varepsilon)|w|+K$ (exactly $|w|+|w|\operatorname{div} q+K$) — matching Hopcroft and Ullman's $(1+\varepsilon)n$, with the additive constant now an explicit closed expression $K = 28 + 8d_0$ — once the block size c is taken large enough (it grows with q and the slope d_0). It is proved both in a deterministic version (M deterministic $\Rightarrow M''$ deterministic) and a nondeterministic version (no determinism hypothesis); the latter is the Hopcroft–Ullman corollary $\mathrm{NTIME}(cn) = \mathrm{NTIME}((1+\varepsilon)n)$, stated there without proof.

- *HU 12.3 (super-linear-T case)*: for T satisfying the growth hypothesis $\forall d.\exists N.\forall n \geq N.\, d \cdot n \leq T(n)$, M'' runs in time $\varepsilon\, T(|w|) + K$ for $|w| \geq N_0$ (the exact integer bound is $T(|w|) \operatorname{div} q + K \leq \varepsilon\, T(|w|) + K$; the linear-in-$|w|$ overhead is absorbed into $T(|w|)/(2q)$ via the growth hypothesis). Again this is proved both in a deterministic and a nondeterministic version, with the latter giving $\mathrm{NTIME}(T(n)) = \mathrm{NTIME}(c \cdot T(n))$ for any $c > 0$.
- *Time-complexity convention*: both linear-T headlines are additionally restated in Hopcroft and Ullman's $\max(n+1, \cdot)$ time convention, in *two* forms — an all-inputs form keeping the explicit constant ($\ldots + K$), and a constant-free form $(1+\varepsilon)|w|$ (exactly $|w| + |w| \operatorname{div} q$) valid once $|w|$ passes the closed threshold $q(q+1)K$. The clean constant-free bound cannot hold on *all* inputs, so the residual K is provably necessary: a minimal counterexample (`clean_convention_unattainable`, proved in the substrate entry `Multitape_TM_Substrate`) accepts its language yet exceeds the convention floor on its shortest input.

The language equivalence, the time bound, and the combined building block above all hold for nondeterministic machines as well as deterministic ones (they assume no determinism). Each Hopcroft–Ullman speedup corollary is proved twice: once for deterministic machines, where the result is also shown to stay deterministic, as in the classical statements, and once with no determinism assumption, giving the corollary that Hopcroft and Ullman state but do not prove.

The nondeterministic multitape substrate lives in the sibling `Multitape_TM_Substrate` session; this session depends on `Multitape_TM_Substrate` and `HOL-Library`.

Acknowledgement. This formalisation was developed with proof engineering by Anthropic's Claude Opus 4.5, 4.6, 4.7, and 4.8 as the work progressed, together with other Claude family models in supporting roles. Claude Code was used to structure the work and for access to Isabelle/HOL. We also thank Arthur Freitas Ramos who provided feedback and also contributed several tooling improvements.

Contents

1 Overview **7**

1.1 High-level overview . . . 7
1.2 Glossary . . . 7
1.3 Scope and limitations . . . 10
1.4 The Hopcroft–Ullman statements . . . 11
1.5 Details of theorem statements . . . 12

2 Alphabet enlargement **15**
2.1 Stage bookkeeping types . . . 15
2.2 Offset arithmetic on $'c$ . . . 16
2.3 Block-encoding helpers . . . 17
2.4 Encoder . . . 17
2.5 Decoder . . . 17
2.6 Initial stage . . . 18
2.7 Encoder-canonicity predicates . . . 18
2.8 Per-substep transition relations . . . 19
2.9 Transition relation (union of per-substep parts) . . . 39
2.10 Per-substep functionality (source determines target, except SS4→SS5) . . . 54
2.11 The combinator . . . 62
2.12 Simulation infrastructure . . . 63
2.12.1 Position-decoding, tape correspondence, initial config . 64
2.12.2 Substrate projection bridges . . . 65
2.12.3 Gamma-block invariants and preservation . . . 67
2.12.4 Simulation relation and mid-stage invariants . . . 71
2.12.5 LE-guard pre-emption invariant . . . 74
2.12.6 Position-link discharges . . . 76
2.12.7 Pre-state idx and void-aux helpers . . . 81
2.12.8 Position-link preservation . . . 83
2.12.9 Per-substep step-existence . . . 86
2.12.10 Substrate and encoder helpers . . . 106
2.13 Block-predicate identities (VFwd exclusion support) . . . 108
2.14 VFwd 6-fold within-phase exclusion . . . 109
2.15 Chain uniqueness — single-step functionality . . . 117
2.16 Buffered head position arithmetic . . . 123
2.16.1 *c_idx* and *bp_linear* arithmetic . . . 124
2.16.2 *bp_advance* commutation, totality, and LE-aware bounds . . . 125
2.16.3 Read-from-window lemmas . . . 131
2.16.4 *buf_lin_at* preservation under *write_bp* . . . 136
2.17 Window-invariant single-step preservation . . . 140
2.18 Encoder and decoder structural lemmas . . . 158
2.18.1 Decoder structural lemmas and ceiling-div arithmetic 158
2.18.2 Encoder list operations and image properties . . . 161
2.18.3 Encoder well-formedness and round-trip . . . 172

2.19 Validation step machinery and initial config . . . 180
2.19.1 Validation canonicity equivalence with encoder image 180
2.19.2 Validation step helpers . . . 185
2.19.3 Initial-config shape and gamma-block . . . 196
2.20 Validation sweeps, post-state, and step bound . . . 198
2.20.1 Validation sweeps and tape correspondence . . . 198
2.20.2 Validation post-state and step count . . . 207
2.21 Forward simulation chain . . . 243
2.21.1 Per-substep state shape and invariant preservation . . 243
2.21.2 *mttm_step* lifts of cross-phase exclusions . . . 249
2.21.3 Buffered c-fold compute correctness . . . 254
2.22 SS4 trace-existence and buffer characterisations . . . 276
2.22.1 SS4→SS5 trace-existence . . . 276
2.22.2 Buffer characterisations at SS4 entry . . . 287
2.23 Home classification and output well-formedness . . . 302
2.23.1 Home classification, stage unpack, load chain . . . 302
2.23.2 Output well-formedness theorem . . . 310
2.24 Forward-stage per-tape reconstruction leaves . . . 315
2.25 Forward-stage SS5–SS8 super-step . . . 363
2.26 Per-tape unified forward stage . . . 384
2.27 Acceptance correspondence and step-count engine . . . 436
2.27.1 Acceptance correspondence and initial setup . . . 436
2.27.2 Step-count machinery and chunked simulation engine 438
2.28 Top-level theorems . . . 448
2.29 Reverse direction: buffered → tape lift . . . 457
2.29.1 Single-step lift to per-tape . . . 457
2.29.2 SS4→SS5 backward destruct . . . 477
2.29.3 Backward stage: load, compute, writeback . . . 479
2.29.4 Window-inversion kernel and SS5→SS8 congruence (*det*-free) . . . 496
2.29.5 SS4→SS5 trace-functional . . . 504
2.29.6 SS5→SS8 blindness to the pos=0 residual . . . 505
2.29.7 Backward stage orchestration . . . 527
2.29.8 Substep cycle progression . . . 539
2.29.9 Reverse-arm dispatcher and substrate consequences . 549
2.29.10 Chunked-reverse engine and language equivalence . . . 551
2.30 Nondeterministic alphabet enlargement . . . 560

3 Encoding-wrap combinator 562
3.1 Wrap alphabet . . . 562
3.2 Wrap tape index . . . 563
3.3 Wrap state space . . . 563
3.4 Generic single-tape rewind . . . 563
3.5 User-facing language and time-bounded acceptance . . . 564

3.6 Encoder image and state-set finiteness 564
3.7 Encoder-image support lemmas 566

4 Faithful (k-tape) encoding wrap: transpose primitives 575
4.1 The tape transposition 576
4.2 Boundary-parameterised encoder families 576
4.3 Transposed run relation 579

5 Faithful (k-tape) encoding wrap: encoder-phase threading 579
5.1 Boundary-parameterised encoder configurations 580

6 Faithful (k-tape) encoding wrap: combined reset and dispatch 581
6.1 The τ-lift of an M-configuration 582
6.2 The combined reset configuration 582

7 AE-wrap glue: *wrap_enc* matches *encode_input* 583

8 Faithful (k-tape) encoding wrap: plant-*le* variant (HU 12.4) 588
8.1 The plant-*le* reset done step 589
8.2 The plant-*le* combinator 589
8.3 Well-formedness of the plant-*le* wrap 590
8.4 Determinism of the plant-*le* wrap 596

9 Faithful (k-tape) plant-*le* wrap: reset threading and the origin float 599
9.1 The storage-tape rewind, input head frozen 599
9.2 The floated engine configuration and the plant-done / dispatch steps 603
9.3 The reset reaches the floated engine configuration 606
9.4 The floated config is a shift of the proper τ-lift 608

10 Faithful (k-tape) plant-*le* wrap: the transposed run 609
10.1 Forward simulation: an M-step lifts to a wrap-step 609
10.2 State-routing exclusion 611
10.3 Reverse simulation: a wrap-step projects to an M-step 611
10.4 Iterated simulation 614

11 Faithful (k-tape) plant-*le* wrap: encoder-phase threading 617
11.1 Helper (1a): the initial encoder step 617
11.2 Helper (1b)(i): one buffer-extend step 618
11.3 Helper (1b)(ii): one buffer-close step 621
11.4 Helper (1b): one block cycle 626
11.5 Helper (1c)(r=0): clean-termination tail step 630
11.6 Helper (1c)(r≠0): partial-block tail step 632

11.7 Helper (1c): tail dispatch 641
11.8 Encoder-phase reachability and step count 642

12 Faithful (k-tape) encoding wrap: forcing machinery 646
12.1 State-routing exclusions and forward determinism 646
12.2 Pack-contract forcing 654

13 Faithful (k-tape) encoding wrap: canonical factor 661
13.1 Canonical chain construction 661
13.2 Pack contracts by strong induction 666
13.3 The canonical-factor lemma 673

14 Faithful (k-tape) plant-*le* wrap: language preservation 679
14.1 Reverse inclusion: chaining the phases through the floated origin 679
14.2 Forward inclusion: factoring through the floated origin 683
14.3 Language equality headline 686

15 Faithful (k-tape) plant-*le* wrap: time bound 686

16 Faithful (k-tape) classical linear-speedup headlines — plant-*le* wrap 690
16.1 Linear speedup theorem (Form 1, raw composition) — faithful 691
16.2 Textbook linear-speedup theorem (HU 12.4) — faithful k-tape 698
16.3 Textbook linear-speedup theorem (HU 12.3) faithful k-tape . 706

17 The linear-speedup headlines in Hopcroft–Ullman's time-complexity convention 713
17.1 Bridge: an accepted word of the wrap is *Raw*-encoded 714
17.2 Arithmetic core of the superlinear absorption 715
17.3 HU 12.4 in the convention (linear T) 716
17.4 HU 12.3 in the convention (superlinear T): the clean bound . 725

1 Overview

1.1 High-level overview

In one sentence: this session proves the *linear speedup theorem* for multitape Turing machines, in both its deterministic and nondeterministic forms. The theorem says that any machine can be made to run a constant factor faster. The idea is to enlarge the machine's tape alphabet so that a fixed number c of the original cells is packed into one "block", and then to simulate c of the original steps at a time.

Two constructions do the work. The first, $M' = \text{alphabet_enlarge}\, M$, rebuilds M so that it works over the block alphabet and simulates c steps of M in each of its own steps. Its input must be presented in the same blocked form. The second, $M'' = \text{encoding_wrap}(M', \ldots)$, folds the block-encoder into the machine, so that the textbook statements can be made over M's original alphabet, with no encoding visible from outside.

The headline results are a time bound and a language equivalence for M' (packaged together, with no determinism assumption, as alphabet_enlarge_nae), together with the two textbook corollaries over the original alphabet: Hopcroft and Ullman's Theorem 12.4 (the linear-time case) and Theorem 12.3 (the super-linear case). Each corollary is proved twice: once for deterministic machines (the `_dae` version, which also shows the result stays deterministic, as in the classical statements) and once with no determinism assumption (the `_nae` version).

Three points help a first-time reader. First, as in the sibling `Multitape_Alphabet_Reduction` session, a machine here may be nondeterministic by default, since its transition is a relation. The `_nae` theorems are therefore genuine statements about nondeterministic machines; they are the NTIME corollaries ($\text{NTIME}(cn) = \text{NTIME}((1+\varepsilon)n)$ and $\text{NTIME}(T(n)) = \text{NTIME}(cT(n))$) that Hopcroft and Ullman state without proof. Second, because the alphabet grows, the plain machine M' reads its input in blocked form $\text{encode_input}\, bl_M\, w$; the wrapped machine M'' hides that encoding, which is why the corollaries can be stated over the original alphabet. Third, the block size c can be any fixed number; the speedup factor is $1+\varepsilon$ with $\varepsilon = 1/q$, and each corollary holds once c is large enough relative to q.

Every theorem is stated for an arbitrary machine M, so each is a genuine "for all M" result. The sibling entry `Multitape_Alphabet_Reduction` goes the other way: it *reduces* the alphabet to a fixed four-symbol set.

1.2 Glossary

The machine type is mttm, the ten-component descriptor MTTM $Q\ \Sigma\ \Gamma\ bl\ le\ \delta\ s\ t\ r\ k$: state set, input alphabet, tape alphabet, blank symbol, left endmarker, transition relation, start/accept/reject

states, and tape count. Throughout, M is the source machine.

Projections. Positional selectors that read one component out of M:

- `k_tm` M — the tape count k_M (a natural number);
- `Sigma_tm` M — the input alphabet Σ_M;
- `Γ_tm` M — the tape alphabet Γ_M;
- `bl_tm` M — the blank symbol;
- `delta_tm` M — the transition relation δ_M;
- `t_tm` M — the accept state.

Shared substrate predicates. Each entry opens with the gist, then the precise conditions.

`valid_mttm` M *"M is a legal machine".* The structural invariant: Q and Γ finite, $\Sigma \subseteq \Gamma$, the three distinguished states lie in Q, blank and endmarker are tape-but-not-input symbols, accept $\neq$ reject, $k > 0$, the transition relation is correctly typed, the *left-endmarker read discipline* holds (reading the endmarker forces re-writing it and never moving off the left edge), and every tape beyond index k is frozen blank.

`well_formed_mttm` M *a legal machine that also starts cleanly and keeps its endmarker.* An abbreviation: `valid_mttm` M strengthened with *non-degeneracy* (start $\neq$ accept, start $\neq$ reject, endmarker $\neq$ blank) and the endmarker *write* discipline `le_unique` M. Required wherever a run must genuinely start, the endmarker must be detectable, and it must never be forged.

`le_unique` M *the left endmarker is never freshly planted.* A machine property in the same family as `det_mttm` — an opt-in structural guarantee a downstream construction may rely on. Precisely: no transition writes the endmarker where it was not already reading it ($a'\,j = le \Rightarrow a\,j = le$ over δ). Folded into `well_formed_mttm`; the alphabet transformations use it to pin the endmarker, and the origin-floating speed-up wrapper is the one construction that deliberately forgoes it (so it is `valid_mttm` but not `well_formed_mttm`).

`det_mttm` M *M is deterministic.* The transition relation is a partial function: each (state, read-vector) pair has at most one successor triple — the functionality predicate layered on the otherwise relational δ.

$L(M) =$ `Lang_mttm` M *the language M accepts.* All input words w (with set $w \subseteq \Sigma_M$) from which some run reaches the accept state; acceptance is *existential* over runs — nondeterministic.

`accepts_in_time_mttm` $M\, w\, t$ *M accepts w within t steps.* Weak time-bounded acceptance: some run of length at most t reaches the accept state.

The input lives on a work tape. The substrate has *no* separate read-only input tape: the input word is placed on tape 0, which is an ordinary work tape. This is why `valid_mttm` requires $\Sigma \subseteq \Gamma$ — every input symbol occupies a tape cell, so it must also be a tape symbol — and it is load-bearing for the alphabet transformations, which re-encode the input *in place* on tape 0 rather than copying it to a fresh tape.

Alphabet-enlargement vocabulary. *Alphabet enlargement* is our name for the block construction; in the classical literature it is the alphabet-growing half of the Hartmanis–Stearns tape-compression / linear-speedup theorem and carries no standard short name of its own.

$c = |'c|$ the *group size*: the number of original cells packed into one block, equal to card (UNIV :: $'c$ set) for a type parameter $'c$ :: enum.

$M' =$ alphabet_enlarge M the enlarged machine, over the block alphabet $'c \to 'a$, with state type $'q \times ('a, 'c)$ ae_stage; it simulates c steps of M per M'-stage following Hopcroft–Ullman's eight-substep scheme.

ae_stage the per-state bookkeeping carried by M' (validation / simulation substep tag, per-tape buffers, head destinations).

encode_input $bl\, w$ the input encoding: it groups w into consecutive blocks of c symbols, each packed as a function $'c \to 'a$ holding those c symbols, padding the final block with the blank bl. The result is a list of $\lceil |w|/c \rceil$ blocks.

M'', the wrapped machine encoding_wrap M' (ae_pack bl_M) $c\, \Sigma_M$: the raw machine M' with an inline block-encoder, on the faithful k-tape plant-$\triangleright$ construction (reusing the spent input tape rather than prepending a fresh one), over the original-alphabet symbols Raw Σ_M (plus the encoded working symbols Enc Γ_M); ae_pack is the per-block pack function matching encode_input.

`Lang_user_wrap` M'' the user-facing language of a wrapped machine, over the *original* alphabet: $\{w \mid \text{map Raw } w \in L(M'')\}$.

`accepts_in_time_user_wrap` $M''\, w\, t$ the corresponding user-facing timed acceptance, `accepts_in_time_mttm` M'' (map Raw w) t.

Theorem-name suffixes: `_nae` marks the determinism-free ("nondeterministic alphabet enlargement") version, `_dae` the deterministic specialisation that additionally preserves determinism.

The slowdown constant. The scheme formalised here is Hopcroft and Ullman's eight-substep simulation [6, Theorems 12.3 and 12.4] (already in their 1969 text [5, §7.1]), whose per-group cost is the factor 8 in the time bound above. The precise constant is a matter of presentation, not of the theorem: Balcázar, Díaz, and Gabarró [2] and Papadimitriou [7] give a factor-6 simulation, while van Emde Boas's survey [8] reads the original Hartmanis–Stearns theorem [4] as intending an amortised, non-oblivious simulation that reaches $(1+\varepsilon)$ directly rather than through any fixed multiplicative constant. Whichever the constant, the group size c divides it away, so we formalise the scheme cleanest to verify rather than the one with the smallest factor.

Related AFP developments. To our knowledge the Hartmanis–Stearns linear speedup theorem has not been formalised in any proof assistant, so this entry has no direct predecessor; the neighbouring Isabelle/HOL developments prove different theorems. Balbach's `Cook_Levin` [1] shares the model — multitape machines with the tape count a value — and, like this entry, states a machine's language and its running-time bound as separate theorems, but treats deterministic polynomial-time reductions for NP-completeness, not a speedup. Xu, Zhang, Urban, and Joosten's `Universal_Turing_Machine` [9] is a single-tape universal machine over a fixed binary alphabet for computability, with no timing analysis. The substrate itself descends from Dalvit and Thiemann's `Multitape_To_Singletape_TM` [3], but makes the number of tapes a machine's *data* rather than a type parameter, so a single speedup theorem here ranges over machines of every tape count.

1.3 Scope and limitations

Three caveats orient a reader deciding whether these results apply.

- *Original-alphabet statements need the wrap.* The raw enlarged machine M' works on block-encoded input (encode_input $bl_M\, w$); the same-alphabet corollaries hold for the *wrapped* machine M'', which folds the encoder in. Statements about M' alone carry the visible encoding.
- *The corollaries need a large enough group.* The speedup factor is $1+\varepsilon$ with $\varepsilon = 1/q$, and each corollary holds only once the group size c is large enough relative to q ($q\,(3+8d_0) \le c$ for the linear case, $16q \le c$ for the super-linear case).

- *The super-linear case needs a growth hypothesis.* HU 12.3 assumes $\forall d.\ \exists N.\ \forall n \geq N.\ d\,n \leq T(n)$ and holds for inputs $|w| \geq N_0$; the linear case HU 12.4 instead assumes $T(n) \leq d_0 n + b$.

1.4 The Hopcroft–Ullman statements

So a reader can check this session against the textbook results it claims, we reproduce the source statements verbatim from Hopcroft and Ullman [6, §12.2, pp. 290–291], with the two conventions needed to read them.

Two reading conventions. First, in these quoted statements c is Hopcroft and Ullman's *time-complexity coefficient*; it is *unrelated* to our group size c (the enlargement block size of the vocabulary above). Second, they use the convention that $cT(n)$ abbreviates $\max(n+1, \lceil cT(n) \rceil)$: the $n+1$ floor is the cost of reading the input and reaching the blank past its end, and it is what lets the bound hold for *all* n — the finitely many short inputs are recognised by finite control in $n+1$ moves, and only the long inputs run the simulation.

Theorem 12.3 (linear speed-up).

> If L is accepted by a k-tape $T(n)$ time-bounded Turing machine M_1, then L is accepted by a k-tape $cT(n)$ time-bounded TM M_2 for any $c > 0$, provided that $k > 1$ and $\inf_{n\to\infty} T(n)/n = \infty$.

Theorem 12.4 (linear-T case).

> If L is accepted by a k-tape cn time-bounded TM, for $k > 1$ and for some constant c, then for every $\varepsilon > 0$, L is accepted by a k-tape $(1+\varepsilon)n$ time-bounded TM.

Corollaries (p. 291).

> *To Theorem 12.3:* if $\inf_{n\to\infty} T(n)/n = \infty$ and $c > 0$, then $\mathrm{DTIME}(T(n)) = \mathrm{DTIME}(cT(n))$.
>
> *To Theorem 12.4:* if $T(n) = cn$ for some $c > 1$, then $\mathrm{DTIME}(T(n)) = \mathrm{DTIME}((1+\varepsilon)n)$ for any $\varepsilon > 0$.
>
> *Of Theorems 12.3 and 12.4:* **(a)** if $\inf_{n\to\infty} T(n)/n = \infty$, then $\mathrm{NTIME}(T(n)) = \mathrm{NTIME}(cT(n))$ for any $c > 0$; **(b)** if $T(n) = cn$ for some constant c, then $\mathrm{NTIME}(T(n)) = \mathrm{NTIME}((1+\varepsilon)n)$ for any $\varepsilon > 0$.

Correspondence. The session's four headline theorems formalise these results over the value-level substrate:

`linear_speedup_HU_12_3_dae` Theorem 12.3, the deterministic linear speed-up (super-linear T).

`linear_speedup_HU_12_3_nae` Corollary (a), its nondeterministic form.

`linear_speedup_HU_12_4_dae` Theorem 12.4, the deterministic linear-T speed-up.

`linear_speedup_HU_12_4_nae` Corollary (b), its nondeterministic form.

The `DTIME`/`NTIME` corollaries are the complexity-class readings of the deterministic and nondeterministic forms respectively. The precise formal statements of all four — read against the $\max(n+1, \lceil \cdot \rceil)$ convention above — follow next.

1.5 Details of theorem statements

We first state the three properties of the raw enlarged machine M', then their determinism-free bundle, then the two textbook corollaries over the original alphabet. The only hypothesis on M for the raw results is `well_formed_mttm`: the time and language results carry *no* determinism hypothesis, and the deterministic corollaries add `det_mttm`.

`alphabet_enlarge_wf.`

$$\text{valid_mttm}\, M \implies \text{valid_mttm}\, M'.$$

The enlarged machine is again a legal machine. Unlike reduction, no alphabet-size side condition is needed: enlargement always yields a well-typed block machine.

`alphabet_enlarge_language.`

$$\begin{aligned}&\text{well_formed_mttm}\, M \implies \forall w.\ \text{set}\, w \subseteq \Sigma_M \longrightarrow \\ &\quad \big(\text{encode_input}\, bl_M\, w \in L(M')\big) = \big(w \in L(M)\big).\end{aligned}$$

The language biconditional for the raw machine, with no determinism hypothesis: M' accepts the block-encoding of w exactly when M accepts w.

`alphabet_enlarge_time_explicit` and `alphabet_enlarge_time`. Under well_formed_mttm M, for every input word w:

$$\begin{aligned}&\text{accepts_in_time_mttm}\, M\, w\, (T(|w|)) \longrightarrow\\ &\quad \text{accepts_in_time_mttm}\, M'\, (\text{encode_input}\, bl_M\, w)\\ &\qquad \big(2\,\lceil |w|/c\rceil + 8\,\lceil T(|w|)/c\rceil + 4\big).\end{aligned}$$

The $\lceil \cdot/c\rceil$ factors are the source of the speedup: both the input length and the running time are divided by the group size c. The coefficients 2, 8, 4 are absolute constants (the 8 counts the eight substeps simulated per group); none of them depends on M, on w, or even on c. The explicit-constant statement is `alphabet_enlarge_time_explicit`; the `obtains`-corollary `alphabet_enlarge_time` repackages it existentially as $\exists\alpha, f.\ \alpha\,\lceil |w|/c\rceil + 8\,\lceil T(|w|)/c\rceil + f$, instantiated at $\alpha = 2$, $f = 4$.

`alphabet_enlarge_nae`. The determinism-free building block: under well_formed_mttm M alone, the language biconditional *and* the same time bound hold together, packaged existentially ($\exists\alpha, f$, with the witnesses $\alpha = 2$, $f = 4$). This is the raw, encoded-input "Form 2" statement over the enlarged alphabet; wrapping it in the inline encoder yields the same-alphabet corollaries below.

`linear_speedup_HU_12_4_nae` / `_dae` (linear-T case). Assume well_formed_mttm M, a linear time bound $T(n) \le d_0 n + b$, and $q > 0$, $c > 0$. Then M'' is valid, $\texttt{Lang_user_wrap}\, M'' = L(M)$ (same language, *over the original alphabet*), and, provided c is large enough ($q\,(3+8d_0) \le c$), for every w,

$$\begin{aligned}&\text{accepts_in_time_mttm}\, M\, w\, (T(|w|)) \longrightarrow\\ &\quad \texttt{accepts_in_time_user_wrap}\, M''\, w\\ &\qquad \big(|w| + |w|\,\mathrm{div}\, q + K\big),\\ &\qquad\quad K = 28 + 8d_0 + 8b.\end{aligned}$$

The additive constant K is a *literal* closed expression in the input machine's linear-bound constants d_0, b; an earlier form left both K and a wrap constant α existential, and threading $\alpha = 2$ up the raw chain removed them, leaving the side condition $q\,(3 + 8d_0) \le c$ free of α. The running time is $|w|\,(1 + 1/q) + K = (1+\varepsilon)\,|w| + K$ with $\varepsilon = 1/q$, the classical linear speedup. The `_dae` version adds the hypothesis det_mttm M and the extra conclusion det_mttm M'' (determinism preserved); the `_nae` version drops both and is the Hopcroft–Ullman corollary $\mathrm{NTIME}(cn) = \mathrm{NTIME}((1+\varepsilon)n)$.

`linear_speedup_HU_12_3_nae` / `_dae` (super-linear-T case). Assume well_formed_mttm M, the growth hypothesis $\forall d.\ \exists N.\ \forall n \ge N.\ d\,n \le T(n)$,

$q > 0$, and $16q \le c$. Then there are constants K, N_0 with M'' valid, `Lang_user_wrap` $M'' = L(M)$, and for every w with $|w| \ge N_0$,

$$\begin{aligned}&\text{accepts_in_time_mttm}\, M\, w\, (T(|w|)) \longrightarrow \\ &\quad \texttt{accepts_in_time_user_wrap}\, M''\, w\, \big(T(|w|) \operatorname{div} q + K\big).\end{aligned}$$

For super-linear T the growth hypothesis absorbs the linear-in-$|w|$ overhead into $T(|w|)/(2q)$, leaving running time $T(|w|)/q + K$, a speedup by an arbitrary factor q. As before, the `_dae` version adds det_mttm preservation, and the `_nae` version is the corollary $\mathrm{NTIME}(T(n)) = \mathrm{NTIME}(cT(n))$ for any $c > 0$.

Convention restatements (`linear_speedup_HU_12_4_nae_conv` / `_eventual`, and the `_dae` pair). The four headlines above bound each accepting run by a per-input additive constant K. Read in Hopcroft and Ullman's $\max(n+1, \cdot)$ time convention — here $\max(n+2, \cdot)$, since the substrate reads the left endmarker before the first input symbol — the linear-T case has two honest forms. The `_conv` form keeps the constant and holds for *all* inputs: M'' accepts every $w \in L(M)$ within $\max(|w| + 2,\ |w| + |w| \operatorname{div} q + K)$, with the same literal $K = 28 + 8d_0 + 8b$. The `_eventual` form drops the constant entirely — M'' then runs in exactly $|w| + |w| \operatorname{div} q$ — but only once $|w|$ exceeds the closed threshold $q(q+1)K$ (proved by running the raw bound one denominator tighter, at $q+1$, then absorbing K into the extra div-slack). The clean constant-free bound on *all* inputs is provably unattainable: the substrate entry `Multitape_TM_Substrate` proves `clean_convention_unattainable`, exhibiting a valid deterministic machine that accepts its language yet needs three steps on the empty input — one past the floor 2 — so no $|w| + |w| \operatorname{div} q$ convention bound can hold there, for any q. This is why the residual K is kept rather than folded into a smaller ε: closing the finitely many small inputs would require a non-effective finite-exceptions table (`time_cleanup` in `Multitape_TM_Substrate`) that the speedup construction does not build. So the textbook's clean $(1+\varepsilon)n$ reading of 12.4 holds for the constructed machine only with the explicit residual K, or only past the threshold; the class equality $\mathrm{DTIME}(cn) = \mathrm{DTIME}((1+\varepsilon)n)$ survives, but is recovered only through that non-effective table. The superlinear case carries no such caveat: there the growth of T dominates the finite-control cleanup, so the convention forms `linear_speedup_HU_12_3_nae_conv` and `_dae_conv` are already clean, with no residual constant.

theory *AlphabetEnlargement_Defs*
imports *Multitape_TM_Substrate.Multitape_Substrate*
begin

2 Alphabet enlargement

Setup for the *alphabet_enlarge* combinator: substrate selectors, stage-bookkeeping types, offset and block helpers, encoder / decoder, encoder-canonicity predicates, per-substep transition relations, the *alphabet_enlarge* combinator itself, the *ae_simulates* relation, and the eight mid-stage invariants *ae_inv_ss1* through *ae_inv_ss8*, plus the upstream structural lemmas, the encoder-roundtrip chain, and the validation-phase chain culminating in *ae_validation_steps_bound*.

The forward-simulation chain that consumes the validation result, and the top-level theorems *alphabet_enlarge_wf*, *alphabet_enlarge_language*, and *alphabet_enlarge_time*, follow further on in this chapter.

Substrate selectors (*bl_tm*, *le_tm*, *delta_tm*, *s_tm*, *t_tm*, *r_tm*, *Sigma_tm*, *mt_tape*) live in the *Multitape_Substrate* theory alongside the functional substrate layer; they're imported transitively.

2.1 Stage bookkeeping types

Substep counter / phase indicator for M''s state machine.

Constructors *VFwd* / *VFwdPad* / *VRet* mark the validation phase: forward scan (no padded block seen yet), forward scan after a trailing-padded block has been observed, and return scan back to the first input block.

Constructors *SS1* *SS8* mark the 8-substep simulation stage of Hopcroft–Ullman [6, Theorem 12.3]; *SSn* corresponds to the spec's "substep n".

datatype *substep_idx* =
VFwd | *VFwdPad* | *VRet*
| *SS1* | *SS2* | *SS3* | *SS4* | *SS5* | *SS6* | *SS7* | *SS8*

instance *substep_idx* :: *finite*
proof (*intro_classes*)
have (*UNIV* :: *substep_idx set*) ⊆
{*VFwd*, *VFwdPad*, *VRet*,
SS1, *SS2*, *SS3*, *SS4*, *SS5*, *SS6*, *SS7*, *SS8*}
using *substep_idx.exhaust* **by** *blast*
thus *finite* (*UNIV* :: *substep_idx set*)
using *finite_subset* **by** *auto*
qed

Per-tape destination indicator: which buffer slot now holds the home block of M's simulated head, after the compute substep (the 4th component of *stage*). The compute substep sets this; the write-back phase consumes it to

choose between move-sequence variants (‘L,R,R,N‘ / ‘L,R,R,L‘ / ‘R,L,L,N‘ for steady-state stages).

```
datatype ae_dest = AE_Left | AE_Home | AE_Right

instance ae_dest :: finite
proof (intro_classes)
  have (UNIV :: ae_dest set) ⊆ {AE_Left, AE_Home, AE_Right}
    using ae_dest.exhaust by blast
  thus finite (UNIV :: ae_dest set)
    using finite_subset by auto
qed
```

Stage bookkeeping carried in the output machine's state set: per-tape within-block offset ($nat \Rightarrow 'c$), per-tape three-block buffer (left, home, right), per-tape destination indicator (*AE_Left* / *AE_Home* / *AE_Right*), and the substep counter within the current 8-substep stage.

```
type_synonym ('a, 'c) ae_stage =
  (nat ⇒ 'c)
   × (nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
   × (nat ⇒ ae_dest)
   × substep_idx
```

2.2 Offset arithmetic on $'c$

The compute substep tracks a simulated head position by an offset within a 3-block buffer; advancing by L / R requires partial successor / predecessor on $'c$. We use the canonical enumeration provided by the *enum* class.

```
definition c_idx :: 'c :: enum ⇒ nat where
  c_idx x = (THE i. i < length (enum_class.enum :: 'c list)
                  ∧ (enum_class.enum :: 'c list) ! i = x)

definition c_first :: 'c :: enum where
  c_first = (enum_class.enum :: 'c list) ! 0

definition c_last :: 'c :: enum where
  c_last = (enum_class.enum :: 'c list)
           ! (length (enum_class.enum :: 'c list) − 1)

definition c_succ :: 'c :: enum ⇒ 'c option where
  c_succ x =
    (let xs = enum_class.enum :: 'c list; i = c_idx x in
      if Suc i < length xs then Some (xs ! Suc i) else None)

definition c_pred :: 'c :: enum ⇒ 'c option where
  c_pred x =
    (let xs = enum_class.enum :: 'c list; i = c_idx x in
      if i = 0 then None else Some (xs ! (i − 1)))
```

2.3 Block-encoding helpers

The constant LE-block: a block whose every cell holds M's left-endmarker symbol. Used as M''s left endmarker. Compatible with the substrate's δLE invariant: any read of this block tests *le* at every cell, so the $a\ k = LE \Longrightarrow a'\ k = LE \wedge d\ k \in \{N, R\}$ obligation lifts pointwise from M's.

definition *LE_block* :: $'a \Rightarrow ('c :: enum \Rightarrow 'a)$ **where**
LE_block le = $(\lambda_.\ le)$

The constant blank-block: a block whose every cell holds M's blank symbol. Used as M''s blank, and also as the semantic placeholder for *buf.left* in LE-stages.

definition *bl_block* :: $'a \Rightarrow ('c :: enum \Rightarrow 'a)$ **where**
bl_block bl = $(\lambda_.\ bl)$

2.4 Encoder

Encoding an input word as a list of c-blocks, padding the final block with the substrate's blank symbol if the input length is not a multiple of c.

The grouping factor $c = card\ (UNIV :: 'c\ set)$ is determined at the type level. Block i (for $i < \lceil length\ w\ /\ c \rceil$) is the function $\lambda x.\ if\ i \cdot c + c_idx\ x < length\ w\ then\ w\ !\ (i \cdot c + c_idx\ x)\ else\ bl$; the output list has length $\lceil length\ w\ /\ c \rceil$, computed in nat arithmetic as $(length\ w + c - 1)\ div\ c$.

definition *encode_input* ::
$'a \Rightarrow 'a\ list \Rightarrow (('c :: enum) \Rightarrow 'a)\ list$ **where**
encode_input bl w =
(*let c* = *card* (*UNIV* :: $'c\ set$) *in*
map $(\lambda i.\ (\lambda x.\ let\ j = i * c + c_idx\ x\ in$
if $j < length\ w\ then\ w\ !\ j\ else\ bl))$
$[0\ ..<\ (length\ w + c - 1)\ div\ c]$)

2.5 Decoder

Decoder: extract M-symbols from a list of blocks by taking each block's *bl*-free prefix under the canonical $'c$-enumeration. For pure blocks this yields all c cells; for trailing-padded blocks it yields the non-blank prefix; for non-canonical blocks (blank-then-non-blank pattern) it yields the leading non-blank prefix only but the validation phase rejects these before decoding is invoked. Inverse to *encode_input* on the encoder image; cited by the forward direction of *ae_validation_canonical_iff_encoder_image*.

definition *ae_decode_block* :: $'a \Rightarrow ('c :: enum \Rightarrow 'a) \Rightarrow 'a\ list$ **where**
ae_decode_block bl f =
takeWhile $(\lambda a.\ a \neq bl)$ (*map f* (*enum_class.enum* :: $'c\ list$))

definition *ae_decode_input* :: $'a \Rightarrow (('c :: enum) \Rightarrow 'a)\ list \Rightarrow 'a\ list$ **where**
ae_decode_input bl w = *concat* (*map* (*ae_decode_block bl*) *w*)

2.6 Initial stage

Initial within-block offset. Every tape's offset starts at a canonical element of $'c$ (treated as the within-block position of M's left endmarker on each tape). The specific element is underspecified at this level; the simulation argument (ğ30.4) fixes a canonical *0*-offset matching M's initial head position.

definition *init_offset* :: *nat* ⇒ ($'c$ :: *enum*) **where**
init_offset = (λ_. *SOME x*. *True*)

Initial buffer. Every tape starts with three constant *LE*-blocks: at M's position 0 the home block is the left-endmarker block, and the buffer phase has not yet executed on the surrounding cells; the SS1→SS4 buffer phase refills the slots from actual tape contents at the start of every stage.

definition *init_buffer* ::
$'a$ ⇒ (*nat* ⇒ (($'c$ :: *enum* ⇒ $'a$)
× ($'c$ ⇒ $'a$)
× ($'c$ ⇒ $'a$))) **where**
init_buffer le = (λ_. (*LE_block le*, *LE_block le*, *LE_block le*))

Initial destination indicator: every tape starts with *AE_Home* (the indicator is meaningful only after the compute substep sets it; the initial value is canonical).

definition *init_dest* :: *nat* ⇒ *ae_dest* **where**
init_dest = (λ_. *AE_Home*)

Initial stage: zero offset, all-*LE* buffers, dest = home, substep counter *SS1*.

definition *init_stage* ::
$'a$ ⇒ ($'a$, $'c$ :: *enum*) *ae_stage* **where**
init_stage le = (*init_offset*, *init_buffer le*, *init_dest*, *VFwd*)

2.7 Encoder-canonicity predicates

A block is *pure* if every cell is non-blank. In the encoder image, every block except possibly the last has this form (every cell is from the original input).

definition *is_pure_block* ::
$'a$ ⇒ ($'c$ :: *enum* ⇒ $'a$) ⇒ *bool* **where**
is_pure_block bl f = (∀ *x*. *f x* ≠ *bl*)

A block is *trailing−padded* if there is a non-empty, proper prefix (under the canonical $'c$-enumeration) of non-blank cells, and the remaining cells are all blank. In the encoder image, the last block has this form when the input length is not a multiple of *c*.

definition *is_padded_block* ::
$'a$ ⇒ ($'c$ :: *enum* ⇒ $'a$) ⇒ *bool* **where**
is_padded_block bl f =
(∃ *k*. *k* ≥ *1* ∧ *k* < *length* (*enum_class.enum* :: $'c$ *list*)

$\wedge\ (\forall x.\ c_idx\ x < k \longrightarrow f\ x \neq bl)$
$\wedge\ (\forall x.\ c_idx\ x \geq k \longrightarrow f\ x = bl))$

Encoder-canonical block: pure or trailing-padded. The validation phase accepts only blocks satisfying this predicate; non-canonical blocks (blanks scattered in non-trailing positions) route the input to r_M'.

definition *is_canonical_block* ::
$'a \Rightarrow ('c :: enum \Rightarrow 'a) \Rightarrow bool$ **where**
is_canonical_block bl f = (*is_pure_block bl f* $\vee$ *is_padded_block bl f*)

Sequence-level encoder-canonicity: the input w is well-formed (= in the encoder image) iff every block is pure, *except possibly the last* which may also be padded. Per block encoder-canonicity (*is_canonical_block*) is necessary but not sufficient a sequence with a padded block followed by anything else is per-cell canonical but not in the encoder image.

The validation phase enforces exactly this distinction: VFwd reject fires on non-canonical cells; VFwdPad reject fires on any non-*bl_block* cell after a padded block.

definition *ae_input_well_formed* ::
$'a \Rightarrow ('c :: enum \Rightarrow 'a)\ list \Rightarrow bool$ **where**
ae_input_well_formed bl w $\longleftrightarrow$
$(\forall s.\ s < length\ w \longrightarrow$
(*is_pure_block bl* (w ! s)
$\vee$ ($s = length\ w - 1 \wedge$ *is_padded_block bl* (w ! s))))

end
theory *AlphabetEnlargement_Substeps*
imports *AlphabetEnlargement_Defs*
begin

2.8 Per-substep transition relations

δ' is decomposed into named per-substep relations for the validation and simulation phases. Validation: 8 relations (forward LE-skip / pure / padded / non-canonical-reject / end-of-input; padded end-of-input / padded-reject; return step / return-LE-to-Sim). Simulation: 8 relations (SS1→SS2, ..., SS7→SS8, SS8→SS1). Total: 16 relations.

The substrate transition shape is (q, a, q', a', d) with state $'q \times ae_stage$, tape symbol $'c \Rightarrow 'a$. Source-state constraint $q \in Q \wedge q \neq t \wedge q \neq r$ matches substrate δ_set for VFwd-source relations (whose *substep_idx* coincides with t', r''s); for the other phases the *substep_idx* mismatch itself rules out source = t' / r'.

Validation, VFwd, advance: read either *LE_block le_M* (initial step from position 0) or a pure block (no blanks); R move on tape 0; N moves on other tapes; phase stays *VFwd*. No write change.

definition *ae_delta_val_fwd_advance* ::

```
('q, 'a) mttm
  ⇒ (('q × ('a, 'c :: enum) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × ('q × ('a, 'c) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × (nat ⇒ dir)) set where
ae_delta_val_fwd_advance M =
  {((q, ofs, buf, dest, VFwd), a,
     (q, ofs, buf, dest, VFwd), a, d) |
   q ofs buf dest a d.
     q ∈ Q_tm M ∧ q ≠ t_tm M ∧ q ≠ r_tm M
     ∧ (a 0 = LE_block (le_tm M)
         ∨ is_pure_block (bl_tm M) (a 0))
     ∧ (∀j≥k_tm M. a j = bl_block (bl_tm M))
     ∧ d = (λk. if k = 0 then dir.R else dir.N)}
```

Validation, VFwd → VFwdPad: read a trailing-padded block on tape 0; R move on tape 0; N moves on other tapes; phase becomes VFwdPad. No write change.

```
definition ae_delta_val_fwd_to_padded ::
  ('q, 'a) mttm
  ⇒ (('q × ('a, 'c :: enum) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × ('q × ('a, 'c) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × (nat ⇒ dir)) set where
ae_delta_val_fwd_to_padded M =
  {((q, ofs, buf, dest, VFwd), a,
     (q, ofs, buf, dest, VFwdPad), a, d) |
   q ofs buf dest a d.
     q ∈ Q_tm M ∧ q ≠ t_tm M ∧ q ≠ r_tm M
     ∧ is_padded_block (bl_tm M) (a 0)
     ∧ (∀j≥k_tm M. a j = bl_block (bl_tm M))
     ∧ d = (λk. if k = 0 then dir.R else dir.N)}
```

Validation, VFwd reject: read a non-canonical block on tape 0 (in Σ' but neither pure nor padded blanks in non-trailing positions); N moves uniformly; transition to r_M' (= the canonical reject state of M').

```
definition ae_delta_val_fwd_reject ::
  ('q, 'a) mttm
  ⇒ (('q × ('a, 'c :: enum) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × ('q × ('a, 'c) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × (nat ⇒ dir)) set where
ae_delta_val_fwd_reject M =
  {((q, ofs, buf, dest, VFwd), a,
     (r_tm M, init_stage (le_tm M)), a, d) |
   q ofs buf dest a d.
```

```
    q ∈ Q_tm M ∧ q ≠ t_tm M ∧ q ≠ r_tm M
    ∧ a 0 ≠ LE_block (le_tm M)
    ∧ a 0 ≠ bl_block (bl_tm M)
    ∧ ¬ is_canonical_block (bl_tm M) (a 0)
    ∧ (∀ j≥k_tm M. a j = bl_block (bl_tm M))
    ∧ d = (λ_. dir.N)}
```

Validation, VFwd end-of-input: read *bl_block bl_M* (past the encoded input); N moves; transition to VRet to begin the return scan.

```
definition ae_delta_val_fwd_to_ret ::
  ('q, 'a) mttm
    ⇒ (('q × ('a, 'c :: enum) ae_stage)
        × (nat ⇒ ('c ⇒ 'a))
        × ('q × ('a, 'c) ae_stage)
        × (nat ⇒ ('c ⇒ 'a))
        × (nat ⇒ dir)) set where
  ae_delta_val_fwd_to_ret M =
    {((q, ofs, buf, dest, VFwd), a,
       (q, ofs, buf, dest, VRet), a, d) |
     q ofs buf dest a d.
       q ∈ Q_tm M ∧ q ≠ t_tm M ∧ q ≠ r_tm M
       ∧ a 0 = bl_block (bl_tm M)
       ∧ (∀ j≥k_tm M. a j = bl_block (bl_tm M))
       ∧ d = (λ_. dir.N)}
```

Validation, VFwdPad end-of-input: read *bl_block bl_M*; N moves; transition to VRet. Same as the VFwd version except the source phase.

```
definition ae_delta_val_pad_to_ret ::
  ('q, 'a) mttm
    ⇒ (('q × ('a, 'c :: enum) ae_stage)
        × (nat ⇒ ('c ⇒ 'a))
        × ('q × ('a, 'c) ae_stage)
        × (nat ⇒ ('c ⇒ 'a))
        × (nat ⇒ dir)) set where
  ae_delta_val_pad_to_ret M =
    {((q, ofs, buf, dest, VFwdPad), a,
       (q, ofs, buf, dest, VRet), a, d) |
     q ofs buf dest a d.
       q ∈ Q_tm M
       ∧ a 0 = bl_block (bl_tm M)
       ∧ (∀ j≥k_tm M. a j = bl_block (bl_tm M))
       ∧ d = (λ_. dir.N)}
```

Validation, VFwdPad reject: read anything on tape 0 other than *bl_block bl_M*. This signals a non-blank block appearing after the trailing-padded block. N moves; transition to *r_M'*.

```
definition ae_delta_val_pad_reject ::
  ('q, 'a) mttm
    ⇒ (('q × ('a, 'c :: enum) ae_stage)
```

```
      × (nat ⇒ ('c ⇒ 'a))
      × ('q × ('a, 'c) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × (nat ⇒ dir)) set where
  ae_delta_val_pad_reject M =
   {((q, ofs, buf, dest, VFwdPad), a,
     (r_tm M, init_stage (le_tm M)), a, d) |
    q ofs buf dest a d.
     q ∈ Q_tm M
     ∧ a 0 ≠ bl_block (bl_tm M)
     ∧ (∀j≥k_tm M. a j = bl_block (bl_tm M))
     ∧ d = (λ_. dir.N)}
```

Validation, VRet step: read a non-LE block on tape 0 (during the return scan); L move on tape 0; N moves on other tapes; phase stays VRet.

```
definition ae_delta_val_ret_step ::
  ('q, 'a) mttm
   ⇒ (('q × ('a, 'c :: enum) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × ('q × ('a, 'c) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × (nat ⇒ dir)) set where
  ae_delta_val_ret_step M =
   {((q, ofs, buf, dest, VRet), a,
     (q, ofs, buf, dest, VRet), a, d) |
    q ofs buf dest a d.
     q ∈ Q_tm M
     ∧ a 0 ≠ LE_block (le_tm M)
     ∧ (∀j≥k_tm M. a j = bl_block (bl_tm M))
     ∧ d = (λk. if k = 0 then dir.L else dir.N)}
```

Validation, VRet → Sim: read *LE_block le_M* on tape 0 (return scan reached position 0); N moves uniformly (head stays at position 0, the LE-block); phase becomes Sim with *substep_idx* = SS1.

The N move on tape 0 rather than R leaves M''s head at block 0 (LE_M') post-validation, so the simulation phase's first stage runs in LE-stage mode ($home = LE_M'$) and the c-fold compute correctly simulates M's first step from (s_M, LE). See *bp_advance_le* below for how the c-fold compute handles M's first R-move out of LE.

```
definition ae_delta_val_ret_to_sim ::
  ('q, 'a) mttm
   ⇒ (('q × ('a, 'c :: enum) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × ('q × ('a, 'c) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × (nat ⇒ dir)) set where
  ae_delta_val_ret_to_sim M =
   {((q, ofs, buf, dest, VRet), a,
```

```
    (q, ofs, buf, dest, SS1), a, d) |
   q ofs buf dest a d.
    q ∈ Q_tm M
    ∧ a 0 = LE_block (le_tm M)
    ∧ (∀j≥k_tm M. a j = bl_block (bl_tm M))
    ∧ d = (λ_. dir.N)}
```

SS1 → SS2: read home into buffer; per-tape move L (steady-state) or N (LE-stage). No write change ($a' = a$); no M-state advance. Buffer-phase substep 1.

```
definition ae_delta_ss1_ss2 ::
 ('q, 'a) mttm
  ⇒ (('q × ('a, 'c :: enum) ae_stage)
     × (nat ⇒ ('c ⇒ 'a))
     × ('q × ('a, 'c) ae_stage)
     × (nat ⇒ ('c ⇒ 'a))
     × (nat ⇒ dir)) set where
 ae_delta_ss1_ss2 M =
  {((q, ofs, buf, dest, SS1), a,
    (q, ofs, buf', dest, SS2), a, d) |
   q ofs buf dest a buf' d.
    q ∈ Q_tm M ∧ q ≠ t_tm M ∧ q ≠ r_tm M
    ∧ (∀j≥k_tm M. a j = bl_block (bl_tm M))
    ∧ buf' = (λk. if k < k_tm M
                  then (fst (buf k), a k, snd (snd (buf k)))
                  else init_buffer (le_tm M) k)
    ∧ d = (λk. if k < k_tm M
               then (if a k = LE_block (le_tm M) then dir.N else dir.L)
               else dir.N)}
```

SS2 → SS3: read left into buffer (or placeholder for LE-stage); per-tape move R (steady-state, returning to home) or N (LE-stage, staying at home). Buffer-phase substep 2.

In LE-stage ($buf\ k.home = LE_M'$), the read is again LE_M' (head didn't move at SS1) and $buf'\ k.left$ is set to $bl_block\ bl_M$ as a semantic placeholder.

```
definition ae_delta_ss2_ss3 ::
 ('q, 'a) mttm
  ⇒ (('q × ('a, 'c :: enum) ae_stage)
     × (nat ⇒ ('c ⇒ 'a))
     × ('q × ('a, 'c) ae_stage)
     × (nat ⇒ ('c ⇒ 'a))
     × (nat ⇒ dir)) set where
 ae_delta_ss2_ss3 M =
  {((q, ofs, buf, dest, SS2), a,
    (q, ofs, buf', dest, SS3), a, d) |
   q ofs buf dest a buf' d.
    q ∈ Q_tm M ∧ q ≠ t_tm M ∧ q ≠ r_tm M
    ∧ (∀j≥k_tm M. a j = bl_block (bl_tm M))
```

```
    ∧ buf' = (λk. if k < k_tm M
                then (let (l, h, r) = buf k in
                        (if h = LE_block (le_tm M)
                           then bl_block (bl_tm M)
                           else a k,
                         h, r))
                else init_buffer (le_tm M) k)
    ∧ d = (λk. if k < k_tm M
             then (if (fst (snd (buf k))) = LE_block (le_tm M)
                    then dir.N else dir.R)
             else dir.N)}
```

SS3 → SS4: uniform R move per tape (no buffer update, no M-state advance). Both steady-state and LE-stage move R (steady-state from home to right; LE-stage from home (= LE) to right neighbour, satisfying δLE since R is allowed from LE). Buffer-phase substep 3.

```
definition ae_delta_ss3_ss4 ::
  ('q, 'a) mttm
    ⇒ (('q × ('a, 'c :: enum) ae_stage)
        × (nat ⇒ ('c ⇒ 'a))
        × ('q × ('a, 'c) ae_stage)
        × (nat ⇒ ('c ⇒ 'a))
        × (nat ⇒ dir)) set where
  ae_delta_ss3_ss4 M =
    {((q, ofs, buf, dest, SS3), a,
       (q, ofs, buf, dest, SS4), a, d) |
     q ofs buf dest a d.
       q ∈ Q_tm M ∧ q ≠ t_tm M ∧ q ≠ r_tm M
       ∧ (∀j≥k_tm M. a j = bl_block (bl_tm M))
       ∧ d = (λk. if k < k_tm M then dir.R else dir.N)}
```

Buffered head position: which buffer slot (*AE_Left* / *AE_Home* / *AE_Right*) and which offset within that slot. Internal to the compute substep; the output of *m_steps_buffered* projects this onto (*nat* ⇒ $'c$) × (*nat* ⇒ *ae_dest*).

```
type_synonym 'c bp = ae_dest × 'c
```

Read the symbol at a buffered head position.

```
definition read_bp ::
  (('c :: enum ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
    ⇒ 'c bp ⇒ 'a where
  read_bp blocks p =
    (let (b, off) = p; (l, h, r) = blocks in
       case b of AE_Left ⇒ l off
               | AE_Home ⇒ h off
               | AE_Right ⇒ r off)
```

Write a symbol at a buffered head position.

```
definition write_bp ::
```

```
(('c :: enum ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
  ⇒ 'c bp ⇒ 'a
  ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)) where
write_bp blocks p x =
  (let (b, off) = p; (l, h, r) = blocks in
    case b of AE_Left  ⇒ (l(off := x), h, r)
            | AE_Home  ⇒ (l, h(off := x), r)
            | AE_Right ⇒ (l, h, r(off := x)))
```

Advance a buffered head position by a direction. Returns *None* if the head would exit the 3-block buffer (which by the per-c-step head-displacement bound of Hopcroft–Ullman [6, Theorem 12.3] cannot happen during a single c-step compute starting from (*AE_Home*, *ofs*); relations using *bp_advance* filter the *None* case as a vacuous side condition).

```
definition bp_advance ::
  ('c :: enum) bp ⇒ dir ⇒ 'c bp option where
  bp_advance p d =
    (let (b, off) = p in
      case d of
        dir.N ⇒ Some (b, off)
      | dir.R ⇒
          (case c_succ off of
             Some off' ⇒ Some (b, off')
           | None ⇒
               (case b of
                  AE_Left  ⇒ Some (AE_Home,  c_first)
                | AE_Home  ⇒ Some (AE_Right, c_first)
                | AE_Right ⇒ None))
      | dir.L ⇒
          (case c_pred off of
             Some off' ⇒ Some (b, off')
           | None ⇒
               (case b of
                  AE_Right ⇒ Some (AE_Home, c_last)
                | AE_Home  ⇒ Some (AE_Left, c_last)
                | AE_Left  ⇒ None)))
```

LE-aware buffered head advance. Wraps *bp_advance* with the substrate-induced LE-skip rule: when the read symbol is *le* (i.e., the head is positioned within an *LE_M'* block) and the move is R, jump to (*AE_Right*, *c_first*) rather than advancing within the home block. This corresponds to M's actual head crossing from position 0 (LE) to position 1 (the first input cell), which the simulation correspondence requires as a single bp-step rather than a sequence of c within-block bp-steps.

Rationale: the *LE_M'* block is a c-tuple but only its slot 0 represents a real M-cell; slots 1..c-1 are structural padding. *bp_advance*'s standard offset arithmetic would walk through these padding slots, which doesn't correspond to any M-step. N stays in place (consistent with M staying at

LE on N), L is forbidden by δLE at LE so the case is unreachable.

definition *bp_advance_le* ::
  $'a \Rightarrow 'a \Rightarrow ('c :: enum)\ bp \Rightarrow dir \Rightarrow 'c\ bp\ option$ **where**
  bp_advance_le le a p d =
    (*if a* = *le* ∧ *fst p* = *AE_Home* ∧ *d* = *dir.R*
      then Some (*AE_Right*, *c_first*)
      else bp_advance p d)

Single buffered M-step: applies M's δ to the current per-tape buffered reads, writes the post-symbols back into the buffer, and advances each per-tape head position via *bp_advance_le* (the LE-aware wrapper around *bp_advance*). The relation is empty for configurations whose source M-state is halting (M's δ excludes those) or whose head movement would exit the 3-block buffer on any tape.

definition *m_step_buffered* ::
  $('q, 'a)\ mttm$
    ⇒ (($'q$
        × (*nat* ⇒ ($'c$ :: *enum* ⇒ $'a$) × ($'c$ ⇒ $'a$) × ($'c$ ⇒ $'a$))
        × (*nat* ⇒ $'c$ *bp*))
       × ($'q$
          × (*nat* ⇒ ($'c$ ⇒ $'a$) × ($'c$ ⇒ $'a$) × ($'c$ ⇒ $'a$))
          × (*nat* ⇒ $'c$ *bp*))) *set* **where**
  m_step_buffered M =
    {((*q*, *blocks*, *pos*), (*q'*, *blocks'*, *pos'*)) |
      q blocks pos q' blocks' pos' a a' d.
        (*q*, *a*, *q'*, *a'*, *d*) ∈ *delta_tm M*
        ∧ (∀ *k*. *a k* = *read_bp* (*blocks k*) (*pos k*))
        ∧ (∀ *k*. *blocks' k* = *write_bp* (*blocks k*) (*pos k*) (*a' k*))
        ∧ (∀ *k*. *bp_advance_le* (*le_tm M*) (*a k*) (*pos k*) (*d k*)
              = *Some* (*pos' k*))}

Introduction rule for *m_step_buffered*: package the four ingredients (δ-tuple, read-from-buffer match, write-back, head-advance) into the relational membership claim. Used by the inductive step of *ae_m_steps_buffered_correct* (*AlphabetEnlargement.thy*) to extend a coupled run by one step.

lemma *m_step_bufferedI*:
  fixes *M* :: $('q, 'a)\ mttm$
    and *a a'* :: *nat* ⇒ $'a$
    and *d* :: *nat* ⇒ *dir*
    and *blocks blocks'* ::
        nat ⇒ (($'c$ :: *enum* ⇒ $'a$) × ($'c$ ⇒ $'a$) × ($'c$ ⇒ $'a$))
    and *pos pos'* :: *nat* ⇒ $'c$ *bp*
  assumes (*q*, *a*, *q'*, *a'*, *d*) ∈ *delta_tm M*
      and ∀ *k*. *a k* = *read_bp* (*blocks k*) (*pos k*)
      and ∀ *k*. *blocks' k* = *write_bp* (*blocks k*) (*pos k*) (*a' k*)
      and ∀ *k*. *bp_advance_le* (*le_tm M*) (*a k*) (*pos k*) (*d k*)
            = *Some* (*pos' k*)

shows $((q,\ blocks,\ pos),\ (q',\ blocks',\ pos')) \in m_step_buffered\ M$
using *assms* **unfolding** *m_step_buffered_def* **by** *blast*

Auxiliary: up-to-c-step composition of M's δ on the buffered representation. Captures the cumulative effect of either c consecutive M-steps, or fewer if M reaches a halting state (*t_M* / *r_M*) earlier.

Defined as the union of n-fold relational compositions of *m_step_buffered M* for $n \le c = card\ (UNIV :: 'c\ set)$, filtered to enforce the early-stop discipline (the run runs the full c steps unless M halts). This is the semantic core of the compute substep (SS4→SS5).

```
definition m_steps_buffered ::
  ('q, 'a) mttm
    ⇒ (('q
         × (nat ⇒ ('c :: enum ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
         × (nat ⇒ 'c bp))
        × ('q
           × (nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
           × (nat ⇒ 'c bp))) set where
  m_steps_buffered M =
    {(s, s') | s s' n.
       n ≤ card (UNIV :: 'c set)
       ∧ (s, s') ∈ (m_step_buffered M) ^^ n
       ∧ (n = card (UNIV :: 'c set)
          ∨ fst s' ∈ {t_tm M, r_tm M})}
```

Functionality of *m_step_buffered* under *det_mttm M*: a single buffered M-step from a fixed source determines the target uniquely. Threaded through *m_step_buffered_relpow_functional* and *m_steps_buffered_functional* below into the SS4→SS5 compute substep's functionality lemma the 16th entry in the per-substep functionality cluster, completing it under *det_mttm M*.

```
lemma m_step_buffered_functional:
  fixes M :: ('q, 'a) mttm
  assumes det: det_mttm M
    and h1: ((q, blocks, pos), (q1, blocks1, pos1)) ∈ m_step_buffered M
    and h2: ((q, blocks, pos), (q2, blocks2, pos2)) ∈ m_step_buffered M
  shows (q1, blocks1, pos1) = (q2, blocks2, pos2)
proof −
  from h1 obtain a1 a1' d1 where
      t1: (q, a1, q1, a1', d1) ∈ delta_tm M
    and r1: ∀k. a1 k = read_bp (blocks k) (pos k)
    and w1: ∀k. blocks1 k = write_bp (blocks k) (pos k) (a1' k)
    and p1: ∀k. bp_advance_le (le_tm M) (a1 k) (pos k) (d1 k)
                  = Some (pos1 k)
    unfolding m_step_buffered_def by blast
  from h2 obtain a2 a2' d2 where
      t2: (q, a2, q2, a2', d2) ∈ delta_tm M
    and r2: ∀k. a2 k = read_bp (blocks k) (pos k)
```

```
    and w2: ∀k. blocks2 k = write_bp (blocks k) (pos k) (a2' k)
    and p2: ∀k. bp_advance_le (le_tm M) (a2 k) (pos k) (d2 k)
                = Some (pos2 k)
    unfolding m_step_buffered_def by blast
  have a_eq: a1 = a2
    using r1 r2 by (intro ext) auto
  have t2': (q, a1, q2, a2', d2) ∈ delta_tm M
    using t2 a_eq by simp
  from det t1 t2'
  have qad: (q1, a1', d1) = (q2, a2', d2)
    unfolding det_mttm_def by blast
  hence q_eq: q1 = q2 and a'_eq: a1' = a2' and d_eq: d1 = d2
    by auto
  have b_eq: blocks1 = blocks2
    using w1 w2 a'_eq by (intro ext) auto
  have p_eq: pos1 = pos2
  proof (intro ext)
    fix k
    have Some (pos1 k) = Some (pos2 k)
      using p1 p2 a_eq d_eq by metis
    thus pos1 k = pos2 k by simp
  qed
  show ?thesis using q_eq b_eq p_eq by simp
qed
```

Relational-power lift of *m_step_buffered_functional*: under *det_mttm M*, n-fold composition is functional too. Standard induction-on-n proof using the single-step lemma.

```
lemma m_step_buffered_relpow_functional:
  fixes M :: ('q, 'a) mttm
  assumes det: det_mttm M
      and h1: (s, s1) ∈ (m_step_buffered M) ^^ n
      and h2: (s, s2) ∈ (m_step_buffered M) ^^ n
  shows s1 = s2
  using h1 h2
proof (induction n arbitrary: s s1 s2)
  case 0
  thus ?case by simp
next
  case (Suc n)
  from Suc.prems(1) obtain s1' where
      step1: (s, s1') ∈ m_step_buffered M
    and rest1: (s1', s1) ∈ (m_step_buffered M) ^^ n
    by (meson relpow_Suc_D2)
  from Suc.prems(2) obtain s2' where
      step2: (s, s2') ∈ m_step_buffered M
    and rest2: (s2', s2) ∈ (m_step_buffered M) ^^ n
    by (meson relpow_Suc_D2)
  obtain q b p where s_eq: s = (q, b, p) by (cases s)
```

```
  obtain q1' b1' p1' where s1'_eq: s1' = (q1', b1', p1') by (cases s1')
  obtain q2' b2' p2' where s2'_eq: s2' = (q2', b2', p2') by (cases s2')
  from step1 s_eq s1'_eq
  have st1: ((q, b, p), (q1', b1', p1')) ∈ m_step_buffered M
    by simp
  from step2 s_eq s2'_eq
  have st2: ((q, b, p), (q2', b2', p2')) ∈ m_step_buffered M
    by simp
  from m_step_buffered_functional[OF det st1 st2]
  have (q1', b1', p1') = (q2', b2', p2') .
  hence s'_eq: s1' = s2' using s1'_eq s2'_eq by simp
  show ?case using Suc.IH[OF rest1] rest2 s'_eq by simp
qed
```

From a halt state, no buffered M-step is possible. Follows from *valid_mttm M*'s structural constraint that *delta_tm M* has no transitions originating in $\{t_tm\ M,\ r_tm\ M\}$. Used below to rule out the case where the two witnesses of *m_steps_buffered*'s functionality argument use different step counts.

```
lemma m_step_buffered_no_halt:
  fixes M :: ('q, 'a) mttm
  assumes vM: valid_mttm M
      and halt: q ∈ {t_tm M, r_tm M}
  shows ((q, blocks, pos), s') ∉ m_step_buffered M
proof
  assume ((q, blocks, pos), s') ∈ m_step_buffered M
  then obtain a q' a' d where
      st: (q, a, q', a', d) ∈ delta_tm M
    unfolding m_step_buffered_def by blast
  from valid_mttm_delta_set[OF vM] st
  have q ∈ Q_tm M − {t_tm M, r_tm M} by auto
  with halt show False by auto
qed
```

Functionality of *m_steps_buffered* under *det_mttm M*: the bounded-and-halt-truncated buffered M-run is functional in its source.

Argument: from membership we obtain step counts *n1*, *n2* with $n_i \le c$ and $n_i = c \vee fst\ s_i \in \{t,\ r\}$. First show *n1* = *n2*: if $n1 < n2$ the prefix run determines the n1-step state to be *s1* (by *m_step_buffered_relpow_functional*), and the remaining $n2 - n1 \ge 1$ steps require a transition from *s1*. The disjunction on *s1* forces either *n1* = *c* (contradicting $n1 < n2 \le c$) or $fst\ s1 \in \{t,\ r\}$, the latter ruled out by *m_step_buffered_no_halt*. Symmetric for $n2 < n1$. With *n1* = *n2*, the relpow functional lemma finishes.

```
lemma m_steps_buffered_functional:
  fixes M :: ('q, 'a) mttm
    and s s1 s2 :: 'q
                  × (nat ⇒ ('c :: enum ⇒ 'a)
```

```
                                  × ('c ⇒ 'a) × ('c ⇒ 'a))
                            × (nat ⇒ 'c bp)
  assumes vM:  valid_mttm M
      and det: det_mttm M
      and h1:  (s, s1) ∈ m_steps_buffered M
      and h2:  (s, s2) ∈ m_steps_buffered M
  shows s1 = s2
proof −
  let ?c = card (UNIV :: 'c set)
  let ?R = m_step_buffered M
  from h1 obtain n1 where
      n1_bnd:  n1 ≤ ?c
    and n1_run: (s, s1) ∈ ?R ^^ n1
    and n1_halt: n1 = ?c ∨ fst s1 ∈ {t_tm M, r_tm M}
    unfolding m_steps_buffered_def by blast
  from h2 obtain n2 where
      n2_bnd:  n2 ≤ ?c
    and n2_run: (s, s2) ∈ ?R ^^ n2
    and n2_halt: n2 = ?c ∨ fst s2 ∈ {t_tm M, r_tm M}
    unfolding m_steps_buffered_def by blast

  — Sub-lemma: under m < n, the run from the m-step state cannot extend,
contradicting the halt-or-full disjunction at the m-step state.
  have no_extend:
    ⋀m n s_m s_n.
       (s, s_m) ∈ ?R ^^ m
       ⟹ (s, s_n) ∈ ?R ^^ n
       ⟹ m < n
       ⟹ n ≤ ?c
       ⟹ m = ?c ∨ fst s_m ∈ {t_tm M, r_tm M}
       ⟹ False
  proof −
    fix m n :: nat and s_m s_n
    assume rm: (s, s_m) ∈ ?R ^^ m
      and rn: (s, s_n) ∈ ?R ^^ n
      and mn: m < n
      and nc: n ≤ ?c
      and hl: m = ?c ∨ fst s_m ∈ {t_tm M, r_tm M}
    define k where k = n − m − 1
    have k_eq: n = m + Suc k
      unfolding k_def using mn by simp
    from rn k_eq have rn': (s, s_n) ∈ ?R ^^ (m + Suc k) by simp
    from rn'[unfolded relpow_add]
    obtain s' where
       rpre: (s, s') ∈ ?R ^^ m
      and rpost: (s', s_n) ∈ ?R ^^ Suc k
      by blast
    from m_step_buffered_relpow_functional[OF det rm rpre]
    have s_eq: s_m = s' .
```

```
    from rpost obtain s'' where step: (s', s'') ∈ ?R
      by (meson relpow_Suc_D2)
    with s_eq have step_sm: (s_m, s'') ∈ ?R by simp
    from hl show False
    proof
      assume m = ?c
      with k_eq nc show False by linarith
    next
      assume halt: fst s_m ∈ {t_tm M, r_tm M}
      obtain q b p where sm_eq: s_m = (q, b, p) by (cases s_m)
      from halt sm_eq have q_halt: q ∈ {t_tm M, r_tm M} by simp
      from step_sm sm_eq have ((q, b, p), s'') ∈ ?R by simp
      with m_step_buffered_no_halt[OF vM q_halt] show False by blast
    qed
  qed

  — The step counts must be equal.
  have n_eq: n1 = n2
  proof (rule ccontr)
    assume n1 ≠ n2
    then consider (lt) n1 < n2 | (gt) n2 < n1 by linarith
    thus False
    proof cases
      case lt
      from no_extend[OF n1_run n2_run lt n2_bnd n1_halt] show False .
    next
      case gt
      from no_extend[OF n2_run n1_run gt n1_bnd n2_halt] show False .
    qed
  qed
  from n1_run n2_run n_eq
  show ?thesis
    using m_step_buffered_relpow_functional[OF det] by metis
qed
```

Linearisation helpers for the 3-block buffer. These map (*block*, *offset*)-pairs to indices in $[0, 3c)$ and read the symbol at a linearised buffer index. Used to express the buffered compute's correctness as a contiguous-tape-window match.

```
definition bp_linear :: ('c :: enum) bp ⇒ nat where
  bp_linear p =
    (case fst p of
       AE_Left  ⇒ 0
     | AE_Home  ⇒ card (UNIV :: 'c set)
     | AE_Right ⇒ 2 * card (UNIV :: 'c set))
    + c_idx (snd p)

definition buf_lin_at ::
  (('c :: enum ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
```

```
  ⇒ nat ⇒ 'a where
 buf_lin_at blocks i =
   (let c = card (UNIV :: 'c set);
        enum_c = (enum_class.enum :: 'c list);
        (l, h, r) = blocks in
      if i < c then l (enum_c ! i)
      else if i < 2 * c then h (enum_c ! (i − c))
      else r (enum_c ! (i − 2 * c)))
```

Window invariant on a 3-block buffer plus buffered head: (1) the buffer's linearisation matches an M-tape window of 3c contiguous positions starting at *p_start*; (2) the buffered head decodes to the actual M-head position via *bp_linear*; (3) the entire window lies in the non-LE region of the M-tape (*p_start* ≥ *1*). The non-LE precondition keeps the predicate steady-state; the LE-edge case (window intersects M-position 0) is handled by a separate lemma at the simulation level.

```
definition ae_window_invariant ::
  (nat ⇒ 'a) ⇒ nat
    ⇒ ('c :: enum) bp
    ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
    ⇒ nat ⇒ bool where
  ae_window_invariant tM nM bp blocks p_start ⟷
     p_start ≥ 1
     ∧ nM = p_start + bp_linear bp
     ∧ (∀ i. i < 3 * card (UNIV :: 'c set)
             ⟶ tM (p_start + i) = buf_lin_at blocks i)
```

LE-edge analogue of *ae_window_invariant* for the *le1* sub-case: *M'*'s head is at block 1 (the first content block), so the buffer's left slot is *LE_block le* (block 0's content under *ae_init_config*) and M's tape position lies in {*0*, ..., *2c*}. The home and right buffer slots linearise to an M-tape window starting at position 1; the left buffer slot is *LE_block* by construction, with no claim about M-tape positions *0*, ..., *c−1* beyond *tM 0* = *le*. Under *δLE* on *delta_tm M*, M's buffered trajectory in this setting visits only (*AE_Left*, *c_last*) within the left block (when reading LE), never the other left slots, so the buffer↔tape mismatch there is harmless.

The buffered head *bp* decodes to the actual M-tape position by a three-way case-split (*AE_Left c_last* ↦ *0*, *AE_Home* ↦ *1* + *c_idx ofs*, *AE_Right* ↦ *1* + *c* + *c_idx ofs*) so the predicate serves as both SS4 entry condition (*bp* = (*AE_Home*, *ofs*)) and post-compute condition (*bp* anywhere in the three-block buffer except *AE_Left* at non-*c_last* offsets, which the *δLE*-respecting buffered trajectory cannot reach).

```
definition ae_window_invariant_le1 ::
  (nat ⇒ 'a) ⇒ nat
    ⇒ ('c :: enum) bp
    ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
    ⇒ 'a ⇒ bool where
```

```
ae_window_invariant_le1 tM nM bp blocks le ⟷
  fst blocks = LE_block le
  ∧ tM 0 = le
  ∧ (case fst bp of
       AE_Left  ⇒ snd bp = c_last ∧ nM = 0
     | AE_Home  ⇒ nM = Suc (c_idx (snd bp))
     | AE_Right ⇒ nM = Suc (card (UNIV :: 'c set) + c_idx (snd bp)))
  ∧ (∀ i. i < 2 * card (UNIV :: 'c set)
        ⟶ tM (Suc i)
            = buf_lin_at blocks (card (UNIV :: 'c set) + i))
```

LE-edge analogue of *ae_window_invariant* for the *le0* sub-case: M''s head is at block 0 (the LE block itself). Buffer shape: the home slot is *LE_block le* (block 0's content), the right slot is block 1's content (real input/blank), and the left slot holds an arbitrary sentinel (SS2 → SS3 installs *bl_block* (*bl_tm M*); the predicate doesn't constrain it because M's buffered trajectory cannot reach the left block under δLE M starts at home reading LE, can only move N or R, and any R-move from home reading LE jumps via *bp_advance_le*'s special case to (*AE_Right*, *c_first*), bypassing the rest of home and never visiting left).

The buffered head *bp* decodes: *AE_Home* ↦ *0* (M sits at LE regardless of *snd bp*, since home is all-LE) and *AE_Right* ↦ *1* + *c_idx ofs* (M is in block 1's range). *AE_Left* is excluded.

Only the right slot's linearisation is asserted: the home slot is fully LE (so linearisation matches *tM* only at position 0, which *tM 0* = *le* covers; the rest of home's linearisation is fake but unread).

```
definition ae_window_invariant_le0 ::
  (nat ⇒ 'a) ⇒ nat
    ⇒ ('c :: enum) bp
    ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
    ⇒ 'a ⇒ bool where
  ae_window_invariant_le0 tM nM bp blocks le ⟷
    fst (snd blocks) = LE_block le
    ∧ tM 0 = le
    ∧ (case fst bp of
         AE_Home  ⇒ nM = 0
       | AE_Right ⇒ nM = Suc (c_idx (snd bp))
       | AE_Left  ⇒ False)
    ∧ (∀ i. i < card (UNIV :: 'c set)
          ⟶ tM (Suc i)
              = buf_lin_at blocks (2 * card (UNIV :: 'c set) + i))
```

Per-tape unified window invariant for the SS4→SS5 trace toolkit. Hybrid encapsulation of the three regime-specific sibling predicates: one extra parameter *pos* (the per-tape *mt_pos c' k* at SS4 entry, a frozen value during the buffered M-side run) selects which sibling fires. Each regime is expressed as an implication, so the dispatch is by partition of *pos* :: *nat* into *0* / *1* /

≥ 2 exactly one implication is non-vacuous on any fixed *pos*.

Design choice rationale: the conjunction-of-implications form avoids forcing consumers to disjunction-eliminate before getting at the relevant conjunct, while the indexing by *pos* (rather than a uniform regime tag) lets the predicate be instantiated directly from the consumer's per-tape *mt_pos* c' k without an extra dispatch parameter. The fixed-regime-per-tape property *pos* doesn't change during the buffered run because *m_step_buffered* is parameterised by (q, *blocks*, pos_{bp}) with no c' in scope makes per-tape regime selection commute with the induction in *ae_coupled_run_aux_general*.

```
definition ae_window_invariant_general ::
  (nat ⇒ 'a) ⇒ nat
    ⇒ ('c :: enum) bp
    ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
    ⇒ nat ⇒ 'a ⇒ bool where
  ae_window_invariant_general tM nM bp blocks pos le ⟷
     (pos = 0 ⟶ ae_window_invariant_le0 tM nM bp blocks le)
     ∧ (pos = 1 ⟶ ae_window_invariant_le1 tM nM bp blocks le)
     ∧ (pos ≥ 2
          ⟶ ae_window_invariant tM nM bp blocks
               ((pos − 2) ∗ card (UNIV :: 'c set) + 1))
```

SS4 → SS5: read right into buffer; apply the compute (c-fold composition of M's δ) to determine the post-stage M-state, the per-tape destination indicator, the modified buffer slots, and the new per-tape offset; per-tape move L back to home. Buffer-phase substep 4 (compute folded in).

The substrate write $a' = a$ is a no-op (SS4 reads the right block but does not modify the on-tape contents; modifications are materialised during the write-back phase SS5→SS8). The compute happens in the state component: *m_steps_buffered* consumes the (fully buffered) blocks plus (*AE_Home*, *ofs*) starting position and produces the post-compute (q', *buf'*, *end_pos*); the new offset and destination are projected from *end_pos*.

```
definition ae_delta_ss4_ss5 ::
  ('q, 'a) mttm
    ⇒ (('q × ('a, 'c :: enum) ae_stage)
       × (nat ⇒ ('c ⇒ 'a))
       × ('q × ('a, 'c) ae_stage)
       × (nat ⇒ ('c ⇒ 'a))
       × (nat ⇒ dir)) set where
  ae_delta_ss4_ss5 M =
     {((q, ofs, buf, dest_old, SS4), a,
       (q', ofs', buf', dest', SS5), a, d) |
       q ofs buf dest_old a q' ofs' buf' dest' d
       buf_full end_pos bufC.
         q ∈ Q_tm M ∧ q ≠ t_tm M ∧ q ≠ r_tm M
         ∧ (∀ j≥k_tm M. a j = bl_block (bl_tm M))
         ∧ buf_full = (λk. (fst (buf k),
                            if k < k_tm M then fst (snd (buf k)) else a k,
```

$$
\begin{aligned}
&\qquad a\ k))\\
&\wedge\ ((q,\ buf_full,\ \lambda k.\ (AE_Home,\ ofs\ k)),\\
&\quad (q',\ bufC,\ end_pos)) \in m_steps_buffered\ M\\
&\wedge\ ofs' = (\lambda k.\ \mathit{if}\ k < k_tm\ M\ \mathit{then}\ snd\ (end_pos\ k)\ \mathit{else}\ init_offset\ k)\\
&\wedge\ buf' = (\lambda k.\ \mathit{if}\ k < k_tm\ M\ \mathit{then}\ bufC\ k\ \mathit{else}\ init_buffer\ (le_tm\ M)\ k)\\
&\wedge\ dest' = (\lambda k.\ \mathit{if}\ k < k_tm\ M\ \mathit{then}\ fst\ (end_pos\ k)\ \mathit{else}\ init_dest\ k)\\
&\wedge\ d = (\lambda k.\ \mathit{if}\ k < k_tm\ M\\
&\qquad\qquad \mathit{then}\ (\mathit{if}\ a\ k = LE_block\ (le_tm\ M)\\
&\qquad\qquad\qquad \mathit{then}\ dir.N\ \mathit{else}\ dir.L)\\
&\qquad\qquad \mathit{else}\ dir.N)\}
\end{aligned}
$$

Per-tape (write, move) action helpers for the four write-back substeps (SS5→SS6, SS6→SS7, SS7→SS8, SS8→SS1). Each takes (le, a, buf, $dest$) and returns (a', d): the per-tape written block and direction.

Convention: $buf\ k = (l,\ h,\ r)$ is the post-compute buffer. LE-stage $\longleftrightarrow$ $h = LE_block\ le$. In steady-state, $dest$ ranges over $\{AE_Left,\ AE_Home,\ AE_Right\}$; in LE-stage, the compute restricts $dest$ to $\{AE_Home,\ AE_Right\}$ but the helpers handle the AE_Left branch as a vacuous fall-through (will never fire under the compute's invariants).

Head trajectory through the writeback chain. The direction rules in the action helpers below thread M''s head through a specific sequence of block positions across SS5 → SS8. Starting from the SS4 → SS5 transition (which moves L when reading non-LE, so SS5 entry is at block s, the original home), the chain walks:

- SS5: at block s; writes h; moves L ($dest \neq AE_Left$) or R ($dest = AE_Left$).
- SS6: at block $s-1$ or $s+1$; writes l/r depending on $dest$; moves back toward home.
- SS7: at block s; writes a (idempotent); moves L ($dest = AE_Left$) or R (otherwise).
- SS8: at block $s+1$ ($dest \in \{AE_Home,\ AE_Right\}$) or $s-1$ ($dest = AE_Left$); writes r/l; lands at block $s + dest_offset$ for the next stage.

  Why this matters for the LE-edge cases. For steady-state ($s \geq 2$), the walk stays in data territory and the LE-guard never fires. For le1 ($s = 1$), the walk reaches **block 0 the LE position** at SS6 entry (when $dest \neq AE_Left$) or at SS8 entry (when $dest = AE_Left$). At those moments the LE-guard branch fires (head reads LE_block), writing LE_block back idempotently. The side-band invariant $ae_position_link$ records that these LE-guard firings happen at exactly the substeps where the default branch would otherwise write the buffer's l-slot (which is LE_block in le1, having been loaded from block 0) to a non-LE position the pre-emption that keeps the encoding consistent.

LE-guard prefix and δLE compatibility. Each action helper prepends an LE-guard *if* $a = LE_block\ le$ *then* (a, N) *else* ... for syntactic δLE-compatibility: the per-substep relations are over all $(state, a, \ldots)$ tuples, not just reachable ones, and the substrate's δLE well-formedness conjunct (a *valid_mttm* clause) is universal. In reachable executions $a = h$ at SS5→SS8 (head at the home position), so the guard agrees with the $h = LE_block\ le$ branch; in unreachable tuples ($a = LE_block\ le$ but $h \neq LE_block\ le$), the guard forces a δLE-safe (a, N) output rather than the buffer-driven $(\ldots, L)$ that would violate δLE.

This guard serves a double purpose: substrate compatibility (the immediate concern above) AND the LE-pre-emption used by le1/le0. In those regimes the head genuinely reaches block 0, the LE-guard's $a = LE_block$ antecedent is true, and the guard fires the LE-block-write branch in preference to the default protecting block 0 from being clobbered by a non-LE buffer slot.

```
fun ae_ss5_action ::
  'a ⇒ ('c :: enum ⇒ 'a)
    ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
    ⇒ ae_dest ⇒ (('c ⇒ 'a) × dir) where
  ae_ss5_action le a (l, h, r) ds =
     (if a = LE_block le then (a, dir.N)
      else if h = LE_block le then (a, dir.N)
      else (h, if ds = AE_Left then dir.R else dir.L))

fun ae_ss6_action ::
  'a ⇒ ('c :: enum ⇒ 'a)
    ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
    ⇒ ae_dest ⇒ (('c ⇒ 'a) × dir) where
  ae_ss6_action le a (l, h, r) ds =
     (if a = LE_block le then (a, dir.R)
      else if h = LE_block le then (a, dir.R)
      else if ds = AE_Left then (r, dir.L) else (l, dir.R))

fun ae_ss7_action ::
  'a ⇒ ('c :: enum ⇒ 'a)
    ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
    ⇒ ae_dest ⇒ (('c ⇒ 'a) × dir) where
  ae_ss7_action le a (l, h, r) ds =
     (if a = LE_block le then (a, dir.N)
      else if h = LE_block le
        then (r, if ds = AE_Right then dir.N else dir.L)
      else (a, if ds = AE_Left then dir.L else dir.R))

fun ae_ss8_action ::
  'a ⇒ ('c :: enum ⇒ 'a)
    ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
    ⇒ ae_dest ⇒ (('c ⇒ 'a) × dir) where
```

```
ae_ss8_action le a (l, h, r) ds =
  (if a = LE_block le then (a, dir.N)
   else if h = LE_block le
     then (if ds = AE_Right then (r, dir.N) else (a, dir.N))
   else (case ds of
           AE_Left  ⇒ (l, dir.N)
         | AE_Home  ⇒ (r, dir.L)
         | AE_Right ⇒ (r, dir.N)))
```

SS5 → SS6: write home block; per-tape move depends on (*stage_kind k*, *dest k*). Write-back substep 5.

```
definition ae_delta_ss5_ss6 ::
  ('q, 'a) mttm
  ⇒ (('q × ('a, 'c :: enum) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × ('q × ('a, 'c) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × (nat ⇒ dir)) set where
  ae_delta_ss5_ss6 M =
    {((q, ofs, buf, dest, SS5), a,
      (q, ofs, buf, dest, SS6), a', d) |
     q ofs buf dest a a' d.
      q ∈ Q_tm M
      ∧ (∀j≥k_tm M. a j = bl_block (bl_tm M))
      ∧ a' = (λk. if k < k_tm M
                  then fst (ae_ss5_action (le_tm M) (a k) (buf k) (dest k))
                  else bl_block (bl_tm M))
      ∧ d  = (λk. if k < k_tm M
                  then snd (ae_ss5_action (le_tm M) (a k) (buf k) (dest k))
                  else dir.N)}
```

SS6 → SS7: write left (steady-state, *dest* ∈ {*AE_Home*, *AE_Right*}) or right (steady-state, *dest* = *AE_Left*) or home (LE-stage); per-tape move per dest. Write-back substep 6.

```
definition ae_delta_ss6_ss7 ::
  ('q, 'a) mttm
  ⇒ (('q × ('a, 'c :: enum) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × ('q × ('a, 'c) ae_stage)
      × (nat ⇒ ('c ⇒ 'a))
      × (nat ⇒ dir)) set where
  ae_delta_ss6_ss7 M =
    {((q, ofs, buf, dest, SS6), a,
      (q, ofs, buf, dest, SS7), a', d) |
     q ofs buf dest a a' d.
      q ∈ Q_tm M
      ∧ (∀j≥k_tm M. a j = bl_block (bl_tm M))
      ∧ a' = (λk. if k < k_tm M
                  then fst (ae_ss6_action (le_tm M) (a k) (buf k) (dest k))
```

```
                else bl_block (bl_tm M))
    ∧ d  = (λk. if k < k_tm M
                then snd (ae_ss6_action (le_tm M) (a k) (buf k) (dest k))
                else dir.N)}
```

SS7 → SS8: idempotent home re-write (steady-state) or right write (LE-stage); per-tape move per dest. Write-back substep 7.

```
definition ae_delta_ss7_ss8 ::
  ('q, 'a) mttm
    ⇒ (('q × ('a, 'c :: enum) ae_stage)
        × (nat ⇒ ('c ⇒ 'a))
        × ('q × ('a, 'c) ae_stage)
        × (nat ⇒ ('c ⇒ 'a))
        × (nat ⇒ dir)) set where
  ae_delta_ss7_ss8 M =
    {((q, ofs, buf, dest, SS7), a,
      (q, ofs, buf, dest, SS8), a', d) |
     q ofs buf dest a a' d.
      q ∈ Q_tm M
      ∧ (∀j≥k_tm M. a j = bl_block (bl_tm M))
      ∧ a' = (λk. if k < k_tm M
                  then fst (ae_ss7_action (le_tm M) (a k) (buf k) (dest k))
                  else bl_block (bl_tm M))
      ∧ d  = (λk. if k < k_tm M
                  then snd (ae_ss7_action (le_tm M) (a k) (buf k) (dest k))
                  else dir.N)}
```

SS8 → SS1: write final block at destination; head ends at *dest* position; substep counter resets to SS1. End of write-back phase.

Halt-aware destination: if q reached a halting state (*t_tm M* / *r_tm M*) during the compute substep, the end-of-stage state is forced to (q, *init_stage le_M*), which equals *t_M'* / *r_M'* by construction. This makes M' actually reach its canonical accept / reject state when M halts mid-stage, rather than stalling at SS5. For non-halting q, the (offset, buffer) pair is preserved for the next stage; *dest* resets to *AE_Home*.

```
definition ae_delta_ss8_ss1 ::
  ('q, 'a) mttm
    ⇒ (('q × ('a, 'c :: enum) ae_stage)
        × (nat ⇒ ('c ⇒ 'a))
        × ('q × ('a, 'c) ae_stage)
        × (nat ⇒ ('c ⇒ 'a))
        × (nat ⇒ dir)) set where
  ae_delta_ss8_ss1 M =
    {((q, ofs, buf, dest, SS8), a,
      (q, stage'), a', d) |
     q ofs buf dest a stage' a' d.
      q ∈ Q_tm M
      ∧ (∀j≥k_tm M. a j = bl_block (bl_tm M))
```

$\wedge$ *stage'* = (*if q* $\in$ {*t_tm M*, *r_tm M*}
then init_stage (*le_tm M*)
else (*ofs*, *buf*, *init_dest*, *SS1*))
$\wedge$ *a'* = (λ*k*. *if k* < *k_tm M*
then fst (*ae_ss8_action* (*le_tm M*) (*a k*) (*buf k*) (*dest k*))
else bl_block (*bl_tm M*))
$\wedge$ *d* = (λ*k*. *if k* < *k_tm M*
then snd (*ae_ss8_action* (*le_tm M*) (*a k*) (*buf k*) (*dest k*))
else dir.N)}

end
theory *AlphabetEnlargement_Delta*
imports *AlphabetEnlargement_Substeps*
begin

2.9 Transition relation (union of per-substep parts)

The output machine's tape alphabet Γ': the set of $'c$-blocks whose every cell is in M's tape alphabet Γ.

definition *gamma_block* ::
'a set $\Rightarrow$ (('*c* :: *enum*) $\Rightarrow$ *'a*) *set* **where**
gamma_block Γ = {*f*. *range f* $\subseteq$ Γ}

lemma *gamma_block_mono*: *A* $\subseteq$ *B* $\Longrightarrow$ *gamma_block A* $\subseteq$ *gamma_block B*
unfolding *gamma_block_def* **by** *auto*

lemma *finite_gamma_block*:
fixes *A* :: *'a set*
assumes *finA*: *finite A*
shows *finite* (*gamma_block A* :: ('*c* :: *enum* $\Rightarrow$ *'a*) *set*)
proof −
have *eq*: *gamma_block A* = Pi_E (*UNIV* :: *'c set*) (λ_. *A*)
by (*auto simp*: *gamma_block_def PiE_def Pi_def extensional_def*)
show *?thesis*
unfolding *eq* **by** (*intro finite_PiE*) (*auto simp*: *finA*)
qed

lemma *LE_block_in_gamma_block*:
le $\in$ Γ $\Longrightarrow$ *LE_block le* $\in$ *gamma_block* Γ
unfolding *gamma_block_def LE_block_def* **by** *auto*

lemma *bl_block_in_gamma_block*:
bl $\in$ Γ $\Longrightarrow$ *bl_block bl* $\in$ *gamma_block* Γ
unfolding *gamma_block_def bl_block_def* **by** *auto*

State-level buffer validity: every per-tape block triple stored in the stage's buffer component has all three blocks in *gamma_block* Γ. Spec-side counterpart to the config-level invariant *ae_buffer_in_gamma_block*; used as the buffer-component restriction in *alphabet_enlarge*'s output state set Q'. This

makes Q' finite from the set-level premise *finite* Γ alone without requiring $UNIV('c \Rightarrow 'a)$ to be a finite type, which would not be derivable from *finite* Γ generically.

```
definition ae_valid_stage ::
  'a set ⇒ 'a ⇒ nat ⇒ ('a, 'c :: enum) ae_stage ⇒ bool where
  ae_valid_stage Γ le K stg ⟷
    (case stg of (off, buf, dst, _) ⇒
      (∀k. fst (buf k) ∈ gamma_block Γ
          ∧ fst (snd (buf k)) ∈ gamma_block Γ
          ∧ snd (snd (buf k)) ∈ gamma_block Γ)
     ∧ (∀j≥K. off j = init_offset j)
     ∧ (∀j≥K. buf j = init_buffer le j)
     ∧ (∀j≥K. dst j = init_dest j))

lemma ae_valid_stage_init:
  assumes le ∈ Γ
  shows ae_valid_stage Γ le K (init_stage le)
  unfolding ae_valid_stage_def init_stage_def init_buffer_def
  using LE_block_in_gamma_block[OF assms] by simp

lemma init_buffer_in_gamma_block:
  fixes le :: 'a and Γ :: 'a set
    and ib :: nat ⇒ ('c :: enum ⇒ 'a)
                          × ('c ⇒ 'a) × ('c ⇒ 'a)
  assumes le_in: le ∈ Γ
      and ib_def: ib = init_buffer le
  shows ∀k. fst (ib k) ∈ gamma_block Γ
          ∧ fst (snd (ib k)) ∈ gamma_block Γ
          ∧ snd (snd (ib k)) ∈ gamma_block Γ
  using LE_block_in_gamma_block[OF le_in]
  by (auto simp: ib_def init_buffer_def)
```

Specialized variant for direct invocation at validation-phase call sites where the buffer is *init_buffer* (*le_tm M*). The buf-gamma claim is in the explicit shape that *ae_step_val_*∗'s new *buf_gamma* hypothesis expects. Anchors the polymorphic $'c$ via type annotations on every occurrence (cf. discussion in commit a812ed0).

```
lemma init_buffer_in_gamma_block_at_M:
  fixes M :: ('q, 'a) mttm
  assumes vM: valid_mttm M
  shows ∀kk :: nat.
          fst (init_buffer (le_tm M) kk
                :: ('c :: enum ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
            ∈ gamma_block (Γ_tm M)
          ∧ fst (snd (init_buffer (le_tm M) kk
                        :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)))
            ∈ gamma_block (Γ_tm M)
          ∧ snd (snd (init_buffer (le_tm M) kk
```

```
                    :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)))
          ∈ gamma_block (Γ_tm M)
proof (intro allI)
  fix kk :: nat
  have lg: (LE_block (le_tm M) :: 'c ⇒ 'a) ∈ gamma_block (Γ_tm M)
     by (rule LE_block_in_gamma_block[OF valid_mttm_LE_in_Gamma[OF
vM]])
  show fst (init_buffer (le_tm M) kk
            :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
          ∈ gamma_block (Γ_tm M)
        ∧ fst (snd (init_buffer (le_tm M) kk
                    :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)))
          ∈ gamma_block (Γ_tm M)
        ∧ snd (snd (init_buffer (le_tm M) kk
                    :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)))
          ∈ gamma_block (Γ_tm M)
    using lg by (simp add: init_buffer_def)
qed

lemma finite_ae_valid_stages:
  fixes Γ :: 'a set and le :: 'a and K :: nat
  assumes finG: finite Γ
  shows finite {stg :: ('a, 'c :: enum) ae_stage.
                ae_valid_stage Γ le K stg}
proof −
  let ?G  = gamma_block Γ :: ('c ⇒ 'a) set
  let ?GGG = ?G × ?G × ?G
  have fG: finite ?G by (rule finite_gamma_block[OF finG])
  have fGGG: finite ?GGG
    using fG by (intro finite_cartesian_product)
  — Each nat-indexed field is finite via finite_tail_const_funcs: constant beyond
K (the frozen-at-init tail), finite codomain.
  let ?offs = {off :: nat ⇒ 'c.
               (∀ j. off j ∈ (UNIV :: 'c set))
               ∧ (∀ j≥K. off j = init_offset j)}
  let ?bufs = {buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a).
               (∀ j. buf j ∈ ?GGG)
               ∧ (∀ j≥K. buf j = init_buffer le j)}
  let ?dsts = {dst :: nat ⇒ ae_dest.
               (∀ j. dst j ∈ (UNIV :: ae_dest set))
               ∧ (∀ j≥K. dst j = init_dest j)}
  have foffs: finite ?offs
    using finite_tail_const_funcs[OF finite_UNIV, of K SOME x :: 'c. True]
    by (simp add: init_offset_def)
  have fbufs: finite ?bufs
    using finite_tail_const_funcs[OF fGGG, of K init_buffer le 0]
    by (simp add: init_buffer_def)
  have fdsts: finite ?dsts
    using finite_tail_const_funcs[OF finite_UNIV, of K AE_Home]
```

```
    by (simp add: init_dest_def)
  let ?ENV = ?offs × ?bufs × ?dsts × (UNIV :: substep_idx set)
  have {stg :: ('a, 'c) ae_stage. ae_valid_stage Γ le K stg} ⊆ ?ENV
    unfolding ae_valid_stage_def
    by (auto simp: mem_Times_iff split: prod.splits)
  moreover have finite ?ENV
    using foffs fbufs fdsts by (intro finite_cartesian_product) simp_all
  ultimately show ?thesis by (rule finite_subset)
qed
```

LE-input simp rules for the per-tape action helpers. When the read block on tape k is the all-LE block, each helper's LE-guard prefix returns (*LE_block le*, *N*) (or (*LE_block le*, *R*) for SS6) regardless of the buffer triple's components. These rules unblock the δLE-preservation case analysis: *fun*-generated simp rules match only literal triples (l, h, r); named LE-input lemmas simplify the general application form via tuple destructuring.

```
lemma ae_ss5_action_LE [simp]:
  ae_ss5_action le (LE_block le) buf ds = (LE_block le, dir.N)
proof −
  obtain l h r where buf = (l, h, r) using prod.exhaust by metis
  thus ?thesis by simp
qed

lemma ae_ss6_action_LE [simp]:
  ae_ss6_action le (LE_block le) buf ds = (LE_block le, dir.R)
proof −
  obtain l h r where buf = (l, h, r) using prod.exhaust by metis
  thus ?thesis by simp
qed

lemma ae_ss7_action_LE [simp]:
  ae_ss7_action le (LE_block le) buf ds = (LE_block le, dir.N)
proof −
  obtain l h r where buf = (l, h, r) using prod.exhaust by metis
  thus ?thesis by simp
qed

lemma ae_ss8_action_LE [simp]:
  ae_ss8_action le (LE_block le) buf ds = (LE_block le, dir.N)
proof −
  obtain l h r where buf = (l, h, r) using prod.exhaust by metis
  thus ?thesis by simp
qed
```

State preservation for a single buffered M-step: if the source state is in Q, so is the destination. Direct consequence of the substrate's δ_set range obligation.

```
lemma m_step_buffered_state_preservation:
  fixes M :: ('q, 'a) mttm
```

```
  assumes valM: valid_mttm M
    and step: ((q, blocks, pos), (q', blocks', pos')) ∈ m_step_buffered M
    and qQ: q ∈ Q_tm M
  shows q' ∈ Q_tm M
proof −
  from step obtain a a' d where
    (q, a, q', a', d) ∈ delta_tm M
    unfolding m_step_buffered_def by blast
  thus q' ∈ Q_tm M
    using valid_mttm_delta_set[OF valM] by auto
qed
```

State preservation for the up-to-c-step composition: if the source state is in Q, so is the destination. Induction on the relation power.

```
lemma m_steps_buffered_state_preservation:
  fixes M :: ('q, 'a) mttm
  assumes valM: valid_mttm M
    and steps: ((q, blocks, pos), (q', blocks', pos')) ∈ m_steps_buffered M
    and qQ: q ∈ Q_tm M
  shows q' ∈ Q_tm M
proof −
  from steps obtain n where
    n_step: ((q, blocks, pos), (q', blocks', pos')) ∈ (m_step_buffered M) ^^ n
    unfolding m_steps_buffered_def by blast
  from n_step qQ show ?thesis
  proof (induction n arbitrary: q' blocks' pos')
    case 0
    then show ?case by auto
  next
    case (Suc n)
    from Suc.prems(1) obtain q'' blocks'' pos'' where
      step_n: ((q, blocks, pos), (q'', blocks'', pos''))
                ∈ (m_step_buffered M) ^^ n
      and step_one: ((q'', blocks'', pos''), (q', blocks', pos'))
                    ∈ m_step_buffered M
      by (auto elim: relpow_Suc_E)
    from Suc.IH[OF step_n Suc.prems(2)] have qQ'': q'' ∈ Q_tm M .
   from m_step_buffered_state_preservation[OF valM step_one qQ''] show ?case
.
  qed
qed
```

Buffer-gamma preservation under a single *m_step_buffered*: if every slot of every tape's buffer is in *gamma_block* (Γ_*tm* M) pre-step, the post-step buffer's slots are too. The substantive fact: *m_step_buffered*'s write updates exactly one cell of one slot per tape, the written value lies in Γ_*tm* M (by *valid_mttm_delta*), and *gamma_block* is closed under such point-updates.

```
lemma m_step_buffered_gamma_preserve:
  fixes M :: ('q, 'a) mttm
```

```
  assumes vM: valid_mttm M
      and step: ((q, blocks, pos), (q', blocks', pos')) ∈ m_step_buffered M
      and pre: ∀k. fst (blocks k) ∈ gamma_block (Γ_tm M)
                 ∧ fst (snd (blocks k)) ∈ gamma_block (Γ_tm M)
                 ∧ snd (snd (blocks k)) ∈ gamma_block (Γ_tm M)
    shows ∀k. fst (blocks' k) ∈ gamma_block (Γ_tm M)
            ∧ fst (snd (blocks' k)) ∈ gamma_block (Γ_tm M)
            ∧ snd (snd (blocks' k)) ∈ gamma_block (Γ_tm M)
proof −
  from step obtain a a' d where
      tr: (q, a, q', a', d) ∈ delta_tm M
      and bk_update: ∀k. blocks' k = write_bp (blocks k) (pos k) (a' k)
    unfolding m_step_buffered_def by auto
  show ?thesis
  proof (intro allI)
    fix k
    obtain l h r where bk_eq: blocks k = (l, h, r)
      by (cases blocks k) auto
    obtain b off where pk_eq: pos k = (b, off)
      by (cases pos k) auto
    from pre[rule_format, of k] bk_eq have
        l_in: l ∈ gamma_block (Γ_tm M) and
        h_in: h ∈ gamma_block (Γ_tm M) and
        r_in: r ∈ gamma_block (Γ_tm M)
      by auto
    from valid_mttm_delta(4)[OF vM tr, of k] have a'_in: a' k ∈ Γ_tm M .
    have l_upd: l(off := a' k) ∈ gamma_block (Γ_tm M)
      using l_in a'_in unfolding gamma_block_def by auto
    have h_upd: h(off := a' k) ∈ gamma_block (Γ_tm M)
      using h_in a'_in unfolding gamma_block_def by auto
    have r_upd: r(off := a' k) ∈ gamma_block (Γ_tm M)
      using r_in a'_in unfolding gamma_block_def by auto
    have bk'_eq: blocks' k = write_bp (l, h, r) (b, off) (a' k)
      using bk_update[rule_format, of k] bk_eq pk_eq by simp
    show fst (blocks' k) ∈ gamma_block (Γ_tm M)
          ∧ fst (snd (blocks' k)) ∈ gamma_block (Γ_tm M)
          ∧ snd (snd (blocks' k)) ∈ gamma_block (Γ_tm M)
      using bk'_eq l_in h_in r_in l_upd h_upd r_upd
      by (cases b) (auto simp: write_bp_def)
  qed
qed
```

Buffer-gamma preservation under *m_steps_buffered* (the iterated up-to-c-step relation). Induction on the relation power; the step case applies *m_step_buffered_gamma_preserve*.

```
lemma m_steps_buffered_gamma_preserve:
  fixes M :: ('q, 'a) mttm
  assumes vM: valid_mttm M
      and steps: ((q, blocks, pos), (q', blocks', pos')) ∈ m_steps_buffered M
```

```
    and pre: ∀k. fst (blocks k) ∈ gamma_block (Γ_tm M)
                ∧ fst (snd (blocks k)) ∈ gamma_block (Γ_tm M)
                ∧ snd (snd (blocks k)) ∈ gamma_block (Γ_tm M)
  shows ∀k. fst (blocks' k) ∈ gamma_block (Γ_tm M)
          ∧ fst (snd (blocks' k)) ∈ gamma_block (Γ_tm M)
          ∧ snd (snd (blocks' k)) ∈ gamma_block (Γ_tm M)
proof −
  from steps obtain n where
    n_step: ((q, blocks, pos), (q', blocks', pos'))
              ∈ (m_step_buffered M) ^^ n
    unfolding m_steps_buffered_def by blast
  from n_step pre show ?thesis
  proof (induction n arbitrary: q' blocks' pos')
    case 0
    then show ?case by auto
  next
    case (Suc n)
    from Suc.prems(1) obtain q'' blocks'' pos'' where
      step_n: ((q, blocks, pos), (q'', blocks'', pos''))
                ∈ (m_step_buffered M) ^^ n
      and step_one: ((q'', blocks'', pos''), (q', blocks', pos'))
                      ∈ m_step_buffered M
      by (auto elim: relpow_Suc_E)
    from Suc.IH[OF step_n Suc.prems(2)] have IH:
      ∀k. fst (blocks'' k) ∈ gamma_block (Γ_tm M)
        ∧ fst (snd (blocks'' k)) ∈ gamma_block (Γ_tm M)
        ∧ snd (snd (blocks'' k)) ∈ gamma_block (Γ_tm M) .
    from m_step_buffered_gamma_preserve[OF vM step_one IH] show ?case .
  qed
qed
```

Per-substep destination-stage-validity helpers. Given that a tuple is in a particular substep delta, source-stage validity, and any auxiliary gamma-block facts (a's gamma, bl_M/le_M in Γ_M as needed for SS2→SS3 and SS8→SS1 halt branch, *m_steps_buffered_gamma_preserve* for SS4→SS5), the destination stage is valid. Companion to *ae_step_alphabet_enlarge_buffer_gamma_preserve* at the tuple level instead of the configuration level. These are used by the *_exists* construction lemmas to discharge the Q'-filter conjunct that *alphabet_enlarge_delta* carries to enforce $Q' = Q \times \{stg.\ ae_valid_stage\ \Gamma_M\ stg\}$ finiteness on a set-finite-Γ premise (instead of type-class-finite-$'a$).

```
lemma ae_delta_ss1_ss2_dest_valid:
  assumes rel: (s, a, s', a', d) ∈ ae_delta_ss1_ss2 M
    and a_gamma: ∀k. a k ∈ gamma_block (Γ_tm M)
    and src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s)
  shows ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
proof −
  from rel obtain q ofs buf dest buf' where
```

```
    s_eq: s = (q, ofs, buf, dest, SS1)
    and s'_eq: s' = (q, ofs, buf', dest, SS2)
    and buf'_def: buf' = (λk. if k < k_tm M
                          then (fst (buf k), a k, snd (snd (buf k)))
                          else init_buffer (le_tm M) k)
  unfolding ae_delta_ss1_ss2_def by auto
from src_valid s_eq have
    src_g: ∀k. fst (buf k) ∈ gamma_block (Γ_tm M)
             ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
             ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
    and src_off: ∀j≥k_tm M. ofs j = init_offset j
    and src_buf: ∀j≥k_tm M. buf j = init_buffer (le_tm M) j
    and src_dst: ∀j≥k_tm M. dest j = init_dest j
  by (simp_all add: ae_valid_stage_def)
have buf'_g: ∀k. fst (buf' k) ∈ gamma_block (Γ_tm M)
               ∧ fst (snd (buf' k)) ∈ gamma_block (Γ_tm M)
               ∧ snd (snd (buf' k)) ∈ gamma_block (Γ_tm M)
proof (intro allI)
  fix k
  show fst (buf' k) ∈ gamma_block (Γ_tm M)
       ∧ fst (snd (buf' k)) ∈ gamma_block (Γ_tm M)
       ∧ snd (snd (buf' k)) ∈ gamma_block (Γ_tm M)
  proof (cases k < k_tm M)
    case True
    thus ?thesis using buf'_def src_g a_gamma by simp
  next
    case False
    hence le_k: k_tm M ≤ k by simp
    have buf' k = init_buffer (le_tm M) k using buf'_def False by simp
    moreover have buf k = init_buffer (le_tm M) k
      using src_buf[rule_format, OF le_k] .
    ultimately have buf' k = buf k by simp
    thus ?thesis using src_g by simp
  qed
qed
have buf'_tail: ∀j≥k_tm M. buf' j = init_buffer (le_tm M) j
  using buf'_def by simp
show ?thesis
  unfolding s'_eq ae_valid_stage_def
  using buf'_g src_off src_dst buf'_tail by simp
qed

lemma ae_delta_ss2_ss3_dest_valid:
  assumes vM: valid_mttm M
    and rel: (s, a, s', a', d) ∈ ae_delta_ss2_ss3 M
    and a_gamma: ∀k. a k ∈ gamma_block (Γ_tm M)
    and src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s)
  shows ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
proof −
```

```
from rel obtain q ofs buf dest buf' where
    s_eq: s = (q, ofs, buf, dest, SS2)
    and s'_eq: s' = (q, ofs, buf', dest, SS3)
    and buf'_def: buf' = (λk. if k < k_tm M
                              then let (l, h, r) = buf k in
                                   (if h = LE_block (le_tm M)
                                      then bl_block (bl_tm M)
                                      else a k,
                                    h, r)
                              else init_buffer (le_tm M) k)
  unfolding ae_delta_ss2_ss3_def by auto
from src_valid s_eq have
    src: ∀k. fst (buf k) ∈ gamma_block (Γ_tm M)
           ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
           ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
    and src_off: ∀j≥k_tm M. ofs j = init_offset j
    and src_buf: ∀j≥k_tm M. buf j = init_buffer (le_tm M) j
    and src_dst: ∀j≥k_tm M. dest j = init_dest j
  by (simp_all add: ae_valid_stage_def)
have bl_gam: bl_block (bl_tm M) ∈ gamma_block (Γ_tm M)
 using bl_block_in_gamma_block[OF valid_mttm_blank_in_Gamma[OF vM]]
.
have per_k: ∀k. fst (buf' k) ∈ gamma_block (Γ_tm M)
              ∧ fst (snd (buf' k)) ∈ gamma_block (Γ_tm M)
              ∧ snd (snd (buf' k)) ∈ gamma_block (Γ_tm M)
proof
  fix k
  show fst (buf' k) ∈ gamma_block (Γ_tm M)
       ∧ fst (snd (buf' k)) ∈ gamma_block (Γ_tm M)
       ∧ snd (snd (buf' k)) ∈ gamma_block (Γ_tm M)
  proof (cases k < k_tm M)
    case True
    obtain l h r where buf_eq: buf k = (l, h, r) using prod.exhaust by metis
    have lhr: l ∈ gamma_block (Γ_tm M)
            ∧ h ∈ gamma_block (Γ_tm M)
            ∧ r ∈ gamma_block (Γ_tm M)
      using src buf_eq by (metis fst_conv snd_conv)
    have buf'_k: buf' k = (if h = LE_block (le_tm M)
                           then bl_block (bl_tm M)
                           else a k, h, r)
      using buf'_def buf_eq True by simp
    show ?thesis using lhr a_gamma bl_gam buf'_k by auto
  next
    case False
    hence le_k: k_tm M ≤ k by simp
    have buf' k = init_buffer (le_tm M) k using buf'_def False by simp
    moreover have buf k = init_buffer (le_tm M) k
      using src_buf[rule_format, OF le_k] .
    ultimately have buf' k = buf k by simp
```

```
    thus ?thesis using src by simp
   qed
  qed
  have buf'_tail: ∀ j≥k_tm M. buf' j = init_buffer (le_tm M) j
   using buf'_def by simp
  show ?thesis
   unfolding s'_eq ae_valid_stage_def
   using per_k src_off src_dst buf'_tail by simp
qed

lemma ae_delta_ss3_ss4_dest_valid:
  assumes rel: (s, a, s', a', d) ∈ ae_delta_ss3_ss4 M
     and src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s)
  shows ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
proof −
  from rel obtain q ofs buf dest where
     s_eq: s = (q, ofs, buf, dest, SS3)
     and s'_eq: s' = (q, ofs, buf, dest, SS4)
   unfolding ae_delta_ss3_ss4_def by auto
  show ?thesis
   using src_valid unfolding s_eq s'_eq ae_valid_stage_def by simp
qed

lemma ae_delta_ss4_ss5_dest_valid:
  assumes vM: valid_mttm M
     and rel: (s, a, s', a', d) ∈ ae_delta_ss4_ss5 M
     and a_gamma: ∀ k. a k ∈ gamma_block (Γ_tm M)
     and src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s)
  shows ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
proof −
   from rel obtain q ofs buf dest_old q' ofs' buf' dest' buf_full end_pos bufC
where
     s_eq: s = (q, ofs, buf, dest_old, SS4)
     and s'_eq: s' = (q', ofs', buf', dest', SS5)
     and buf_full_def: buf_full = (λk. (fst (buf k),
                        if k < k_tm M then fst (snd (buf k)) else a k,
                        a k))
     and mst: ((q, buf_full, λk. (AE_Home, ofs k)),
              (q', bufC, end_pos)) ∈ m_steps_buffered M
    and ofs'_def: ofs' = (λk. if k < k_tm M then snd (end_pos k) else init_offset
k)
     and buf'_def: buf' = (λk. if k < k_tm M then bufC k else init_buffer (le_tm
M) k)
     and dest'_def: dest' = (λk. if k < k_tm M then fst (end_pos k) else init_dest
k)
   unfolding ae_delta_ss4_ss5_def by auto
  from src_valid s_eq have
     src: ∀ k. fst (buf k) ∈ gamma_block (Γ_tm M)
            ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
```

```
                ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
    and src_buf: ∀ j≥k_tm M. buf j = init_buffer (le_tm M) j
   by (simp_all add: ae_valid_stage_def)
  have full: ∀ k. fst (buf_full k) ∈ gamma_block (Γ_tm M)
                ∧ fst (snd (buf_full k)) ∈ gamma_block (Γ_tm M)
                ∧ snd (snd (buf_full k)) ∈ gamma_block (Γ_tm M)
   unfolding buf_full_def using src a_gamma by (auto split: if_splits)
  have bufC_gam: ∀ k. fst (bufC k) ∈ gamma_block (Γ_tm M)
                    ∧ fst (snd (bufC k)) ∈ gamma_block (Γ_tm M)
                    ∧ snd (snd (bufC k)) ∈ gamma_block (Γ_tm M)
   by (rule m_steps_buffered_gamma_preserve[OF vM mst full])
  have buf'_gam: ∀ k. fst (buf' k) ∈ gamma_block (Γ_tm M)
                    ∧ fst (snd (buf' k)) ∈ gamma_block (Γ_tm M)
                    ∧ snd (snd (buf' k)) ∈ gamma_block (Γ_tm M)
  proof (intro allI)
    fix k
    show fst (buf' k) ∈ gamma_block (Γ_tm M)
          ∧ fst (snd (buf' k)) ∈ gamma_block (Γ_tm M)
          ∧ snd (snd (buf' k)) ∈ gamma_block (Γ_tm M)
    proof (cases k < k_tm M)
      case True
      thus ?thesis using bufC_gam by (simp add: buf'_def)
    next
      case False
      hence le_k: k_tm M ≤ k by simp
      have buf' k = init_buffer (le_tm M) k by (simp add: buf'_def False)
      also have ... = buf k using src_buf[rule_format, OF le_k] by simp
      finally have buf' k = buf k .
      thus ?thesis using src by simp
    qed
  qed
  have ofs'_tail: ∀ j≥k_tm M. ofs' j = init_offset j by (simp add: ofs'_def)
  have buf'_tail: ∀ j≥k_tm M. buf' j = init_buffer (le_tm M) j by (simp add:
buf'_def)
  have dest'_tail: ∀ j≥k_tm M. dest' j = init_dest j by (simp add: dest'_def)
  show ?thesis unfolding s'_eq ae_valid_stage_def
    using buf'_gam ofs'_tail buf'_tail dest'_tail by simp
qed

lemma ae_delta_ss5_ss6_dest_valid:
  assumes rel: (s, a, s', a', d) ∈ ae_delta_ss5_ss6 M
      and src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s)
  shows ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
proof −
  from rel obtain q ofs buf dest where
      s_eq: s = (q, ofs, buf, dest, SS5)
      and s'_eq: s' = (q, ofs, buf, dest, SS6)
    unfolding ae_delta_ss5_ss6_def by auto
  show ?thesis
```

```
  using src_valid unfolding s_eq s'_eq ae_valid_stage_def by simp
qed

lemma ae_delta_ss6_ss7_dest_valid:
  assumes rel: (s, a, s', a', d) ∈ ae_delta_ss6_ss7 M
      and src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s)
  shows ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
proof −
  from rel obtain q ofs buf dest where
      s_eq: s = (q, ofs, buf, dest, SS6)
      and s'_eq: s' = (q, ofs, buf, dest, SS7)
    unfolding ae_delta_ss6_ss7_def by auto
  show ?thesis
    using src_valid unfolding s_eq s'_eq ae_valid_stage_def by simp
qed

lemma ae_delta_ss7_ss8_dest_valid:
  assumes rel: (s, a, s', a', d) ∈ ae_delta_ss7_ss8 M
      and src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s)
  shows ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
proof −
  from rel obtain q ofs buf dest where
      s_eq: s = (q, ofs, buf, dest, SS7)
      and s'_eq: s' = (q, ofs, buf, dest, SS8)
    unfolding ae_delta_ss7_ss8_def by auto
  show ?thesis
    using src_valid unfolding s_eq s'_eq ae_valid_stage_def by simp
qed

lemma ae_delta_ss8_ss1_dest_valid:
  assumes vM: valid_mttm M
      and rel: (s, a, s', a', d) ∈ ae_delta_ss8_ss1 M
      and src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s)
  shows ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
proof −
  from rel obtain q ofs buf dest stage' where
      s_eq: s = (q, ofs, buf, dest, SS8)
      and s'_eq: s' = (q, stage')
      and stage'_def: stage' = (if q ∈ {t_tm M, r_tm M}
                                then init_stage (le_tm M)
                                else (ofs, buf, init_dest, SS1))
    unfolding ae_delta_ss8_ss1_def by auto
  from src_valid s_eq have
      buf_src: ∀ k. fst (buf k) ∈ gamma_block (Γ_tm M)
                  ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
                  ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
      and src_off: ∀ j≥k_tm M. ofs j = init_offset j
      and src_buf: ∀ j≥k_tm M. buf j = init_buffer (le_tm M) j
    by (simp_all add: ae_valid_stage_def)
```

```
  show ?thesis
  proof (cases q ∈ {t_tm M, r_tm M})
    case True
    hence stage' = init_stage (le_tm M) using stage'_def by simp
    moreover have ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (init_stage
(le_tm M))
      using ae_valid_stage_init[OF valid_mttm_LE_in_Gamma[OF vM]] .
    ultimately show ?thesis unfolding s'_eq by simp
  next
    case False
    hence stage' = (ofs, buf, init_dest, SS1) using stage'_def by simp
    thus ?thesis unfolding s'_eq ae_valid_stage_def
      using buf_src src_off src_buf by simp
  qed
qed
```

Validation-phase destination-stage-validity helpers. All 8 validation substeps preserve the buffer component, so the proofs are mechanical: extract the stage destructuring from the relation and propagate *src_valid*.

```
lemma ae_delta_val_fwd_advance_dest_valid:
  assumes rel: (s, a, s', a', d) ∈ ae_delta_val_fwd_advance M
      and src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s)
  shows ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
proof −
  from rel obtain q ofs buf dest where
      s_eq: s = (q, ofs, buf, dest, VFwd)
      and s'_eq: s' = (q, ofs, buf, dest, VFwd)
    unfolding ae_delta_val_fwd_advance_def by auto
  show ?thesis using src_valid unfolding s_eq s'_eq by simp
qed

lemma ae_delta_val_fwd_to_padded_dest_valid:
  assumes rel: (s, a, s', a', d) ∈ ae_delta_val_fwd_to_padded M
      and src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s)
  shows ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
proof −
  from rel obtain q ofs buf dest where
      s_eq: s = (q, ofs, buf, dest, VFwd)
      and s'_eq: s' = (q, ofs, buf, dest, VFwdPad)
    unfolding ae_delta_val_fwd_to_padded_def by auto
  show ?thesis
    using src_valid unfolding s_eq s'_eq ae_valid_stage_def by simp
qed

lemma ae_delta_val_fwd_reject_dest_valid:
  assumes vM: valid_mttm M
      and rel: (s, a, s', a', d) ∈ ae_delta_val_fwd_reject M
      and src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s)
  shows ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
```

proof −
  from *rel* **have** *s′_eq*: *s′* = (*r_tm M*, *init_stage* (*le_tm M*))
    unfolding *ae_delta_val_fwd_reject_def* **by** *auto*
  have *ae_valid_stage* (Γ*_tm M*) (*le_tm M*) (*k_tm M*) (*init_stage* (*le_tm M*))
    using *ae_valid_stage_init*[*OF valid_mttm_LE_in_Gamma*[*OF vM*]] **.**
  thus *?thesis* **unfolding** *s′_eq* **by** *simp*
qed

lemma *ae_delta_val_fwd_to_ret_dest_valid*:
  assumes *rel*: (*s*, *a*, *s′*, *a′*, *d*) ∈ *ae_delta_val_fwd_to_ret M*
      and *src_valid*: *ae_valid_stage* (Γ*_tm M*) (*le_tm M*) (*k_tm M*) (*snd s*)
  shows *ae_valid_stage* (Γ*_tm M*) (*le_tm M*) (*k_tm M*) (*snd s′*)
proof −
  from *rel* **obtain** *q ofs buf dest* **where**
      s_eq: *s* = (*q*, *ofs*, *buf*, *dest*, *VFwd*)
      and *s′_eq*: *s′* = (*q*, *ofs*, *buf*, *dest*, *VRet*)
    unfolding *ae_delta_val_fwd_to_ret_def* **by** *auto*
  show *?thesis*
    using *src_valid* **unfolding** *s_eq s′_eq ae_valid_stage_def* **by** *simp*
qed

lemma *ae_delta_val_pad_to_ret_dest_valid*:
  assumes *rel*: (*s*, *a*, *s′*, *a′*, *d*) ∈ *ae_delta_val_pad_to_ret M*
      and *src_valid*: *ae_valid_stage* (Γ*_tm M*) (*le_tm M*) (*k_tm M*) (*snd s*)
  shows *ae_valid_stage* (Γ*_tm M*) (*le_tm M*) (*k_tm M*) (*snd s′*)
proof −
  from *rel* **obtain** *q ofs buf dest* **where**
      s_eq: *s* = (*q*, *ofs*, *buf*, *dest*, *VFwdPad*)
      and *s′_eq*: *s′* = (*q*, *ofs*, *buf*, *dest*, *VRet*)
    unfolding *ae_delta_val_pad_to_ret_def* **by** *auto*
  show *?thesis*
    using *src_valid* **unfolding** *s_eq s′_eq ae_valid_stage_def* **by** *simp*
qed

lemma *ae_delta_val_pad_reject_dest_valid*:
  assumes *vM*: *valid_mttm M*
      and *rel*: (*s*, *a*, *s′*, *a′*, *d*) ∈ *ae_delta_val_pad_reject M*
      and *src_valid*: *ae_valid_stage* (Γ*_tm M*) (*le_tm M*) (*k_tm M*) (*snd s*)
  shows *ae_valid_stage* (Γ*_tm M*) (*le_tm M*) (*k_tm M*) (*snd s′*)
proof −
  from *rel* **have** *s′_eq*: *s′* = (*r_tm M*, *init_stage* (*le_tm M*))
    unfolding *ae_delta_val_pad_reject_def* **by** *auto*
  have *ae_valid_stage* (Γ*_tm M*) (*le_tm M*) (*k_tm M*) (*init_stage* (*le_tm M*))
    using *ae_valid_stage_init*[*OF valid_mttm_LE_in_Gamma*[*OF vM*]] **.**
  thus *?thesis* **unfolding** *s′_eq* **by** *simp*
qed

lemma *ae_delta_val_ret_step_dest_valid*:
  assumes *rel*: (*s*, *a*, *s′*, *a′*, *d*) ∈ *ae_delta_val_ret_step M*

```
    and src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s)
  shows ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
proof −
  from rel obtain q ofs buf dest where
      s_eq: s = (q, ofs, buf, dest, VRet)
    and s'_eq: s' = (q, ofs, buf, dest, VRet)
    unfolding ae_delta_val_ret_step_def by auto
  show ?thesis using src_valid unfolding s_eq s'_eq by simp
qed

lemma ae_delta_val_ret_to_sim_dest_valid:
  assumes rel: (s, a, s', a', d) ∈ ae_delta_val_ret_to_sim M
    and src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s)
  shows ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
proof −
  from rel obtain q ofs buf dest where
      s_eq: s = (q, ofs, buf, dest, VRet)
    and s'_eq: s' = (q, ofs, buf, dest, SS1)
    unfolding ae_delta_val_ret_to_sim_def by auto
  show ?thesis
    using src_valid unfolding s_eq s'_eq ae_valid_stage_def by simp
qed
```

δ' for the alphabet-enlargement combinator: the union of the 16 per-substep relations, intersected with two restrictions.

First restriction: read / write blocks lie in $\Gamma' = gamma_block\ (\Gamma_tm\ M)$. This makes δ' satisfy the substrate's δ_set-shape obligation.

Second restriction (added 2026-05-09 alongside the substrate's δLE-no-write strengthening): a transition writes $LE_block\ (le_tm\ M)$ on tape k only when reading the same. The substep relations for SS5→SS8 produce write-back tuples whose $a'\ k$ can in unreachable buffer states (left or right block holding $LE_block\ (le_tm\ M)$) equal $LE_block\ (le_tm\ M)$ without the read $a\ k$ matching; the intersection drops those tuples. Since reachable buffer states do not put LE_block in left or right blocks (the LE block stays at M-position 0, below the simulation phase's $p_start \geq 1$ window), this is semantically inert in reachable executions and exists only to syntactically satisfy the substrate's universal δLE-no-write obligation.

```
definition alphabet_enlarge_delta ::
  ('q, 'a) mttm
    ⇒ (('q × ('a, 'c :: enum) ae_stage)
        × (nat ⇒ ('c ⇒ 'a))
        × ('q × ('a, 'c) ae_stage)
        × (nat ⇒ ('c ⇒ 'a))
        × (nat ⇒ dir)) set where
  alphabet_enlarge_delta M =
    (ae_delta_val_fwd_advance M
      ∪ ae_delta_val_fwd_to_padded M
      ∪ ae_delta_val_fwd_reject M
```

$\cup$ *ae_delta_val_fwd_to_ret M*
$\cup$ *ae_delta_val_pad_to_ret M*
$\cup$ *ae_delta_val_pad_reject M*
$\cup$ *ae_delta_val_ret_step M*
$\cup$ *ae_delta_val_ret_to_sim M*
$\cup$ *ae_delta_ss1_ss2 M*
$\cup$ *ae_delta_ss2_ss3 M*
$\cup$ *ae_delta_ss3_ss4 M*
$\cup$ *ae_delta_ss4_ss5 M*
$\cup$ *ae_delta_ss5_ss6 M*
$\cup$ *ae_delta_ss6_ss7 M*
$\cup$ *ae_delta_ss7_ss8 M*
$\cup$ *ae_delta_ss8_ss1 M*)
$\cap$ {(*s*, *a*, *s′*, *a′*, *d*).
(∀ *k*. *a k* ∈ *gamma_block* (Γ_*tm M*))
∧ (∀ *k*. *a′ k* ∈ *gamma_block* (Γ_*tm M*))}
$\cap$ {(*s*, *a*, *s′*, *a′*, *d*).
∀ *k*. *a′ k* = *LE_block* (*le_tm M*) ⟶ *a k* = *LE_block* (*le_tm M*)}
$\cap$ {(*s*, *a*, *s′*, *a′*, *d*).
ae_valid_stage (Γ_*tm M*) (*le_tm M*) (*k_tm M*) (*snd s*)
∧ *ae_valid_stage* (Γ_*tm M*) (*le_tm M*) (*k_tm M*) (*snd s′*)}

2.10 Per-substep functionality (source determines target, except SS4→SS5)

For each of the 15 non-compute substep relations, the source configuration uniquely determines the target. These functionality lemmas underpin the reverse-arm trace decoder: given an *M′*-step from a config with a known substep index, the target components are mechanically read off.

lemma *ae_delta_val_fwd_advance_functional*:
assumes (*s*, *a*, *s1*, *a1*, *d1*) ∈ *ae_delta_val_fwd_advance M*
and (*s*, *a*, *s2*, *a2*, *d2*) ∈ *ae_delta_val_fwd_advance M*
shows *s1* = *s2* ∧ *a1* = *a2* ∧ *d1* = *d2*
using *assms* **unfolding** *ae_delta_val_fwd_advance_def* **by** *auto*

lemma *ae_delta_val_fwd_to_padded_functional*:
assumes (*s*, *a*, *s1*, *a1*, *d1*) ∈ *ae_delta_val_fwd_to_padded M*
and (*s*, *a*, *s2*, *a2*, *d2*) ∈ *ae_delta_val_fwd_to_padded M*
shows *s1* = *s2* ∧ *a1* = *a2* ∧ *d1* = *d2*
using *assms* **unfolding** *ae_delta_val_fwd_to_padded_def* **by** *auto*

lemma *ae_delta_val_fwd_reject_functional*:
assumes (*s*, *a*, *s1*, *a1*, *d1*) ∈ *ae_delta_val_fwd_reject M*
and (*s*, *a*, *s2*, *a2*, *d2*) ∈ *ae_delta_val_fwd_reject M*
shows *s1* = *s2* ∧ *a1* = *a2* ∧ *d1* = *d2*
using *assms* **unfolding** *ae_delta_val_fwd_reject_def* **by** *auto*

lemma *ae_delta_val_fwd_to_ret_functional*:

assumes $(s,\ a,\ s1,\ a1,\ d1) \in ae_delta_val_fwd_to_ret\ M$
and $(s,\ a,\ s2,\ a2,\ d2) \in ae_delta_val_fwd_to_ret\ M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
using *assms* **unfolding** *ae_delta_val_fwd_to_ret_def* **by** *auto*

lemma *ae_delta_val_pad_to_ret_functional*:
assumes $(s,\ a,\ s1,\ a1,\ d1) \in ae_delta_val_pad_to_ret\ M$
and $(s,\ a,\ s2,\ a2,\ d2) \in ae_delta_val_pad_to_ret\ M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
using *assms* **unfolding** *ae_delta_val_pad_to_ret_def* **by** *auto*

lemma *ae_delta_val_pad_reject_functional*:
assumes $(s,\ a,\ s1,\ a1,\ d1) \in ae_delta_val_pad_reject\ M$
and $(s,\ a,\ s2,\ a2,\ d2) \in ae_delta_val_pad_reject\ M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
using *assms* **unfolding** *ae_delta_val_pad_reject_def* **by** *auto*

lemma *ae_delta_val_ret_step_functional*:
assumes $(s,\ a,\ s1,\ a1,\ d1) \in ae_delta_val_ret_step\ M$
and $(s,\ a,\ s2,\ a2,\ d2) \in ae_delta_val_ret_step\ M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
using *assms* **unfolding** *ae_delta_val_ret_step_def* **by** *auto*

lemma *ae_delta_val_ret_to_sim_functional*:
assumes $(s,\ a,\ s1,\ a1,\ d1) \in ae_delta_val_ret_to_sim\ M$
and $(s,\ a,\ s2,\ a2,\ d2) \in ae_delta_val_ret_to_sim\ M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
using *assms* **unfolding** *ae_delta_val_ret_to_sim_def* **by** *auto*

lemma *ae_delta_ss1_ss2_functional*:
assumes $(s,\ a,\ s1,\ a1,\ d1) \in ae_delta_ss1_ss2\ M$
and $(s,\ a,\ s2,\ a2,\ d2) \in ae_delta_ss1_ss2\ M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
using *assms* **unfolding** *ae_delta_ss1_ss2_def* **by** *auto*

lemma *ae_delta_ss2_ss3_functional*:
assumes $(s,\ a,\ s1,\ a1,\ d1) \in ae_delta_ss2_ss3\ M$
and $(s,\ a,\ s2,\ a2,\ d2) \in ae_delta_ss2_ss3\ M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
using *assms* **unfolding** *ae_delta_ss2_ss3_def* **by** *auto*

lemma *ae_delta_ss3_ss4_functional*:
assumes $(s,\ a,\ s1,\ a1,\ d1) \in ae_delta_ss3_ss4\ M$
and $(s,\ a,\ s2,\ a2,\ d2) \in ae_delta_ss3_ss4\ M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
using *assms* **unfolding** *ae_delta_ss3_ss4_def* **by** *auto*

SS4→SS5 (compute substep) functionality under *det_mttm M*: the source determines the target uniquely. Unlike the other 15 substep functionality lemmas which follow from pure definitional unfolding, this one con-

sumes M's determinism via *m_steps_buffered_functional* because SS4→SS5 is precisely where M's δ enters AE's δ'. Completes the 16th functionality entry; together with the 15 simpler entries, this discharges the per-substep functionality obligations of the reverse-arm chain-uniqueness argument.

lemma *ae_delta_ss4_ss5_functional*:
fixes *M* :: ('*q*, '*a*) *mttm*
assumes *vM*: *valid_mttm M*
and *det*: *det_mttm M*
and *h1*: (*s*, *a*, *s1*, *a1*, *d1*) ∈ *ae_delta_ss4_ss5 M*
and *h2*: (*s*, *a*, *s2*, *a2*, *d2*) ∈ *ae_delta_ss4_ss5 M*
shows *s1* = *s2* ∧ *a1* = *a2* ∧ *d1* = *d2*
proof −
from *h1* **obtain** *q ofs buf dest_old q1' ofs1' buf1' dest1' buf_full1 end_pos1 bufC1*
where *sd1*: *s* = (*q*, *ofs*, *buf*, *dest_old*, *SS4*)
and *rs1*: *s1* = (*q1'*, *ofs1'*, *buf1'*, *dest1'*, *SS5*)
and *ad1*: *a1* = *a*
and *dd1*: *d1* = (λ*k*. *if k* < *k_tm M*
then (*if a k* = *LE_block* (*le_tm M*) *then dir.N else dir.L*)
else dir.N)
and *bf1*: *buf_full1* = (λ*k*. (*fst* (*buf k*),
if k < *k_tm M then fst* (*snd* (*buf k*)) *else a k*,
a k))
and *ms1*: ((*q*, *buf_full1*, λ*k*. (*AE_Home*, *ofs k*)),
(*q1'*, *bufC1*, *end_pos1*)) ∈ *m_steps_buffered M*
and *of1*: *ofs1'* = (λ*k*. *if k* < *k_tm M then snd* (*end_pos1 k*) *else init_offset k*)
and *bd1*: *buf1'* = (λ*k*. *if k* < *k_tm M then bufC1 k else init_buffer* (*le_tm M*) *k*)
and *de1*: *dest1'* = (λ*k*. *if k* < *k_tm M then fst* (*end_pos1 k*) *else init_dest k*)
unfolding *ae_delta_ss4_ss5_def* **by** *auto*
from *h2* **obtain** *q2 ofs2v buf2v dest_old2 q2' ofs2' buf2' dest2' buf_full2 end_pos2 bufC2*
where *sd2*: *s* = (*q2*, *ofs2v*, *buf2v*, *dest_old2*, *SS4*)
and *rs2*: *s2* = (*q2'*, *ofs2'*, *buf2'*, *dest2'*, *SS5*)
and *ad2*: *a2* = *a*
and *dd2*: *d2* = (λ*k*. *if k* < *k_tm M*
then (*if a k* = *LE_block* (*le_tm M*) *then dir.N else dir.L*)
else dir.N)
and *bf2*: *buf_full2* = (λ*k*. (*fst* (*buf2v k*),
if k < *k_tm M then fst* (*snd* (*buf2v k*)) *else a k*,
a k))
and *ms2*: ((*q2*, *buf_full2*, λ*k*. (*AE_Home*, *ofs2v k*)),
(*q2'*, *bufC2*, *end_pos2*)) ∈ *m_steps_buffered M*
and *of2*: *ofs2'* = (λ*k*. *if k* < *k_tm M then snd* (*end_pos2 k*) *else init_offset k*)
and *bd2*: *buf2'* = (λ*k*. *if k* < *k_tm M then bufC2 k else init_buffer* (*le_tm M*) *k*)

```
    and de2: dest2' = (λk. if k < k_tm M then fst (end_pos2 k) else init_dest
k)
    unfolding ae_delta_ss4_ss5_def by auto

  — Source matches force the buffer / offset components to agree, and hence buf_full
agrees.
  from sd1 sd2 have q2_eq: q2 = q and ofs2_eq: ofs2v = ofs
    and buf2_eq: buf2v = buf by simp_all
  have bf_eq: buf_full2 = buf_full1
    unfolding bf1 bf2 buf2_eq by (rule refl)

  — Apply m_steps_buffered_functional: with the compute-substep input identi-
fied across the two witnesses, determinism yields a unique (q', buf', end_pos).
  have ms2': ((q, buf_full1, λk. (AE_Home, ofs k)),
              (q2', bufC2, end_pos2)) ∈ m_steps_buffered M
    using ms2 q2_eq ofs2_eq bf_eq by simp
  from m_steps_buffered_functional[OF vM det ms1 ms2']
  have target_eq: (q1', bufC1, end_pos1) = (q2', bufC2, end_pos2) .
  hence q'_eq: q1' = q2' and bufC_eq: bufC1 = bufC2
    and ep_eq: end_pos1 = end_pos2 by auto

  show ?thesis
  proof (intro conjI)
    show s1 = s2
      using rs1 rs2 of1 of2 bd1 bd2 de1 de2 q'_eq bufC_eq ep_eq
      by (simp add: fun_eq_iff)
    show a1 = a2 using ad1 ad2 by simp
    show d1 = d2 using dd1 dd2 by simp
  qed
qed

lemma ae_delta_ss5_ss6_functional:
  assumes (s, a, s1, a1, d1) ∈ ae_delta_ss5_ss6 M
      and (s, a, s2, a2, d2) ∈ ae_delta_ss5_ss6 M
  shows s1 = s2 ∧ a1 = a2 ∧ d1 = d2
  using assms unfolding ae_delta_ss5_ss6_def by auto

lemma ae_delta_ss6_ss7_functional:
  assumes (s, a, s1, a1, d1) ∈ ae_delta_ss6_ss7 M
      and (s, a, s2, a2, d2) ∈ ae_delta_ss6_ss7 M
  shows s1 = s2 ∧ a1 = a2 ∧ d1 = d2
  using assms unfolding ae_delta_ss6_ss7_def by auto

lemma ae_delta_ss7_ss8_functional:
  assumes (s, a, s1, a1, d1) ∈ ae_delta_ss7_ss8 M
      and (s, a, s2, a2, d2) ∈ ae_delta_ss7_ss8 M
  shows s1 = s2 ∧ a1 = a2 ∧ d1 = d2
  using assms unfolding ae_delta_ss7_ss8_def by auto
```

lemma *ae_delta_ss8_ss1_functional*:
assumes $(s, a, s1, a1, d1) \in ae_delta_ss8_ss1\ M$
and $(s, a, s2, a2, d2) \in ae_delta_ss8_ss1\ M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
using *assms* **unfolding** *ae_delta_ss8_ss1_def* **by** (*auto split*: *if_split_asm*)

Within-phase exclusion: for each source substep index with multiple relations (VFwd: 4, VFwdPad: 2, VRet: 2), the relations do not overlap. These lemmas commit the reverse-arm trace decoder to a specific branch.

lemma *ae_delta_val_pad_to_ret_pad_reject_disjoint*:
assumes $(s, a, s1, a1, d1) \in ae_delta_val_pad_to_ret\ M$
and $(s, a, s2, a2, d2) \in ae_delta_val_pad_reject\ M$
shows *False*
using *assms* **unfolding** *ae_delta_val_pad_to_ret_def ae_delta_val_pad_reject_def*
by *auto*

lemma *ae_delta_val_ret_step_ret_to_sim_disjoint*:
assumes $(s, a, s1, a1, d1) \in ae_delta_val_ret_step\ M$
and $(s, a, s2, a2, d2) \in ae_delta_val_ret_to_sim\ M$
shows *False*
using *assms* **unfolding** *ae_delta_val_ret_step_def ae_delta_val_ret_to_sim_def*
by *auto*

Cross-phase exclusion: a tuple in *alphabet_enlarge_delta M* whose source state has substep index *SSn* lies in the matching *ae_delta_ssn_ssm M* (and not in any other relation of the union). Each of the 16 union components pins its source substep index to a specific value; the 15 other relations have source substep indices in $\{VFwd, VFwdPad, VRet, SS1, \ldots, SS8\} \setminus \{SSn\}$, hence cannot match. The alphabet intersection clauses are preserved through the conclusion (membership in *ae_delta_ssn_ssm M* does not require them, so we discard them). Used by the reverse-arm trace decoder to commit each peeled δ'-step to its canonical substep.

lemma *ae_delta_ss1_only*:
fixes $M :: ('q, 'a)\ mttm$
assumes $(s, a, s', a', d) \in alphabet_enlarge_delta\ M$
and $snd\ (snd\ (snd\ (snd\ s))) = SS1$
shows $(s, a, s', a', d) \in ae_delta_ss1_ss2\ M$
using *assms*
unfolding *alphabet_enlarge_delta_def*
ae_delta_val_fwd_advance_def
ae_delta_val_fwd_to_padded_def
ae_delta_val_fwd_reject_def
ae_delta_val_fwd_to_ret_def
ae_delta_val_pad_to_ret_def
ae_delta_val_pad_reject_def
ae_delta_val_ret_step_def

ae_delta_val_ret_to_sim_def
ae_delta_ss2_ss3_def
ae_delta_ss3_ss4_def
ae_delta_ss4_ss5_def
ae_delta_ss5_ss6_def
ae_delta_ss6_ss7_def
ae_delta_ss7_ss8_def
ae_delta_ss8_ss1_def
by *auto*

lemma *ae_delta_ss2_only*:
fixes $M :: ('q, 'a)$ *mttm*
assumes $(s, a, s', a', d) \in$ *alphabet_enlarge_delta M*
and *snd* (*snd* (*snd* (*snd s*))) = *SS2*
shows $(s, a, s', a', d) \in$ *ae_delta_ss2_ss3 M*
using *assms*
unfolding *alphabet_enlarge_delta_def*
ae_delta_val_fwd_advance_def
ae_delta_val_fwd_to_padded_def
ae_delta_val_fwd_reject_def
ae_delta_val_fwd_to_ret_def
ae_delta_val_pad_to_ret_def
ae_delta_val_pad_reject_def
ae_delta_val_ret_step_def
ae_delta_val_ret_to_sim_def
ae_delta_ss1_ss2_def
ae_delta_ss3_ss4_def
ae_delta_ss4_ss5_def
ae_delta_ss5_ss6_def
ae_delta_ss6_ss7_def
ae_delta_ss7_ss8_def
ae_delta_ss8_ss1_def
by *auto*

lemma *ae_delta_ss3_only*:
fixes $M :: ('q, 'a)$ *mttm*
assumes $(s, a, s', a', d) \in$ *alphabet_enlarge_delta M*
and *snd* (*snd* (*snd* (*snd s*))) = *SS3*
shows $(s, a, s', a', d) \in$ *ae_delta_ss3_ss4 M*
using *assms*
unfolding *alphabet_enlarge_delta_def*
ae_delta_val_fwd_advance_def
ae_delta_val_fwd_to_padded_def
ae_delta_val_fwd_reject_def
ae_delta_val_fwd_to_ret_def
ae_delta_val_pad_to_ret_def
ae_delta_val_pad_reject_def
ae_delta_val_ret_step_def
ae_delta_val_ret_to_sim_def

ae_delta_ss1_ss2_def
ae_delta_ss2_ss3_def
ae_delta_ss4_ss5_def
ae_delta_ss5_ss6_def
ae_delta_ss6_ss7_def
ae_delta_ss7_ss8_def
ae_delta_ss8_ss1_def
by *auto*

lemma *ae_delta_ss4_only*:
fixes $M :: ('q, 'a)$ *mttm*
assumes $(s, a, s', a', d) \in$ *alphabet_enlarge_delta M*
and *snd* (*snd* (*snd* (*snd s*))) = *SS4*
shows $(s, a, s', a', d) \in$ *ae_delta_ss4_ss5 M*
using *assms*
unfolding *alphabet_enlarge_delta_def*
ae_delta_val_fwd_advance_def
ae_delta_val_fwd_to_padded_def
ae_delta_val_fwd_reject_def
ae_delta_val_fwd_to_ret_def
ae_delta_val_pad_to_ret_def
ae_delta_val_pad_reject_def
ae_delta_val_ret_step_def
ae_delta_val_ret_to_sim_def
ae_delta_ss1_ss2_def
ae_delta_ss2_ss3_def
ae_delta_ss3_ss4_def
ae_delta_ss5_ss6_def
ae_delta_ss6_ss7_def
ae_delta_ss7_ss8_def
ae_delta_ss8_ss1_def
by *auto*

lemma *ae_delta_ss5_only*:
fixes $M :: ('q, 'a)$ *mttm*
assumes $(s, a, s', a', d) \in$ *alphabet_enlarge_delta M*
and *snd* (*snd* (*snd* (*snd s*))) = *SS5*
shows $(s, a, s', a', d) \in$ *ae_delta_ss5_ss6 M*
using *assms*
unfolding *alphabet_enlarge_delta_def*
ae_delta_val_fwd_advance_def
ae_delta_val_fwd_to_padded_def
ae_delta_val_fwd_reject_def
ae_delta_val_fwd_to_ret_def
ae_delta_val_pad_to_ret_def
ae_delta_val_pad_reject_def
ae_delta_val_ret_step_def
ae_delta_val_ret_to_sim_def
ae_delta_ss1_ss2_def

```
        ae_delta_ss2_ss3_def
        ae_delta_ss3_ss4_def
        ae_delta_ss4_ss5_def
        ae_delta_ss6_ss7_def
        ae_delta_ss7_ss8_def
        ae_delta_ss8_ss1_def
  by auto

lemma ae_delta_ss6_only:
  fixes M :: ('q, 'a) mttm
  assumes (s, a, s', a', d) ∈ alphabet_enlarge_delta M
      and snd (snd (snd (snd s))) = SS6
    shows (s, a, s', a', d) ∈ ae_delta_ss6_ss7 M
  using assms
  unfolding alphabet_enlarge_delta_def
        ae_delta_val_fwd_advance_def
        ae_delta_val_fwd_to_padded_def
        ae_delta_val_fwd_reject_def
        ae_delta_val_fwd_to_ret_def
        ae_delta_val_pad_to_ret_def
        ae_delta_val_pad_reject_def
        ae_delta_val_ret_step_def
        ae_delta_val_ret_to_sim_def
        ae_delta_ss1_ss2_def
        ae_delta_ss2_ss3_def
        ae_delta_ss3_ss4_def
        ae_delta_ss4_ss5_def
        ae_delta_ss5_ss6_def
        ae_delta_ss7_ss8_def
        ae_delta_ss8_ss1_def
  by auto

lemma ae_delta_ss7_only:
  fixes M :: ('q, 'a) mttm
  assumes (s, a, s', a', d) ∈ alphabet_enlarge_delta M
      and snd (snd (snd (snd s))) = SS7
    shows (s, a, s', a', d) ∈ ae_delta_ss7_ss8 M
  using assms
  unfolding alphabet_enlarge_delta_def
        ae_delta_val_fwd_advance_def
        ae_delta_val_fwd_to_padded_def
        ae_delta_val_fwd_reject_def
        ae_delta_val_fwd_to_ret_def
        ae_delta_val_pad_to_ret_def
        ae_delta_val_pad_reject_def
        ae_delta_val_ret_step_def
        ae_delta_val_ret_to_sim_def
        ae_delta_ss1_ss2_def
        ae_delta_ss2_ss3_def
```

```
    ae_delta_ss3_ss4_def
    ae_delta_ss4_ss5_def
    ae_delta_ss5_ss6_def
    ae_delta_ss6_ss7_def
    ae_delta_ss8_ss1_def
  by auto

lemma ae_delta_ss8_only:
  fixes M :: ('q, 'a) mttm
  assumes (s, a, s', a', d) ∈ alphabet_enlarge_delta M
      and snd (snd (snd (snd s))) = SS8
    shows (s, a, s', a', d) ∈ ae_delta_ss8_ss1 M
  using assms
  unfolding alphabet_enlarge_delta_def
    ae_delta_val_fwd_advance_def
    ae_delta_val_fwd_to_padded_def
    ae_delta_val_fwd_reject_def
    ae_delta_val_fwd_to_ret_def
    ae_delta_val_pad_to_ret_def
    ae_delta_val_pad_reject_def
    ae_delta_val_ret_step_def
    ae_delta_val_ret_to_sim_def
    ae_delta_ss1_ss2_def
    ae_delta_ss2_ss3_def
    ae_delta_ss3_ss4_def
    ae_delta_ss4_ss5_def
    ae_delta_ss5_ss6_def
    ae_delta_ss6_ss7_def
    ae_delta_ss7_ss8_def
  by auto

end
theory AlphabetEnlargement_Simulation
  imports AlphabetEnlargement_Delta
begin
```

2.11 The combinator

The alphabet-enlargement combinator following Hopcroft–Ullman [6, Theorem 12.3]. Takes a substrate machine over alphabet $'a$; produces a substrate machine over alphabet $'c \Rightarrow 'a$ (blocks) with state set $'q \times ('a, 'c)$ *ae_stage*. Tape count *k_tm M* is preserved.

The output machine simulates M on the externally encoded input via a stage-based simulation: each stage consumes 8 super-steps of M' to simulate exactly c consecutive steps of M (pre-fetch home + neighbour blocks in 4 substeps; compute next c M-steps internally; write back up to 3 modified blocks and reposition heads in 4 substeps).

Pieces:

- Q': $Q_M \times UNIV$ over *ae_stage*.
- Σ': blocks composed of $\Sigma_M \cup \{bl_M\}$ cells (the encoder's image), excluding the all-blank and all-LE blocks.
- Γ': blocks composed of Γ_M cells.
- blank: *bl_block bl_M* (the constant blank-block).
- LE: *LE_block le_M* (the constant LE-block).
- δ': *alphabet_enlarge_delta M* (union of 8 per-substep relations; bodies partially filled in this pass).
- start: (s_M, *init_stage le_M*).
- accept / reject: (t_M, *init_stage le_M*) / (r_M, *init_stage le_M*); distinguished by the M-state component, which inherits $t_M \neq r_M$ from M's wf.

```
definition alphabet_enlarge ::
  ('q, 'a) mttm
    ⇒ ('q × ('a, ('c :: enum)) ae_stage, 'c ⇒ 'a) mttm
  where
    alphabet_enlarge M =
     (case M of MTTM Q_M Sigma_M Gamma_M bl_M le_M _ s_M t_M r_M
k_M ⇒
        MTTM (Q_M × {stg. ae_valid_stage Gamma_M le_M k_M stg})
            (gamma_block (Sigma_M ∪ {bl_M})
               − {bl_block bl_M, LE_block le_M})
            (gamma_block Gamma_M)
            (bl_block bl_M)
            (LE_block le_M)
            (alphabet_enlarge_delta M)
            (s_M, init_stage le_M)
            (t_M, init_stage le_M)
            (r_M, init_stage le_M)
            k_M)
```

2.12 Simulation infrastructure

The two top-level theorems below *alphabet_enlarge_language* and *alphabet_enlarge_time* both route through a shared simulation argument relating M's configurations to $M' =$ *alphabet_enlarge* M's configurations at SS1 stage boundaries (or at halt configurations reached via SS8→SS1's halt-routing). The relation *ae_simulates*, the per-substep mid-stage invariants *ae_inv_ss1*, ..., *ae_inv_ss8*, and the lemma roster below carry the structure of that argument.

2.12.1 Position-decoding, tape correspondence, initial config

The position-decoding map: given M''s block position s and a within-block offset $i :: 'c$, return the corresponding M-tape position. For $s = 0$ the result is 0 (the LE position is shared between M and M'). For $s \geq 1$, the block at s covers M's positions $(s - 1) \cdot c + 1$ through $(s - 1) \cdot c + c$ in the order determined by the canonical $'c$-enumeration via *c_idx*.

definition *ae_decode_pos* :: *nat* ⇒ ('*c* :: *enum*) ⇒ *nat* **where**
ae_decode_pos s i =
(*if s* = *0 then 0*
else (*s* − *1*) ∗ *card* (*UNIV* :: '*c set*) + *c_idx i* + *1*)

Tape-content correspondence under the encoding: for every M-tape position p, the value at p equals the value of the corresponding M'-block at the corresponding offset. Position 0 of M's tape is fixed at *le* (the substrate's LE invariant); positions $p \geq 1$ map bijectively to block-and-offset pairs (s, i) with $s \geq 1$ via $p = (s - 1) \cdot c + c_idx\ i + 1$.

definition *ae_tape_correspondence* ::
'*a* ⇒ (*nat* ⇒ '*a*) ⇒ (*nat* ⇒ ('*c* :: *enum* ⇒ '*a*)) ⇒ *bool* **where**
ae_tape_correspondence le tM tM' ⟷
tM 0 = *le* ∧
(∀ *s i*. *s* ≥ *1*
⟶ *tM* ((*s* − *1*) ∗ *card* (*UNIV* :: '*c set*) + *c_idx i* + *1*)
= *tM' s i*)

Initial configuration of M' = *alphabet_enlarge M* on a block input w :: ('*c* ⇒ '*a*) *list*. Mirrors the substrate's top-level *init_config_mttm* shape, restated here with the explicit block tape layout the downstream simulation lemmas need. Position 0 holds the LE-block; positions *1*...*length w* on tape 0 hold w's blocks; all other positions hold the blank-block; all heads at position 0.

definition *ae_init_config* ::
('*q*, '*a*) *mttm*
⇒ (('*c* :: *enum*) ⇒ '*a*) *list*
⇒ ('*c* ⇒ '*a*, '*q* × ('*a*, '*c*) *ae_stage*) *mt_config* **where**
ae_init_config M w =
Config$_M$ (*s_tm M*, *init_stage* (*le_tm M*))
(λ*i n*. *if i* < *k_tm M*
then if n = *0 then LE_block* (*le_tm M*)
else if i = *0* ∧ *n* ≤ *length w*
then w ! (*n* − *1*)
else bl_block (*bl_tm M*)
else bl_block (*bl_tm M*))
(λ_. *0*)

Bridge: the substrate's *init_config_mttm* of *alphabet_enlarge M* on a block input w coincides with the AE-specific *ae_init_config M w*. The substrate's *init_config_mttm* uses *alphabet_enlarge M*'s field projections,

which inherit from M via the construction (start state becomes (*s_tm M*, *init_stage* (*le_tm M*)); blank becomes *bl_block* (*bl_tm M*); LE becomes *LE_block* (*le_tm M*)). Used by *alphabet_enlarge_language* and *alphabet_enlarge_time* to translate the substrate-level *accepts_in_time_mttm* conclusion into the AE-side *ae_init_config* form on which the validation / simulation chains operate.

```
lemma init_config_alphabet_enlarge:
  fixes M :: ('q, 'a) mttm
    and w :: ('c :: enum ⇒ 'a) list
  shows init_config_mttm
          (alphabet_enlarge M
             :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm) w
        = ae_init_config M w
proof −
  obtain Q Σ Γ bl le δ s t r k where
    M_eq: M = MTTM Q Σ Γ bl le δ s t r k
    by (cases M)
  have s_eq:  s_tm M = s   by (simp only: M_eq s_tm.simps)
  have le_eq: le_tm M = le by (simp only: M_eq le_tm.simps)
  have bl_eq: bl_tm M = bl by (simp only: M_eq bl_tm.simps)
  show ?thesis
    unfolding ae_init_config_def
    unfolding s_eq le_eq bl_eq
    unfolding M_eq alphabet_enlarge_def
    by simp
qed
```

2.12.2 Substrate projection bridges

Projection bridges for *alphabet_enlarge M*: the field accessors *s_tm*, *t_tm*, *r_tm*, *bl_tm*, *le_tm*, *delta_tm* are positional pattern-matches on the *MTTM* constructor. Used by *alphabet_enlarge_language* and *alphabet_enlarge_time* to translate substrate-level field references on *alphabet_enlarge M* into the AE-side expressions (the construction's start / accept / reject states inherit from M with an attached *init_stage le*; blank and LE become block-encoded; delta is *alphabet_enlarge_delta M*).

```
lemma s_tm_alphabet_enlarge:
  fixes M :: ('q, 'a) mttm
  shows s_tm (alphabet_enlarge M
                :: ('q × ('a, ('c :: enum)) ae_stage,
                    'c ⇒ 'a) mttm)
        = (s_tm M, init_stage (le_tm M))
  by (cases M) (simp add: alphabet_enlarge_def)

lemma t_tm_alphabet_enlarge:
  fixes M :: ('q, 'a) mttm
  shows t_tm (alphabet_enlarge M
```

$$:: ('q \times ('a, ('c :: enum))\ ae_stage, 'c \Rightarrow 'a)\ mttm)$$
$$= (t_tm\ M, init_stage\ (le_tm\ M))$$
by (*cases M*) (*simp add*: *alphabet_enlarge_def*)

lemma *r_tm_alphabet_enlarge*:
fixes $M :: ('q, 'a)\ mttm$
shows $r_tm\ (alphabet_enlarge\ M :: ('q \times ('a, ('c :: enum))\ ae_stage, 'c \Rightarrow 'a)\ mttm) = (r_tm\ M, init_stage\ (le_tm\ M))$
by (*cases M*) (*simp add*: *alphabet_enlarge_def*)

lemma *bl_tm_alphabet_enlarge*:
fixes $M :: ('q, 'a)\ mttm$
shows $bl_tm\ (alphabet_enlarge\ M :: ('q \times ('a, ('c :: enum))\ ae_stage, 'c \Rightarrow 'a)\ mttm) = bl_block\ (bl_tm\ M)$
by (*cases M*) (*simp add*: *alphabet_enlarge_def*)

lemma *le_tm_alphabet_enlarge*:
fixes $M :: ('q, 'a)\ mttm$
shows $le_tm\ (alphabet_enlarge\ M :: ('q \times ('a, ('c :: enum))\ ae_stage, 'c \Rightarrow 'a)\ mttm) = LE_block\ (le_tm\ M)$
by (*cases M*) (*simp add*: *alphabet_enlarge_def*)

lemma *delta_tm_alphabet_enlarge*:
fixes $M :: ('q, 'a)\ mttm$
shows $delta_tm\ (alphabet_enlarge\ M :: ('q \times ('a, ('c :: enum))\ ae_stage, 'c \Rightarrow 'a)\ mttm) = alphabet_enlarge_delta\ M$
by (*cases M*) (*simp add*: *alphabet_enlarge_def*)

lemma *Sigma_tm_alphabet_enlarge*:
fixes $M :: ('q, 'a)\ mttm$
shows $Sigma_tm\ (alphabet_enlarge\ M :: ('q \times ('a, ('c :: enum))\ ae_stage, 'c \Rightarrow 'a)\ mttm) = gamma_block\ (Sigma_tm\ M \cup \{bl_tm\ M\}) - \{bl_block\ (bl_tm\ M), LE_block\ (le_tm\ M)\}$
by (*cases M*) (*simp add*: *alphabet_enlarge_def*)

2.12.3 Gamma-block invariants and preservation

Tape well-formedness invariant on an M'-configuration: every cell of every tape lies in *gamma_block* (Γ_*tm* *M*), and every cell of every *inactive* tape (index ≥ *k_tm M*) holds the blank block *bl_block* (*bl_tm M*). The blank-tail conjunct is the value-level analogue of the substrate's config support: with the tape count a runtime *nat*, the inactive tapes must read blank so that a constructed δ'-transition meets the substrate δ-support's read-side condition $\forall j{\geq}k.\ a\ j = bl$. Carried by the simulation (and by each *ae_inv_ss*<*N*>) so that downstream chain steps in *alphabet_enlarge_delta M* can discharge both the intersection guard's read-side membership and the support's blank-tail locally.

definition *ae_tape_in_gamma_block* ::
  ('*q*, '*a*) *mttm*
    ⇒ ('*c* :: *enum* ⇒ '*a*,
        '*q* × ('*a*, '*c*) *ae_stage*) *mt_config*
    ⇒ *bool* **where**
  ae_tape_in_gamma_block M cM' ⟷
    (∀ *k p*. *mt_tape cM' k p* ∈ *gamma_block* (Γ_*tm M*))
    ∧ (∀ *j*≥*k_tm M*. ∀ *p*. *mt_tape cM' j p* = *bl_block* (*bl_tm M*))

Write-side blank-tail of *alphabet_enlarge_delta*: every member writes the blank block *bl_block* (*bl_tm M*) on every inactive tape (index ≥ *k_tm M*). Holds builder-by-builder the validation and buffer-load substeps leave the tape unchanged ($a' = a$, so the write-tail is the read-tail conjunct), and the write-back substeps guard their write to *bl_block* (*bl_tm M*) beyond *k_tm M*. Discharges the δ-support write-side obligation when threading *ae_tape_in_gamma_block*'s blank-tail across a step.

lemma *alphabet_enlarge_delta_write_tail*:
  fixes *M* :: ('*q*, '*a*) *mttm*
  assumes *mem*: (*s*, *a*, *s'*, *a'*, *d*) ∈ *alphabet_enlarge_delta M*
  shows ∀ *j*≥*k_tm M*. *a' j* = *bl_block* (*bl_tm M*)
  using *mem*
  unfolding *alphabet_enlarge_delta_def*
      ae_delta_val_fwd_advance_def ae_delta_val_fwd_to_padded_def
      ae_delta_val_fwd_reject_def ae_delta_val_fwd_to_ret_def
      ae_delta_val_pad_to_ret_def ae_delta_val_pad_reject_def
      ae_delta_val_ret_step_def ae_delta_val_ret_to_sim_def
      ae_delta_ss1_ss2_def ae_delta_ss2_ss3_def ae_delta_ss3_ss4_def
      ae_delta_ss4_ss5_def ae_delta_ss5_ss6_def ae_delta_ss6_ss7_def
      ae_delta_ss7_ss8_def ae_delta_ss8_ss1_def
  by *auto*

Single-step preservation of the gamma-block invariant under *alphabet_enlarge_delta*. The intersection guard in *alphabet_enlarge_delta*'s definition forces the written value *a' k* to lie in *gamma_block* (Γ_*tm M*); untouched cells inherit gamma-block-ness from the precondition on *c'*. Itera-

tion to $\widehat{\ }\widehat{\ }$ k-step chains follows by induction on k at use sites; not packaged separately here pending a concrete use site.

```
lemma ae_step_alphabet_enlarge_gamma_preserve:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes pre:  ae_tape_in_gamma_block M c'
      and step: (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
    shows ae_tape_in_gamma_block M c''
proof −
  from step obtain s ts n s' a' d where
      c'_eq:  c' = Config_M s ts n
      and c''_eq: c'' = Config_M s' (λk. (ts k)(n k := a' k))
                                    (λk. go_dir (d k) (n k))
      and rel: (s, (λk. ts k (n k)), s', a', d) ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  from rel have a'_gamma: ∀ k. a' k ∈ gamma_block (Γ_tm M)
    unfolding alphabet_enlarge_delta_def by auto
  from rel have a'_tail: ∀ j≥k_tm M. a' j = bl_block (bl_tm M)
    by (rule alphabet_enlarge_delta_write_tail)
  have pre_gamma: ∀ k p. mt_tape c' k p ∈ gamma_block (Γ_tm M)
    using pre unfolding ae_tape_in_gamma_block_def by simp
  have pre_tail: ∀ j≥k_tm M. ∀ p. mt_tape c' j p = bl_block (bl_tm M)
    using pre unfolding ae_tape_in_gamma_block_def by simp
  have gamma_part: ∀ k p. mt_tape c'' k p ∈ gamma_block (Γ_tm M)
  proof (intro allI)
    fix k p
    show mt_tape c'' k p ∈ gamma_block (Γ_tm M)
    proof (cases p = n k)
      case True
      with c''_eq have mt_tape c'' k p = a' k by simp
      thus ?thesis using a'_gamma by simp
    next
      case False
      with c''_eq have eq_c'': mt_tape c'' k p = ts k p by simp
      from c'_eq have eq_c': mt_tape c' k p = ts k p by simp
      from eq_c'' eq_c' have mt_tape c'' k p = mt_tape c' k p by simp
      thus ?thesis using pre_gamma by simp
    qed
  qed
  have tail_part: ∀ j≥k_tm M. ∀ p. mt_tape c'' j p = bl_block (bl_tm M)
  proof (intro allI impI)
    fix j p assume jk: k_tm M ≤ j
    show mt_tape c'' j p = bl_block (bl_tm M)
    proof (cases p = n j)
      case True
      with c''_eq have mt_tape c'' j p = a' j by simp
      thus ?thesis using a'_tail jk by simp
    next
```

```
    case False
    with c''_eq have eq_c'': mt_tape c'' j p = ts j p by simp
    from c'_eq have mt_tape c' j p = ts j p by simp
    with eq_c'' have mt_tape c'' j p = mt_tape c' j p by simp
    thus ?thesis using pre_tail jk by simp
   qed
  qed
  show ?thesis
   unfolding ae_tape_in_gamma_block_def
   using gamma_part tail_part by blast
qed
```

Side-band invariant: the M'-state's stage is *valid* in the sense of *ae_valid_stage* at Γ_*tm M*, *le_tm M*, *k_tm M* every block stored in the three-block buffer lies in *gamma_block* (Γ_*tm M*), and the per-tape offset, buffer and destination fields are all frozen at their initial values beyond the tape count *k_tm M*. Companion to *ae_tape_in_gamma_block*: the gamma conjunct discharges the *alphabet_enlarge_delta* intersection's write-side membership at the write-back substeps (SS5→SS6, SS6→SS7, SS7→SS8), and the frozen-tail conjuncts supply the *ae_valid_stage* source obligation the step-existence lemmas feed to the ∗*_dest_valid* preservation lemmas neither is provable from the minimal *ae_inv_ss*<*N*> bodies alone. Equal by construction to *ae_valid_stage* (Γ_*tm M*) (*le_tm M*) (*k_tm M*) (*snd* (*mt_state cM'*)).

```
definition ae_buffer_in_gamma_block ::
  ('q, 'a) mttm
   ⇒ ('c :: enum ⇒ 'a,
      'q × ('a, 'c) ae_stage) mt_config
   ⇒ bool where
  ae_buffer_in_gamma_block M cM' ⟷
    (case mt_state cM' of (_, off, buf, dst, _) ⇒
      (∀ k. fst (buf k) ∈ gamma_block (Γ_tm M)
          ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
          ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M))
     ∧ (∀ j≥k_tm M. off j = init_offset j)
     ∧ (∀ j≥k_tm M. buf j = init_buffer (le_tm M) j)
     ∧ (∀ j≥k_tm M. dst j = init_dest j))
```

Preservation of *ae_buffer_in_gamma_block* (stage validity) under a single *mttm_step* via *alphabet_enlarge_delta*. Companion to *ae_step_alphabet_enlarge_gamma_preserve* (tape-side). The destination stage's *ae_valid_stage* is already an explicit conjunct of *alphabet_enlarge_delta*'s intersection, so the invariant is preserved by reading it straight off the post-state no case split on the 16 sub-deltas is required. (The per-substep discharge of that conjunct lives in the ∗*_dest_valid* lemmas, which the step-existence lemmas invoke when building each δ'-membership.)

lemma *ae_step_alphabet_enlarge_buffer_gamma_preserve*:

```
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes vM:       valid_mttm M
      and pre_buf:  ae_buffer_in_gamma_block M c'
      and pre_tape: ae_tape_in_gamma_block M c'
      and step:     (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
    shows ae_buffer_in_gamma_block M c''
proof −
  obtain s ts n s' a' d where
      rel: (s, (λk. ts k (n k)), s', a', d) ∈ alphabet_enlarge_delta M
      and c'_eq: c' = Config_M s ts n
      and c''_eq: c'' = Config_M s' (λk. (ts k)(n k := a' k))
                                    (λk. go_dir (d k) (n k))
    using step by (auto elim: mttm_step.cases)
  — The destination stage's validity is already an explicit conjunct of alphabet_enlarge_delta's intersection, so the step preserves ae_buffer_in_gamma_block (stage validity) directly  no case split on the 16 sub-deltas is needed.
  obtain q' off' buf' dst' idx' where s'_eq: s' = (q', off', buf', dst', idx')
    by (cases s')
  have vs': ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M) (snd s')
    using rel unfolding alphabet_enlarge_delta_def by auto
  have mt_c'': mt_state c'' = s' using c''_eq by simp
  show ?thesis
    using vs' mt_c'' s'_eq
    unfolding ae_buffer_in_gamma_block_def ae_valid_stage_def by simp
qed
```

Joint preservation of *ae_tape_in_gamma_block* and *ae_buffer_in_gamma_block* across an n-step chain of *mttm_step* (*alphabet_enlarge_delta M*). The per-step buffer-side preservation requires both invariants at the source, so they must thread jointly across the chain.

```
lemma ae_gamma_preserve_relpow:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
    and n :: nat
  assumes vM:       valid_mttm M
      and pre_buf:  ae_buffer_in_gamma_block M c'
      and pre_tape: ae_tape_in_gamma_block M c'
      and chain:    (c', c'') ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
    shows ae_buffer_in_gamma_block M c'' ∧ ae_tape_in_gamma_block M c''
proof −
  have main:
    ∀ d. (c', d) ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
         ⟶ ae_buffer_in_gamma_block M d ∧ ae_tape_in_gamma_block M d
  proof (induct n)
    case 0
```

```
    show ?case
    proof (intro allI impI)
      fix d
      assume (c', d) ∈ mttm_step (alphabet_enlarge_delta M) ^^ 0
      hence d = c' by simp
      thus ae_buffer_in_gamma_block M d ∧ ae_tape_in_gamma_block M d
        using pre_buf pre_tape by simp
    qed
  next
    case (Suc n)
    show ?case
    proof (intro allI impI)
      fix d
      assume chain_Sn:
          (c', d) ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc n
      obtain c_mid where
          chain_n: (c', c_mid) ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
          and step: (c_mid, d) ∈ mttm_step (alphabet_enlarge_delta M)
        using chain_Sn by (auto elim: relpow_Suc_E)
      from Suc.hyps chain_n have
          mid_buf:  ae_buffer_in_gamma_block M c_mid
          and mid_tape: ae_tape_in_gamma_block M c_mid
        by blast+
      have d_tape: ae_tape_in_gamma_block M d
        using ae_step_alphabet_enlarge_gamma_preserve[OF mid_tape step] .
      have d_buf: ae_buffer_in_gamma_block M d
        using ae_step_alphabet_enlarge_buffer_gamma_preserve
              [OF vM mid_buf mid_tape step] .
      show ae_buffer_in_gamma_block M d ∧ ae_tape_in_gamma_block M d
        using d_buf d_tape by simp
    qed
  qed
  show ?thesis using main chain by blast
qed
```

2.12.4 Simulation relation and mid-stage invariants

The simulation relation, stage-granular. Holds at SS1 boundaries (or at halt configurations) between M-configurations and M'-configurations. Captures five invariants: *substep_idx* is SS1 (or the halt branch fires); M-state matches the projected $'q$-component of M''s state; tape contents correspond cell-by-block-and-offset per *ae_tape_correspondence*; head positions correspond per *ae_decode_pos*; and every M'-tape cell lies in *gamma_block* (Γ_*tm* M) per *ae_tape_in_gamma_block*.

Mid-stage configurations (*substep_idx* $\in \{SS2, \ldots, SS7\}$ or any validation phase) are not in this relation; they're handled by the per-substep invariants *ae_inv_ss*<N> below that thread through individual substep proofs.

definition *ae_simulates* ::
(′*q*, ′*a*) *mttm*
⇒ (′*a*, ′*q*) *mt_config*
⇒ (′*c* :: *enum* ⇒ ′*a*,
′*q* × (′*a*, ′*c*) *ae_stage*) *mt_config*
⇒ *bool* **where**
ae_simulates M cM cM′ ⟷
(*let qM* = *mt_state cM*; *tsM* = *mt_tape cM*; *nM* = *mt_pos cM*;
full = *mt_state cM′*;
tsM′ = *mt_tape cM′*; *nM′* = *mt_pos cM′ in*
(*case full of* (*qM′*, *off*, *buf*, *dest*, *idx*) ⇒
((*idx* = *SS1* ∧ *qM′* ∉ {*t_tm M*, *r_tm M*})
∨ (*qM′* ∈ {*t_tm M*, *r_tm M*}
∧ (*off*, *buf*, *dest*, *idx*) = *init_stage* (*le_tm M*)))
∧ *qM* = *qM′*
∧ (∀ *k*<*k_tm M*. *ae_tape_correspondence* (*le_tm M*) (*tsM k*) (*tsM′ k*))
∧ (*idx* = *SS1*
⟶ (∀ *k*<*k_tm M*. *nM k* = *ae_decode_pos* (*nM′ k*) (*off k*)))
∧ *ae_tape_in_gamma_block M cM′*))

Mid-stage invariants: one per simulation substep boundary. Each *ae_inv_ss*<*N*> describes *M*′'s configuration shape at the substep boundary entering substep *SS*<*N*>. The bodies are deliberately minimal: they pin only the *substep_idx* component and the *M*-state's membership in *Q_tm M*. The heavier tape- and buffer-content properties are tracked separately by the companion side-band invariants *ae_tape_in_gamma_block* and *ae_buffer_in_gamma_block* below, rather than folded into these per-substep bodies.

definition *ae_inv_ss1* ::
(′*q*, ′*a*) *mttm*
⇒ (′*c* :: *enum* ⇒ ′*a*,
′*q* × (′*a*, ′*c*) *ae_stage*) *mt_config*
⇒ *bool* **where**
ae_inv_ss1 M cM′ ⟷
(*case mt_state cM′ of* (*qM′*, _, _, _, *idx*) ⇒
idx = *SS1* ∧ *qM′* ∈ *Q_tm M*)

definition *ae_inv_ss2* ::
(′*q*, ′*a*) *mttm*
⇒ (′*c* :: *enum* ⇒ ′*a*,
′*q* × (′*a*, ′*c*) *ae_stage*) *mt_config*
⇒ *bool* **where**
ae_inv_ss2 M cM′ ⟷
(*case mt_state cM′ of* (*qM′*, _, _, _, *idx*) ⇒
idx = *SS2* ∧ *qM′* ∈ *Q_tm M*)

definition *ae_inv_ss3* ::
(′*q*, ′*a*) *mttm*
⇒ (′*c* :: *enum* ⇒ ′*a*,

```
    'q × ('a, 'c) ae_stage) mt_config
  ⇒ bool where
 ae_inv_ss3 M cM' ⟷
   (case mt_state cM' of (qM', _, _, _, idx) ⇒
     idx = SS3 ∧ qM' ∈ Q_tm M)

definition ae_inv_ss4 ::
 ('q, 'a) mttm
  ⇒ ('c :: enum ⇒ 'a,
     'q × ('a, 'c) ae_stage) mt_config
  ⇒ bool where
 ae_inv_ss4 M cM' ⟷
   (case mt_state cM' of (qM', _, _, _, idx) ⇒
     idx = SS4 ∧ qM' ∈ Q_tm M)

definition ae_inv_ss5 ::
 ('q, 'a) mttm
  ⇒ ('c :: enum ⇒ 'a,
     'q × ('a, 'c) ae_stage) mt_config
  ⇒ bool where
 ae_inv_ss5 M cM' ⟷
   (case mt_state cM' of (qM', _, _, _, idx) ⇒
     idx = SS5 ∧ qM' ∈ Q_tm M)

definition ae_inv_ss6 ::
 ('q, 'a) mttm
  ⇒ ('c :: enum ⇒ 'a,
     'q × ('a, 'c) ae_stage) mt_config
  ⇒ bool where
 ae_inv_ss6 M cM' ⟷
   (case mt_state cM' of (qM', _, _, _, idx) ⇒
     idx = SS6 ∧ qM' ∈ Q_tm M)

definition ae_inv_ss7 ::
 ('q, 'a) mttm
  ⇒ ('c :: enum ⇒ 'a,
     'q × ('a, 'c) ae_stage) mt_config
  ⇒ bool where
 ae_inv_ss7 M cM' ⟷
   (case mt_state cM' of (qM', _, _, _, idx) ⇒
     idx = SS7 ∧ qM' ∈ Q_tm M)

definition ae_inv_ss8 ::
 ('q, 'a) mttm
  ⇒ ('c :: enum ⇒ 'a,
     'q × ('a, 'c) ae_stage) mt_config
  ⇒ bool where
 ae_inv_ss8 M cM' ⟷
   (case mt_state cM' of (qM', _, _, _, idx) ⇒
```

idx = SS8 ∧ qM′ ∈ Q_tm M)

2.12.5 LE-guard pre-emption invariant

Per-substep LE-compatibility predicates (SS6, SS7, SS8). At these substeps, the action returns one of the buffer's side slots l, r (or in SS8's halt case, the home slot) in branches the LE-no-write guard cannot discharge structurally (gamma-block alone allows l, $r \in LE_block$). Each predicate asserts the substrate-level guard exactly: at the current configuration, if the substep's action returns *LE_block* on tape k, then *ts k* (*n k*) already equals *LE_block*. These are contracts the chain proof discharges from a richer position-buffer correspondence; the per-substep step-existence lemmas consume them as additional hypotheses.

definition *ae_le_compat_ss6* ::
 ('q, 'a) *mttm*
  ⇒ ('c :: *enum* ⇒ 'a,
   'q × ('a, 'c) *ae_stage*) *mt_config*
  ⇒ *bool* **where**
 ae_le_compat_ss6 M cM′ ⟷
  (*case cM′ of Config*$_M$ (_, _, *buf*, *dest*, _) *ts n* ⇒
   ∀ *k*<*k_tm M*. *fst* (*ae_ss6_action* (*le_tm M*) (*ts k* (*n k*)) (*buf k*) (*dest k*))
    = *LE_block* (*le_tm M*)
   ⟶ *ts k* (*n k*) = *LE_block* (*le_tm M*))

definition *ae_le_compat_ss7* ::
 ('q, 'a) *mttm*
  ⇒ ('c :: *enum* ⇒ 'a,
   'q × ('a, 'c) *ae_stage*) *mt_config*
  ⇒ *bool* **where**
 ae_le_compat_ss7 M cM′ ⟷
  (*case cM′ of Config*$_M$ (_, _, *buf*, *dest*, _) *ts n* ⇒
   ∀ *k*<*k_tm M*. *fst* (*ae_ss7_action* (*le_tm M*) (*ts k* (*n k*)) (*buf k*) (*dest k*))
    = *LE_block* (*le_tm M*)
   ⟶ *ts k* (*n k*) = *LE_block* (*le_tm M*))

definition *ae_le_compat_ss8* ::
 ('q, 'a) *mttm*
  ⇒ ('c :: *enum* ⇒ 'a,
   'q × ('a, 'c) *ae_stage*) *mt_config*
  ⇒ *bool* **where**
 ae_le_compat_ss8 M cM′ ⟷
  (*case cM′ of Config*$_M$ (_, _, *buf*, *dest*, _) *ts n* ⇒
   ∀ *k*<*k_tm M*. *fst* (*ae_ss8_action* (*le_tm M*) (*ts k* (*n k*)) (*buf k*) (*dest k*))
    = *LE_block* (*le_tm M*)
   ⟶ *ts k* (*n k*) = *LE_block* (*le_tm M*))

The LE-guard pre-emption invariant. This is the load-bearing side-band that makes the writeback chain correct across all three regimes (steady-

state, le1, le0).

Why we need it. Each writeback substep's action (*ae_ss5_action* through *ae_ss8_action*) has an LE-guard prefix branch (*a = LE_block le*: write *LE_block* back, idempotent see the action-helper preamble) and a default branch that writes some buffer slot. If the buffer slot the default branch *would* write is itself *LE_block* and the head is at a non-LE position, the default branch would corrupt the tape encoding by writing *LE_block* where non-LE data should be. The construction prevents this by **threading the head trajectory through block 0 (the LE position) at exactly the moments when a buffer-LE-write would otherwise occur** so the LE-guard branch fires first and pre-empts the default branch's dangerous write.

What the predicate captures. The three conjuncts are exactly the structural facts needed for this pre-emption to hold:

1. Post-buffer-load (*idx* $\in$ {*SS5*, *SS6*, *SS7*, *SS8*}): the buffer's r slot is not *LE_block*. This rules out the dangerous write whenever SS6 (*dest = AE_Left*) or SS7/SS8 (various *dest*) would write the r-slot those branches' antecedent (*slot = LE_block*) is vacuously false. Justification: r is loaded from M'-position *p_start + 1* $\geq$ *1*; the substrate's δLE forbids *LE_block* at any position $\geq$ *1*, and the c-step compute (via *ae_m_steps_buffered_correct*) preserves this.

2. SS6 (*dest* $\neq$ *AE_Left*): when SS6's default branch would write the l-slot, and that l-slot is *LE_block*, then the head IS at a position holding *LE_block*. The LE-guard fires and pre-empts the default branch. In steady-state ($s \geq 2$) the antecedent l = *LE_block* is vacuously false (l is loaded from block $s-1 \geq 1$, non-LE). In le1 ($s = 1$) the antecedent is true (l is loaded from block 0, which IS *LE_block*), and the consequent is established by le1's head trajectory: SS5's *move L* rule (for *dest* $\neq$ *AE_Left*) takes the head from block 1 (SS5 entry) to block 0 (SS6 entry) where it reads *LE_block*.

3. SS8 (*dest = AE_Left*): symmetric to the SS6 clause. When SS8's default branch would write the l-slot for *dest = AE_Left*, and that slot is *LE_block*, the head is at *LE_block*. Steady-state vacuous; in le1 with *dest = AE_Left*, SS5's *move R* + SS6's *move L* + SS7's *move L* threads the head from block 1 to block 0 by SS8 entry.

   Other substep+dest combinations either write r (covered by conjunct 1, vacuous antecedent), write a back idempotently (SS7's default; no *LE_block* ever introduced), or fire only on the LE-guard branch (writing *LE_block* only when a = *LE_block*, which is trivially safe).

   Why this isn't separate predicates per regime. The predicate's antecedents (*slot = LE_block*) are inherently regime-discriminating:

steady-state satisfies them vacuously, le1 satisfies them via head-trajectory tracing. A single uniform predicate works because the sub-step actions' direction rules (*ae_ss5_action*'s "move L when *dest* ≠ *AE_Left*", etc.) deliver the head to block 0 at exactly the right sub-steps in le1, and don't need to in steady-state.

```
definition ae_position_link ::
  ('q, 'a) mttm
    ⇒ ('c :: enum ⇒ 'a,
       'q × ('a, 'c) ae_stage) mt_config
    ⇒ bool where
  ae_position_link M cM' ⟷
    (case mt_state cM' of (_, _, buf, dest, idx) ⇒
      (idx ∈ {SS5, SS6, SS7, SS8} ⟶
        (∀ k<k_tm M. snd (snd (buf k)) ≠ LE_block (le_tm M)))
      ∧ (idx = SS6 ⟶
        (∀ k<k_tm M. dest k ≠ AE_Left
              ⟶ fst (buf k) = LE_block (le_tm M)
              ⟶ mt_tape cM' k (mt_pos cM' k)
                  = LE_block (le_tm M)))
      ∧ (idx = SS8 ⟶
        (∀ k<k_tm M. dest k = AE_Left
              ⟶ fst (buf k) = LE_block (le_tm M)
              ⟶ mt_tape cM' k (mt_pos cM' k)
                  = LE_block (le_tm M))))
```

2.12.6 Position-link discharges

Discharge lemmas: at SS<*N*> entry (per *ae_inv_ss<N>*), the side-band *ae_position_link* implies the per-substep contract *ae_le_compat_ss<N>*. Each proof case-splits on the *ae_ss<N>_action*'s branches: r-write branches close via the first conjunct of *ae_position_link* (*r* ≠ *LE_block* makes the antecedent vacuous); a-write branches close trivially via *a'* = *a*; the l-write branch (SS6 with *dest* ≠ *AE_Left*; SS8 with *dest* = *AE_Left*) closes via the second conjunct (the position-link's *l* = *LE_block* ⟶ *ts* (*n*) = *LE_block* guard for that substep+dest). Used at the chain proof's call site to discharge the *ae_le_compat_ss<N>* hypotheses on the *ae_step_ss<N>_ss<N+1>_exists* step-existence helpers.

```
lemma ae_position_link_discharges_ss6:
  fixes M :: ('q, 'a) mttm
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
  assumes inv: ae_inv_ss6 M c'
      and pos: ae_position_link M c'
    shows ae_le_compat_ss6 M c'
proof −
  obtain q ofs buf dest where
```

```
    state_eq: mt_state c' = (q, ofs, buf, dest, SS6)
  using inv unfolding ae_inv_ss6_def
  by (cases mt_state c') auto
obtain ts n where
    c'_eq: c' = Config_M (q, ofs, buf, dest, SS6) ts n
  using state_eq by (cases c') auto
have r_not_le: ∀ k<k_tm M. snd (snd (buf k)) ≠ LE_block (le_tm M)
  using pos state_eq unfolding ae_position_link_def by simp
have l_guard:
    ∀ k<k_tm M. dest k ≠ AE_Left
          ⟶ fst (buf k) = LE_block (le_tm M)
          ⟶ ts k (n k) = LE_block (le_tm M)
  using pos state_eq c'_eq unfolding ae_position_link_def by simp
have body:
    ∀ k<k_tm M. fst (ae_ss6_action (le_tm M) (ts k (n k)) (buf k) (dest k))
          = LE_block (le_tm M)
        ⟶ ts k (n k) = LE_block (le_tm M)
proof (intro allI impI)
  fix k
  assume klt: k < k_tm M
  let ?a = ts k (n k)
  obtain l h r where buf_k: buf k = (l, h, r)
    by (cases buf k) auto
  assume act_le:
      fst (ae_ss6_action (le_tm M) ?a (buf k) (dest k))
        = LE_block (le_tm M)
  show ?a = LE_block (le_tm M)
  proof (cases ?a = LE_block (le_tm M))
    case True thus ?thesis .
  next
    case a_neq: False
    show ?thesis
    proof (cases h = LE_block (le_tm M))
      case h_le: True
      from a_neq h_le buf_k have
          fst (ae_ss6_action (le_tm M) ?a (buf k) (dest k)) = ?a
        by simp
      with act_le show ?thesis by simp
    next
      case h_neq: False
      show ?thesis
      proof (cases dest k = AE_Left)
        case ds_l: True
        from a_neq h_neq ds_l buf_k have
            fst (ae_ss6_action (le_tm M) ?a (buf k) (dest k)) = r
          by simp
        with act_le have r = LE_block (le_tm M) by simp
        with r_not_le[rule_format, OF klt] buf_k have False by simp
        thus ?thesis ..
```

```
      next
        case ds_nl: False
        from a_neq h_neq ds_nl buf_k have
            fst (ae_ss6_action (le_tm M) ?a (buf k) (dest k)) = l
          by simp
        with act_le have l_le: l = LE_block (le_tm M) by simp
        have fst (buf k) = l using buf_k by simp
        with l_le ds_nl l_guard[rule_format, OF klt] show ?thesis by simp
      qed
    qed
  qed
  show ?thesis
    using body unfolding ae_le_compat_ss6_def c'_eq by simp
qed

lemma ae_position_link_discharges_ss7:
  fixes M :: ('q, 'a) mttm
    and c' :: ('c :: enum ⇒ 'a,
                'q × ('a, 'c) ae_stage) mt_config
  assumes inv: ae_inv_ss7 M c'
      and pos: ae_position_link M c'
    shows ae_le_compat_ss7 M c'
proof −
  obtain q ofs buf dest where
      state_eq: mt_state c' = (q, ofs, buf, dest, SS7)
    using inv unfolding ae_inv_ss7_def
    by (cases mt_state c') auto
  obtain ts n where
      c'_eq: c' = Config_M (q, ofs, buf, dest, SS7) ts n
    using state_eq by (cases c') auto
  have r_not_le: ∀ k<k_tm M. snd (snd (buf k)) ≠ LE_block (le_tm M)
    using pos state_eq unfolding ae_position_link_def by simp
  have body:
      ∀ k<k_tm M. fst (ae_ss7_action (le_tm M) (ts k (n k)) (buf k) (dest k))
            = LE_block (le_tm M)
          ⟶ ts k (n k) = LE_block (le_tm M)
  proof (intro allI impI)
    fix k
    assume klt: k < k_tm M
    let ?a = ts k (n k)
    obtain l h r where buf_k: buf k = (l, h, r)
      by (cases buf k) auto
    assume act_le:
        fst (ae_ss7_action (le_tm M) ?a (buf k) (dest k))
          = LE_block (le_tm M)
    show ?a = LE_block (le_tm M)
    proof (cases ?a = LE_block (le_tm M))
      case True thus ?thesis .
```

```
    next
      case a_neq: False
      show ?thesis
      proof (cases h = LE_block (le_tm M))
        case h_le: True
        from a_neq h_le buf_k have
            fst (ae_ss7_action (le_tm M) ?a (buf k) (dest k)) = r
          by simp
        with act_le have r = LE_block (le_tm M) by simp
        with r_not_le[rule_format, OF klt] buf_k have False by simp
        thus ?thesis ..
      next
        case h_neq: False
        from a_neq h_neq buf_k have
            fst (ae_ss7_action (le_tm M) ?a (buf k) (dest k)) = ?a
          by simp
        with act_le show ?thesis by simp
      qed
    qed
  qed
  show ?thesis
    using body unfolding ae_le_compat_ss7_def c'_eq by simp
qed

lemma ae_position_link_discharges_ss8:
  fixes M :: ('q, 'a) mttm
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
  assumes inv: ae_inv_ss8 M c'
      and pos: ae_position_link M c'
    shows ae_le_compat_ss8 M c'
proof −
  obtain q ofs buf dest where
      state_eq: mt_state c' = (q, ofs, buf, dest, SS8)
    using inv unfolding ae_inv_ss8_def
    by (cases mt_state c') auto
  obtain ts n where
      c'_eq: c' = Config_M (q, ofs, buf, dest, SS8) ts n
    using state_eq by (cases c') auto
  have r_not_le: ∀ k<k_tm M. snd (snd (buf k)) ≠ LE_block (le_tm M)
    using pos state_eq unfolding ae_position_link_def by simp
  have l_guard:
      ∀ k<k_tm M. dest k = AE_Left
            ⟶ fst (buf k) = LE_block (le_tm M)
            ⟶ ts k (n k) = LE_block (le_tm M)
    using pos state_eq c'_eq unfolding ae_position_link_def by simp
  have body:
      ∀ k<k_tm M. fst (ae_ss8_action (le_tm M) (ts k (n k)) (buf k) (dest k))
            = LE_block (le_tm M)
```

```
        ⟶ ts k (n k) = LE_block (le_tm M)
proof (intro allI impI)
  fix k
  assume klt: k < k_tm M
  let ?a = ts k (n k)
  obtain l h r where buf_k: buf k = (l, h, r)
    by (cases buf k) auto
  assume act_le:
     fst (ae_ss8_action (le_tm M) ?a (buf k) (dest k))
         = LE_block (le_tm M)
  show ?a = LE_block (le_tm M)
  proof (cases ?a = LE_block (le_tm M))
    case True thus ?thesis .
  next
    case a_neq: False
    show ?thesis
    proof (cases h = LE_block (le_tm M))
      case h_le: True
      show ?thesis
      proof (cases dest k = AE_Right)
        case ds_r: True
        from a_neq h_le ds_r buf_k have
            fst (ae_ss8_action (le_tm M) ?a (buf k) (dest k)) = r
          by simp
        with act_le have r = LE_block (le_tm M) by simp
        with r_not_le[rule_format, OF klt] buf_k have False by simp
        thus ?thesis ..
      next
        case ds_nr: False
        from a_neq h_le ds_nr buf_k have
            fst (ae_ss8_action (le_tm M) ?a (buf k) (dest k)) = ?a
          by simp
        with act_le show ?thesis by simp
      qed
    next
      case h_neq: False
      show ?thesis
      proof (cases dest k)
        case AE_Left
        from a_neq h_neq AE_Left buf_k have
            fst (ae_ss8_action (le_tm M) ?a (buf k) (dest k)) = l
          by simp
        with act_le have l_le: l = LE_block (le_tm M) by simp
        have fst (buf k) = l using buf_k by simp
        with l_le AE_Left l_guard[rule_format, OF klt] show ?thesis by simp
      next
        case AE_Home
        from a_neq h_neq AE_Home buf_k have
            fst (ae_ss8_action (le_tm M) ?a (buf k) (dest k)) = r
```

```
        by simp
      with act_le have r = LE_block (le_tm M) by simp
      with r_not_le[rule_format, OF klt] buf_k have False by simp
      thus ?thesis ..
    next
      case AE_Right
      from a_neq h_neq AE_Right buf_k have
          fst (ae_ss8_action (le_tm M) ?a (buf k) (dest k)) = r
        by simp
      with act_le have r = LE_block (le_tm M) by simp
      with r_not_le[rule_format, OF klt] buf_k have False by simp
      thus ?thesis ..
    qed
  qed
 qed
 show ?thesis
  using body unfolding ae_le_compat_ss8_def c'_eq by simp
qed
```

2.12.7 Pre-state idx and void-aux helpers

Pre-state idx determinations for the three substantive sub-deltas of *ae_step_alphabet_enlarge_position_link_preserve*'s aux hypotheses. Used by the chain proof to discharge vacuous-aux cases of the preservation lemma: when the actual sub-step is e.g. *ss1→ss2*, the pre-state idx is SS1 ≠ SS4, so the *aux_ss4_ss5* antecedent is unsatisfiable.

```
lemma ae_delta_ss4_ss5_pre_idx:
 assumes (s, a, s', a', d) ∈ ae_delta_ss4_ss5 M
 shows snd (snd (snd (snd s))) = SS4
 using assms unfolding ae_delta_ss4_ss5_def by force

lemma ae_delta_ss5_ss6_pre_idx:
 assumes (s, a, s', a', d) ∈ ae_delta_ss5_ss6 M
 shows snd (snd (snd (snd s))) = SS5
 using assms unfolding ae_delta_ss5_ss6_def by force

lemma ae_delta_ss7_ss8_pre_idx:
 assumes (s, a, s', a', d) ∈ ae_delta_ss7_ss8 M
 shows snd (snd (snd (snd s))) = SS7
 using assms unfolding ae_delta_ss7_ss8_def by force
```

Vacuous-aux helpers: when the actual sub-step is not SS4→SS5 / SS5→SS6 / SS7→SS8 (per the pre-state idx), the corresponding aux antecedent of *ae_step_alphabet_enlarge_position_link_preserve* is unsatisfiable, so the aux holds vacuously. Used in the chain proof to discharge the three aux hypotheses at substeps whose pre-state idx doesn't match.

```
lemma ae_pos_link_aux_void_ss4_ss5:
```

```
  fixes M :: ('q, 'a) mttm
    and c_pre c_post :: ('c :: enum ⇒ 'a,
                         'q × ('a, 'c) ae_stage) mt_config
  assumes pre_neq: snd (snd (snd (snd (mt_state c_pre)))) ≠ SS4
      and step:    (c_pre, c_post) ∈ mttm_step (ae_delta_ss4_ss5 M)
    shows ae_position_link M c_post
proof −
  from step obtain s ts n s' a' d where
      eq: c_pre = Config_M s ts n
      and rel: (s, (λk. ts k (n k)), s', a', d) ∈ ae_delta_ss4_ss5 M
    by (auto elim: mttm_step.cases)
  have snd (snd (snd (snd s))) = SS4
    using ae_delta_ss4_ss5_pre_idx[OF rel] .
  with eq pre_neq have False by simp
  thus ?thesis ..
qed

lemma ae_pos_link_aux_void_ss5_ss6:
  fixes M :: ('q, 'a) mttm
    and c_pre c_post :: ('c :: enum ⇒ 'a,
                         'q × ('a, 'c) ae_stage) mt_config
  assumes pre_neq: snd (snd (snd (snd (mt_state c_pre)))) ≠ SS5
      and step:    (c_pre, c_post) ∈ mttm_step (ae_delta_ss5_ss6 M)
    shows ae_position_link M c_post
proof −
  from step obtain s ts n s' a' d where
      eq: c_pre = Config_M s ts n
      and rel: (s, (λk. ts k (n k)), s', a', d) ∈ ae_delta_ss5_ss6 M
    by (auto elim: mttm_step.cases)
  have snd (snd (snd (snd s))) = SS5
    using ae_delta_ss5_ss6_pre_idx[OF rel] .
  with eq pre_neq have False by simp
  thus ?thesis ..
qed

lemma ae_pos_link_aux_void_ss7_ss8:
  fixes M :: ('q, 'a) mttm
    and c_pre c_post :: ('c :: enum ⇒ 'a,
                         'q × ('a, 'c) ae_stage) mt_config
  assumes pre_neq: snd (snd (snd (snd (mt_state c_pre)))) ≠ SS7
      and step:    (c_pre, c_post) ∈ mttm_step (ae_delta_ss7_ss8 M)
    shows ae_position_link M c_post
proof −
  from step obtain s ts n s' a' d where
      eq: c_pre = Config_M s ts n
      and rel: (s, (λk. ts k (n k)), s', a', d) ∈ ae_delta_ss7_ss8 M
    by (auto elim: mttm_step.cases)
  have snd (snd (snd (snd s))) = SS7
    using ae_delta_ss7_ss8_pre_idx[OF rel] .
```

```
  with eq pre_neq have False by simp
  thus ?thesis ..
qed
```

2.12.8 Position-link preservation

Preservation of *ae_position_link* under a single *mttm_step* via *alphabet_enlarge_delta*. The proof case-splits on which of the 16 sub-deltas fires. Validation transitions (8) and simulation transitions whose post-idx is outside the post-buffer-load regime (*ss1→ss2*, *ss2→ss3*, *ss3→ss4*, *ss8→ss1*) close vacuously the predicate's content is conditional on $idx \in \{SS5, SS6, SS7, SS8\}$. The buffer-stable transition *ss6→ss7* closes by transferring the pre-state's $r \neq LE_block$ across the buffer-preserving step. The three substantive transitions *ss4→ss5*, *ss5→ss6*, *ss7→ss8* require auxiliary hypotheses that the chain proof discharges from the broader simulation context (*m_steps_buffered_correct*'s δLE preservation, the buffer-load invariants, and the cadence at SS5/SS7 entry).

```
lemma ae_step_alphabet_enlarge_position_link_preserve:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes pre: ae_position_link M c'
      and step: (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
      and aux_ss4_ss5:
          (c', c'') ∈ mttm_step (ae_delta_ss4_ss5 M)
            ⟹ ae_position_link M c''
      and aux_ss5_ss6:
          (c', c'') ∈ mttm_step (ae_delta_ss5_ss6 M)
            ⟹ ae_position_link M c''
      and aux_ss7_ss8:
          (c', c'') ∈ mttm_step (ae_delta_ss7_ss8 M)
            ⟹ ae_position_link M c''
    shows ae_position_link M c''
proof −
  obtain s ts n s' a' d where
      rel: (s, (λk. ts k (n k)), s', a', d) ∈ alphabet_enlarge_delta M
      and c'_eq: c' = Config_M s ts n
      and c''_eq: c'' = Config_M s' (λk. (ts k)(n k := a' k))
                                    (λk. go_dir (d k) (n k))
    using step by (auto elim: mttm_step.cases)
  let ?rel_tup = (s, (λk. ts k (n k)), s', a', d)
  consider
      (vfa) ?rel_tup ∈ ae_delta_val_fwd_advance M
    | (vfp) ?rel_tup ∈ ae_delta_val_fwd_to_padded M
    | (vfr) ?rel_tup ∈ ae_delta_val_fwd_reject M
    | (vfr2) ?rel_tup ∈ ae_delta_val_fwd_to_ret M
    | (vpr) ?rel_tup ∈ ae_delta_val_pad_to_ret M
    | (vpr2) ?rel_tup ∈ ae_delta_val_pad_reject M
```

```
  | (vrs) ?rel_tup ∈ ae_delta_val_ret_step M
  | (vrts) ?rel_tup ∈ ae_delta_val_ret_to_sim M
  | (s12) ?rel_tup ∈ ae_delta_ss1_ss2 M
  | (s23) ?rel_tup ∈ ae_delta_ss2_ss3 M
  | (s34) ?rel_tup ∈ ae_delta_ss3_ss4 M
  | (s45) ?rel_tup ∈ ae_delta_ss4_ss5 M
  | (s56) ?rel_tup ∈ ae_delta_ss5_ss6 M
  | (s67) ?rel_tup ∈ ae_delta_ss6_ss7 M
  | (s78) ?rel_tup ∈ ae_delta_ss7_ss8 M
  | (s81) ?rel_tup ∈ ae_delta_ss8_ss1 M
  using rel unfolding alphabet_enlarge_delta_def by blast
 thus ?thesis
 proof cases
  case vfa
  thus ?thesis
                          using c''_eq unfolding ae_position_link_def
ae_delta_val_fwd_advance_def
    by auto
 next
  case vfp
  thus ?thesis
                          using c''_eq unfolding ae_position_link_def
ae_delta_val_fwd_to_padded_def
    by auto
 next
  case vfr
  thus ?thesis
    using c''_eq
   unfolding ae_position_link_def ae_delta_val_fwd_reject_def init_stage_def
    by auto
 next
  case vfr2
  thus ?thesis
   using c''_eq unfolding ae_position_link_def ae_delta_val_fwd_to_ret_def
    by auto
 next
  case vpr
  thus ?thesis
   using c''_eq unfolding ae_position_link_def ae_delta_val_pad_to_ret_def
    by auto
 next
  case vpr2
  thus ?thesis
    using c''_eq
   unfolding ae_position_link_def ae_delta_val_pad_reject_def init_stage_def
    by auto
 next
  case vrs
  thus ?thesis
```

```
    using c''_eq unfolding ae_position_link_def ae_delta_val_ret_step_def
    by auto
  next
   case vrts
   thus ?thesis
    using c''_eq unfolding ae_position_link_def ae_delta_val_ret_to_sim_def
    by auto
  next
   case s12
   thus ?thesis
    using c''_eq unfolding ae_position_link_def ae_delta_ss1_ss2_def
    by auto
  next
   case s23
   thus ?thesis
    using c''_eq unfolding ae_position_link_def ae_delta_ss2_ss3_def
    by auto
  next
   case s34
   thus ?thesis
    using c''_eq unfolding ae_position_link_def ae_delta_ss3_ss4_def
    by auto
  next
   case s45
   have step_s45: (c', c'') ∈ mttm_step (ae_delta_ss4_ss5 M)
    using s45 c'_eq c''_eq by (auto intro: mttm_step.intros)
   thus ?thesis by (rule aux_ss4_ss5)
  next
   case s56
   have step_s56: (c', c'') ∈ mttm_step (ae_delta_ss5_ss6 M)
    using s56 c'_eq c''_eq by (auto intro: mttm_step.intros)
   thus ?thesis by (rule aux_ss5_ss6)
  next
   case s67
   — Buffer unchanged from SS6 (pre) to SS7 (post). Pre's r ≠ LE_block at SS6
transfers to SS7.
   obtain q ofs buf dest where
     pre_state: s = (q, ofs, buf, dest, SS6)
     and post_state: s' = (q, ofs, buf, dest, SS7)
    using s67 unfolding ae_delta_ss6_ss7_def by auto
   have pre_state': mt_state c' = (q, ofs, buf, dest, SS6)
    using c'_eq pre_state by simp
   have post_state': mt_state c'' = (q, ofs, buf, dest, SS7)
    using c''_eq post_state by simp
   have r_not_le:
     ∀ k<k_tm M. snd (snd (buf k)) ≠ LE_block (le_tm M)
    using pre pre_state' unfolding ae_position_link_def by simp
   show ?thesis
    unfolding ae_position_link_def post_state'
```

```
    using r_not_le by simp
  next
    case s78
    have step_s78: (c', c'') ∈ mttm_step (ae_delta_ss7_ss8 M)
      using s78 c'_eq c''_eq by (auto intro: mttm_step.intros)
    thus ?thesis by (rule aux_ss7_ss8)
  next
    case s81
   — Post-state is SS1 (non-halt branch) or init_stage le (halt branch); both have
idx outside the regime.
    obtain q ofs buf dest where
        pre_state: s = (q, ofs, buf, dest, SS8)
        and post_state:
          s' = (q, if q ∈ {t_tm M, r_tm M}
                   then init_stage (le_tm M)
                   else (ofs, buf, init_dest, SS1))
      using s81 unfolding ae_delta_ss8_ss1_def by auto
    show ?thesis
    proof (cases q ∈ {t_tm M, r_tm M})
      case True
      hence s' = (q, init_stage (le_tm M)) using post_state by simp
      thus ?thesis
        using c''_eq
        unfolding ae_position_link_def init_stage_def
        by auto
    next
      case False
      hence s' = (q, ofs, buf, init_dest, SS1) using post_state by simp
      thus ?thesis
        using c''_eq
        unfolding ae_position_link_def
        by auto
    qed
  qed
qed
```

2.12.9 Per-substep step-existence

Per-substep step-existence lemmas (forward simulation, buffer phase). Each lemma, given a configuration c' satisfying the appropriate *ae_inv_ss<N>* precondition (plus whichever side-band invariants the substep requires), produces a witness c'' for which both the per-substep step *mttm_step* (*ae_delta_ss<N>_ss<N+1>* *M*) and the full-delta step *mttm_step* (*alphabet_enlarge_delta M*) hold. Per-substep delta is forwarded to the existing *ae_step_ss<N>_ss<N+1>_invariant* lemma at the chain orchestration site (*ae_simulates_forward_stage*); full-delta step accumulates into the chain's *mttm_step** closure.

Hypothesis pattern by substep:

- SS1→SS2, SS2→SS3, SS3→SS4 (buffer-load): *ae_inv_ss<N>* + *ae_tape_in_gamma_block* + non-halt; substrate write is a no-op ($a' = a$).

- SS4→SS5 (c-fold compute): adds *valid_mttm*, *delta_total*, the per-tape *ae_window_invariant*, and a no-LE-in-window hypothesis; the substrate witness is obtained by invoking *ae_m_steps_buffered_correct*. Substrate write is still a no-op. Located in *AlphabetEnlargement* rather than this theory because *ae_m_steps_buffered_correct* lives there.

- SS5→SS6 (write-back, first cell): *ae_inv_ss5* + *ae_tape_in_gamma_block* + *ae_buffer_in_gamma_block*; no non-halt hypothesis (the SS5 action is total in its inputs; halt routing kicks in only at SS8→SS1).

- SS6→SS7 (write-back, side cell): *ae_inv_ss6* + *ae_tape_in_gamma_block* + *ae_buffer_in_gamma_block* + *ae_le_compat_ss6*. The fourth hypothesis is the per-substep LE-no-write contract (the action returns the buffer's l or r slot in steady-state, and gamma-block alone does not preclude $l, r \in$ *LE_block*).

- SS7→SS8: same shape as SS6, with *ae_le_compat_ss7*.

- SS8→SS1: same shape, with *ae_le_compat_ss8*; halt-aware destination routing is internal to *ae_delta_ss8_ss1*'s *stage'* if-then-else, so step-existence is uniform across halt and non-halt.

```
lemma ae_step_ss1_ss2_exists:
  fixes M :: ('q, 'a) mttm
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
  assumes inv:     ae_inv_ss1 M c'
      and gamma:   ae_tape_in_gamma_block M c'
      and buf:     ae_buffer_in_gamma_block M c'
      and q_neq_t: fst (mt_state c') ≠ t_tm M
      and q_neq_r: fst (mt_state c') ≠ r_tm M
  shows ∃ c''. (c', c'') ∈ mttm_step (ae_delta_ss1_ss2 M)
               ∧ (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  obtain q ofs buf dest where
      state_eq: mt_state c' = (q, ofs, buf, dest, SS1)
      and q_in: q ∈ Q_tm M
    using inv unfolding ae_inv_ss1_def
    by (cases mt_state c') auto
  obtain ts n where c'_eq: c' = Config_M (q, ofs, buf, dest, SS1) ts n
```

using *state_eq* **by** (*cases c'*) *auto*
let *?a* = λ*k*. *ts k* (*n k*)
let *?buf'* = λ*k*. *if k* < *k_tm M*
then (*fst* (*buf k*), *?a k*, *snd* (*snd* (*buf k*)))
else init_buffer (*le_tm M*) *k*
let *?d* = λ*k*. *if k* < *k_tm M*
then (*if ?a k* = *LE_block* (*le_tm M*) *then dir.N else dir.L*)
else dir.N
have *q_neq_t'*: *q* ≠ *t_tm M* **using** *q_neq_t state_eq* **by** *simp*
have *q_neq_r'*: *q* ≠ *r_tm M* **using** *q_neq_r state_eq* **by** *simp*
have *a_tail*: ∀ *j*≥*k_tm M*. *?a j* = *bl_block* (*bl_tm M*)
using *gamma c'_eq* **unfolding** *ae_tape_in_gamma_block_def* **by** *simp*
have *rel_in*: ((*q*, *ofs*, *buf*, *dest*, *SS1*), *?a*,
(*q*, *ofs*, *?buf'*, *dest*, *SS2*), *?a*, *?d*)
∈ *ae_delta_ss1_ss2 M*
unfolding *ae_delta_ss1_ss2_def*
using *q_in q_neq_t' q_neq_r' a_tail* **by** *auto*
have *gamma_a*: ∀ *k*. *?a k* ∈ *gamma_block* (Γ_*tm M*)
using *gamma c'_eq* **unfolding** *ae_tape_in_gamma_block_def* **by** *simp*
have *src_valid*: *ae_valid_stage* (Γ_*tm M*) (*le_tm M*) (*k_tm M*)
(*snd* ((*q*, *ofs*, *buf*, *dest*, *SS1*)
:: 'q × ('a, 'c) *ae_stage*))
using *buf state_eq*
unfolding *ae_buffer_in_gamma_block_def ae_valid_stage_def* **by** *simp*
have *dst_valid*: *ae_valid_stage* (Γ_*tm M*) (*le_tm M*) (*k_tm M*)
(*snd* ((*q*, *ofs*, *?buf'*, *dest*, *SS2*)
:: 'q × ('a, 'c) *ae_stage*))
by (*rule ae_delta_ss1_ss2_dest_valid*[*OF rel_in gamma_a src_valid*])
have *aed_in*: ((*q*, *ofs*, *buf*, *dest*, *SS1*), *?a*,
(*q*, *ofs*, *?buf'*, *dest*, *SS2*), *?a*, *?d*)
∈ *alphabet_enlarge_delta M*
unfolding *alphabet_enlarge_delta_def*
using *rel_in gamma_a src_valid dst_valid* **by** *auto*
have *ts_unchanged*: (λ*k*. (*ts k*)(*n k* := *?a k*)) = *ts*
by (*rule ext*) *auto*
let *?c''* = *Config*$_M$ (*q*, *ofs*, *?buf'*, *dest*, *SS2*) *ts*
(λ*k*. *go_dir* (*?d k*) (*n k*))
have *step_sub_raw*:
(*Config*$_M$ (*q*, *ofs*, *buf*, *dest*, *SS1*) *ts n*,
Config$_M$ (*q*, *ofs*, *?buf'*, *dest*, *SS2*)
(λ*k*. (*ts k*)(*n k* := *?a k*))
(λ*k*. *go_dir* (*?d k*) (*n k*)))
∈ *mttm_step* (*ae_delta_ss1_ss2 M*)
proof (*rule mttm_step.intros*)
show ((*q*, *ofs*, *buf*, *dest*, *SS1*), *?a*,
(*q*, *ofs*, *?buf'*, *dest*, *SS2*), *?a*, *?d*)
∈ *ae_delta_ss1_ss2 M* **by** (*rule rel_in*)
qed
have *step_sub*: (*c'*, *?c''*) ∈ *mttm_step* (*ae_delta_ss1_ss2 M*)

```
    using step_sub_raw c'_eq ts_unchanged by simp
  have step_full_raw:
    (Config_M (q, ofs, buf, dest, SS1) ts n,
       Config_M (q, ofs, ?buf', dest, SS2)
         (λk. (ts k)(n k := ?a k))
         (λk. go_dir (?d k) (n k)))
       ∈ mttm_step (alphabet_enlarge_delta M)
  proof (rule mttm_step.intros)
    show ((q, ofs, buf, dest, SS1), ?a,
          (q, ofs, ?buf', dest, SS2), ?a, ?d)
          ∈ alphabet_enlarge_delta M by (rule aed_in)
  qed
  have step_full: (c', ?c'') ∈ mttm_step (alphabet_enlarge_delta M)
    using step_full_raw c'_eq ts_unchanged by simp
  show ?thesis using step_sub step_full by blast
qed

lemma ae_step_ss2_ss3_exists:
  fixes M :: ('q, 'a) mttm
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
  assumes vM:      valid_mttm M
      and inv:     ae_inv_ss2 M c'
      and gamma:   ae_tape_in_gamma_block M c'
      and buf:     ae_buffer_in_gamma_block M c'
      and q_neq_t: fst (mt_state c') ≠ t_tm M
      and q_neq_r: fst (mt_state c') ≠ r_tm M
  shows ∃ c''. (c', c'') ∈ mttm_step (ae_delta_ss2_ss3 M)
             ∧ (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  obtain q ofs buf dest where
      state_eq: mt_state c' = (q, ofs, buf, dest, SS2)
      and q_in: q ∈ Q_tm M
    using inv unfolding ae_inv_ss2_def
    by (cases mt_state c') auto
  obtain ts n where c'_eq: c' = Config_M (q, ofs, buf, dest, SS2) ts n
    using state_eq by (cases c') auto
  let ?a = λk. ts k (n k)
  let ?buf' = λk. if k < k_tm M
                  then (let (l, h, r) = buf k in
                          (if h = LE_block (le_tm M)
                             then bl_block (bl_tm M)
                             else ?a k,
                           h, r))
                  else init_buffer (le_tm M) k
  let ?d = λk. if k < k_tm M
               then (if (fst (snd (buf k))) = LE_block (le_tm M)
                     then dir.N else dir.R)
               else dir.N
```

have *q_neq_t'*: $q \neq t_tm\ M$ **using** *q_neq_t state_eq* **by** *simp*
have *q_neq_r'*: $q \neq r_tm\ M$ **using** *q_neq_r state_eq* **by** *simp*
have *a_tail*: $\forall j{\geq}k_tm\ M.\ ?a\ j = bl_block\ (bl_tm\ M)$
  using *gamma c'_eq* **unfolding** *ae_tape_in_gamma_block_def* **by** *simp*
have *rel_in*: $((q,\ ofs,\ buf,\ dest,\ SS2),\ ?a,$
  $(q,\ ofs,\ ?buf',\ dest,\ SS3),\ ?a,\ ?d)$
  $\in ae_delta_ss2_ss3\ M$
  unfolding *ae_delta_ss2_ss3_def*
  using *q_in q_neq_t' q_neq_r' a_tail* **by** *auto*
have *gamma_a*: $\forall k.\ ?a\ k \in gamma_block\ (\Gamma_tm\ M)$
  using *gamma c'_eq* **unfolding** *ae_tape_in_gamma_block_def* **by** *simp*
have *src_valid*: $ae_valid_stage\ (\Gamma_tm\ M)\ (le_tm\ M)\ (k_tm\ M)$
  $(snd\ ((q,\ ofs,\ buf,\ dest,\ SS2)$
  $::\ 'q \times ('a,\ 'c)\ ae_stage))$
  using *buf state_eq*
  unfolding *ae_buffer_in_gamma_block_def ae_valid_stage_def* **by** *simp*
have *dst_valid*: $ae_valid_stage\ (\Gamma_tm\ M)\ (le_tm\ M)\ (k_tm\ M)$
  $(snd\ ((q,\ ofs,\ ?buf',\ dest,\ SS3)$
  $::\ 'q \times ('a,\ 'c)\ ae_stage))$
  by (*rule ae_delta_ss2_ss3_dest_valid*[*OF vM rel_in gamma_a src_valid*])
have *aed_in*: $((q,\ ofs,\ buf,\ dest,\ SS2),\ ?a,$
  $(q,\ ofs,\ ?buf',\ dest,\ SS3),\ ?a,\ ?d)$
  $\in alphabet_enlarge_delta\ M$
  unfolding *alphabet_enlarge_delta_def*
  using *rel_in gamma_a src_valid dst_valid* **by** *auto*
have *ts_unchanged*: $(\lambda k.\ (ts\ k)(n\ k := ?a\ k)) = ts$
  by (*rule ext*) *auto*
let *?c''* = $Config_M\ (q,\ ofs,\ ?buf',\ dest,\ SS3)\ ts$
  $(\lambda k.\ go_dir\ (?d\ k)\ (n\ k))$
have *step_sub_raw*:
  $(Config_M\ (q,\ ofs,\ buf,\ dest,\ SS2)\ ts\ n,$
  $Config_M\ (q,\ ofs,\ ?buf',\ dest,\ SS3)$
  $(\lambda k.\ (ts\ k)(n\ k := ?a\ k))$
  $(\lambda k.\ go_dir\ (?d\ k)\ (n\ k)))$
  $\in mttm_step\ (ae_delta_ss2_ss3\ M)$
proof (*rule mttm_step.intros*)
  show $((q,\ ofs,\ buf,\ dest,\ SS2),\ ?a,$
  $(q,\ ofs,\ ?buf',\ dest,\ SS3),\ ?a,\ ?d)$
  $\in ae_delta_ss2_ss3\ M$ **by** (*rule rel_in*)
qed
have *step_sub*: $(c',\ ?c'') \in mttm_step\ (ae_delta_ss2_ss3\ M)$
  using *step_sub_raw c'_eq ts_unchanged* **by** *simp*
have *step_full_raw*:
  $(Config_M\ (q,\ ofs,\ buf,\ dest,\ SS2)\ ts\ n,$
  $Config_M\ (q,\ ofs,\ ?buf',\ dest,\ SS3)$
  $(\lambda k.\ (ts\ k)(n\ k := ?a\ k))$
  $(\lambda k.\ go_dir\ (?d\ k)\ (n\ k)))$
  $\in mttm_step\ (alphabet_enlarge_delta\ M)$
proof (*rule mttm_step.intros*)

```
    show ((q, ofs, buf, dest, SS2), ?a,
          (q, ofs, ?buf', dest, SS3), ?a, ?d)
          ∈ alphabet_enlarge_delta M by (rule aed_in)
  qed
  have step_full: (c', ?c'') ∈ mttm_step (alphabet_enlarge_delta M)
    using step_full_raw c'_eq ts_unchanged by simp
  show ?thesis using step_sub step_full by blast
qed

lemma ae_step_ss3_ss4_exists:
  fixes M :: ('q, 'a) mttm
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
  assumes inv:      ae_inv_ss3 M c'
      and gamma:    ae_tape_in_gamma_block M c'
      and buf:      ae_buffer_in_gamma_block M c'
      and q_neq_t: fst (mt_state c') ≠ t_tm M
      and q_neq_r: fst (mt_state c') ≠ r_tm M
  shows ∃ c''. (c', c'') ∈ mttm_step (ae_delta_ss3_ss4 M)
               ∧ (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  obtain q ofs buf dest where
      state_eq: mt_state c' = (q, ofs, buf, dest, SS3)
      and q_in: q ∈ Q_tm M
    using inv unfolding ae_inv_ss3_def
    by (cases mt_state c') auto
  obtain ts n where c'_eq: c' = Config_M (q, ofs, buf, dest, SS3) ts n
    using state_eq by (cases c') auto
  let ?a = λk. ts k (n k)
  let ?d = λk. if k < k_tm M then dir.R else dir.N
  have q_neq_t': q ≠ t_tm M using q_neq_t state_eq by simp
  have q_neq_r': q ≠ r_tm M using q_neq_r state_eq by simp
  have a_tail: ∀ j≥k_tm M. ?a j = bl_block (bl_tm M)
    using gamma c'_eq unfolding ae_tape_in_gamma_block_def by simp
  have rel_in: ((q, ofs, buf, dest, SS3), ?a,
                (q, ofs, buf, dest, SS4), ?a, ?d)
                ∈ ae_delta_ss3_ss4 M
    unfolding ae_delta_ss3_ss4_def
    using q_in q_neq_t' q_neq_r' a_tail by auto
  have gamma_a: ∀ k. ?a k ∈ gamma_block (Γ_tm M)
    using gamma c'_eq unfolding ae_tape_in_gamma_block_def by simp
  have src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                    (snd ((q, ofs, buf, dest, SS3)
                          :: 'q × ('a, 'c) ae_stage))
    using buf state_eq
    unfolding ae_buffer_in_gamma_block_def ae_valid_stage_def by simp
  have dst_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                    (snd ((q, ofs, buf, dest, SS4)
                          :: 'q × ('a, 'c) ae_stage))
```

```
  by (rule ae_delta_ss3_ss4_dest_valid[OF rel_in src_valid])
 have aed_in: ((q, ofs, buf, dest, SS3), ?a,
               (q, ofs, buf, dest, SS4), ?a, ?d)
               ∈ alphabet_enlarge_delta M
  unfolding alphabet_enlarge_delta_def
  using rel_in gamma_a src_valid dst_valid by auto
 have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts
  by (rule ext) auto
 let ?c'' = Config_M (q, ofs, buf, dest, SS4) ts
                     (λk. go_dir (?d k) (n k))
 have step_sub_raw:
  (Config_M (q, ofs, buf, dest, SS3) ts n,
     Config_M (q, ofs, buf, dest, SS4)
       (λk. (ts k)(n k := ?a k))
       (λk. go_dir (?d k) (n k)))
     ∈ mttm_step (ae_delta_ss3_ss4 M)
 proof (rule mttm_step.intros)
  show ((q, ofs, buf, dest, SS3), ?a,
        (q, ofs, buf, dest, SS4), ?a, ?d)
        ∈ ae_delta_ss3_ss4 M by (rule rel_in)
 qed
 have step_sub: (c', ?c'') ∈ mttm_step (ae_delta_ss3_ss4 M)
  using step_sub_raw c'_eq ts_unchanged by simp
 have step_full_raw:
  (Config_M (q, ofs, buf, dest, SS3) ts n,
     Config_M (q, ofs, buf, dest, SS4)
       (λk. (ts k)(n k := ?a k))
       (λk. go_dir (?d k) (n k)))
     ∈ mttm_step (alphabet_enlarge_delta M)
 proof (rule mttm_step.intros)
  show ((q, ofs, buf, dest, SS3), ?a,
        (q, ofs, buf, dest, SS4), ?a, ?d)
        ∈ alphabet_enlarge_delta M by (rule aed_in)
 qed
 have step_full: (c', ?c'') ∈ mttm_step (alphabet_enlarge_delta M)
  using step_full_raw c'_eq ts_unchanged by simp
 show ?thesis using step_sub step_full by blast
qed
```

SS5→SS6 step-existence. The first of the write-back substeps' step-existence cluster (SS5→SS6, SS6→SS7, SS7→SS8) each writes a buffer cell back to the tape, so each needs *ae_buffer_in_gamma_block* to discharge the *alphabet_enlarge_delta* intersection guard's $a'\ k \in gamma_block$ obligation.

The SS5 action is total in its inputs (no halt-state precondition), so unlike the buffer-load substeps (SS1→SS4) the lemma does not need q_neq_t/q_neq_r hypotheses. Halt routing kicks in only at SS8→SS1.

```
lemma ae_step_ss5_ss6_exists:
 fixes M :: ('q, 'a) mttm
```

```
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
  assumes inv:   ae_inv_ss5 M c'
      and gamma: ae_tape_in_gamma_block M c'
      and buf:   ae_buffer_in_gamma_block M c'
  shows ∃ c''. (c', c'') ∈ mttm_step (ae_delta_ss5_ss6 M)
              ∧ (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  obtain q ofs buf dest where
      state_eq: mt_state c' = (q, ofs, buf, dest, SS5)
      and q_in: q ∈ Q_tm M
    using inv unfolding ae_inv_ss5_def
    by (cases mt_state c') auto
  obtain ts n where c'_eq: c' = Config_M (q, ofs, buf, dest, SS5) ts n
    using state_eq by (cases c') auto
  let ?a  = λk. ts k (n k)
  let ?a' = λk. if k < k_tm M
                then fst (ae_ss5_action (le_tm M) (?a k) (buf k) (dest k))
                else bl_block (bl_tm M)
  let ?d  = λk. if k < k_tm M
                then snd (ae_ss5_action (le_tm M) (?a k) (buf k) (dest k))
                else dir.N
  have a_tail: ∀ j≥k_tm M. ?a j = bl_block (bl_tm M)
    using gamma c'_eq unfolding ae_tape_in_gamma_block_def by simp
  have rel_in: ((q, ofs, buf, dest, SS5), ?a,
                (q, ofs, buf, dest, SS6), ?a', ?d)
                ∈ ae_delta_ss5_ss6 M
    unfolding ae_delta_ss5_ss6_def using q_in a_tail by auto
  have gamma_a: ∀ k. ?a k ∈ gamma_block (Γ_tm M)
    using gamma c'_eq unfolding ae_tape_in_gamma_block_def by simp
  have h_gamma: ∀ k. fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
    using buf state_eq unfolding ae_buffer_in_gamma_block_def by simp
  have act_gamma: ∀ k. fst (ae_ss5_action (le_tm M) (?a k) (buf k) (dest k))
                       ∈ gamma_block (Γ_tm M)
  proof (intro allI)
    fix k
    obtain l h r where buf_k: buf k = (l, h, r)
      by (cases buf k) auto
    have h_in_gamma: h ∈ gamma_block (Γ_tm M)
      using spec[OF h_gamma, of k] buf_k by simp
    have a_k_in_gamma: ?a k ∈ gamma_block (Γ_tm M)
      using spec[OF gamma_a, of k] by simp
    show fst (ae_ss5_action (le_tm M) (?a k) (buf k) (dest k))
          ∈ gamma_block (Γ_tm M)
    proof (cases ?a k = LE_block (le_tm M))
      case True
      thus ?thesis using a_k_in_gamma buf_k by simp
    next
      case a_neq_le: False
```

```
    show ?thesis
    proof (cases h = LE_block (le_tm M))
      case True
      thus ?thesis using a_neq_le a_k_in_gamma buf_k by simp
    next
      case False
      thus ?thesis using a_neq_le h_in_gamma buf_k by simp
    qed
  qed
qed
have gamma_a': ∀ k. ?a' k ∈ gamma_block (Γ_tm M)
proof (intro allI)
  fix k
  show ?a' k ∈ gamma_block (Γ_tm M)
  proof (cases k < k_tm M)
    case True thus ?thesis using spec[OF act_gamma, of k] by simp
  next
    case False
    hence ?a' k = ?a k using a_tail by simp
    thus ?thesis using spec[OF gamma_a, of k] by simp
  qed
qed
have act_le: ∀ k. fst (ae_ss5_action (le_tm M) (?a k) (buf k) (dest k))
                    = LE_block (le_tm M)
                  ⟶ ?a k = LE_block (le_tm M)
proof (intro allI impI)
  fix k
  assume a'_le: fst (ae_ss5_action (le_tm M) (?a k) (buf k) (dest k))
                  = LE_block (le_tm M)
  obtain l h r where buf_k: buf k = (l, h, r)
    by (cases buf k) auto
  show ?a k = LE_block (le_tm M)
  proof (cases ?a k = LE_block (le_tm M))
    case True thus ?thesis .
  next
    case a_neq_le: False
    show ?thesis
    proof (cases h = LE_block (le_tm M))
      case True
      hence fst (ae_ss5_action (le_tm M) (?a k) (buf k) (dest k)) = ?a k
        using a_neq_le buf_k by simp
      with a'_le a_neq_le show ?thesis by simp
    next
      case h_neq_le: False
      hence fst (ae_ss5_action (le_tm M) (?a k) (buf k) (dest k)) = h
        using a_neq_le buf_k by simp
      with a'_le h_neq_le show ?thesis by simp
    qed
  qed
```

qed
have *le_no_write*: $\forall k.\ ?a'\ k = LE_block\ (le_tm\ M)$
$\longrightarrow\ ?a\ k = LE_block\ (le_tm\ M)$
proof (*intro allI impI*)
fix k
assume *a'_le*: $?a'\ k = LE_block\ (le_tm\ M)$
show $?a\ k = LE_block\ (le_tm\ M)$
proof (*cases* $k < k_tm\ M$)
case *True*
hence *fst* (*ae_ss5_action* (*le_tm M*) (*?a k*) (*buf k*) (*dest k*))
$= LE_block\ (le_tm\ M)$
using *a'_le* **by** *simp*
thus *?thesis* **using** *spec*[*OF act_le*, *of k*] **by** *simp*
next
case *False*
hence $?a\ k = ?a'\ k$ **using** *a_tail* **by** *simp*
thus *?thesis* **using** *a'_le* **by** *simp*
qed
qed
have *src_valid*: *ae_valid_stage* (Γ_*tm M*) (*le_tm M*) (*k_tm M*)
(*snd* ((*q*, *ofs*, *buf*, *dest*, *SS5*)
:: $'q \times ('a, 'c)$ *ae_stage*))
using *buf state_eq*
unfolding *ae_buffer_in_gamma_block_def ae_valid_stage_def* **by** *simp*
have *dst_valid*: *ae_valid_stage* (Γ_*tm M*) (*le_tm M*) (*k_tm M*)
(*snd* ((*q*, *ofs*, *buf*, *dest*, *SS6*)
:: $'q \times ('a, 'c)$ *ae_stage*))
by (*rule ae_delta_ss5_ss6_dest_valid*[*OF rel_in src_valid*])
have *aed_in*: ((*q*, *ofs*, *buf*, *dest*, *SS5*), *?a*,
(*q*, *ofs*, *buf*, *dest*, *SS6*), *?a'*, *?d*)
∈ *alphabet_enlarge_delta M*
unfolding *alphabet_enlarge_delta_def*
using *rel_in gamma_a gamma_a' le_no_write src_valid dst_valid* **by** *auto*
have *ts_unchanged_when_N*:
$(\lambda k.\ (ts\ k)(n\ k := ?a\ k)) = ts$
by (*rule ext*) *auto*
let $?ts' = \lambda k.\ (ts\ k)(n\ k := ?a'\ k)$
let $?n' = \lambda k.\ go_dir\ (?d\ k)\ (n\ k)$
let $?c'' = Config_M\ (q, ofs, buf, dest, SS6)\ ?ts'\ ?n'$
have *step_sub_raw*:
($Config_M\ (q, ofs, buf, dest, SS5)\ ts\ n$,
$Config_M\ (q, ofs, buf, dest, SS6)\ ?ts'\ ?n'$)
∈ *mttm_step* (*ae_delta_ss5_ss6 M*)
proof (*rule mttm_step.intros*)
show ((*q*, *ofs*, *buf*, *dest*, *SS5*), *?a*,
(*q*, *ofs*, *buf*, *dest*, *SS6*), *?a'*, *?d*)
∈ *ae_delta_ss5_ss6 M* **by** (*rule rel_in*)
qed
have *step_sub*: $(c', ?c'') \in mttm_step\ (ae_delta_ss5_ss6\ M)$

```
    using step_sub_raw c'_eq by simp
  have step_full_raw:
    (Config_M (q, ofs, buf, dest, SS5) ts n,
       Config_M (q, ofs, buf, dest, SS6) ?ts' ?n')
       ∈ mttm_step (alphabet_enlarge_delta M)
  proof (rule mttm_step.intros)
    show ((q, ofs, buf, dest, SS5), ?a,
          (q, ofs, buf, dest, SS6), ?a', ?d)
          ∈ alphabet_enlarge_delta M by (rule aed_in)
  qed
  have step_full: (c', ?c'') ∈ mttm_step (alphabet_enlarge_delta M)
    using step_full_raw c'_eq by simp
  show ?thesis using step_sub step_full by blast
qed
```

Step-existence at SS6: writes the home block's previous contents in steady-state ($ds = AE_Left$ writes r; otherwise writes l); LE-stage branches return a idempotently. The l/r branches force the per-substep *ae_le_compat_ss6* hypothesis: l may legitimately equal *LE_block* when $p_start = 1$, so structural discharge fails and the chain proof must thread the position-buffer correspondence in.

```
lemma ae_step_ss6_ss7_exists:
  fixes M :: ('q, 'a) mttm
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
  assumes inv:        ae_inv_ss6 M c'
      and gamma:      ae_tape_in_gamma_block M c'
      and buf:        ae_buffer_in_gamma_block M c'
      and le_compat:  ae_le_compat_ss6 M c'
  shows ∃ c''. (c', c'') ∈ mttm_step (ae_delta_ss6_ss7 M)
               ∧ (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  obtain q ofs buf dest where
      state_eq: mt_state c' = (q, ofs, buf, dest, SS6)
      and q_in: q ∈ Q_tm M
    using inv unfolding ae_inv_ss6_def
    by (cases mt_state c') auto
  obtain ts n where c'_eq: c' = Config_M (q, ofs, buf, dest, SS6) ts n
    using state_eq by (cases c') auto
  let ?a  = λk. ts k (n k)
  let ?a' = λk. if k < k_tm M
                then fst (ae_ss6_action (le_tm M) (?a k) (buf k) (dest k))
                else bl_block (bl_tm M)
  let ?d  = λk. if k < k_tm M
                then snd (ae_ss6_action (le_tm M) (?a k) (buf k) (dest k))
                else dir.N
  have a_tail: ∀ j≥k_tm M. ?a j = bl_block (bl_tm M)
    using gamma c'_eq unfolding ae_tape_in_gamma_block_def by simp
  have rel_in: ((q, ofs, buf, dest, SS6), ?a,
```

```
            (q, ofs, buf, dest, SS7), ?a', ?d)
            ∈ ae_delta_ss6_ss7 M
  unfolding ae_delta_ss6_ss7_def using q_in a_tail by auto
have gamma_a: ∀ k. ?a k ∈ gamma_block (Γ_tm M)
  using gamma c'_eq unfolding ae_tape_in_gamma_block_def by simp
have buf_gamma:
    ∀ k. fst (buf k) ∈ gamma_block (Γ_tm M)
         ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
         ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
  using buf state_eq unfolding ae_buffer_in_gamma_block_def by simp
have act_gamma: ∀ k. fst (ae_ss6_action (le_tm M) (?a k) (buf k) (dest k))
                  ∈ gamma_block (Γ_tm M)
proof (intro allI)
  fix k
  obtain l h r where buf_k: buf k = (l, h, r)
    by (cases buf k) auto
  have l_in: l ∈ gamma_block (Γ_tm M)
    using spec[OF buf_gamma, of k] buf_k by simp
  have h_in: h ∈ gamma_block (Γ_tm M)
    using spec[OF buf_gamma, of k] buf_k by simp
  have r_in: r ∈ gamma_block (Γ_tm M)
    using spec[OF buf_gamma, of k] buf_k by simp
  have a_in: ?a k ∈ gamma_block (Γ_tm M)
    using spec[OF gamma_a, of k] by simp
  show fst (ae_ss6_action (le_tm M) (?a k) (buf k) (dest k))
         ∈ gamma_block (Γ_tm M)
  proof (cases ?a k = LE_block (le_tm M))
    case True thus ?thesis using a_in buf_k by simp
  next
    case a_neq_le: False
    show ?thesis
    proof (cases h = LE_block (le_tm M))
      case True thus ?thesis using a_neq_le a_in buf_k by simp
    next
      case h_neq_le: False
      show ?thesis
      proof (cases dest k = AE_Left)
        case True
        thus ?thesis using a_neq_le h_neq_le r_in buf_k by simp
      next
        case False
        thus ?thesis using a_neq_le h_neq_le l_in buf_k by simp
      qed
    qed
  qed
qed
have gamma_a': ∀ k. ?a' k ∈ gamma_block (Γ_tm M)
proof (intro allI)
  fix k
```

show *?a' k ∈ gamma_block (Γ_tm M)*
proof (*cases k < k_tm M*)
case *True* **thus** *?thesis* **using** *spec*[*OF act_gamma*, *of k*] **by** *simp*
next
case *False*
hence *?a' k = ?a k* **using** *a_tail* **by** *simp*
thus *?thesis* **using** *spec*[*OF gamma_a*, *of k*] **by** *simp*
qed
qed
have *act_le*: ∀ *k*<*k_tm M*. *fst* (*ae_ss6_action* (*le_tm M*) (*?a k*) (*buf k*) (*dest k*))
= *LE_block* (*le_tm M*)
⟶ *?a k* = *LE_block* (*le_tm M*)
using *le_compat c'_eq* **unfolding** *ae_le_compat_ss6_def* **by** *simp*
have *le_no_write*: ∀ *k*. *?a' k* = *LE_block* (*le_tm M*)
⟶ *?a k* = *LE_block* (*le_tm M*)
proof (*intro allI impI*)
fix *k*
assume *a'_le*: *?a' k* = *LE_block* (*le_tm M*)
show *?a k* = *LE_block* (*le_tm M*)
proof (*cases k < k_tm M*)
case *True*
hence *fst* (*ae_ss6_action* (*le_tm M*) (*?a k*) (*buf k*) (*dest k*))
= *LE_block* (*le_tm M*)
using *a'_le* **by** *simp*
thus *?thesis* **using** *act_le*[*rule_format*, *OF True*] **by** *simp*
next
case *False*
hence *?a k* = *?a' k* **using** *a_tail* **by** *simp*
thus *?thesis* **using** *a'_le* **by** *simp*
qed
qed
have *src_valid*: *ae_valid_stage* (*Γ_tm M*) (*le_tm M*) (*k_tm M*)
(*snd* ((*q*, *ofs*, *buf*, *dest*, *SS6*)
:: *'q* × (*'a*, *'c*) *ae_stage*))
using *buf state_eq*
unfolding *ae_buffer_in_gamma_block_def ae_valid_stage_def* **by** *simp*
have *dst_valid*: *ae_valid_stage* (*Γ_tm M*) (*le_tm M*) (*k_tm M*)
(*snd* ((*q*, *ofs*, *buf*, *dest*, *SS7*)
:: *'q* × (*'a*, *'c*) *ae_stage*))
by (*rule ae_delta_ss6_ss7_dest_valid*[*OF rel_in src_valid*])
have *aed_in*: ((*q*, *ofs*, *buf*, *dest*, *SS6*), *?a*,
(*q*, *ofs*, *buf*, *dest*, *SS7*), *?a'*, *?d*)
∈ *alphabet_enlarge_delta M*
unfolding *alphabet_enlarge_delta_def*
using *rel_in gamma_a gamma_a' le_no_write src_valid dst_valid* **by** *auto*
let *?ts'* = λ*k*. (*ts k*)(*n k* := *?a' k*)
let *?n'* = λ*k*. *go_dir* (*?d k*) (*n k*)
let *?c''* = *Config*$_M$ (*q*, *ofs*, *buf*, *dest*, *SS7*) *?ts' ?n'*

```
  have step_sub_raw:
    (Config_M (q, ofs, buf, dest, SS6) ts n,
       Config_M (q, ofs, buf, dest, SS7) ?ts' ?n')
       ∈ mttm_step (ae_delta_ss6_ss7 M)
  proof (rule mttm_step.intros)
    show ((q, ofs, buf, dest, SS6), ?a,
          (q, ofs, buf, dest, SS7), ?a', ?d)
          ∈ ae_delta_ss6_ss7 M by (rule rel_in)
  qed
  have step_sub: (c', ?c'') ∈ mttm_step (ae_delta_ss6_ss7 M)
    using step_sub_raw c'_eq by simp
  have step_full_raw:
    (Config_M (q, ofs, buf, dest, SS6) ts n,
       Config_M (q, ofs, buf, dest, SS7) ?ts' ?n')
       ∈ mttm_step (alphabet_enlarge_delta M)
  proof (rule mttm_step.intros)
    show ((q, ofs, buf, dest, SS6), ?a,
          (q, ofs, buf, dest, SS7), ?a', ?d)
          ∈ alphabet_enlarge_delta M by (rule aed_in)
  qed
  have step_full: (c', ?c'') ∈ mttm_step (alphabet_enlarge_delta M)
    using step_full_raw c'_eq by simp
  show ?thesis using step_sub step_full by blast
qed
```

Step-existence at SS7: idempotent home re-write in steady-state (returns a) or right write in LE-stage (returns r); the LE-stage branch needs *ae_le_compat_ss7*. Steady-state is structurally safe ($a' = a$); the LE-stage branch fires only when $h = LE_block$ and $a \neq LE_block$, returning r, where the chain proof must show $r = LE_block$ contradicts the position invariant.

```
lemma ae_step_ss7_ss8_exists:
  fixes M :: ('q, 'a) mttm
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
  assumes inv:        ae_inv_ss7 M c'
      and gamma:      ae_tape_in_gamma_block M c'
      and buf:        ae_buffer_in_gamma_block M c'
      and le_compat:  ae_le_compat_ss7 M c'
  shows ∃ c''. (c', c'') ∈ mttm_step (ae_delta_ss7_ss8 M)
               ∧ (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  obtain q ofs buf dest where
      state_eq: mt_state c' = (q, ofs, buf, dest, SS7)
      and q_in: q ∈ Q_tm M
    using inv unfolding ae_inv_ss7_def
    by (cases mt_state c') auto
  obtain ts n where c'_eq: c' = Config_M (q, ofs, buf, dest, SS7) ts n
    using state_eq by (cases c') auto
  let ?a = λk. ts k (n k)
```

let *?a′* = λ*k*. *if k* < *k_tm M*
then fst (*ae_ss7_action* (*le_tm M*) (*?a k*) (*buf k*) (*dest k*))
else bl_block (*bl_tm M*)
let *?d* = λ*k*. *if k* < *k_tm M*
then snd (*ae_ss7_action* (*le_tm M*) (*?a k*) (*buf k*) (*dest k*))
else dir.N
have *a_tail*: ∀ *j*≥*k_tm M*. *?a j* = *bl_block* (*bl_tm M*)
using *gamma c′_eq* **unfolding** *ae_tape_in_gamma_block_def* **by** *simp*
have *rel_in*: ((*q*, *ofs*, *buf*, *dest*, *SS7*), *?a*,
(*q*, *ofs*, *buf*, *dest*, *SS8*), *?a′*, *?d*)
∈ *ae_delta_ss7_ss8 M*
unfolding *ae_delta_ss7_ss8_def* **using** *q_in a_tail* **by** *auto*
have *gamma_a*: ∀ *k*. *?a k* ∈ *gamma_block* (Γ_*tm M*)
using *gamma c′_eq* **unfolding** *ae_tape_in_gamma_block_def* **by** *simp*
have *buf_gamma*:
∀ *k*. *fst* (*buf k*) ∈ *gamma_block* (Γ_*tm M*)
∧ *fst* (*snd* (*buf k*)) ∈ *gamma_block* (Γ_*tm M*)
∧ *snd* (*snd* (*buf k*)) ∈ *gamma_block* (Γ_*tm M*)
using *buf state_eq* **unfolding** *ae_buffer_in_gamma_block_def* **by** *simp*
have *act_gamma*: ∀ *k*. *fst* (*ae_ss7_action* (*le_tm M*) (*?a k*) (*buf k*) (*dest k*))
∈ *gamma_block* (Γ_*tm M*)
proof (*intro allI*)
fix *k*
obtain *l h r* **where** *buf_k*: *buf k* = (*l*, *h*, *r*)
by (*cases buf k*) *auto*
have *r_in*: *r* ∈ *gamma_block* (Γ_*tm M*)
using *spec*[*OF buf_gamma*, *of k*] *buf_k* **by** *simp*
have *a_in*: *?a k* ∈ *gamma_block* (Γ_*tm M*)
using *spec*[*OF gamma_a*, *of k*] **by** *simp*
show *fst* (*ae_ss7_action* (*le_tm M*) (*?a k*) (*buf k*) (*dest k*))
∈ *gamma_block* (Γ_*tm M*)
proof (*cases ?a k* = *LE_block* (*le_tm M*))
case *True* **thus** *?thesis* **using** *a_in buf_k* **by** *simp*
next
case *a_neq_le*: *False*
show *?thesis*
proof (*cases h* = *LE_block* (*le_tm M*))
case *True* **thus** *?thesis* **using** *a_neq_le r_in buf_k* **by** *simp*
next
case *h_neq_le*: *False*
thus *?thesis* **using** *a_neq_le a_in buf_k* **by** *simp*
qed
qed
qed
have *gamma_a′*: ∀ *k*. *?a′ k* ∈ *gamma_block* (Γ_*tm M*)
proof (*intro allI*)
fix *k*
show *?a′ k* ∈ *gamma_block* (Γ_*tm M*)
proof (*cases k* < *k_tm M*)

case *True* **thus** *?thesis* **using** *spec*[*OF act_gamma*, *of k*] **by** *simp*
next
case *False*
hence *?a' k* = *?a k* **using** *a_tail* **by** *simp*
thus *?thesis* **using** *spec*[*OF gamma_a*, *of k*] **by** *simp*
qed
qed
have *act_le*: ∀ *k*<*k_tm M*. *fst* (*ae_ss7_action* (*le_tm M*) (*?a k*) (*buf k*) (*dest k*))
= *LE_block* (*le_tm M*)
⟶ *?a k* = *LE_block* (*le_tm M*)
using *le_compat c'_eq* **unfolding** *ae_le_compat_ss7_def* **by** *simp*
have *le_no_write*: ∀ *k*. *?a' k* = *LE_block* (*le_tm M*)
⟶ *?a k* = *LE_block* (*le_tm M*)
proof (*intro allI impI*)
fix *k*
assume *a'_le*: *?a' k* = *LE_block* (*le_tm M*)
show *?a k* = *LE_block* (*le_tm M*)
proof (*cases k* < *k_tm M*)
case *True*
hence *fst* (*ae_ss7_action* (*le_tm M*) (*?a k*) (*buf k*) (*dest k*))
= *LE_block* (*le_tm M*)
using *a'_le* **by** *simp*
thus *?thesis* **using** *act_le*[*rule_format*, *OF True*] **by** *simp*
next
case *False*
hence *?a k* = *?a' k* **using** *a_tail* **by** *simp*
thus *?thesis* **using** *a'_le* **by** *simp*
qed
qed
have *src_valid*: *ae_valid_stage* (Γ*_tm M*) (*le_tm M*) (*k_tm M*)
(*snd* ((*q*, *ofs*, *buf*, *dest*, *SS7*)
:: *'q* × (*'a*, *'c*) *ae_stage*))
using *buf state_eq*
unfolding *ae_buffer_in_gamma_block_def ae_valid_stage_def* **by** *simp*
have *dst_valid*: *ae_valid_stage* (Γ*_tm M*) (*le_tm M*) (*k_tm M*)
(*snd* ((*q*, *ofs*, *buf*, *dest*, *SS8*)
:: *'q* × (*'a*, *'c*) *ae_stage*))
by (*rule ae_delta_ss7_ss8_dest_valid*[*OF rel_in src_valid*])
have *aed_in*: ((*q*, *ofs*, *buf*, *dest*, *SS7*), *?a*,
(*q*, *ofs*, *buf*, *dest*, *SS8*), *?a'*, *?d*)
∈ *alphabet_enlarge_delta M*
unfolding *alphabet_enlarge_delta_def*
using *rel_in gamma_a gamma_a' le_no_write src_valid dst_valid* **by** *auto*
let *?ts'* = λ*k*. (*ts k*)(*n k* := *?a' k*)
let *?n'* = λ*k*. *go_dir* (*?d k*) (*n k*)
let *?c''* = *Config*$_M$ (*q*, *ofs*, *buf*, *dest*, *SS8*) *?ts' ?n'*
have *step_sub_raw*:
(*Config*$_M$ (*q*, *ofs*, *buf*, *dest*, *SS7*) *ts n*,

```
      Config_M (q, ofs, buf, dest, SS8) ?ts' ?n')
      ∈ mttm_step (ae_delta_ss7_ss8 M)
  proof (rule mttm_step.intros)
    show ((q, ofs, buf, dest, SS7), ?a,
          (q, ofs, buf, dest, SS8), ?a', ?d)
          ∈ ae_delta_ss7_ss8 M by (rule rel_in)
  qed
  have step_sub: (c', ?c'') ∈ mttm_step (ae_delta_ss7_ss8 M)
    using step_sub_raw c'_eq by simp
  have step_full_raw:
    (Config_M (q, ofs, buf, dest, SS7) ts n,
      Config_M (q, ofs, buf, dest, SS8) ?ts' ?n')
      ∈ mttm_step (alphabet_enlarge_delta M)
  proof (rule mttm_step.intros)
    show ((q, ofs, buf, dest, SS7), ?a,
          (q, ofs, buf, dest, SS8), ?a', ?d)
          ∈ alphabet_enlarge_delta M by (rule aed_in)
  qed
  have step_full: (c', ?c'') ∈ mttm_step (alphabet_enlarge_delta M)
    using step_full_raw c'_eq by simp
  show ?thesis using step_sub step_full by blast
qed
```

Step-existence at SS8: end of the write-back phase. The action selects l, r, or a per stage-kind $\times$ dest; the next stage is determined by halt routing $q \in \{t, r\}$ routes to *init_stage le*, otherwise re-enters SS1 with buffer + offset preserved and dest reset to *init_dest*. Step-existence is uniform across the halt branch: both branches are total in the action's input. The chain-level forward-simulation argument splits halt-emission from cycle continuation downstream, but step-existence itself does not require a non-halting hypothesis.

```
lemma ae_step_ss8_ss1_exists:
  fixes M :: ('q, 'a) mttm
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
  assumes vM:          valid_mttm M
      and inv:         ae_inv_ss8 M c'
      and gamma:       ae_tape_in_gamma_block M c'
      and buf:         ae_buffer_in_gamma_block M c'
      and le_compat:   ae_le_compat_ss8 M c'
  shows ∃ c''. (c', c'') ∈ mttm_step (ae_delta_ss8_ss1 M)
              ∧ (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  obtain q ofs buf dest where
      state_eq: mt_state c' = (q, ofs, buf, dest, SS8)
      and q_in: q ∈ Q_tm M
    using inv unfolding ae_inv_ss8_def
    by (cases mt_state c') auto
  obtain ts n where c'_eq: c' = Config_M (q, ofs, buf, dest, SS8) ts n
```

```
  using state_eq by (cases c') auto
let ?a  = λk. ts k (n k)
let ?a' = λk. if k < k_tm M
              then fst (ae_ss8_action (le_tm M) (?a k) (buf k) (dest k))
              else bl_block (bl_tm M)
let ?d  = λk. if k < k_tm M
              then snd (ae_ss8_action (le_tm M) (?a k) (buf k) (dest k))
              else dir.N
let ?stage' = if q ∈ {t_tm M, r_tm M}
              then init_stage (le_tm M)
              else (ofs, buf, init_dest, SS1)
have a_tail: ∀ j≥k_tm M. ?a j = bl_block (bl_tm M)
  using gamma c'_eq unfolding ae_tape_in_gamma_block_def by simp
have rel_in: ((q, ofs, buf, dest, SS8), ?a,
              (q, ?stage'), ?a', ?d)
              ∈ ae_delta_ss8_ss1 M
  unfolding ae_delta_ss8_ss1_def init_stage_def
  using q_in a_tail by auto
have gamma_a: ∀ k. ?a k ∈ gamma_block (Γ_tm M)
  using gamma c'_eq unfolding ae_tape_in_gamma_block_def by simp
have buf_gamma:
   ∀ k. fst (buf k) ∈ gamma_block (Γ_tm M)
        ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
        ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
  using buf state_eq unfolding ae_buffer_in_gamma_block_def by simp
have act_gamma: ∀ k. fst (ae_ss8_action (le_tm M) (?a k) (buf k) (dest k))
                     ∈ gamma_block (Γ_tm M)
proof (intro allI)
  fix k
  obtain l h r where buf_k: buf k = (l, h, r)
    by (cases buf k) auto
  have l_in: l ∈ gamma_block (Γ_tm M)
    using spec[OF buf_gamma, of k] buf_k by simp
  have r_in: r ∈ gamma_block (Γ_tm M)
    using spec[OF buf_gamma, of k] buf_k by simp
  have a_in: ?a k ∈ gamma_block (Γ_tm M)
    using spec[OF gamma_a, of k] by simp
  show fst (ae_ss8_action (le_tm M) (?a k) (buf k) (dest k))
        ∈ gamma_block (Γ_tm M)
  proof (cases ?a k = LE_block (le_tm M))
    case True thus ?thesis using a_in buf_k by simp
  next
    case a_neq_le: False
    show ?thesis
    proof (cases h = LE_block (le_tm M))
      case True
      show ?thesis
      proof (cases dest k = AE_Right)
        case True thus ?thesis
```

```
        using a_neq_le ‹h = LE_block (le_tm M)› r_in buf_k by simp
      next
        case False thus ?thesis
          using a_neq_le ‹h = LE_block (le_tm M)› a_in buf_k by simp
      qed
    next
      case h_neq_le: False
      show ?thesis
      proof (cases dest k)
        case AE_Left thus ?thesis
          using a_neq_le h_neq_le l_in buf_k by simp
      next
        case AE_Home thus ?thesis
          using a_neq_le h_neq_le r_in buf_k by simp
      next
        case AE_Right thus ?thesis
          using a_neq_le h_neq_le r_in buf_k by simp
      qed
    qed
  qed
  have gamma_a': ∀k. ?a' k ∈ gamma_block (Γ_tm M)
  proof (intro allI)
    fix k
    show ?a' k ∈ gamma_block (Γ_tm M)
    proof (cases k < k_tm M)
      case True thus ?thesis using spec[OF act_gamma, of k] by simp
    next
      case False
      hence ?a' k = ?a k using a_tail by simp
      thus ?thesis using spec[OF gamma_a, of k] by simp
    qed
  qed
  have act_le: ∀k<k_tm M. fst (ae_ss8_action (le_tm M) (?a k) (buf k) (dest
k))
                    = LE_block (le_tm M)
                  ⟶ ?a k = LE_block (le_tm M)
    using le_compat c'_eq unfolding ae_le_compat_ss8_def by simp
  have le_no_write: ∀k. ?a' k = LE_block (le_tm M)
                        ⟶ ?a k = LE_block (le_tm M)
  proof (intro allI impI)
    fix k
    assume a'_le: ?a' k = LE_block (le_tm M)
    show ?a k = LE_block (le_tm M)
    proof (cases k < k_tm M)
      case True
      hence fst (ae_ss8_action (le_tm M) (?a k) (buf k) (dest k))
            = LE_block (le_tm M)
        using a'_le by simp
```

```
    thus ?thesis using act_le[rule_format, OF True] by simp
  next
    case False
    hence ?a k = ?a' k using a_tail by simp
    thus ?thesis using a'_le by simp
  qed
qed
have src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                  (snd ((q, ofs, buf, dest, SS8)
                        :: 'q × ('a, 'c) ae_stage))
  using buf state_eq
  unfolding ae_buffer_in_gamma_block_def ae_valid_stage_def by simp
have dst_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                  (snd ((q, ?stage')
                        :: 'q × ('a, 'c) ae_stage))
  by (rule ae_delta_ss8_ss1_dest_valid[OF vM rel_in src_valid])
have aed_in: ((q, ofs, buf, dest, SS8), ?a,
              (q, ?stage'), ?a', ?d)
              ∈ alphabet_enlarge_delta M
  unfolding alphabet_enlarge_delta_def
  using rel_in gamma_a gamma_a' le_no_write src_valid dst_valid by auto
let ?ts' = λk. (ts k)(n k := ?a' k)
let ?n' = λk. go_dir (?d k) (n k)
let ?c'' = Config_M (q, ?stage') ?ts' ?n'
have step_sub_raw:
  (Config_M (q, ofs, buf, dest, SS8) ts n,
     Config_M (q, ?stage') ?ts' ?n')
     ∈ mttm_step (ae_delta_ss8_ss1 M)
proof (rule mttm_step.intros)
  show ((q, ofs, buf, dest, SS8), ?a,
        (q, ?stage'), ?a', ?d)
        ∈ ae_delta_ss8_ss1 M by (rule rel_in)
qed
have step_sub: (c', ?c'') ∈ mttm_step (ae_delta_ss8_ss1 M)
  using step_sub_raw c'_eq by simp
have step_full_raw:
  (Config_M (q, ofs, buf, dest, SS8) ts n,
     Config_M (q, ?stage') ?ts' ?n')
     ∈ mttm_step (alphabet_enlarge_delta M)
proof (rule mttm_step.intros)
  show ((q, ofs, buf, dest, SS8), ?a,
        (q, ?stage'), ?a', ?d)
        ∈ alphabet_enlarge_delta M by (rule aed_in)
qed
have step_full: (c', ?c'') ∈ mttm_step (alphabet_enlarge_delta M)
  using step_full_raw c'_eq by simp
show ?thesis using step_sub step_full by blast
qed
```

2.12.10 Substrate and encoder helpers

Substrate-derived helpers used by the validation lemmas.

lemma *bl_tm_notin_Sigma_tm*:
assumes *valid_mttm M*
shows *bl_tm M* $\notin$ *Sigma_tm M*
by (*rule valid_mttm_blank_not_Sigma*[*OF assms*])

lemma *s_tm_in_Q_tm*:
assumes *valid_mttm M*
shows *s_tm M* $\in$ *Q_tm M*
by (*rule valid_mttm_s_in_Q*[*OF assms*])

Encoder structural helpers. *length_encode_input* and *encode_input_nth* unfold the encoder's definition into the shape that downstream proofs about block content cite.

lemma *length_encode_input*:
length (*encode_input bl u* :: (*'c* :: *enum* $\Rightarrow$ *'a*) *list*)
= (*length u* + *card* (*UNIV* :: *'c set*) − *1*) *div card* (*UNIV* :: *'c set*)
by (*simp add*: *encode_input_def Let_def*)

lemma *encode_input_nth*:
fixes *u* :: *'a list*
and *x* :: *'c* :: *enum*
assumes *i* < *length* (*encode_input bl u* :: (*'c* $\Rightarrow$ *'a*) *list*)
shows ((*encode_input bl u* :: (*'c* $\Rightarrow$ *'a*) *list*) ! *i*) *x*
= (*let j* = *i* ∗ *card* (*UNIV* :: *'c set*) + *c_idx x*
in if j < *length u then u* ! *j else bl*)
proof −
let *?c* = *card* (*UNIV* :: *'c set*)
have *len_eq*: *length* (*encode_input bl u* :: (*'c* $\Rightarrow$ *'a*) *list*)
= (*length u* + *?c* − *1*) *div ?c*
by (*simp add*: *encode_input_def Let_def*)
hence *i_lt*: *i* < (*length u* + *?c* − *1*) *div ?c* **using** *assms* **by** *simp*
show *?thesis*
unfolding *encode_input_def Let_def*
using *i_lt* **by** *simp*
qed

Index range for *c_idx*: every element of *'c* :: *enum* has an index strictly less than the length of the canonical enumeration, and the enumeration retrieves that element back.

lemma *c_idx_in_range*:
fixes *x* :: *'c* :: *enum*
shows *c_idx x* < *length* (*enum_class.enum* :: *'c list*)
and (*enum_class.enum* :: *'c list*) ! *c_idx x* = *x*
proof −
have *ex_unique*: ∃!*i*. *i* < *length* (*enum_class.enum* :: *'c list*)
∧ (*enum_class.enum* :: *'c list*) ! *i* = *x*

```
  proof −
    have x_in: x ∈ set (enum_class.enum :: 'c list)
      using enum_class.UNIV_enum by blast
    then obtain i where
      i_lt: i < length (enum_class.enum :: 'c list) and
      i_eq: (enum_class.enum :: 'c list) ! i = x
      by (auto simp: in_set_conv_nth)
    moreover have ⋀j. j < length (enum_class.enum :: 'c list)
                  ⟹ (enum_class.enum :: 'c list) ! j = x
                  ⟹ j = i
      using i_lt i_eq enum_class.enum_distinct
      by (metis nth_eq_iff_index_eq)
    ultimately show ?thesis by blast
  qed
  have idx_eq: c_idx x < length (enum_class.enum :: 'c list)
               ∧ (enum_class.enum :: 'c list) ! c_idx x = x
    unfolding c_idx_def using theI'[OF ex_unique] .
  show c_idx x < length (enum_class.enum :: 'c list) using idx_eq ..
  show (enum_class.enum :: 'c list) ! c_idx x = x using idx_eq ..
qed

lemma c_idx_lt_card: c_idx (x :: 'c :: enum) < card (UNIV :: 'c set)
proof −
  have card (UNIV :: 'c set) = length (enum_class.enum :: 'c list)
    using enum_class.UNIV_enum enum_class.enum_distinct
    by (metis distinct_card length_remdups_card_conv set_remdups)
  thus ?thesis using c_idx_in_range(1) by simp
qed

lemma card_eq_length_enum:
  card (UNIV :: 'c :: enum set) = length (enum_class.enum :: 'c list)
  using enum_class.UNIV_enum enum_class.enum_distinct
  by (metis distinct_card length_remdups_card_conv set_remdups)

lemma c_idx_enum_nth:
  fixes i :: nat
  assumes i < length (enum_class.enum :: 'c :: enum list)
  shows c_idx ((enum_class.enum :: 'c list) ! i) = i
proof −
  let ?xs = enum_class.enum :: 'c list
  have nth_eq: ?xs ! c_idx (?xs ! i) = ?xs ! i
    using c_idx_in_range(2) .
  have lt: c_idx (?xs ! i) < length ?xs
    by (rule c_idx_in_range(1))
  show ?thesis
    using nth_eq lt assms enum_class.enum_distinct
    by (metis nth_eq_iff_index_eq)
qed
```

end
theory *AlphabetEnlargement_Uniqueness*
imports *AlphabetEnlargement_Simulation*
begin

2.13 Block-predicate identities (VFwd exclusion support)

Identities relating *is_pure_block*, *is_padded_block*, *LE_block*, and *bl_block*. Used by the VFwd 6-fold pairwise exclusion lemmas below (and downstream by the reverse-arm trace decoder). Located here because the proofs need *c_idx_enum_nth* to construct index witnesses inside the existential in *is_padded_block*.

lemma *is_pure_not_padded*:
fixes $f :: 'c :: enum \Rightarrow 'a$
shows *is_pure_block bl f* $\Longrightarrow \neg$ *is_padded_block bl f*
unfolding *is_pure_block_def is_padded_block_def*
proof *clarify*
fix $k :: nat$
assume *pure*: $\forall x :: 'c.\ f\ x \neq bl$
and *k_lb*: $1 \leq k$
and *k_ub*: $k <$ *length* (*enum_class.enum* $:: 'c$ *list*)
and *suffix*: $\forall x :: 'c.\ k \leq$ *c_idx x* $\longrightarrow f\ x = bl$
let *?x* = (*enum_class.enum* $:: 'c$ *list*) ! *k*
have *c_idx ?x* = *k* **using** *c_idx_enum_nth*[*OF k_ub*] **.**
hence $k \leq$ *c_idx ?x* **by** *simp*
hence *f ?x* = *bl* **using** *suffix* **by** *blast*
thus *False* **using** *pure* **by** *blast*
qed

lemma *not_is_pure_bl_block*:
$\neg$ *is_pure_block* (*bl* $:: 'a$) (*bl_block bl* $:: 'c :: enum \Rightarrow 'a$)
unfolding *is_pure_block_def bl_block_def* **by** *simp*

lemma *not_is_padded_bl_block*:
$\neg$ *is_padded_block* (*bl* $:: 'a$) (*bl_block bl* $:: 'c :: enum \Rightarrow 'a$)
unfolding *is_padded_block_def bl_block_def*
proof *clarify*
fix $k :: nat$
assume *k_lb*: $1 \leq k$
and *k_ub*: $k <$ *length* (*enum_class.enum* $:: 'c$ *list*)
and *prefix*: $\forall x :: 'c.$ *c_idx x* $< k \longrightarrow (\lambda_.\ bl)\ x \neq bl$
have *len_pos*: $0 <$ *length* (*enum_class.enum* $:: 'c$ *list*) **using** *k_lb k_ub* **by** *linarith*
let *?x* = (*enum_class.enum* $:: 'c$ *list*) ! *0*
have *c_idx ?x* = *0* **using** *c_idx_enum_nth*[*OF len_pos*] **.**
hence *c_idx ?x* $< k$ **using** *k_lb* **by** *linarith*
hence $(\lambda_ :: 'c.\ bl)$ *?x* $\neq bl$ **using** *prefix* **by** *blast*
thus *False* **by** *simp*

qed

lemma *is_pure_LE_block*:
  (*le* :: *'a*) ≠ *bl* ⟹
    is_pure_block bl (*LE_block le* :: *'c* :: *enum* ⇒ *'a*)
  unfolding *is_pure_block_def LE_block_def* **by** *simp*

lemma *not_is_padded_LE_block*:
  (*le* :: *'a*) ≠ *bl* ⟹
    ¬ *is_padded_block bl* (*LE_block le* :: *'c* :: *enum* ⇒ *'a*)
  using *is_pure_LE_block is_pure_not_padded* **by** *metis*

lemma *LE_block_eq_bl_block_imp_eq*:
  fixes *le bl* :: *'a*
  assumes (*LE_block le* :: *'c* :: *enum* ⇒ *'a*) = *bl_block bl*
  shows *le* = *bl*
proof −
  have *len_pos*: *0* < *length* (*enum_class.enum* :: *'c list*)
    using *c_idx_in_range*(*1*)[*of c_first* :: *'c*] **by** *linarith*
  let *?x* = (*enum_class.enum* :: *'c list*) ! *0*
  from *assms* **have** (*LE_block le* :: *'c* ⇒ *'a*) *?x* = *bl_block bl ?x* **by** *simp*
  thus *?thesis* **unfolding** *LE_block_def bl_block_def* **by** *simp*
qed

2.14 VFwd 6-fold within-phase exclusion

The four VFwd-source relations (advance, *to_padded*, *to_ret*, reject) are pairwise disjoint under *le_neq_bl*.

lemma *ae_delta_val_fwd_advance_to_padded_disjoint*:
  fixes *M* :: (*'q*, *'a*) *mttm*
    and *a* :: *nat* ⇒ *'c* :: *enum* ⇒ *'a*
  assumes *le_tm M* ≠ *bl_tm M*
      and (*s*, *a*, *s1*, *a1*, *d1*) ∈ *ae_delta_val_fwd_advance M*
      and (*s*, *a*, *s2*, *a2*, *d2*) ∈ *ae_delta_val_fwd_to_padded M*
  shows *False*
proof −
  have *alt*: *a 0* = *LE_block* (*le_tm M*)
            ∨ *is_pure_block* (*bl_tm M*) (*a 0*)
    using *assms*(*2*) **unfolding** *ae_delta_val_fwd_advance_def* **by** *auto*
  have *padded*: *is_padded_block* (*bl_tm M*) (*a 0*)
    using *assms*(*3*) **unfolding** *ae_delta_val_fwd_to_padded_def* **by** *auto*
  from *alt* **show** *False*
  proof
    assume *a 0* = *LE_block* (*le_tm M*)
    thus *False* **using** *padded assms*(*1*) *not_is_padded_LE_block* **by** *metis*
  next
    assume *is_pure_block* (*bl_tm M*) (*a 0*)
    thus *False* **using** *padded is_pure_not_padded* **by** *metis*
  qed

qed

lemma *ae_delta_val_fwd_advance_to_ret_disjoint*:
fixes $M :: ('q, 'a)$ *mttm*
and $a :: nat \Rightarrow 'c :: enum \Rightarrow 'a$
assumes *le_tm M* $\neq$ *bl_tm M*
and $(s, a, s1, a1, d1) \in$ *ae_delta_val_fwd_advance M*
and $(s, a, s2, a2, d2) \in$ *ae_delta_val_fwd_to_ret M*
shows *False*
proof −
have *alt*: *a 0* = *LE_block* (*le_tm M*)
$\vee$ *is_pure_block* (*bl_tm M*) (*a 0*)
using *assms*(*2*) **unfolding** *ae_delta_val_fwd_advance_def* **by** *auto*
have *to_ret*: *a 0* = *bl_block* (*bl_tm M*)
using *assms*(*3*) **unfolding** *ae_delta_val_fwd_to_ret_def* **by** *auto*
from *alt* **show** *False*
proof
assume *a 0* = *LE_block* (*le_tm M*)
with *to_ret* **have** *eq*: (*LE_block* (*le_tm M*) :: $'c \Rightarrow 'a$) = *bl_block* (*bl_tm M*)
by *simp*
hence *le_tm M* = *bl_tm M* **by** (*rule LE_block_eq_bl_block_imp_eq*)
thus *False* **using** *assms*(*1*) **by** *simp*
next
assume *is_pure_block* (*bl_tm M*) (*a 0*)
with *to_ret* **have** *is_pure_block* (*bl_tm M*) (*bl_block* (*bl_tm M*) :: $'c \Rightarrow 'a$)
by *simp*
thus *False* **using** *not_is_pure_bl_block*[**where** $'c='c$ **and** *bl*=*bl_tm M*] **by**
simp
qed
qed

lemma *ae_delta_val_fwd_advance_reject_disjoint*:
assumes $(s, a, s1, a1, d1) \in$ *ae_delta_val_fwd_advance M*
and $(s, a, s2, a2, d2) \in$ *ae_delta_val_fwd_reject M*
shows *False*
using *assms*
unfolding *ae_delta_val_fwd_advance_def ae_delta_val_fwd_reject_def*
is_canonical_block_def
by *auto*

lemma *ae_delta_val_fwd_to_padded_to_ret_disjoint*:
fixes $M :: ('q, 'a)$ *mttm*
and $a :: nat \Rightarrow 'c :: enum \Rightarrow 'a$
assumes $(s, a, s1, a1, d1) \in$ *ae_delta_val_fwd_to_padded M*
and $(s, a, s2, a2, d2) \in$ *ae_delta_val_fwd_to_ret M*
shows *False*
proof −
have *padded*: *is_padded_block* (*bl_tm M*) (*a 0*)
using *assms*(*1*) **unfolding** *ae_delta_val_fwd_to_padded_def* **by** *auto*

```
  have to_ret: a 0 = bl_block (bl_tm M)
    using assms(2) unfolding ae_delta_val_fwd_to_ret_def by auto
  from padded to_ret
    have is_padded_block (bl_tm M) (bl_block (bl_tm M) :: 'c ⇒ 'a) by simp
  thus False using not_is_padded_bl_block[where 'c='c and bl=bl_tm M] by
simp
qed

lemma ae_delta_val_fwd_to_padded_reject_disjoint:
  assumes (s, a, s1, a1, d1) ∈ ae_delta_val_fwd_to_padded M
      and (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_reject M
  shows False
  using assms
  unfolding ae_delta_val_fwd_to_padded_def ae_delta_val_fwd_reject_def
          is_canonical_block_def
  by auto

lemma ae_delta_val_fwd_to_ret_reject_disjoint:
  assumes (s, a, s1, a1, d1) ∈ ae_delta_val_fwd_to_ret M
      and (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_reject M
  shows False
  using assms
  unfolding ae_delta_val_fwd_to_ret_def ae_delta_val_fwd_reject_def
  by auto
```

V-marker dispatch helpers: package the within-cluster multi-relation disjunction + pairwise disjointness + per-relation functionality into a single *_functional* lemma per V marker. Used by *alphabet_enlarge_delta_functional*'s V cases.

```
lemma ae_delta_VFwd_functional:
  fixes M :: ('q, 'a) mttm
  assumes le_neq_bl: le_tm M ≠ bl_tm M
      and h1: (s, a, s1, a1, d1) ∈ alphabet_enlarge_delta M
      and h2: (s, a, s2, a2, d2) ∈ alphabet_enlarge_delta M
      and idx: snd (snd (snd (snd s))) = VFwd
  shows s1 = s2 ∧ a1 = a2 ∧ d1 = d2
proof −
  have alt1: (s, a, s1, a1, d1) ∈ ae_delta_val_fwd_advance M
             ∨ (s, a, s1, a1, d1) ∈ ae_delta_val_fwd_to_padded M
             ∨ (s, a, s1, a1, d1) ∈ ae_delta_val_fwd_reject M
             ∨ (s, a, s1, a1, d1) ∈ ae_delta_val_fwd_to_ret M
    using h1 idx
    unfolding alphabet_enlarge_delta_def
            ae_delta_val_pad_to_ret_def
            ae_delta_val_pad_reject_def
            ae_delta_val_ret_step_def
            ae_delta_val_ret_to_sim_def
            ae_delta_ss1_ss2_def
            ae_delta_ss2_ss3_def
```

```
        ae_delta_ss3_ss4_def
        ae_delta_ss4_ss5_def
        ae_delta_ss5_ss6_def
        ae_delta_ss6_ss7_def
        ae_delta_ss7_ss8_def
        ae_delta_ss8_ss1_def
  by auto
 have alt2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_advance M
          ∨ (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_to_padded M
          ∨ (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_reject M
          ∨ (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_to_ret M
  using h2 idx
  unfolding alphabet_enlarge_delta_def
        ae_delta_val_pad_to_ret_def
        ae_delta_val_pad_reject_def
        ae_delta_val_ret_step_def
        ae_delta_val_ret_to_sim_def
        ae_delta_ss1_ss2_def
        ae_delta_ss2_ss3_def
        ae_delta_ss3_ss4_def
        ae_delta_ss4_ss5_def
        ae_delta_ss5_ss6_def
        ae_delta_ss6_ss7_def
        ae_delta_ss7_ss8_def
        ae_delta_ss8_ss1_def
  by auto
 from alt1 show ?thesis
 proof (elim disjE)
  assume A1: (s, a, s1, a1, d1) ∈ ae_delta_val_fwd_advance M
  from alt2 show ?thesis
  proof (elim disjE)
   assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_advance M
    show ?thesis using ae_delta_val_fwd_advance_functional[OF A1 A2] by
simp
  next
   assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_to_padded M
      have False using ae_delta_val_fwd_advance_to_padded_disjoint[OF
le_neq_bl A1 A2] .
    thus ?thesis ..
  next
   assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_reject M
   have False using ae_delta_val_fwd_advance_reject_disjoint[OF A1 A2] .
   thus ?thesis ..
  next
   assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_to_ret M
   have False using ae_delta_val_fwd_advance_to_ret_disjoint[OF le_neq_bl
A1 A2] .
   thus ?thesis ..
  qed
```

```
  next
    assume A1: (s, a, s1, a1, d1) ∈ ae_delta_val_fwd_to_padded M
    from alt2 show ?thesis
    proof (elim disjE)
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_advance M
          have False using ae_delta_val_fwd_advance_to_padded_disjoint[OF
le_neq_bl A2 A1] .
      thus ?thesis ..
    next
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_to_padded M
     show ?thesis using ae_delta_val_fwd_to_padded_functional[OF A1 A2] by
simp
    next
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_reject M
      have False using ae_delta_val_fwd_to_padded_reject_disjoint[OF A1 A2]
.
      thus ?thesis ..
    next
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_to_ret M
     have False using ae_delta_val_fwd_to_padded_to_ret_disjoint[OF A1 A2]
.
      thus ?thesis ..
    qed
  next
    assume A1: (s, a, s1, a1, d1) ∈ ae_delta_val_fwd_reject M
    from alt2 show ?thesis
    proof (elim disjE)
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_advance M
      have False using ae_delta_val_fwd_advance_reject_disjoint[OF A2 A1] .
      thus ?thesis ..
    next
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_to_padded M
      have False using ae_delta_val_fwd_to_padded_reject_disjoint[OF A2 A1]
.
      thus ?thesis ..
    next
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_reject M
     show ?thesis using ae_delta_val_fwd_reject_functional[OF A1 A2] by simp
    next
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_to_ret M
      have False using ae_delta_val_fwd_to_ret_reject_disjoint[OF A2 A1] .
      thus ?thesis ..
    qed
  next
    assume A1: (s, a, s1, a1, d1) ∈ ae_delta_val_fwd_to_ret M
    from alt2 show ?thesis
    proof (elim disjE)
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_advance M
     have False using ae_delta_val_fwd_advance_to_ret_disjoint[OF le_neq_bl
```

```
A2 A1] .
      thus ?thesis ..
    next
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_to_padded M
     have False using ae_delta_val_fwd_to_padded_to_ret_disjoint[OF A2 A1]
.
      thus ?thesis ..
    next
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_reject M
      have False using ae_delta_val_fwd_to_ret_reject_disjoint[OF A1 A2] .
      thus ?thesis ..
    next
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_fwd_to_ret M
       show ?thesis using ae_delta_val_fwd_to_ret_functional[OF A1 A2] by
simp
    qed
  qed
qed

lemma ae_delta_VRet_functional:
  fixes M :: ('q, 'a) mttm
  assumes h1: (s, a, s1, a1, d1) ∈ alphabet_enlarge_delta M
      and h2: (s, a, s2, a2, d2) ∈ alphabet_enlarge_delta M
      and idx: snd (snd (snd (snd s))) = VRet
  shows s1 = s2 ∧ a1 = a2 ∧ d1 = d2
proof −
  have alt1: (s, a, s1, a1, d1) ∈ ae_delta_val_ret_step M
               ∨ (s, a, s1, a1, d1) ∈ ae_delta_val_ret_to_sim M
    using h1 idx
    unfolding alphabet_enlarge_delta_def
              ae_delta_val_fwd_advance_def
              ae_delta_val_fwd_to_padded_def
              ae_delta_val_fwd_reject_def
              ae_delta_val_fwd_to_ret_def
              ae_delta_val_pad_to_ret_def
              ae_delta_val_pad_reject_def
              ae_delta_ss1_ss2_def
              ae_delta_ss2_ss3_def
              ae_delta_ss3_ss4_def
              ae_delta_ss4_ss5_def
              ae_delta_ss5_ss6_def
              ae_delta_ss6_ss7_def
              ae_delta_ss7_ss8_def
              ae_delta_ss8_ss1_def
    by auto
  have alt2: (s, a, s2, a2, d2) ∈ ae_delta_val_ret_step M
               ∨ (s, a, s2, a2, d2) ∈ ae_delta_val_ret_to_sim M
    using h2 idx
    unfolding alphabet_enlarge_delta_def
```

```
            ae_delta_val_fwd_advance_def
            ae_delta_val_fwd_to_padded_def
            ae_delta_val_fwd_reject_def
            ae_delta_val_fwd_to_ret_def
            ae_delta_val_pad_to_ret_def
            ae_delta_val_pad_reject_def
            ae_delta_ss1_ss2_def
            ae_delta_ss2_ss3_def
            ae_delta_ss3_ss4_def
            ae_delta_ss4_ss5_def
            ae_delta_ss5_ss6_def
            ae_delta_ss6_ss7_def
            ae_delta_ss7_ss8_def
            ae_delta_ss8_ss1_def
    by auto
  from alt1 show ?thesis
  proof
    assume A1: (s, a, s1, a1, d1) ∈ ae_delta_val_ret_step M
    from alt2 show ?thesis
    proof
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_ret_step M
      show ?thesis using ae_delta_val_ret_step_functional[OF A1 A2] by simp
    next
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_ret_to_sim M
      have False using ae_delta_val_ret_step_ret_to_sim_disjoint[OF A1 A2] .
      thus ?thesis ..
    qed
  next
    assume A1: (s, a, s1, a1, d1) ∈ ae_delta_val_ret_to_sim M
    from alt2 show ?thesis
    proof
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_ret_step M
      have False using ae_delta_val_ret_step_ret_to_sim_disjoint[OF A2 A1] .
      thus ?thesis ..
    next
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_ret_to_sim M
       show ?thesis using ae_delta_val_ret_to_sim_functional[OF A1 A2] by
simp
    qed
  qed
qed

lemma ae_delta_VFwdPad_functional:
  fixes M :: ('q, 'a) mttm
  assumes h1: (s, a, s1, a1, d1) ∈ alphabet_enlarge_delta M
      and h2: (s, a, s2, a2, d2) ∈ alphabet_enlarge_delta M
      and idx: snd (snd (snd (snd s))) = VFwdPad
  shows s1 = s2 ∧ a1 = a2 ∧ d1 = d2
proof −
```

```
  have alt1: (s, a, s1, a1, d1) ∈ ae_delta_val_pad_to_ret M
             ∨ (s, a, s1, a1, d1) ∈ ae_delta_val_pad_reject M
    using h1 idx
    unfolding alphabet_enlarge_delta_def
              ae_delta_val_fwd_advance_def
              ae_delta_val_fwd_to_padded_def
              ae_delta_val_fwd_reject_def
              ae_delta_val_fwd_to_ret_def
              ae_delta_val_ret_step_def
              ae_delta_val_ret_to_sim_def
              ae_delta_ss1_ss2_def
              ae_delta_ss2_ss3_def
              ae_delta_ss3_ss4_def
              ae_delta_ss4_ss5_def
              ae_delta_ss5_ss6_def
              ae_delta_ss6_ss7_def
              ae_delta_ss7_ss8_def
              ae_delta_ss8_ss1_def
    by auto
  have alt2: (s, a, s2, a2, d2) ∈ ae_delta_val_pad_to_ret M
             ∨ (s, a, s2, a2, d2) ∈ ae_delta_val_pad_reject M
    using h2 idx
    unfolding alphabet_enlarge_delta_def
              ae_delta_val_fwd_advance_def
              ae_delta_val_fwd_to_padded_def
              ae_delta_val_fwd_reject_def
              ae_delta_val_fwd_to_ret_def
              ae_delta_val_ret_step_def
              ae_delta_val_ret_to_sim_def
              ae_delta_ss1_ss2_def
              ae_delta_ss2_ss3_def
              ae_delta_ss3_ss4_def
              ae_delta_ss4_ss5_def
              ae_delta_ss5_ss6_def
              ae_delta_ss6_ss7_def
              ae_delta_ss7_ss8_def
              ae_delta_ss8_ss1_def
    by auto
  from alt1 show ?thesis
  proof
    assume A1: (s, a, s1, a1, d1) ∈ ae_delta_val_pad_to_ret M
    from alt2 show ?thesis
    proof
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_pad_to_ret M
       show ?thesis using ae_delta_val_pad_to_ret_functional[OF A1 A2] by
simp
    next
      assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_pad_reject M
     have False using ae_delta_val_pad_to_ret_pad_reject_disjoint[OF A1 A2]
```

```
.
    thus ?thesis ..
  qed
 next
  assume A1: (s, a, s1, a1, d1) ∈ ae_delta_val_pad_reject M
  from alt2 show ?thesis
  proof
    assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_pad_to_ret M
   have False using ae_delta_val_pad_to_ret_pad_reject_disjoint[OF A2 A1]
.
    thus ?thesis ..
  next
    assume A2: (s, a, s2, a2, d2) ∈ ae_delta_val_pad_reject M
   show ?thesis using ae_delta_val_pad_reject_functional[OF A1 A2] by simp
  qed
 qed
qed
```

2.15 Chain uniqueness — single-step functionality

Under *valid_mttm M*, *det_mttm M*, and *le_tm M* $\neq$ *bl_tm M*, the alphabet-enlargement transition relation is functional in its source pair. Proof case-splits on the source substep index: *SSN* cases dispatch via the matching *ae_delta_ssN_only* + **_functional* pair; V-marker cases (*VFwd*, *VFwdPad*, *VRet*) use within-cluster disjointness to collapse the multi-relation alternative, then apply the surviving sub-relation's functionality lemma.

```
lemma alphabet_enlarge_delta_functional:
 fixes M :: ('q, 'a) mttm
 assumes vM:  valid_mttm M
    and det: det_mttm M
    and le_neq_bl: le_tm M ≠ bl_tm M
    and h1: (s, a, s1, a1, d1) ∈ alphabet_enlarge_delta M
    and h2: (s, a, s2, a2, d2) ∈ alphabet_enlarge_delta M
 shows s1 = s2 ∧ a1 = a2 ∧ d1 = d2
proof (cases snd (snd (snd (snd s))))
 case VFwd
 show ?thesis by (rule ae_delta_VFwd_functional[OF le_neq_bl h1 h2 VFwd])
next
 case VFwdPad
 show ?thesis by (rule ae_delta_VFwdPad_functional[OF h1 h2 VFwdPad])
next
 case VRet
 show ?thesis by (rule ae_delta_VRet_functional[OF h1 h2 VRet])
next
 case SS1
 have e1: (s, a, s1, a1, d1) ∈ ae_delta_ss1_ss2 M
   by (rule ae_delta_ss1_only[OF h1 SS1])
 have e2: (s, a, s2, a2, d2) ∈ ae_delta_ss1_ss2 M
```

```
    by (rule ae_delta_ss1_only[OF h2 SS1])
  show ?thesis using ae_delta_ss1_ss2_functional[OF e1 e2] by simp
next
  case SS2
  have e1: (s, a, s1, a1, d1) ∈ ae_delta_ss2_ss3 M
    by (rule ae_delta_ss2_only[OF h1 SS2])
  have e2: (s, a, s2, a2, d2) ∈ ae_delta_ss2_ss3 M
    by (rule ae_delta_ss2_only[OF h2 SS2])
  show ?thesis using ae_delta_ss2_ss3_functional[OF e1 e2] by simp
next
  case SS3
  have e1: (s, a, s1, a1, d1) ∈ ae_delta_ss3_ss4 M
    by (rule ae_delta_ss3_only[OF h1 SS3])
  have e2: (s, a, s2, a2, d2) ∈ ae_delta_ss3_ss4 M
    by (rule ae_delta_ss3_only[OF h2 SS3])
  show ?thesis using ae_delta_ss3_ss4_functional[OF e1 e2] by simp
next
  case SS4
  have e1: (s, a, s1, a1, d1) ∈ ae_delta_ss4_ss5 M
    by (rule ae_delta_ss4_only[OF h1 SS4])
  have e2: (s, a, s2, a2, d2) ∈ ae_delta_ss4_ss5 M
    by (rule ae_delta_ss4_only[OF h2 SS4])
  show ?thesis using ae_delta_ss4_ss5_functional[OF vM det e1 e2] by simp
next
  case SS5
  have e1: (s, a, s1, a1, d1) ∈ ae_delta_ss5_ss6 M
    by (rule ae_delta_ss5_only[OF h1 SS5])
  have e2: (s, a, s2, a2, d2) ∈ ae_delta_ss5_ss6 M
    by (rule ae_delta_ss5_only[OF h2 SS5])
  show ?thesis using ae_delta_ss5_ss6_functional[OF e1 e2] by simp
next
  case SS6
  have e1: (s, a, s1, a1, d1) ∈ ae_delta_ss6_ss7 M
    by (rule ae_delta_ss6_only[OF h1 SS6])
  have e2: (s, a, s2, a2, d2) ∈ ae_delta_ss6_ss7 M
    by (rule ae_delta_ss6_only[OF h2 SS6])
  show ?thesis using ae_delta_ss6_ss7_functional[OF e1 e2] by simp
next
  case SS7
  have e1: (s, a, s1, a1, d1) ∈ ae_delta_ss7_ss8 M
    by (rule ae_delta_ss7_only[OF h1 SS7])
  have e2: (s, a, s2, a2, d2) ∈ ae_delta_ss7_ss8 M
    by (rule ae_delta_ss7_only[OF h2 SS7])
  show ?thesis using ae_delta_ss7_ss8_functional[OF e1 e2] by simp
next
  case SS8
  have e1: (s, a, s1, a1, d1) ∈ ae_delta_ss8_ss1 M
    by (rule ae_delta_ss8_only[OF h1 SS8])
  have e2: (s, a, s2, a2, d2) ∈ ae_delta_ss8_ss1 M
```

```
    by (rule ae_delta_ss8_only[OF h2 SS8])
  show ?thesis using ae_delta_ss8_ss1_functional[OF e1 e2] by simp
qed
```

Lifting chain uniqueness from *alphabet_enlarge_delta* to *mttm_step* (*alphabet_enlarge_delta M*): single-step then n-fold via standard induction. Consumed in stage 6 to identify the forward arm's produced c''' with the backward arm's given c''.

```
lemma mttm_step_alphabet_enlarge_functional:
  fixes M :: ('q, 'a) mttm
  assumes vM: valid_mttm M
      and det: det_mttm M
      and le_neq_bl: le_tm M ≠ bl_tm M
      and h1: (c, c1) ∈ mttm_step (alphabet_enlarge_delta M)
      and h2: (c, c2) ∈ mttm_step (alphabet_enlarge_delta M)
  shows c1 = c2
proof −
  from h1 obtain q ts n q1 a1 dir1 where
      c_eq1: c = Config_M q ts n
    and c1_eq: c1 = Config_M q1 (λk. (ts k)(n k := a1 k))
                                (λk. go_dir (dir1 k) (n k))
    and tr1: (q, λk. ts k (n k), q1, a1, dir1)
             ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  from h2 obtain q' ts' n' q2 a2 dir2 where
      c_eq2: c = Config_M q' ts' n'
    and c2_eq: c2 = Config_M q2 (λk. (ts' k)(n' k := a2 k))
                                (λk. go_dir (dir2 k) (n' k))
    and tr2: (q', λk. ts' k (n' k), q2, a2, dir2)
             ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  from c_eq1 c_eq2
  have qq: q = q' and tsts: ts = ts' and nn: n = n'
    by auto
  from tr1 qq tsts nn
  have tr1': (q, λk. ts k (n k), q1, a1, dir1)
             ∈ alphabet_enlarge_delta M by simp
  from tr2 qq tsts nn
  have tr2': (q, λk. ts k (n k), q2, a2, dir2)
             ∈ alphabet_enlarge_delta M by simp
  have q1 = q2 ∧ a1 = a2 ∧ dir1 = dir2
    using alphabet_enlarge_delta_functional[OF vM det le_neq_bl tr1' tr2'] .
  thus ?thesis using c1_eq c2_eq tsts nn by simp
qed

lemma mttm_step_alphabet_enlarge_relpow_functional:
  fixes M :: ('q, 'a) mttm
  assumes vM: valid_mttm M
      and det: det_mttm M
```

```
    and le_neq_bl: le_tm M ≠ bl_tm M
    and h1: (c, c1) ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
    and h2: (c, c2) ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
  shows c1 = c2
  using h1 h2
proof (induction n arbitrary: c c1 c2)
  case 0
  thus ?case by simp
next
  case (Suc n)
  from Suc.prems(1) obtain c1' where
      step1: (c, c1') ∈ mttm_step (alphabet_enlarge_delta M)
    and rest1: (c1', c1) ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
    by (meson relpow_Suc_D2)
  from Suc.prems(2) obtain c2' where
      step2: (c, c2') ∈ mttm_step (alphabet_enlarge_delta M)
    and rest2: (c2', c2) ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
    by (meson relpow_Suc_D2)
  have c'_eq: c1' = c2'
    by (rule mttm_step_alphabet_enlarge_functional
          [OF vM det le_neq_bl step1 step2])
  show ?case using Suc.IH[OF rest1] rest2 c'_eq by simp
qed
```

det-free chain uniqueness for the *validation* prefix. *alphabet_enlarge_delta*'s only nondeterministic source is the SS4→SS5 macro-step; every other stage marker dispatches to a functional substep builder. In particular, from a validation marker (*VFwd* / *VFwdPad* / *VRet*) the transition is single-valued with no appeal to *det_mttm M* — the det-free analogue of *alphabet_enlarge_delta_functional* restricted to validation sources. Consumed by *alphabet_enlarge_language*'s reverse arm to identify the unique validation prefix from *ae_init_config* without the *det* hypothesis of Hopcroft–Ullman's speed-up theorems [6, Theorems 12.3, 12.4].

```
lemma alphabet_enlarge_delta_val_functional:
  fixes M :: ('q, 'a) mttm
  assumes le_neq_bl: le_tm M ≠ bl_tm M
    and h1: (s, a, s1, a1, d1) ∈ alphabet_enlarge_delta M
    and h2: (s, a, s2, a2, d2) ∈ alphabet_enlarge_delta M
    and idx: snd (snd (snd (snd s))) ∈ {VFwd, VFwdPad, VRet}
  shows s1 = s2 ∧ a1 = a2 ∧ d1 = d2
proof −
  consider (VFwd) snd (snd (snd (snd s))) = VFwd
      | (VFwdPad) snd (snd (snd (snd s))) = VFwdPad
      | (VRet) snd (snd (snd (snd s))) = VRet
    using idx by blast
  thus ?thesis
  proof cases
    case VFwd
```

```
    show ?thesis by (rule ae_delta_VFwd_functional[OF le_neq_bl h1 h2 VFwd])
  next
    case VFwdPad
    show ?thesis by (rule ae_delta_VFwdPad_functional[OF h1 h2 VFwdPad])
  next
    case VRet
    show ?thesis by (rule ae_delta_VRet_functional[OF h1 h2 VRet])
  qed
qed

lemma mttm_step_alphabet_enlarge_val_functional:
  fixes M :: ('q, 'a) mttm
  assumes le_neq_bl: le_tm M ≠ bl_tm M
      and h1: (c, c1) ∈ mttm_step (alphabet_enlarge_delta M)
      and h2: (c, c2) ∈ mttm_step (alphabet_enlarge_delta M)
      and idx: snd (snd (snd (snd (mt_state c)))) ∈ {VFwd, VFwdPad, VRet}
  shows c1 = c2
proof −
  from h1 obtain q ts n q1 a1 dir1 where
      c_eq1:  c  = Config_M q ts n
    and c1_eq: c1 = Config_M q1 (λk. (ts k)(n k := a1 k))
                                (λk. go_dir (dir1 k) (n k))
    and tr1: (q, λk. ts k (n k), q1, a1, dir1) ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  from h2 obtain q' ts' n' q2 a2 dir2 where
      c_eq2:  c  = Config_M q' ts' n'
    and c2_eq: c2 = Config_M q2 (λk. (ts' k)(n' k := a2 k))
                                (λk. go_dir (dir2 k) (n' k))
    and tr2: (q', λk. ts' k (n' k), q2, a2, dir2) ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  from c_eq1 c_eq2 have qq: q = q' and tsts: ts = ts' and nn: n = n'
    by auto
  from tr2 qq tsts nn
  have tr2': (q, λk. ts k (n k), q2, a2, dir2) ∈ alphabet_enlarge_delta M
    by simp
  have idx': snd (snd (snd (snd q))) ∈ {VFwd, VFwdPad, VRet}
    using idx c_eq1 by simp
  have q1 = q2 ∧ a1 = a2 ∧ dir1 = dir2
    by (rule alphabet_enlarge_delta_val_functional[OF le_neq_bl tr1 tr2' idx'])
  thus ?thesis using c1_eq c2_eq tsts nn by simp
qed
```

The relpow lift: an R-chain whose every source config (every config strictly before the end) sits at a validation marker is unique. The discharge supplies the invariant for an *ae_init_config* prefix that stays within the validation sweep.

```
lemma mttm_step_alphabet_enlarge_val_relpow_functional:
  fixes M :: ('q, 'a) mttm
  assumes le_neq_bl: le_tm M ≠ bl_tm M
```

```
    and h1: (c, c1) ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
    and h2: (c, c2) ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
    and inv: ∀ i<n. ∀ d. (c, d) ∈ mttm_step (alphabet_enlarge_delta M) ^^ i
                ⟶ snd (snd (snd (snd (mt_state d))))
                    ∈ {VFwd, VFwdPad, VRet}
  shows c1 = c2
  using h1 h2 inv
proof (induction n arbitrary: c c1 c2)
  case 0
  thus ?case by simp
next
  case (Suc n)
  from Suc.prems(1) obtain c1' where
      step1: (c, c1') ∈ mttm_step (alphabet_enlarge_delta M)
    and rest1: (c1', c1) ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
    by (meson relpow_Suc_D2)
  from Suc.prems(2) obtain c2' where
      step2: (c, c2') ∈ mttm_step (alphabet_enlarge_delta M)
    and rest2: (c2', c2) ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
    by (meson relpow_Suc_D2)
  have c_val: snd (snd (snd (snd (mt_state c)))) ∈ {VFwd, VFwdPad, VRet}
  proof −
    have a: (c, c) ∈ mttm_step (alphabet_enlarge_delta M) ^^ 0 by simp
    have b: (0::nat) < Suc n by simp
    show ?thesis using Suc.prems(3) a b by blast
  qed
  have c'_eq: c1' = c2'
    by (rule mttm_step_alphabet_enlarge_val_functional
            [OF le_neq_bl step1 step2 c_val])
  have rest2': (c1', c2) ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
    using rest2 c'_eq by simp
  have inv':
      ∀ i<n. ∀ d. (c1', d) ∈ mttm_step (alphabet_enlarge_delta M) ^^ i
                ⟶ snd (snd (snd (snd (mt_state d))))
                    ∈ {VFwd, VFwdPad, VRet}
  proof (intro allI impI)
    fix i d
    assume i_lt: i < n
      and reach: (c1', d) ∈ mttm_step (alphabet_enlarge_delta M) ^^ i
    have (c, d) ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc i
      by (rule relpow_Suc_I2[OF step1 reach])
    moreover have Suc i < Suc n using i_lt by simp
    ultimately show snd (snd (snd (snd (mt_state d))))
                      ∈ {VFwd, VFwdPad, VRet}
      using Suc.prems(3) by blast
  qed
  show ?case by (rule Suc.IH[OF rest1 rest2' inv'])
qed
```

end
theory *AlphabetEnlargement_BufferArith*
imports *AlphabetEnlargement_Uniqueness*
begin

Arithmetic lemmas on the 3-block buffer abstraction used by the *alphabet_enlarge* simulation chain. Definitions of *bp_advance*, *bp_linear*, *buf_lin_at*, *write_bp*, *read_bp*, and the *ae_window_invariant* family live in *AlphabetEnlargement_Substeps*; this theory collects the lemmas that relate them and forms a self-contained arithmetic layer between the encoder/decoder and the forward simulation proof.

Contents, in dependency order:

- *c_idx* / *bp_linear* base arithmetic: *c_succ* / *c_pred* on *c_idx*, *bp_linear* at the three block-anchor patterns, the universal $< 3c$ bound, and the *c_idx* values at the first / last enum element.
- *bp_advance*'s commutation with *bp_linear*: each successful R-move increments *bp_linear*, each successful L-move decrements it; failure cases correspond to buffer-edge positions. Head-stays-in-buffer totality lemmas for strict-interior R / L moves. LE-aware totality and single-step linear-bound preservation for *bp_advance_le*.
- Read-from-window lemmas: bridge between buffer-read at a buffered head position and the linearised buffer access (steady / LE-edge / per-tape unified regimes).
- *buf_lin_at* effect of *write_bp* at the matched buffered head's linear index and at all other indices.
- Window-invariant single-step preservation under one buffered M-step: steady regime, *le1* regime, *le0* regime, and the per-tape unified companion dispatching on the regime selector *pos*.

  These lemmas are the standalone arithmetic underpinning piece 4 of the AE simulation correctness proof (*ae_m_steps_buffered_correct*): the buffered compute's head displacement matches the actual head displacement step-by-step, and the head stays in the buffer for any prefix of $\leq c$ moves starting from the home block.

2.16 Buffered head position arithmetic

Lemmas relating *bp_linear* (linearised buffer index in $[0, 3c)$) to *bp_advance*'s case structure and to the *c_succ* / *c_pred* partial-step operations on $'c$. These are the standalone arithmetic underpinning piece 4

(*ae_m_steps_buffered_correct*): the buffered compute's head displacement matches the actual head displacement step-by-step, and the head stays in the buffer for any prefix of $\leq c$ moves starting from the home block.

2.16.1 *c_idx* and *bp_linear* arithmetic

c_succ and *c_pred*'s effect on *c_idx*.

```
lemma c_succ_idx_some:
  fixes x y :: 'c :: enum
  assumes c_succ x = Some y
  shows c_idx y = Suc (c_idx x)
proof −
  let ?xs = enum_class.enum :: 'c list
  from assms have suc_lt: Suc (c_idx x) < length ?xs
                and y_eq: y = ?xs ! Suc (c_idx x)
    unfolding c_succ_def Let_def
    by (auto split: if_split_asm)
  show ?thesis using c_idx_enum_nth[OF suc_lt] y_eq by simp
qed

lemma c_succ_idx_none:
  fixes x :: 'c :: enum
  assumes c_succ x = None
  shows Suc (c_idx x) = card (UNIV :: 'c set)
proof −
  let ?xs = enum_class.enum :: 'c list
  from assms have ¬ Suc (c_idx x) < length ?xs
    unfolding c_succ_def Let_def by (auto split: if_split_asm)
  hence Suc (c_idx x) ≥ length ?xs by simp
  moreover have c_idx x < length ?xs by (rule c_idx_in_range)
  ultimately have Suc (c_idx x) = length ?xs by linarith
  thus ?thesis by (simp add: card_eq_length_enum)
qed

lemma c_pred_idx_some:
  fixes x y :: 'c :: enum
  assumes c_pred x = Some y
  shows Suc (c_idx y) = c_idx x
proof −
  let ?xs = enum_class.enum :: 'c list
  from assms have idx_pos: c_idx x ≠ 0
                and y_eq: y = ?xs ! (c_idx x − 1)
    unfolding c_pred_def Let_def
    by (auto split: if_split_asm)
  from idx_pos have idx_lt: c_idx x − 1 < length ?xs
    using c_idx_in_range(1)[of x] by linarith
  have c_idx y = c_idx x − 1
    using c_idx_enum_nth[OF idx_lt] y_eq by simp
```

```
  thus ?thesis using idx_pos by simp
qed

lemma c_pred_idx_none:
  fixes x :: 'c :: enum
  assumes c_pred x = None
  shows c_idx x = 0
  using assms unfolding c_pred_def Let_def
  by (auto split: if_split_asm)
```

bp_linear's value at the three home/left/right block patterns, plus the universal $< 3c$ bound.

```
lemma bp_linear_AE_Left [simp]:
  fixes off :: 'c :: enum
  shows bp_linear (AE_Left, off) = c_idx off
  by (simp add: bp_linear_def)

lemma bp_linear_AE_Home [simp]:
  fixes off :: 'c :: enum
  shows bp_linear (AE_Home, off) = card (UNIV :: 'c set) + c_idx off
  by (simp add: bp_linear_def)

lemma bp_linear_AE_Right [simp]:
  fixes off :: 'c :: enum
  shows bp_linear (AE_Right, off)
        = 2 * card (UNIV :: 'c set) + c_idx off
  by (simp add: bp_linear_def)

lemma bp_linear_lt_3c:
  fixes p :: ('c :: enum) bp
  shows bp_linear p < 3 * card (UNIV :: 'c set)
proof -
  have idx_lt: c_idx (snd p) < card (UNIV :: 'c set)
    by (rule c_idx_lt_card)
  show ?thesis using idx_lt
    by (cases p; cases fst p) auto
qed
```

2.16.2 *bp_advance* commutation, totality, and LE-aware bounds

bp_advance's commutation with *bp_linear*: each successful R-move increments *bp_linear* by 1, each successful L-move decrements by 1, N is identity.

```
lemma bp_advance_N_some:
  bp_advance p dir.N = Some p
  unfolding bp_advance_def by (cases p) simp

lemma bp_advance_N_linear:
  bp_advance p dir.N = Some p ∧ bp_linear p = bp_linear p
  using bp_advance_N_some by simp
```

The first / last enum element decode to indices 0 and $c - 1$.

```
lemma c_idx_first:
  shows c_idx (c_first :: 'c :: enum) = 0
proof −
  have len_pos: 0 < length (enum_class.enum :: 'c list)
    using c_idx_in_range(1)[of c_first :: 'c] by linarith
  have c_idx ((enum_class.enum :: 'c list) ! 0) = 0
    using c_idx_enum_nth[OF len_pos] .
  thus ?thesis unfolding c_first_def .
qed

lemma c_idx_last:
  shows c_idx (c_last :: 'c :: enum) = card (UNIV :: 'c set) − 1
proof −
  let ?xs = enum_class.enum :: 'c list
  have len_pos: 0 < length ?xs
    using c_idx_in_range(1)[of c_last :: 'c] by linarith
  have last_lt: length ?xs − 1 < length ?xs using len_pos by simp
  have c_idx ((?xs :: 'c list) ! (length ?xs − 1)) = length ?xs − 1
    using c_idx_enum_nth[OF last_lt] .
  thus ?thesis
    by (simp add: c_last_def card_eq_length_enum)
qed

lemma bp_advance_R_linear:
  fixes p p' :: ('c :: enum) bp
  assumes bp_advance p dir.R = Some p'
  shows bp_linear p' = Suc (bp_linear p)
proof (cases c_succ (snd p))
  case (Some off')
  from Some assms have p'_eq: p' = (fst p, off')
    unfolding bp_advance_def by (cases p) simp
  have idx_eq: c_idx off' = Suc (c_idx (snd p))
    using c_succ_idx_some[OF Some] .
  show ?thesis using p'_eq idx_eq
    by (cases p; cases fst p) auto
next
  case None
  from None have idx_at_last: Suc (c_idx (snd p)) = card (UNIV :: 'c set)
    using c_succ_idx_none by blast
  show ?thesis
  proof (cases fst p)
    case AE_Left
    with None assms have p'_eq: p' = (AE_Home, c_first)
      unfolding bp_advance_def by (cases p) simp
    show ?thesis using p'_eq AE_Left idx_at_last c_idx_first
      by (cases p) auto
  next
    case AE_Home
```

```
    with None assms have p'_eq: p' = (AE_Right, c_first)
      unfolding bp_advance_def by (cases p) simp
    show ?thesis using p'_eq AE_Home idx_at_last c_idx_first
      by (cases p) auto
  next
    case AE_Right
    with None assms show ?thesis
      unfolding bp_advance_def by (cases p) simp
  qed
qed

lemma bp_advance_L_linear:
  fixes p p' :: ('c :: enum) bp
  assumes bp_advance p dir.L = Some p'
  shows Suc (bp_linear p') = bp_linear p
proof (cases c_pred (snd p))
  case (Some off')
  from Some assms have p'_eq: p' = (fst p, off')
    unfolding bp_advance_def by (cases p) simp
  have idx_eq: Suc (c_idx off') = c_idx (snd p)
    using c_pred_idx_some[OF Some] .
  show ?thesis using p'_eq idx_eq
    by (cases p; cases fst p) auto
next
  case None
  from None have idx_zero: c_idx (snd p) = 0
    using c_pred_idx_none by blast
  have c_pos: 0 < card (UNIV :: 'c set)
    using c_idx_lt_card[of snd p] by linarith
  show ?thesis
  proof (cases fst p)
    case AE_Left
    with None assms show ?thesis
      unfolding bp_advance_def by (cases p) simp
  next
    case AE_Home
    with None assms have p'_eq: p' = (AE_Left, c_last)
      unfolding bp_advance_def by (cases p) simp
    have lin_p': bp_linear p' = card (UNIV :: 'c set) - 1
      using p'_eq c_idx_last by simp
    have lin_p: bp_linear p = card (UNIV :: 'c set)
      using AE_Home idx_zero by (cases p) auto
    show ?thesis using lin_p' lin_p c_pos by linarith
  next
    case AE_Right
    with None assms have p'_eq: p' = (AE_Home, c_last)
      unfolding bp_advance_def by (cases p) simp
    have lin_p': bp_linear p' = card (UNIV :: 'c set) + (card (UNIV :: 'c set) -
1)
```

```
    using p'_eq c_idx_last by simp
  have lin_p: bp_linear p = 2 * card (UNIV :: 'c set)
    using AE_Right idx_zero by (cases p) auto
  show ?thesis using lin_p' lin_p c_pos by linarith
 qed
qed
```

Head-stays-in-buffer: any R-move from a strict-interior position succeeds; any L-move from a strict-interior position succeeds. These are the standalone arithmetic underpinning the inductive proof of *ae_m_steps_buffered_correct*'s step case.

```
lemma bp_advance_R_some:
 fixes p :: ('c :: enum) bp
 assumes bp_linear p < 3 * card (UNIV :: 'c set) - 1
 shows ∃p'. bp_advance p dir.R = Some p'
proof (cases c_succ (snd p))
 case (Some off')
 thus ?thesis unfolding bp_advance_def by (cases p) simp
next
 case None
 hence idx_at_last: Suc (c_idx (snd p)) = card (UNIV :: 'c set)
   using c_succ_idx_none by blast
 show ?thesis
 proof (cases fst p)
   case AE_Left
   with None show ?thesis
     unfolding bp_advance_def by (cases p) simp
 next
   case AE_Home
   with None show ?thesis
     unfolding bp_advance_def by (cases p) simp
 next
   case AE_Right
   — Contradicts bp_linear p < 3c - 1: at (AE_Right, c_last), bp_linear p =
3c - 1.
   have lin_eq: bp_linear p = 3 * card (UNIV :: 'c set) - 1
     using AE_Right idx_at_last
     by (cases p) auto
   with assms show ?thesis by linarith
 qed
qed

lemma bp_advance_L_some:
 fixes p :: ('c :: enum) bp
 assumes 0 < bp_linear p
 shows ∃p'. bp_advance p dir.L = Some p'
proof (cases c_pred (snd p))
 case (Some off')
 thus ?thesis unfolding bp_advance_def by (cases p) simp
```

```
next
  case None
  hence idx_zero: c_idx (snd p) = 0
    using c_pred_idx_none by blast
  show ?thesis
  proof (cases fst p)
    case AE_Left
    — Contradicts 0 < bp_linear p: at (AE_Left, c_first), bp_linear p = 0.
    have lin_eq: bp_linear p = 0
      using AE_Left idx_zero by (cases p) auto
    with assms show ?thesis by linarith
  next
    case AE_Home
    with None show ?thesis
      unfolding bp_advance_def by (cases p) simp
  next
    case AE_Right
    with None show ?thesis
      unfolding bp_advance_def by (cases p) simp
  qed
qed
```

LE-aware totality of *bp_advance_le*: given the same position bounds that make *bp_advance* total (*bp_linear* $< 3c - 1$ excludes the only None case for R, $0 <$ *bp_linear* excludes the only None case for L), the LE-aware wrapper is also total the only "extra" behaviour is the LE-jump, which always returns *Some* (*AE_Right*, *c_first*).

The two position bounds correspond to "head not at the rightmost cell of the right block" and "head not at the leftmost cell of the left block". In *ae_coupled_run_aux_general*'s inductive step these bounds hold strictly at the precondition step n' thanks to the fifth output conjunct (*bp_linear bound*) and the budget $n' \leq c - 1$.

```
lemma bp_advance_le_total:
  fixes le a :: 'a
    and p :: ('c :: enum) bp
    and d :: dir
  assumes bp_lt_max: bp_linear p < 3 * card (UNIV :: 'c set) - 1
      and bp_gt_zero: 0 < bp_linear p
  shows ∃p'. bp_advance_le le a p d = Some p'
proof (cases a = le ∧ fst p = AE_Home ∧ d = dir.R)
  case True
  hence bp_advance_le le a p d = Some (AE_Right, c_first)
    unfolding bp_advance_le_def by simp
  thus ?thesis by auto
next
  case False
  hence skip: bp_advance_le le a p d = bp_advance p d
    unfolding bp_advance_le_def by auto
```

```
  show ?thesis
  proof (cases d)
    case N
    have bp_advance_le le a p d = Some p
      using skip N bp_advance_N_some[of p] by simp
    thus ?thesis by blast
  next
    case R
    thus ?thesis using skip bp_advance_R_some[OF bp_lt_max] by auto
  next
    case L
    thus ?thesis using skip bp_advance_L_some[OF bp_gt_zero] by auto
  qed
qed
```

Single-step *bp_linear* preservation for *bp_advance_le*: the $c \le$ *bp_linear* $+ n$ / *bp_linear* $< 2c + n$ invariant is preserved when n increases by one. Four cases by *bp_advance_le*'s outcome:

- LE-jump fires: *bp_linear* $p' = 2c$, both bounds hold trivially. - *bp_advance* N: *bp_linear* p' = *bp_linear* p; bounds widen. - *bp_advance* R: *bp_linear* p' = *Suc* (*bp_linear* p) (*bp_advance_R_linear*); upper bound +1, lower +0. - *bp_advance* L: *Suc* (*bp_linear* p') = *bp_linear* p (*bp_advance_L_linear*); upper -1, lower +1 (needs $0 <$ *bp_linear* p to avoid nat truncation).

Used in *ae_coupled_run_aux_general*'s inductive step to discharge the fifth output conjunct at *cM_n*.

```
lemma bp_advance_le_lin_bounded:
  fixes le a :: 'a
    and p p' :: ('c :: enum) bp
    and d :: dir and n :: nat
  assumes adv: bp_advance_le le a p d = Some p'
      and p_pos: 0 < bp_linear p
      and p_upper: bp_linear p < 2 * card (UNIV :: 'c set) + n
      and p_lower: card (UNIV :: 'c set) ≤ bp_linear p + n
  shows card (UNIV :: 'c set) ≤ bp_linear p' + Suc n
        ∧ bp_linear p' < 2 * card (UNIV :: 'c set) + Suc n
proof (cases a = le ∧ fst p = AE_Home ∧ d = dir.R)
  case True
  hence p'_eq: p' = (AE_Right, c_first)
    using adv unfolding bp_advance_le_def by simp
  hence lin_p': bp_linear p' = 2 * card (UNIV :: 'c set)
    by (simp add: c_idx_first)
  show ?thesis using lin_p' by simp
next
  case False
  hence skip: bp_advance_le le a p d = bp_advance p d
    unfolding bp_advance_le_def by auto
  with adv have adv_bp: bp_advance p d = Some p' by simp
```

```
  show ?thesis
  proof (cases d)
    case N
    have bp_advance p dir.N = Some p by (rule bp_advance_N_some)
    with adv_bp N have p' = p by simp
    hence bp_linear p' = bp_linear p by simp
    thus ?thesis using p_upper p_lower by linarith
  next
    case R
    with adv_bp have bp_linear p' = Suc (bp_linear p)
      using bp_advance_R_linear by blast
    thus ?thesis using p_upper p_lower by linarith
  next
    case L
    with adv_bp have Suc (bp_linear p') = bp_linear p
      using bp_advance_L_linear by blast
    thus ?thesis using p_upper p_lower p_pos by linarith
  qed
qed
```

2.16.3 Read-from-window lemmas

Bridge between buffer-read at a buffered head position and the linearised buffer access. Together with the window invariant this gives *read_bp blocks bp = tM nM*, the key step that lets a buffered δ-tuple be matched against an M-step's actual symbol read.

```
lemma buf_lin_at_eq_read_bp:
  fixes blocks :: (('c :: enum) ⇒ 'a)
                  × ('c ⇒ 'a) × ('c ⇒ 'a)
    and bp :: 'c bp
  shows buf_lin_at blocks (bp_linear bp) = read_bp blocks bp
proof −
  obtain l h r where blocks_eq: blocks = (l, h, r)
    by (cases blocks) auto
  obtain b off where bp_eq: bp = (b, off) by (cases bp)
  let ?c = card (UNIV :: 'c set)
  let ?xs = enum_class.enum :: 'c list
  have idx_lt: c_idx off < ?c by (rule c_idx_lt_card)
  have len_eq: ?c = length ?xs by (rule card_eq_length_enum)
  have nth_off: ?xs ! c_idx off = off by (rule c_idx_in_range(2))
  show ?thesis
  proof (cases b)
    case AE_Left
    have bp_linear bp = c_idx off
      using bp_eq AE_Left by simp
    moreover have c_idx off < ?c using idx_lt .
    ultimately show ?thesis
      using bp_eq blocks_eq AE_Left nth_off
      unfolding buf_lin_at_def read_bp_def by simp
```

```
  next
    case AE_Home
    have lin_eq: bp_linear bp = ?c + c_idx off
      using bp_eq AE_Home by simp
    have not_lt_c: ¬ ?c + c_idx off < ?c by simp
    have lt_2c: ?c + c_idx off < 2 * ?c using idx_lt by simp
    have sub_eq: ?c + c_idx off − ?c = c_idx off by simp
    show ?thesis
      using bp_eq blocks_eq AE_Home lin_eq not_lt_c lt_2c sub_eq nth_off
      unfolding buf_lin_at_def read_bp_def by simp
  next
    case AE_Right
    have lin_eq: bp_linear bp = 2 * ?c + c_idx off
      using bp_eq AE_Right by simp
    have not_lt_c: ¬ 2 * ?c + c_idx off < ?c by simp
    have not_lt_2c: ¬ 2 * ?c + c_idx off < 2 * ?c by simp
    have sub_eq: 2 * ?c + c_idx off − 2 * ?c = c_idx off by simp
    show ?thesis
      using bp_eq blocks_eq AE_Right lin_eq not_lt_c not_lt_2c sub_eq nth_off
      unfolding buf_lin_at_def read_bp_def by simp
  qed
qed

lemma read_bp_via_window:
  fixes tM :: nat ⇒ 'a
    and bp :: ('c :: enum) bp
    and blocks :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
  assumes ae_window_invariant tM nM bp blocks p_start
  shows read_bp blocks bp = tM nM
proof −
  let ?c = card (UNIV :: 'c set)
  from assms have nM_eq: nM = p_start + bp_linear bp
    unfolding ae_window_invariant_def by blast
  from assms have win:
      ∀ i. i < 3 * ?c ⟶ tM (p_start + i) = buf_lin_at blocks i
    unfolding ae_window_invariant_def by blast
  have lin_lt: bp_linear bp < 3 * ?c by (rule bp_linear_lt_3c)
  have e1: tM nM = buf_lin_at blocks (bp_linear bp)
    using win lin_lt nM_eq by simp
  have e2: buf_lin_at blocks (bp_linear bp) = read_bp blocks bp
    by (rule buf_lin_at_eq_read_bp)
  from e1 e2 show ?thesis by simp
qed
```

LE-edge analogue of *read_bp_via_window* for the *le1* regime: the buffered read at the head equals the M-tape read at the M-side head position, under *ae_window_invariant_le1*. Three sub-cases on *fst bp*: - *AE_Left* (head at (*AE_Left*, *c_last*)): *nM = 0*; M reads *le*; buffer reads *l c_last = LE_block le c_last = le*. Match. - *AE_Home* (head in home block, off-

set *off*): $nM = Suc\ (c_idx\ off)$; routes through the right slot of the buf-linearisation $tM\ (Suc\ i) = buf_lin_at\ blocks\ (c + i)$ at $i = c_idx\ off$. - *AE_Right* (head in right block, offset *off*): $nM = Suc\ (c + c_idx\ off)$; same buf-linearisation at $i = c + c_idx\ off$.

lemma *read_bp_via_window_le1*:
  fixes *tM* :: *nat* ⇒ *'a*
    and *bp* :: (*'c* :: *enum*) *bp*
    and *blocks* :: (*'c* ⇒ *'a*) × (*'c* ⇒ *'a*) × (*'c* ⇒ *'a*)
  assumes *wi*: *ae_window_invariant_le1 tM nM bp blocks le*
  shows *read_bp blocks bp = tM nM*
proof −
  let *?c* = *card* (*UNIV* :: *'c set*)
  obtain *b off* **where** *bp_eq*: *bp* = (*b*, *off*) **by** (*cases bp*)
  obtain *l h r* **where** *blocks_eq*: *blocks* = (*l*, *h*, *r*)
    by (*cases blocks*) *auto*
  from *wi* **have** *tM_0*: *tM 0 = le*
    unfolding *ae_window_invariant_le1_def* **by** *blast*
  from *wi* **have** *l_eq*: *fst blocks = LE_block le*
    unfolding *ae_window_invariant_le1_def* **by** *blast*
  from *wi* **have** *win*:
      ∀ *i*. *i* < *2* ∗ *?c*
            ⟶ *tM* (*Suc i*) = *buf_lin_at blocks* (*?c* + *i*)
    unfolding *ae_window_invariant_le1_def* **by** *blast*
  show *?thesis*
  proof (*cases b*)
    case *AE_Left*
    have *off_c_last*: *off = c_last*
      using *wi bp_eq AE_Left*
      unfolding *ae_window_invariant_le1_def* **by** *auto*
    have *nM_eq*: *nM = 0*
      using *wi bp_eq AE_Left*
      unfolding *ae_window_invariant_le1_def* **by** *auto*
    have *rd_l*: *read_bp blocks bp = l c_last*
      using *bp_eq AE_Left off_c_last blocks_eq*
      unfolding *read_bp_def* **by** *simp*
    have *l_const*: *l c_last = le*
      using *l_eq blocks_eq* **unfolding** *LE_block_def* **by** *simp*
    show *?thesis* **using** *rd_l l_const nM_eq tM_0* **by** *simp*
  next
    case *AE_Home*
    have *nM_eq*: *nM* = *Suc* (*c_idx off*)
      using *wi bp_eq AE_Home*
      unfolding *ae_window_invariant_le1_def* **by** *auto*
    have *idx_lt*: *c_idx off* < *?c* **by** (*rule c_idx_lt_card*)
    have *idx_lt_2c*: *c_idx off* < *2* ∗ *?c* **using** *idx_lt* **by** *simp*
    have *tM_eq*:
        tM (*Suc* (*c_idx off*)) = *buf_lin_at blocks* (*?c* + *c_idx off*)
      using *win idx_lt_2c* **by** *simp*
    have *lin_eq*: *bp_linear bp = ?c + c_idx off*

```
    using bp_eq AE_Home unfolding bp_linear_def by simp
  have rd_e: buf_lin_at blocks (bp_linear bp) = read_bp blocks bp
    by (rule buf_lin_at_eq_read_bp)
  show ?thesis using nM_eq tM_eq lin_eq rd_e by simp
next
  case AE_Right
  have nM_eq: nM = Suc (?c + c_idx off)
    using wi bp_eq AE_Right
    unfolding ae_window_invariant_le1_def by auto
  have idx_lt: c_idx off < ?c by (rule c_idx_lt_card)
  have lt_2c: ?c + c_idx off < 2 * ?c using idx_lt by simp
  have tM_eq:
      tM (Suc (?c + c_idx off))
        = buf_lin_at blocks (?c + (?c + c_idx off))
    using win lt_2c by simp
  have lin_eq: bp_linear bp = 2 * ?c + c_idx off
    using bp_eq AE_Right unfolding bp_linear_def by simp
  have eq2: ?c + (?c + c_idx off) = 2 * ?c + c_idx off
    by simp
  have rd_e: buf_lin_at blocks (bp_linear bp) = read_bp blocks bp
    by (rule buf_lin_at_eq_read_bp)
  show ?thesis using nM_eq tM_eq lin_eq rd_e eq2 by simp
qed
qed
```

LE-edge analogue of *read_bp_via_window* for the *le0* regime. Buffered head can be: - *AE_Home*: *nM = 0*; M reads *le*; buffer reads *h off = LE_block le off = le* (home slot is the *LE_block*). Match. - *AE_Right*: *nM = Suc (c_idx off)*; routes through the right slot's buf-linearisation *tM (Suc i) = buf_lin_at blocks (2c + i)* at *i = c_idx off*. - *AE_Left* excluded by the window invariant.

```
lemma read_bp_via_window_le0:
  fixes tM :: nat ⇒ 'a
    and bp :: ('c :: enum) bp
    and blocks :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
  assumes wi: ae_window_invariant_le0 tM nM bp blocks le
  shows read_bp blocks bp = tM nM
proof −
  let ?c = card (UNIV :: 'c set)
  obtain b off where bp_eq: bp = (b, off) by (cases bp)
  obtain l h r where blocks_eq: blocks = (l, h, r)
    by (cases blocks) auto
  from wi have tM_0: tM 0 = le
    unfolding ae_window_invariant_le0_def by blast
  from wi have h_eq: fst (snd blocks) = LE_block le
    unfolding ae_window_invariant_le0_def by blast
  from wi have win:
      ∀ i. i < ?c
          ⟶ tM (Suc i) = buf_lin_at blocks (2 * ?c + i)
```

```
    unfolding ae_window_invariant_le0_def by blast
  show ?thesis
  proof (cases b)
    case AE_Left
    — Excluded by ae_window_invariant_le0's case analysis on fst bp.
    from wi bp_eq AE_Left have False
      unfolding ae_window_invariant_le0_def by auto
    thus ?thesis ..
  next
    case AE_Home
    have nM_eq: nM = 0
      using wi bp_eq AE_Home
      unfolding ae_window_invariant_le0_def by auto
    have rd_h: read_bp blocks bp = h off
      using bp_eq AE_Home blocks_eq
      unfolding read_bp_def by simp
    have h_const: h off = le
      using h_eq blocks_eq unfolding LE_block_def by simp
    show ?thesis using rd_h h_const nM_eq tM_0 by simp
  next
    case AE_Right
    have nM_eq: nM = Suc (c_idx off)
      using wi bp_eq AE_Right
      unfolding ae_window_invariant_le0_def by auto
    have idx_lt: c_idx off < ?c by (rule c_idx_lt_card)
    have tM_eq:
        tM (Suc (c_idx off))
          = buf_lin_at blocks (2 * ?c + c_idx off)
      using win idx_lt by simp
    have lin_eq: bp_linear bp = 2 * ?c + c_idx off
      using bp_eq AE_Right unfolding bp_linear_def by simp
    have rd_e: buf_lin_at blocks (bp_linear bp) = read_bp blocks bp
      by (rule buf_lin_at_eq_read_bp)
    show ?thesis using nM_eq tM_eq lin_eq rd_e by simp
  qed
qed
```

Per-tape unified read-from-window companion: dispatches on the regime selector *pos* into the three sibling read-match lemmas (*read_bp_via_window_le0*, *read_bp_via_window_le1*, *read_bp_via_window*). This is the load-bearing read-match lemma at the inductive step of *ae_coupled_run_aux_general*: *read_bp* at the buffered head equals M-side read.

```
lemma read_bp_via_window_general:
  fixes tM :: nat ⇒ 'a
    and bp :: ('c :: enum) bp
    and blocks :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
  assumes wi: ae_window_invariant_general tM nM bp blocks pos le
  shows read_bp blocks bp = tM nM
```

```
proof −
  let ?c = card (UNIV :: 'c set)
  consider (steady) pos ≥ 2 | (le1) pos = 1 | (le0) pos = 0
    by linarith
  then show ?thesis
  proof cases
    case steady
    have wi_st: ae_window_invariant tM nM bp blocks
                  ((pos − 2) * ?c + 1)
      using wi steady
      unfolding ae_window_invariant_general_def by simp
    show ?thesis by (rule read_bp_via_window[OF wi_st])
  next
    case le1
    have wi_l1: ae_window_invariant_le1 tM nM bp blocks le
      using wi le1
      unfolding ae_window_invariant_general_def by simp
    show ?thesis by (rule read_bp_via_window_le1[OF wi_l1])
  next
    case le0
    have wi_l0: ae_window_invariant_le0 tM nM bp blocks le
      using wi le0
      unfolding ae_window_invariant_general_def by simp
    show ?thesis by (rule read_bp_via_window_le0[OF wi_l0])
  qed
qed
```

2.16.4 *buf_lin_at* preservation under *write_bp*

Effect of *write_bp* on *buf_lin_at*: at the buffered head's linearised position, the linearised access reads back the freshly written symbol; at any other in-window position, the linearised access is unchanged. Used in the window-invariant preservation step of the Suc case of *ae_coupled_run_aux*.

```
lemma buf_lin_at_write_bp_match:
  fixes blocks :: (('c :: enum) ⇒ 'a)
                  × ('c ⇒ 'a) × ('c ⇒ 'a)
    and bp :: 'c bp
    and x :: 'a
  shows buf_lin_at (write_bp blocks bp x) (bp_linear bp) = x
proof −
  obtain l h r where blocks_eq: blocks = (l, h, r)
    by (cases blocks) auto
  obtain b off where bp_eq: bp = (b, off) by (cases bp)
  let ?c = card (UNIV :: 'c set)
  let ?xs = enum_class.enum :: 'c list
  have idx_lt: c_idx off < ?c by (rule c_idx_lt_card)
  have nth_off: ?xs ! c_idx off = off by (rule c_idx_in_range(2))
  show ?thesis
  proof (cases b)
```

```
    case AE_Left
    have lin_eq: bp_linear bp = c_idx off
      using bp_eq AE_Left by simp
    have wb: write_bp blocks bp x = (l(off := x), h, r)
      using bp_eq blocks_eq AE_Left unfolding write_bp_def by simp
    show ?thesis
      using lin_eq idx_lt wb nth_off
      unfolding buf_lin_at_def by simp
  next
    case AE_Home
    have lin_eq: bp_linear bp = ?c + c_idx off
      using bp_eq AE_Home by simp
    have not_lt_c: ¬ ?c + c_idx off < ?c by simp
    have lt_2c: ?c + c_idx off < 2 * ?c using idx_lt by simp
    have sub_eq: ?c + c_idx off − ?c = c_idx off by simp
    have wb: write_bp blocks bp x = (l, h(off := x), r)
      using bp_eq blocks_eq AE_Home unfolding write_bp_def by simp
    show ?thesis
      using lin_eq not_lt_c lt_2c sub_eq wb nth_off
      unfolding buf_lin_at_def by simp
  next
    case AE_Right
    have lin_eq: bp_linear bp = 2 * ?c + c_idx off
      using bp_eq AE_Right by simp
    have not_lt_c: ¬ 2 * ?c + c_idx off < ?c by simp
    have not_lt_2c: ¬ 2 * ?c + c_idx off < 2 * ?c by simp
    have sub_eq: 2 * ?c + c_idx off − 2 * ?c = c_idx off by simp
    have wb: write_bp blocks bp x = (l, h, r(off := x))
      using bp_eq blocks_eq AE_Right unfolding write_bp_def by simp
    show ?thesis
      using lin_eq not_lt_c not_lt_2c sub_eq wb nth_off
      unfolding buf_lin_at_def by simp
  qed
qed

lemma buf_lin_at_write_bp_other:
  fixes blocks :: (('c :: enum) ⇒ 'a)
                  × ('c ⇒ 'a) × ('c ⇒ 'a)
    and bp :: 'c bp
    and x :: 'a
    and i :: nat
  assumes neq:    i ≠ bp_linear bp
      and i_lt:   i < 3 * card (UNIV :: 'c set)
  shows buf_lin_at (write_bp blocks bp x) i = buf_lin_at blocks i
proof −
  obtain l h r where blocks_eq: blocks = (l, h, r)
    by (cases blocks) auto
  obtain b off where bp_eq: bp = (b, off) by (cases bp)
  let ?c = card (UNIV :: 'c set)
```

```
let ?xs = enum_class.enum :: 'c list
have idx_lt: c_idx off < ?c by (rule c_idx_lt_card)
have len_eq: length ?xs = ?c by (rule card_eq_length_enum[symmetric])
have c_idx_eq:
   ⋀j. j < ?c ⟹ c_idx ((?xs :: 'c list) ! j) = j
proof −
  fix j assume j < ?c
  with len_eq have j < length ?xs by simp
  thus c_idx ((?xs :: 'c list) ! j) = j by (rule c_idx_enum_nth)
qed
show ?thesis
proof (cases b)
  case AE_Left
  have lin_eq: bp_linear bp = c_idx off
    using bp_eq AE_Left by simp
  have wb: write_bp blocks bp x = (l(off := x), h, r)
    using bp_eq blocks_eq AE_Left unfolding write_bp_def by simp
  show ?thesis
  proof (cases i < ?c)
    case True
    — i lands in the left block; offset is ?xs ! i.
    have idx_neq: ?xs ! i ≠ off
    proof
      assume eq: ?xs ! i = off
      from True len_eq have i_in: i < length ?xs by simp
      have c_idx (?xs ! i) = i using c_idx_eq True by simp
      with eq have c_idx off = i by simp
      with neq lin_eq show False by simp
    qed
    show ?thesis using True wb idx_neq blocks_eq
      unfolding buf_lin_at_def by simp
  next
    case False
    — i lands in home or right block; left-block update doesn't affect.
    show ?thesis using False wb blocks_eq
      unfolding buf_lin_at_def by simp
  qed
next
  case AE_Home
  have lin_eq: bp_linear bp = ?c + c_idx off
    using bp_eq AE_Home by simp
  have wb: write_bp blocks bp x = (l, h(off := x), r)
    using bp_eq blocks_eq AE_Home unfolding write_bp_def by simp
  show ?thesis
  proof (cases i < ?c)
    case True
    show ?thesis using True wb blocks_eq
      unfolding buf_lin_at_def by simp
  next
```

```
    case False
    hence ge_c: i ≥ ?c by simp
    show ?thesis
    proof (cases i < 2 * ?c)
      case True
      — home block; offset is ?xs ! (i − ?c).
      have sub_lt: i − ?c < ?c using True ge_c by simp
      have idx_neq: ?xs ! (i − ?c) ≠ off
      proof
        assume eq: ?xs ! (i − ?c) = off
        have c_idx (?xs ! (i − ?c)) = i − ?c
          using c_idx_eq sub_lt by simp
        with eq have c_idx off = i − ?c by simp
        with neq lin_eq ge_c show False by simp
      qed
      show ?thesis using True ge_c wb idx_neq blocks_eq
        unfolding buf_lin_at_def by simp
    next
      case False
      show ?thesis using False ge_c wb blocks_eq
        unfolding buf_lin_at_def by simp
    qed
  qed
next
  case AE_Right
  have lin_eq: bp_linear bp = 2 * ?c + c_idx off
    using bp_eq AE_Right by simp
  have wb: write_bp blocks bp x = (l, h, r(off := x))
    using bp_eq blocks_eq AE_Right unfolding write_bp_def by simp
  show ?thesis
  proof (cases i < 2 * ?c)
    case True
    show ?thesis using True wb blocks_eq
      unfolding buf_lin_at_def Let_def by simp
  next
    case False
    hence ge_2c: i ≥ 2 * ?c by simp
    — right block; offset is ?xs ! (i − 2 * ?c).
    have sub_lt: i − 2 * ?c < ?c using ge_2c i_lt by simp
    have idx_neq: ?xs ! (i − 2 * ?c) ≠ off
    proof
      assume eq: ?xs ! (i − 2 * ?c) = off
      have c_idx (?xs ! (i − 2 * ?c)) = i − 2 * ?c
        using c_idx_eq sub_lt by simp
      with eq have c_idx off = i − 2 * ?c by simp
      with neq lin_eq ge_2c show False by simp
    qed
    show ?thesis using False ge_2c wb idx_neq blocks_eq
      unfolding buf_lin_at_def Let_def by simp
```

```
  qed
 qed
qed

end
theory AlphabetEnlargement_Window
 imports AlphabetEnlargement_BufferArith
begin
```

2.17 Window-invariant single-step preservation

Steady-regime window-invariant preservation under one buffered M-step. Takes the IH-side invariant *ae_window_invariant tM nM bp blocks p_start*, the buffered head advance *bp_advance_le le (tM nM) bp d = Some bp'*, the position bounds excluding *bp_advance*'s None cases (*bp_linear < 3c − 1* for R, *0 < bp_linear* for L), and a no-LE-at-head fact (*tM nM ≠ le*) excluding *bp_advance_le*'s LE-jump. Produces the new invariant for the parallel tape fun-update and buffer *write_bp*.

In the steady regime (*pos ≥ 2*) the no-LE precondition is forced by the regime's no-LE window: the buffered head's *nM = p_start + bp_linear bp* lies inside the no-LE window [*p_start*, *p_start + 3c − 1*], so *tM nM ≠ le*.

Used by the Suc case of *ae_coupled_run_aux_general* to discharge conjunct 2 in the *pos ≥ 2* regime.

```
lemma ae_window_invariant_step:
 fixes le a' :: 'a
   and tM :: nat ⇒ 'a
   and nM p_start :: nat
   and bp bp' :: ('c :: enum) bp
   and blocks :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
   and d :: dir
 assumes wi:     ae_window_invariant tM nM bp blocks p_start
     and adv:    bp_advance_le le (tM nM) bp d = Some bp'
     and bp_lt:  bp_linear bp < 3 * card (UNIV :: 'c set) - 1
     and bp_pos: 0 < bp_linear bp
     and no_le:  tM nM ≠ le
 shows ae_window_invariant
         (tM(nM := a')) (go_dir d nM)
         bp' (write_bp blocks bp a') p_start
proof −
 let ?c = card (UNIV :: 'c set)
 from wi have p_start_ge: p_start ≥ 1
   unfolding ae_window_invariant_def by simp
 from wi have head_corr: nM = p_start + bp_linear bp
   unfolding ae_window_invariant_def by simp
 from wi have win_eq:
    ∀ i. i < 3 * ?c ⟶ tM (p_start + i) = buf_lin_at blocks i
   unfolding ae_window_invariant_def by simp
```

— The LE-jump in *bp_advance_le* doesn't fire because $tM\ nM \neq le$; so *adv* collapses to a plain *bp_advance*.
have *adv_plain*: *bp_advance bp d* = *Some bp′*
proof −
from *no_le* **have** ¬ (*tM nM* = *le* ∧ *fst bp* = *AE_Home* ∧ *d* = *dir.R*)
by *simp*
hence *bp_advance_le le* (*tM nM*) *bp d* = *bp_advance bp d*
unfolding *bp_advance_le_def* **by** *auto*
with *adv* **show** *?thesis* **by** *simp*
qed
— Head-position correspondence under the step: case-split on direction. N keeps *bp*, R increments *bp_linear*, L decrements (needs $nM \geq 1$, supplied by *p_start_ge*).
have *head_new*: *go_dir d nM* = *p_start* + *bp_linear bp′*
proof (*cases d*)
case *N*
from *adv_plain N* **have** *Some bp′* = *Some bp*
using *bp_advance_N_some*[*of bp*] **by** *simp*
hence *bp′_eq*: *bp′* = *bp* **by** *simp*
show *?thesis* **using** *N bp′_eq head_corr* **by** *simp*
next
case *R*
from *adv_plain R* **have** *adv_R*: *bp_advance bp dir.R* = *Some bp′* **by** *simp*
have *lin_R*: *bp_linear bp′* = *Suc* (*bp_linear bp*)
by (*rule bp_advance_R_linear*[*OF adv_R*])
show *?thesis* **using** *R lin_R head_corr* **by** *simp*
next
case *L*
from *adv_plain L* **have** *adv_L*: *bp_advance bp dir.L* = *Some bp′* **by** *simp*
have *lin_L*: *Suc* (*bp_linear bp′*) = *bp_linear bp*
by (*rule bp_advance_L_linear*[*OF adv_L*])
have *nM_form*: *nM* = *Suc* (*p_start* + *bp_linear bp′*)
using *head_corr lin_L* **by** *simp*
show *?thesis* **using** *L nM_form* **by** *simp*
qed
— Window-content alignment: the only changed cell in the M-tape (*nM*) corresponds to the only changed buffer cell (the linearised *bp_linear bp*), so the alignment is preserved by *buf_lin_at_write_bp_match* at the matching index and *buf_lin_at_write_bp_other* elsewhere.
have *win_new*:
∀ *i*. *i* < *3* ∗ *?c* ⟶
(*tM*(*nM* := *a′*)) (*p_start* + *i*)
= *buf_lin_at* (*write_bp blocks bp a′*) *i*
proof (*intro allI impI*)
fix *i*
assume *i_lt*: *i* < *3* ∗ *?c*
show (*tM*(*nM* := *a′*)) (*p_start* + *i*)
= *buf_lin_at* (*write_bp blocks bp a′*) *i*
proof (*cases i* = *bp_linear bp*)
case *True*

```
    have pos_eq: p_start + i = nM using True head_corr by simp
    have (tM(nM := a')) (p_start + i) = a' using pos_eq by simp
    moreover have buf_lin_at (write_bp blocks bp a') i = a'
      using True buf_lin_at_write_bp_match[of blocks bp a'] by simp
    ultimately show ?thesis by simp
   next
    case False
    hence pos_neq: p_start + i ≠ nM using head_corr by simp
    have (tM(nM := a')) (p_start + i) = tM (p_start + i)
      using pos_neq by simp
    also have ... = buf_lin_at blocks i using win_eq i_lt by blast
    also have ... = buf_lin_at (write_bp blocks bp a') i
      by (rule buf_lin_at_write_bp_other[OF False i_lt, symmetric])
    finally show ?thesis .
   qed
  qed
  show ?thesis
   unfolding ae_window_invariant_def
   using p_start_ge head_new win_new by simp
qed
```

Structural-case equivalence for the *le1* regime: the case-on-*fst bp* head-position constraint in *ae_window_invariant_le1* is equivalent to the single algebraic relation $Suc\ (bp_linear\ bp) = nM + c$.

Forward direction ($\Rightarrow$): case-split on *fst bp*, compute *bp_linear* from the case, conclude the algebraic relation.

Backward direction ($\Leftarrow$): given the algebraic relation, case-split on *fst bp* and recover the original case-clause from *bp_linear*'s known form per case (using *c_idx_last* for the *AE_Left* sub-case where the algebraic relation forces *snd bp = c_last*).

Used by *ae_window_invariant_le1_step* to reduce the structural-case preservation goal to a single algebraic check, avoiding a per-direction-per-block-boundary nested case analysis.

```
lemma bp_linear_le1_struct_iff:
  fixes bp :: ('c :: enum) bp
    and nM :: nat
  shows (case fst bp of
           AE_Left  ⇒ snd bp = c_last ∧ nM = 0
         | AE_Home  ⇒ nM = Suc (c_idx (snd bp))
         | AE_Right ⇒ nM = Suc (card (UNIV :: 'c set)
                                + c_idx (snd bp)))
       ⟷ Suc (bp_linear bp) = nM + card (UNIV :: 'c set)
proof −
  let ?c = card (UNIV :: 'c set)
  have idx_lt: c_idx (snd bp) < ?c by (rule c_idx_lt_card)
  show ?thesis
  proof (cases fst bp)
    case AE_Left
```

```
  have lin_eq: bp_linear bp = c_idx (snd bp)
    using AE_Left by (cases bp) simp
  show ?thesis
  proof
    assume case fst bp of
             AE_Left  ⇒ snd bp = c_last ∧ nM = 0
           | AE_Home  ⇒ nM = Suc (c_idx (snd bp))
           | AE_Right ⇒ nM = Suc (?c + c_idx (snd bp))
    with AE_Left have snd_eq: snd bp = c_last and nM_eq: nM = 0 by auto
    have c_idx (snd bp) = ?c − 1 using snd_eq c_idx_last by simp
    thus Suc (bp_linear bp) = nM + ?c
      using lin_eq nM_eq idx_lt by simp
  next
    assume Suc (bp_linear bp) = nM + ?c
    hence Suc (c_idx (snd bp)) = nM + ?c using lin_eq by simp
    hence nM_zero: nM = 0
      using idx_lt by linarith
    with ‹Suc (c_idx (snd bp)) = nM + ?c›
    have idx_eq: c_idx (snd bp) = ?c − 1 using idx_lt by simp
    have c_idx_last_eq: c_idx (c_last :: 'c) = ?c − 1
      by (rule c_idx_last)
    have idx_at: c_idx (snd bp) = c_idx (c_last :: 'c)
      using idx_eq c_idx_last_eq by simp
    hence snd_eq: snd bp = c_last
      by (metis c_idx_in_range(2))
    show case fst bp of
           AE_Left  ⇒ snd bp = c_last ∧ nM = 0
         | AE_Home  ⇒ nM = Suc (c_idx (snd bp))
         | AE_Right ⇒ nM = Suc (?c + c_idx (snd bp))
      using AE_Left snd_eq nM_zero by simp
  qed
next
  case AE_Home
  have lin_eq: bp_linear bp = ?c + c_idx (snd bp)
    using AE_Home by (cases bp) simp
  show ?thesis
  proof
    assume case fst bp of
             AE_Left  ⇒ snd bp = c_last ∧ nM = 0
           | AE_Home  ⇒ nM = Suc (c_idx (snd bp))
           | AE_Right ⇒ nM = Suc (?c + c_idx (snd bp))
    with AE_Home have nM_eq: nM = Suc (c_idx (snd bp)) by simp
    thus Suc (bp_linear bp) = nM + ?c using lin_eq by simp
  next
    assume Suc (bp_linear bp) = nM + ?c
    hence Suc (?c + c_idx (snd bp)) = nM + ?c using lin_eq by simp
    hence nM = Suc (c_idx (snd bp)) by simp
    thus case fst bp of
           AE_Left  ⇒ snd bp = c_last ∧ nM = 0
```

```
        | AE_Home  ⇒ nM = Suc (c_idx (snd bp))
        | AE_Right ⇒ nM = Suc (?c + c_idx (snd bp))
      using AE_Home by simp
  qed
 next
  case AE_Right
  have lin_eq: bp_linear bp = 2 * ?c + c_idx (snd bp)
   using AE_Right by (cases bp) simp
  show ?thesis
  proof
   assume case fst bp of
            AE_Left  ⇒ snd bp = c_last ∧ nM = 0
          | AE_Home  ⇒ nM = Suc (c_idx (snd bp))
          | AE_Right ⇒ nM = Suc (?c + c_idx (snd bp))
   with AE_Right have nM_eq: nM = Suc (?c + c_idx (snd bp)) by simp
   thus Suc (bp_linear bp) = nM + ?c using lin_eq by simp
  next
   assume Suc (bp_linear bp) = nM + ?c
   hence Suc (2 * ?c + c_idx (snd bp)) = nM + ?c using lin_eq by simp
   hence nM = Suc (?c + c_idx (snd bp)) by simp
   thus case fst bp of
          AE_Left  ⇒ snd bp = c_last ∧ nM = 0
        | AE_Home  ⇒ nM = Suc (c_idx (snd bp))
        | AE_Right ⇒ nM = Suc (?c + c_idx (snd bp))
     using AE_Right by simp
  qed
 qed
qed
```

le1-regime window-invariant preservation under one buffered M-step. Takes the IH-side invariant *ae_window_invariant_le1 tM nM bp blocks le*, the buffered head advance, the position bounds, and the substrate-derived facts on a' / d's response to the read symbol's LE-status: when reading *le*, M writes *le* and moves N or R (from *valid_mttm_deltaLE*); when reading non-*le*, M doesn't write *le* (from *valid_mttm_deltaLE_no_write*'s contrapositive). Also takes a no-LE-in-window precondition (from the IH-side *no_le* invariant restricted to the *le1* regime's window cells *1..2c*) which forces the LE-jump in *bp_advance_le* to never fire.

Used by the Suc case of *ae_coupled_run_aux_general* to discharge conjunct 2 in the *pos = 1* regime.

```
lemma ae_window_invariant_le1_step:
 fixes le a' :: 'a
   and tM :: nat ⇒ 'a
   and nM :: nat
   and bp bp' :: ('c :: enum) bp
   and blocks :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
   and d :: dir
 assumes wi:        ae_window_invariant_le1 tM nM bp blocks le
```

```
    and adv:       bp_advance_le le (tM nM) bp d = Some bp'
    and bp_lt:     bp_linear bp < 3 * card (UNIV :: 'c set) - 1
    and bp_pos:    0 < bp_linear bp
    and a_le:      tM nM = le ⟹ a' = le ∧ d ∈ {dir.N, dir.R}
    and a_not_le:  tM nM ≠ le ⟹ a' ≠ le
    and no_le_win: ∀ i. 0 < i ∧ i ≤ 2 * card (UNIV :: 'c set)
                        ⟶ tM i ≠ le
  shows ae_window_invariant_le1
          (tM(nM := a')) (go_dir d nM)
          bp' (write_bp blocks bp a') le
proof −
  let ?c = card (UNIV :: 'c set)
  from wi have l_LE: fst blocks = LE_block le
    unfolding ae_window_invariant_le1_def by simp
  from wi have tM_0: tM 0 = le
    unfolding ae_window_invariant_le1_def by simp
  have struct_lin: Suc (bp_linear bp) = nM + ?c
  proof −
    have (case fst bp of
            AE_Left  ⇒ snd bp = c_last ∧ nM = 0
          | AE_Home  ⇒ nM = Suc (c_idx (snd bp))
          | AE_Right ⇒ nM = Suc (?c + c_idx (snd bp)))
      using wi unfolding ae_window_invariant_le1_def by simp
    thus ?thesis
      using bp_linear_le1_struct_iff[where bp = bp and nM = nM]
      by blast
  qed
  from wi have align:
     ∀ i. i < 2 * ?c ⟶ tM (Suc i) = buf_lin_at blocks (?c + i)
    unfolding ae_window_invariant_le1_def by simp
  — LE-jump never fires in le1: at fst bp = AE_Home, nM lies in {1..c}, so tM
nM ≠ le by no_le_win.
  have not_le_jump: ¬ (tM nM = le ∧ fst bp = AE_Home ∧ d = dir.R)
  proof
    assume tM nM = le ∧ fst bp = AE_Home ∧ d = dir.R
    hence le_at: tM nM = le and fst_home: fst bp = AE_Home by auto
    have lin: bp_linear bp = ?c + c_idx (snd bp)
      using fst_home by (cases bp) simp
    from struct_lin lin have nM_eq: nM = Suc (c_idx (snd bp)) by simp
    have idx_lt: c_idx (snd bp) < ?c by (rule c_idx_lt_card)
    have nM_bd: 0 < nM ∧ nM ≤ ?c using nM_eq idx_lt by simp
    hence tM nM ≠ le using no_le_win by simp
    with le_at show False by simp
  qed
  have adv_plain: bp_advance bp d = Some bp'
  proof −
    have bp_advance_le le (tM nM) bp d = bp_advance bp d
      using not_le_jump unfolding bp_advance_le_def by auto
    with adv show ?thesis by simp
```

qed
— Algebraic post-step relation: *Suc (bp_linear bp′) = go_dir d nM + c*. By the equivalence *bp_linear_le1_struct_iff* this implies the structural case-on-*fst bp′* at *nM′ = go_dir d nM*.
have *struct_new*: *Suc (bp_linear bp′) = go_dir d nM + ?c*
proof (*cases d*)
case *N*
from *adv_plain N* **have** *Some bp′ = Some bp*
using *bp_advance_N_some*[*of bp*] **by** *simp*
hence *bp′_eq*: *bp′ = bp* **by** *simp*
show *?thesis* **using** *N bp′_eq struct_lin* **by** *simp*
next
case *R*
from *adv_plain R* **have** *adv_R*: *bp_advance bp dir.R = Some bp′* **by** *simp*
have *lin_R*: *bp_linear bp′ = Suc (bp_linear bp)*
by (*rule bp_advance_R_linear*[*OF adv_R*])
show *?thesis* **using** *R lin_R struct_lin* **by** *simp*
next
case *L*
from *adv_plain L* **have** *adv_L*: *bp_advance bp dir.L = Some bp′* **by** *simp*
have *lin_L*: *Suc (bp_linear bp′) = bp_linear bp*
by (*rule bp_advance_L_linear*[*OF adv_L*])
have *nM_pos*: *0 < nM*
proof (*rule ccontr*)
assume ¬ *0 < nM*
hence *nM = 0* **by** *simp*
hence *tM nM = le* **using** *tM_0* **by** *simp*
with *a_le* **have** *d_in*: *d ∈ {dir.N, dir.R}* **by** *blast*
with *L* **show** *False* **by** *simp*
qed
obtain *m* **where** *m_eq*: *nM = Suc m* **using** *nM_pos* **by** (*cases nM*) *auto*
have *Suc (Suc (bp_linear bp′)) = Suc (bp_linear bp)*
using *lin_L* **by** *simp*
also have *... = nM + ?c* **using** *struct_lin* **by** *simp*
also have *... = Suc (m + ?c)* **using** *m_eq* **by** *simp*
finally have *lhs_eq*: *Suc (bp_linear bp′) = m + ?c* **by** *simp*
have *rhs_eq*: *go_dir dir.L nM = m* **using** *m_eq* **by** *simp*
show *?thesis* **using** *L lhs_eq rhs_eq* **by** *simp*
qed
have *struct_post*:
(*case fst bp′ of*
AE_Left ⇒ *snd bp′ = c_last* ∧ *go_dir d nM = 0*
| *AE_Home* ⇒ *go_dir d nM = Suc (c_idx (snd bp′))*
| *AE_Right* ⇒ *go_dir d nM = Suc (?c + c_idx (snd bp′))*)
using *struct_new*
bp_linear_le1_struct_iff[**where** *bp = bp′* **and** *nM = go_dir d nM*]
by *blast*
— Window-content alignment over *{1..2c}* ↔ *{c..3c−1}*: the write lands at buffer index *bp_linear bp*; if that index lies in the alignment range (*?c + i*), the tape write

at *nM* matches; otherwise both sides unchanged.
have *align_new*:
$\forall i.\ i < 2 * ?c \longrightarrow$
$(tM(nM := a'))\ (Suc\ i)$
$= buf_lin_at\ (write_bp\ blocks\ bp\ a')\ (?c + i)$
proof (*intro allI impI*)
fix *i*
assume *i_lt*: $i < 2 * ?c$
show $(tM(nM := a'))\ (Suc\ i)$
$= buf_lin_at\ (write_bp\ blocks\ bp\ a')\ (?c + i)$
proof (*cases* $?c + i = bp_linear\ bp$)
case *True*
hence $Suc\ (?c + i) = Suc\ (bp_linear\ bp)$ **by** *simp*
hence $Suc\ i + ?c = nM + ?c$ **using** *struct_lin* **by** *simp*
hence *Suc_i_eq*: $Suc\ i = nM$ **by** *simp*
have *lhs*: $(tM(nM := a'))\ (Suc\ i) = a'$ **using** *Suc_i_eq* **by** *simp*
have *rhs*: $buf_lin_at\ (write_bp\ blocks\ bp\ a')\ (?c + i) = a'$
using *True buf_lin_at_write_bp_match*[*of blocks bp a'*] **by** *simp*
show *?thesis* **using** *lhs rhs* **by** *simp*
next
case *False*
hence *c_plus_i_neq*: $?c + i \neq bp_linear\ bp$ **by** *simp*
have *Suc_i_neq*: $Suc\ i \neq nM$
proof
assume $Suc\ i = nM$
hence $Suc\ i + ?c = nM + ?c$ **by** *simp*
hence $Suc\ (?c + i) = Suc\ (bp_linear\ bp)$ **using** *struct_lin* **by** *simp*
hence $?c + i = bp_linear\ bp$ **by** *simp*
with *c_plus_i_neq* **show** *False* **..**
qed
have *c_plus_i_lt*: $?c + i < 3 * ?c$ **using** *i_lt* **by** *simp*
have $(tM(nM := a'))\ (Suc\ i) = tM\ (Suc\ i)$ **using** *Suc_i_neq* **by** *simp*
also have $... = buf_lin_at\ blocks\ (?c + i)$ **using** *align i_lt* **by** *blast*
also have $... = buf_lin_at\ (write_bp\ blocks\ bp\ a')\ (?c + i)$
by (*rule buf_lin_at_write_bp_other*
[*OF c_plus_i_neq c_plus_i_lt*, *symmetric*])
finally show *?thesis* **.**
qed
qed
— First conjunct: $fst\ (new\ blocks) = LE_block\ le$. Write site case-split on *fst bp*: in *AE_Left* case (forced $snd\ bp = c_last$, reads *le*, writes *le*) the left slot stays $LE_block\ le$; in *AE_Home* / *AE_Right* cases the write doesn't touch the left slot.
have *fst_new*: $fst\ (write_bp\ blocks\ bp\ a') = LE_block\ le$
proof (*cases fst bp*)
case *AE_Left*
obtain $b'\ off$ **where** *bp_eq*: $bp = (b', off)$ **by** (*cases bp*)
have *b'_eq*: $b' = AE_Left$ **using** *AE_Left bp_eq* **by** *simp*
have *lin_eq*: $bp_linear\ bp = c_idx\ off$ **using** *bp_eq AE_Left* **by** *simp*
have *idx_lt*: $c_idx\ off < ?c$ **by** (*rule c_idx_lt_card*)

from *struct_lin lin_eq* **have** *suc_eq*: *Suc* (*c_idx off*) = *nM* + *?c* **by** *simp*
have *nM_zero*: *nM* = *0* **using** *suc_eq idx_lt* **by** *linarith*
have *idx_top*: *c_idx off* = *?c* − *1* **using** *suc_eq nM_zero* **by** *linarith*
have *c_idx_last_eq*: *c_idx* (*c_last* :: *'c*) = *?c* − *1* **by** (*rule c_idx_last*)
have *idx_at*: *c_idx off* = *c_idx* (*c_last* :: *'c*)
using *idx_top c_idx_last_eq* **by** *simp*
hence *off_eq*: *off* = *c_last* **by** (*metis c_idx_in_range*(*2*))
have *tM_nM_eq*: *tM nM* = *le* **using** *nM_zero tM_0* **by** *simp*
have *a'_eq*: *a'* = *le* **using** *a_le tM_nM_eq* **by** *blast*
obtain *l h r* **where** *blocks_eq*: *blocks* = (*l*, *h*, *r*)
by (*cases blocks*) *auto*
have *l_eq*: *l* = *LE_block le* **using** *l_LE blocks_eq* **by** *simp*
have *wb_eq*: *write_bp blocks bp a'* = (*l*(*off* := *a'*), *h*, *r*)
using *bp_eq b'_eq blocks_eq* **unfolding** *write_bp_def* **by** *simp*
have *l*(*off* := *a'*) = *LE_block le*
proof −
have *l*(*off* := *a'*) = (*LE_block le*)(*c_last* := *le*)
using *l_eq off_eq a'_eq* **by** *simp*
also have ... = *LE_block le* **unfolding** *LE_block_def* **by** *auto*
finally show *?thesis* **.**
qed
thus *?thesis* **using** *wb_eq* **by** *simp*
next
case *AE_Home*
obtain *b' off* **where** *bp_eq*: *bp* = (*b'*, *off*) **by** (*cases bp*)
have *b'_eq*: *b'* = *AE_Home* **using** *AE_Home bp_eq* **by** *simp*
obtain *l h r* **where** *blocks_eq*: *blocks* = (*l*, *h*, *r*)
by (*cases blocks*) *auto*
have *wb_eq*: *write_bp blocks bp a'* = (*l*, *h*(*off* := *a'*), *r*)
using *bp_eq b'_eq blocks_eq* **unfolding** *write_bp_def* **by** *simp*
show *?thesis* **using** *wb_eq l_LE blocks_eq* **by** *simp*
next
case *AE_Right*
obtain *b' off* **where** *bp_eq*: *bp* = (*b'*, *off*) **by** (*cases bp*)
have *b'_eq*: *b'* = *AE_Right* **using** *AE_Right bp_eq* **by** *simp*
obtain *l h r* **where** *blocks_eq*: *blocks* = (*l*, *h*, *r*)
by (*cases blocks*) *auto*
have *wb_eq*: *write_bp blocks bp a'* = (*l*, *h*, *r*(*off* := *a'*))
using *bp_eq b'_eq blocks_eq* **unfolding** *write_bp_def* **by** *simp*
show *?thesis* **using** *wb_eq l_LE blocks_eq* **by** *simp*
qed
— Second conjunct: (*new tM*) *0* = *le*. Either the fun-update doesn't touch cell 0 (*nM* ≠ *0*), or *nM* = *0* and *a'* = *le* by the LE-write discipline.
have *tM_0_new*: (*tM*(*nM* := *a'*)) *0* = *le*
proof (*cases nM* = *0*)
case *True*
have *tM nM* = *le* **using** *True tM_0* **by** *simp*
hence *a'* = *le* **using** *a_le* **by** *blast*
thus *?thesis* **using** *True* **by** *simp*

```
  next
    case False
    have (tM(nM := a')) 0 = tM 0 using False by simp
    thus ?thesis using tM_0 by simp
  qed
  show ?thesis
    unfolding ae_window_invariant_le1_def
    using fst_new tM_0_new struct_post align_new by simp
qed
```

le0-regime window-invariant preservation under one buffered M-step. Differs from *le1* in that:

· the LE block lives in the home slot, not the left slot; · *fst bp* = *AE_Left* is forbidden by the invariant; · *fst bp* = *AE_Home* always reads *le* (the whole home slot is *LE_block le*) so the LE-jump in *bp_advance_le* DOES fire on R from *AE_Home*; · the alignment is only over right-slot cells $\{1..c\} \leftrightarrow \{2c..3c-1\}$.

Structurally the proof splits on *fst bp*: *AE_Left* is closed by contradiction with the invariant; *AE_Home* case-splits on $d \in \{N, R\}$ (L forbidden by *valid_mttm_deltaLE*); *AE_Right* falls through to the standard *bp_advance* with displacement $\pm 1/0$ and the no-LE precondition rules out the LE-jump.

```
lemma ae_window_invariant_le0_step:
  fixes le a' :: 'a
    and tM :: nat ⇒ 'a
    and nM :: nat
    and bp bp' :: ('c :: enum) bp
    and blocks :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
    and d :: dir
  assumes wi:        ae_window_invariant_le0 tM nM bp blocks le
      and adv:       bp_advance_le le (tM nM) bp d = Some bp'
      and bp_lt:     bp_linear bp < 3 * card (UNIV :: 'c set) - 1
      and bp_pos:    0 < bp_linear bp
      and a_le:      tM nM = le ⟹ a' = le ∧ d ∈ {dir.N, dir.R}
      and a_not_le:  tM nM ≠ le ⟹ a' ≠ le
      and no_le_win: ∀ i. 0 < i ∧ i ≤ card (UNIV :: 'c set)
                          ⟶ tM i ≠ le
  shows ae_window_invariant_le0
          (tM(nM := a')) (go_dir d nM)
          bp' (write_bp blocks bp a') le
proof −
  let ?c = card (UNIV :: 'c set)
  from wi have h_LE: fst (snd blocks) = LE_block le
    unfolding ae_window_invariant_le0_def by simp
  from wi have tM_0: tM 0 = le
    unfolding ae_window_invariant_le0_def by simp
  from wi have align:
     ∀ i. i < ?c ⟶ tM (Suc i) = buf_lin_at blocks (2 * ?c + i)
    unfolding ae_window_invariant_le0_def by simp
```

```
  from wi have struct_pre:
     (case fst bp of
        AE_Home  ⇒ nM = 0
      | AE_Right ⇒ nM = Suc (c_idx (snd bp))
      | AE_Left  ⇒ False)
    unfolding ae_window_invariant_le0_def by simp
  have not_left: fst bp ≠ AE_Left
    using struct_pre by (cases fst bp) auto
  obtain l h r where blocks_eq: blocks = (l, h, r)
    by (cases blocks) auto
  have h_eq: h = LE_block le using h_LE blocks_eq by simp
  obtain b off where bp_eq: bp = (b, off) by (cases bp)
  show ?thesis
  proof (cases b)
    case AE_Left
    with bp_eq not_left show ?thesis by simp
  next
    case AE_Home
    have b_home: b = AE_Home using AE_Home .
    from struct_pre bp_eq AE_Home have nM_zero: nM = 0 by simp
    have tM_nM_le: tM nM = le using nM_zero tM_0 by simp
    from a_le tM_nM_le have a'_le: a' = le
      and d_in: d ∈ {dir.N, dir.R} by auto
    — Write-back: (LE_block le)(off := le) = LE_block le, so the home slot stays
LE_block le.
    have wb_eq: write_bp blocks bp a' = (l, h(off := a'), r)
      using bp_eq b_home blocks_eq unfolding write_bp_def by simp
    have h_new: h(off := a') = LE_block le
      using h_eq a'_le unfolding LE_block_def by auto
    have new_blocks: write_bp blocks bp a' = (l, LE_block le, r)
      using wb_eq h_new by simp
    — Tape: tM(0 := le) = tM.
    have tM_new: tM(nM := a') = tM
      using nM_zero tM_0 a'_le by auto
    show ?thesis
    proof (cases d)
      case N
      from adv N have bp_advance_le le (tM nM) bp dir.N = Some bp' by simp
      hence bp'_eq: bp' = bp
        unfolding bp_advance_le_def using bp_advance_N_some[of bp] by auto
      have go_dir_N: go_dir dir.N nM = nM by simp
      show ?thesis
        unfolding ae_window_invariant_le0_def
      proof (intro conjI)
        show fst (snd (write_bp blocks bp a')) = LE_block le
          using new_blocks by simp
        show (tM(nM := a')) 0 = le using tM_new tM_0 by simp
        show (case fst bp' of
                AE_Home  ⇒ go_dir d nM = 0
```

```
        | AE_Right ⇒ go_dir d nM = Suc (c_idx (snd bp'))
        | AE_Left  ⇒ False)
      using bp'_eq bp_eq b_home N nM_zero by simp
    show ∀ i. i < ?c ⟶
           (tM(nM := a')) (Suc i)
             = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
    proof (intro allI impI)
      fix i
      assume i_lt: i < ?c
      have lhs: (tM(nM := a')) (Suc i) = tM (Suc i)
        using tM_new by simp
      also have ... = buf_lin_at blocks (2 * ?c + i)
        using align i_lt by blast
      also have ... = buf_lin_at (l, LE_block le, r) (2 * ?c + i)
        using h_eq blocks_eq
        unfolding buf_lin_at_def by simp
      also have ... = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
        using new_blocks by simp
      finally show (tM(nM := a')) (Suc i)
                     = buf_lin_at (write_bp blocks bp a') (2 * ?c + i) .
    qed
  qed
next
  case R
  — LE-jump fires: bp' = (AE_Right, c_first), go_dir R 0 = 1 = Suc (c_idx
c_first).
  from adv R tM_nM_le b_home bp_eq
  have bp'_eq: bp' = (AE_Right, c_first)
    unfolding bp_advance_le_def by simp
  have nM'_eq: go_dir dir.R nM = 1 using nM_zero by simp
  have c_idx_first_eq: c_idx (c_first :: 'c) = 0
    by (rule c_idx_first)
  show ?thesis
    unfolding ae_window_invariant_le0_def
  proof (intro conjI)
    show fst (snd (write_bp blocks bp a')) = LE_block le
      using new_blocks by simp
    show (tM(nM := a')) 0 = le using tM_new tM_0 by simp
    show (case fst bp' of
          AE_Home  ⇒ go_dir d nM = 0
        | AE_Right ⇒ go_dir d nM = Suc (c_idx (snd bp'))
        | AE_Left  ⇒ False)
      using bp'_eq R nM'_eq c_idx_first_eq by simp
    show ∀ i. i < ?c ⟶
           (tM(nM := a')) (Suc i)
             = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
    proof (intro allI impI)
      fix i
      assume i_lt: i < ?c
```

```
      have (tM(nM := a')) (Suc i) = tM (Suc i) using tM_new by simp
      also have ... = buf_lin_at blocks (2 * ?c + i)
        using align i_lt by blast
      also have ... = buf_lin_at (l, LE_block le, r) (2 * ?c + i)
        using h_eq blocks_eq unfolding buf_lin_at_def by simp
      also have ... = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
        using new_blocks by simp
      finally show (tM(nM := a')) (Suc i)
                   = buf_lin_at (write_bp blocks bp a') (2 * ?c + i) .
    qed
   qed
  next
   case L
   with d_in show ?thesis by simp
  qed
 next
  case AE_Right
  have b_right: b = AE_Right using AE_Right .
  from struct_pre bp_eq AE_Right
  have nM_form: nM = Suc (c_idx off) by simp
  have idx_lt: c_idx off < ?c by (rule c_idx_lt_card)
  have nM_bd: 0 < nM ∧ nM ≤ ?c using nM_form idx_lt by simp
  have tM_nM_ne: tM nM ≠ le using nM_bd no_le_win by simp
  have a'_ne: a' ≠ le using a_not_le tM_nM_ne by blast
  — LE-jump doesn't fire (fst bp = AE_Right ≠ AE_Home). So bp_advance_le
= bp_advance.
  have adv_plain: bp_advance bp d = Some bp'
  proof −
   have ¬ (tM nM = le ∧ fst bp = AE_Home ∧ d = dir.R)
    using bp_eq b_right by simp
   hence bp_advance_le le (tM nM) bp d = bp_advance bp d
    unfolding bp_advance_le_def by auto
   with adv show ?thesis by simp
  qed
  — Write-back: writes to right slot, home and left unchanged.
  have wb_eq: write_bp blocks bp a' = (l, h, r(off := a'))
   using bp_eq b_right blocks_eq unfolding write_bp_def by simp
  have new_h: fst (snd (write_bp blocks bp a')) = LE_block le
   using wb_eq h_eq by simp
  — Tape: cell 0 unchanged (nM ≥ 1).
  have tM_0_new: (tM(nM := a')) 0 = le
   using nM_bd tM_0 by simp
  — Alignment under the parallel write: the only updated buffer cell is at index 2c
+ c_idx off = bp_linear bp; the tape update at nM = Suc (c_idx off) corresponds.
  have bp_lin: bp_linear bp = 2 * ?c + c_idx off
   using bp_eq b_right by simp
  show ?thesis
  proof (cases d)
   case N
```

```
from adv_plain N have Some bp' = Some bp
  using bp_advance_N_some[of bp] by simp
hence bp'_eq: bp' = bp by simp
show ?thesis
  unfolding ae_window_invariant_le0_def
proof (intro conjI)
  show fst (snd (write_bp blocks bp a')) = LE_block le
    by (rule new_h)
  show (tM(nM := a')) 0 = le by (rule tM_0_new)
  show (case fst bp' of
          AE_Home  ⇒ go_dir d nM = 0
        | AE_Right ⇒ go_dir d nM = Suc (c_idx (snd bp'))
        | AE_Left  ⇒ False)
    using bp'_eq bp_eq b_right N nM_form by simp
  show ∀ i. i < ?c ⟶
           (tM(nM := a')) (Suc i)
             = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
  proof (intro allI impI)
    fix i
    assume i_lt: i < ?c
    show (tM(nM := a')) (Suc i)
           = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
    proof (cases i = c_idx off)
      case True
      have Suc i = nM using True nM_form by simp
      hence lhs: (tM(nM := a')) (Suc i) = a' by simp
      have idx_eq: 2 * ?c + i = bp_linear bp
        using True bp_lin by simp
      have rhs: buf_lin_at (write_bp blocks bp a') (2 * ?c + i) = a'
        using idx_eq buf_lin_at_write_bp_match[of blocks bp a'] by simp
      show ?thesis using lhs rhs by simp
    next
      case False
      have idx_neq: 2 * ?c + i ≠ bp_linear bp
        using False bp_lin by simp
      have idx_2c_lt: 2 * ?c + i < 3 * ?c using i_lt by simp
      have Suc_i_neq: Suc i ≠ nM
        using False nM_form by simp
      have (tM(nM := a')) (Suc i) = tM (Suc i)
        using Suc_i_neq by simp
      also have ... = buf_lin_at blocks (2 * ?c + i)
        using align i_lt by blast
      also have ... = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
        by (rule buf_lin_at_write_bp_other
                 [OF idx_neq idx_2c_lt, symmetric])
      finally show ?thesis .
    qed
  qed
qed
```

```
  next
    case R
    from adv_plain R have adv_R: bp_advance bp dir.R = Some bp' by simp
    have lin_R: bp_linear bp' = Suc (bp_linear bp)
      by (rule bp_advance_R_linear[OF adv_R])
    — Either bp' = (AE_Right, ofs') with c_idx ofs' = Suc (c_idx off), or bp' =
  None via the AE_Right R-boundary (c_succ off = None), which contradicts adv.
  Hence c_succ off = Some ofs'.
    obtain ofs' where c_succ_off: c_succ off = Some ofs'
    proof (cases c_succ off)
      case None
      have bp_advance bp dir.R = None
        using bp_eq b_right None unfolding bp_advance_def by simp
      with adv_R show thesis by simp
    next
      case (Some ofs')
      thus thesis using that by simp
    qed
    have bp'_eq: bp' = (AE_Right, ofs')
      using bp_eq b_right c_succ_off adv_R
      unfolding bp_advance_def by simp
    have idx_succ: c_idx ofs' = Suc (c_idx off)
      by (rule c_succ_idx_some[OF c_succ_off])
    have nM_succ: go_dir dir.R nM = Suc (c_idx ofs')
      using nM_form idx_succ by simp
    show ?thesis
      unfolding ae_window_invariant_le0_def
    proof (intro conjI)
      show fst (snd (write_bp blocks bp a')) = LE_block le
        by (rule new_h)
      show (tM(nM := a')) 0 = le by (rule tM_0_new)
      show (case fst bp' of
              AE_Home  ⇒ go_dir d nM = 0
            | AE_Right ⇒ go_dir d nM = Suc (c_idx (snd bp'))
            | AE_Left  ⇒ False)
        using bp'_eq R nM_succ by simp
      show ∀ i. i < ?c ⟶
              (tM(nM := a')) (Suc i)
                = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
      proof (intro allI impI)
        fix i
        assume i_lt: i < ?c
        show (tM(nM := a')) (Suc i)
              = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
        proof (cases i = c_idx off)
          case True
          have Suc i = nM using True nM_form by simp
          hence lhs: (tM(nM := a')) (Suc i) = a' by simp
          have idx_eq: 2 * ?c + i = bp_linear bp
```

```
        using True bp_lin by simp
      have rhs: buf_lin_at (write_bp blocks bp a') (2 * ?c + i) = a'
        using idx_eq buf_lin_at_write_bp_match[of blocks bp a'] by simp
      show ?thesis using lhs rhs by simp
    next
      case False
      have idx_neq: 2 * ?c + i ≠ bp_linear bp
        using False bp_lin by simp
      have idx_2c_lt: 2 * ?c + i < 3 * ?c using i_lt by simp
      have Suc_i_neq: Suc i ≠ nM
        using False nM_form by simp
      have (tM(nM := a')) (Suc i) = tM (Suc i)
        using Suc_i_neq by simp
      also have ... = buf_lin_at blocks (2 * ?c + i)
        using align i_lt by blast
      also have ... = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
        by (rule buf_lin_at_write_bp_other
              [OF idx_neq idx_2c_lt, symmetric])
      finally show ?thesis .
    qed
  qed
qed
next
  case L
  from adv_plain L have adv_L: bp_advance bp dir.L = Some bp' by simp
  have lin_L: Suc (bp_linear bp') = bp_linear bp
    by (rule bp_advance_L_linear[OF adv_L])
  — Two sub-cases: c_pred off = Some ofs' gives bp' = (AE_Right, ofs'); c_pred
off = None (off = c_first) gives bp' = (AE_Home, c_last).
  show ?thesis
  proof (cases c_pred off)
    case (Some ofs')
    have bp'_eq: bp' = (AE_Right, ofs')
      using bp_eq b_right Some adv_L
      unfolding bp_advance_def by simp
    have idx_pred: Suc (c_idx ofs') = c_idx off
      by (rule c_pred_idx_some[OF Some])
    obtain m where nM_eq: nM = Suc m
      using nM_bd by (cases nM) auto
    have m_eq: m = c_idx off using nM_form nM_eq by simp
    have nM'_eq: go_dir dir.L nM = Suc (c_idx ofs')
      using nM_eq m_eq idx_pred by simp
    show ?thesis
      unfolding ae_window_invariant_le0_def
    proof (intro conjI)
      show fst (snd (write_bp blocks bp a')) = LE_block le
        by (rule new_h)
      show (tM(nM := a')) 0 = le by (rule tM_0_new)
      show (case fst bp' of
```

```
          AE_Home  ⇒ go_dir d nM = 0
        | AE_Right ⇒ go_dir d nM = Suc (c_idx (snd bp'))
        | AE_Left  ⇒ False)
    using bp'_eq L nM'_eq by simp
  show ∀ i. i < ?c ⟶
          (tM(nM := a')) (Suc i)
            = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
  proof (intro allI impI)
    fix i
    assume i_lt: i < ?c
    show (tM(nM := a')) (Suc i)
          = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
    proof (cases i = c_idx off)
      case True
      have Suc i = nM using True nM_form by simp
      hence lhs: (tM(nM := a')) (Suc i) = a' by simp
      have idx_eq: 2 * ?c + i = bp_linear bp
        using True bp_lin by simp
      have rhs: buf_lin_at (write_bp blocks bp a') (2 * ?c + i) = a'
        using idx_eq buf_lin_at_write_bp_match[of blocks bp a'] by simp
      show ?thesis using lhs rhs by simp
    next
      case False
      have idx_neq: 2 * ?c + i ≠ bp_linear bp
        using False bp_lin by simp
      have idx_2c_lt: 2 * ?c + i < 3 * ?c using i_lt by simp
      have Suc_i_neq: Suc i ≠ nM
        using False nM_form by simp
      have (tM(nM := a')) (Suc i) = tM (Suc i)
        using Suc_i_neq by simp
      also have ... = buf_lin_at blocks (2 * ?c + i)
        using align i_lt by blast
      also have ... = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
        by (rule buf_lin_at_write_bp_other
              [OF idx_neq idx_2c_lt, symmetric])
      finally show ?thesis .
    qed
  qed
qed
next
  case None
  have idx_zero: c_idx off = 0
    using c_pred_idx_none[OF None] by simp
  have bp'_eq: bp' = (AE_Home, c_last)
    using bp_eq b_right None adv_L
    unfolding bp_advance_def by simp
  have nM_one: nM = 1 using nM_form idx_zero by simp
  have nM'_eq: go_dir dir.L nM = 0 using nM_one by simp
  show ?thesis
```

```
      unfolding ae_window_invariant_le0_def
    proof (intro conjI)
      show fst (snd (write_bp blocks bp a')) = LE_block le
        by (rule new_h)
      show (tM(nM := a')) 0 = le by (rule tM_0_new)
      show (case fst bp' of
              AE_Home  ⇒ go_dir d nM = 0
            | AE_Right ⇒ go_dir d nM = Suc (c_idx (snd bp'))
            | AE_Left  ⇒ False)
        using bp'_eq L nM'_eq by simp
      show ∀ i. i < ?c ⟶
              (tM(nM := a')) (Suc i)
                = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
      proof (intro allI impI)
        fix i
        assume i_lt: i < ?c
        show (tM(nM := a')) (Suc i)
              = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
        proof (cases i = c_idx off)
          case True
          have Suc i = nM using True nM_form by simp
          hence lhs: (tM(nM := a')) (Suc i) = a' by simp
          have idx_eq: 2 * ?c + i = bp_linear bp
            using True bp_lin by simp
          have rhs: buf_lin_at (write_bp blocks bp a') (2 * ?c + i) = a'
            using idx_eq buf_lin_at_write_bp_match[of blocks bp a'] by simp
          show ?thesis using lhs rhs by simp
        next
          case False
          have idx_neq: 2 * ?c + i ≠ bp_linear bp
            using False bp_lin by simp
          have idx_2c_lt: 2 * ?c + i < 3 * ?c using i_lt by simp
          have Suc_i_neq: Suc i ≠ nM
            using False nM_form by simp
          have (tM(nM := a')) (Suc i) = tM (Suc i)
            using Suc_i_neq by simp
          also have ... = buf_lin_at blocks (2 * ?c + i)
            using align i_lt by blast
          also have ... = buf_lin_at (write_bp blocks bp a') (2 * ?c + i)
            by (rule buf_lin_at_write_bp_other
                  [OF idx_neq idx_2c_lt, symmetric])
          finally show ?thesis .
        qed
      qed
    qed
  qed
 qed
qed
```

end
theory *AlphabetEnlargement_Codec*
imports *AlphabetEnlargement_Window*
begin

2.18 Encoder and decoder structural lemmas

2.18.1 Decoder structural lemmas and ceiling-div arithmetic

Decoder structural lemmas. *ae_decode_block_pure* resolves to the full enum image (length c); the analogous fact for padded blocks (length = the padded witness) splits the enum list at the witness index and uses the takeWhile / append decomposition.

lemma *ae_decode_block_pure*:
fixes $f :: 'c :: enum \Rightarrow 'a$
assumes *is_pure_block bl f*
shows *ae_decode_block bl f*
= *map f* (*enum_class.enum* :: *'c list*)
proof −
have *all_neq*: $\forall a \in set\ (map\ f\ (enum_class.enum :: 'c\ list)).\ a \neq bl$
using *assms* **unfolding** *is_pure_block_def* **by** *auto*
show *?thesis*
unfolding *ae_decode_block_def*
using *all_neq* **by** (*simp add*: *takeWhile_eq_all_conv*)
qed

lemma *length_ae_decode_block_pure*:
fixes $f :: 'c :: enum \Rightarrow 'a$
assumes *is_pure_block bl f*
shows *length* (*ae_decode_block bl f*) = *card* (*UNIV* :: *'c set*)
using *ae_decode_block_pure*[*OF assms*] *card_eq_length_enum*[*symmetric*]
by *simp*

lemma *ae_decode_block_padded*:
fixes $f :: 'c :: enum \Rightarrow 'a$
assumes *k_lt*: $k < length\ (enum_class.enum :: 'c\ list)$
and *prefix*: $\forall x.\ c_idx\ x < k \longrightarrow f\ x \neq bl$
and *suffix*: $\forall x.\ k \leq c_idx\ x \longrightarrow f\ x = bl$
shows *ae_decode_block bl f*
= *map f* (*take k* (*enum_class.enum* :: *'c list*))
proof −
let *?xs* = *enum_class.enum* :: *'c list*
let *?P* = $\lambda a.\ a \neq bl$
have *at_k*: $f\ (?xs\ !\ k) = bl$
proof −
have $c_idx\ (?xs\ !\ k) = k$ **using** *k_lt* **by** (*rule c_idx_enum_nth*)
thus *?thesis* **using** *suffix* **by** *simp*
qed

```
  have prefix_neq: ∀ a ∈ set (map f (take k ?xs)). ?P a
  proof
    fix a assume a_in: a ∈ set (map f (take k ?xs))
    then obtain i where i_lt: i < length (take k ?xs)
      and a_eq: a = map f (take k ?xs) ! i
      by (auto simp: in_set_conv_nth)
    from i_lt have i_lt_k: i < k using k_lt by simp
    hence i_lt_n: i < length ?xs using k_lt by linarith
    have a_alt: a = f (?xs ! i)
      using a_eq i_lt_k k_lt by (simp add: nth_map nth_take)
    have c_idx_at: c_idx (?xs ! i) = i using i_lt_n by (rule c_idx_enum_nth)
    show ?P a using prefix c_idx_at i_lt_k a_alt by simp
  qed
  have xs_split: ?xs = take k ?xs @ ?xs ! k # drop (Suc k) ?xs
    using k_lt by (rule id_take_nth_drop)
  have map_split: map f ?xs = map f (take k ?xs) @ f (?xs ! k) # map f (drop
(Suc k) ?xs)
  proof −
    have map f ?xs = map f (take k ?xs @ ?xs ! k # drop (Suc k) ?xs)
      using xs_split by simp
    thus ?thesis by simp
  qed
  have step1: takeWhile ?P (map f ?xs)
                = map f (take k ?xs)
                    @ takeWhile ?P (f (?xs ! k) # map f (drop (Suc k) ?xs))
  proof −
    have takeWhile ?P (map f ?xs)
          = takeWhile ?P (map f (take k ?xs)
                            @ f (?xs ! k) # map f (drop (Suc k) ?xs))
      using map_split by simp
    also have ... = map f (take k ?xs)
                    @ takeWhile ?P (f (?xs ! k) # map f (drop (Suc k) ?xs))
      using prefix_neq by (subst takeWhile_append) auto
    finally show ?thesis .
  qed
  have step2: takeWhile ?P (f (?xs ! k) # map f (drop (Suc k) ?xs)) = []
    using at_k by simp
  show ?thesis
    unfolding ae_decode_block_def
    using step1 step2 by simp
qed
```

Decoder structural lemmas (continued). *ae_decode_input* is compositional under list append; the length-of-decode for an all-pure prefix is exactly *length* · *c*.

```
lemma ae_decode_input_append:
  ae_decode_input bl (xs @ ys)
    = ae_decode_input bl xs @ ae_decode_input bl ys
  by (simp add: ae_decode_input_def)
```

```
lemma length_ae_decode_input_pure:
  fixes xs :: ('c :: enum ⇒ 'a) list
  assumes ∀ s < length xs. is_pure_block bl (xs ! s)
  shows length (ae_decode_input bl xs) = length xs * card (UNIV :: 'c set)
  using assms
proof (induction xs)
  case Nil
  show ?case by (simp add: ae_decode_input_def)
next
  case (Cons x xs)
  have x_pure: is_pure_block bl x
    using Cons.prems[rule_format, of 0] by simp
  have rest_pure: ∀ s < length xs. is_pure_block bl (xs ! s)
  proof (intro allI impI)
    fix s assume s < length xs
    hence Suc s < length (x # xs) by simp
    thus is_pure_block bl (xs ! s)
      using Cons.prems[rule_format, of Suc s] by simp
  qed
  have length (ae_decode_input bl (x # xs))
        = length (ae_decode_block bl x) + length (ae_decode_input bl xs)
    by (simp add: ae_decode_input_def)
  also have ... = card (UNIV :: 'c set) + length xs * card (UNIV :: 'c set)
    using length_ae_decode_block_pure[OF x_pure] Cons.IH[OF rest_pure]
    by simp
  also have ... = length (x # xs) * card (UNIV :: 'c set)
    by simp
  finally show ?case .
qed
```

Ceiling-division arithmetic. Used by the encoder/decoder round-trip lemmas below.

```
lemma ceil_div_mult_c:
  fixes m c :: nat
  assumes 0 < c
  shows (m * c + c - 1) div c = m
  using assms
proof (induction m)
  case 0
  show ?case using ‘0 < c‘ by simp
next
  case (Suc m)
  have c_ne: c ≠ 0 using ‘0 < c‘ by simp
  have eq: Suc m * c + c - 1 = (m * c + c - 1) + c
    using ‘0 < c‘ by simp
  have step: ((m * c + c - 1) + c) div c = (m * c + c - 1) div c + 1
    by (rule div_add_self2[OF c_ne])
  have ih: (m * c + c - 1) div c = m using Suc.IH[OF ‘0 < c‘] .
```

```
  have (Suc m * c + c - 1) div c = ((m * c + c - 1) + c) div c
    using eq by (rule arg_cong[where f = λx. x div c])
  also have ... = (m * c + c - 1) div c + 1 using step .
  also have ... = m + 1 using ih by simp
  finally show ?case by simp
qed

lemma ceil_div_2c_minus_1:
  fixes c :: nat
  assumes 0 < c
  shows (c + c - 1) div c = 1
proof -
  have c + c - 1 = 1 * c + c - 1 by simp
  thus ?thesis using ceil_div_mult_c[OF assms, of 1] by simp
qed

lemma ceil_div_k_plus_c_minus_1:
  fixes k c :: nat
  assumes 0 < c 0 < k k ≤ c
  shows (k + c - 1) div c = 1
proof -
  have lo: k + c - 1 ≥ c using assms by simp
  have hi: k + c - 1 < c + c
    using assms by linarith
  have (k + c - 1) div c = ((k + c - 1) - c) div c + 1
    using lo assms by (auto simp: le_div_geq)
  also have (k + c - 1) - c = k - 1 by simp
  also have (k - 1) div c = 0 using assms by simp
  finally show ?thesis by simp
qed
```

2.18.2 Encoder list operations and image properties

Encoding an input that is the image of a block function under the canonical enumeration yields the singleton list containing that block. Used by the round-trip lemma in the forward direction of *ae_validation_canonical_iff_encoder_image*: a pure last-block decodes to its full enum image, and that image re-encodes back to the original block.

```
lemma encode_input_map_enum:
  fixes f :: 'c :: enum ⇒ 'a
  shows encode_input bl (map f (enum_class.enum :: 'c list))
        = ([f] :: ('c ⇒ 'a) list)
proof -
  let ?c = card (UNIV :: 'c set)
  let ?xs = enum_class.enum :: 'c list
  let ?w = map f ?xs
  let ?lhs = encode_input bl ?w :: ('c ⇒ 'a) list
  have len_xs: length ?xs = ?c
```

```
    using card_eq_length_enum[symmetric] .
  have len_w: length ?w = ?c using len_xs by simp
  have c_pos: 0 < ?c
  proof −
    have (c_first :: 'c) ∈ UNIV by simp
    thus ?thesis by (simp add: card_gt_0_iff)
  qed
  have len_lhs: length ?lhs = 1
  proof −
    have length ?lhs = (length ?w + ?c − 1) div ?c
      using length_encode_input[where bl = bl and u = ?w] .
    also have ... = (?c + ?c − 1) div ?c using len_w by simp
    also have ... = 1 using ceil_div_2c_minus_1[OF c_pos] .
    finally show ?thesis .
  qed
  have zero_lt_lhs: 0 < length ?lhs using len_lhs by simp
  have nth0_at: ⋀x :: 'c. (?lhs ! 0) x = f x
  proof −
    fix x :: 'c
    have idx_lt_xs: c_idx x < length ?xs
      using c_idx_in_range(1) .
    have idx_lt_w: c_idx x < length ?w using idx_lt_xs by simp
    have nth_step: (?lhs ! 0) x =
                   (let j = 0 * ?c + c_idx x
                      in if j < length ?w then ?w ! j else bl)
      by (rule encode_input_nth[OF zero_lt_lhs])
    have w_at: ?w ! c_idx x = f x
      using idx_lt_xs c_idx_in_range(2)[of x] by (simp add: nth_map)
    show (?lhs ! 0) x = f x
      using nth_step idx_lt_w w_at by (simp add: Let_def)
  qed
  have nth0: ?lhs ! 0 = f
    using nth0_at by (rule ext)
  show ?thesis
  proof (rule nth_equalityI)
    show length ?lhs = length ([f] :: ('c ⇒ 'a) list)
      using len_lhs by simp
  next
    fix i assume i_lt: i < length ?lhs
    hence i = 0 using len_lhs by simp
    thus ?lhs ! i = ([f] :: ('c ⇒ 'a) list) ! i
      using nth0 by simp
  qed
qed
```

Generalisation of *encode_input_map_enum*: encoding the image of f over a length-m prefix of the canonical enumeration (where $0 < m \leq c$, and $f\ x = bl$ on the out-of-prefix indices) yields the singleton list $[f]$. Used by the round-trip in the forward direction of

ae_validation_canonical_iff_encoder_image: a padded last block decodes to its enum-prefix, and that prefix re-encodes back to the original block.

```
lemma encode_input_map_take_enum:
  fixes f :: 'c :: enum ⇒ 'a
  assumes m_pos: 0 < m
      and m_le: m ≤ length (enum_class.enum :: 'c list)
      and pad: ∀ x. c_idx x ≥ m ⟶ f x = bl
  shows encode_input bl (map f (take m (enum_class.enum :: 'c list)))
          = ([f] :: ('c ⇒ 'a) list)
proof −
  let ?c = card (UNIV :: 'c set)
  let ?xs = enum_class.enum :: 'c list
  let ?w = map f (take m ?xs)
  let ?lhs = encode_input bl ?w :: ('c ⇒ 'a) list
  have len_xs: length ?xs = ?c using card_eq_length_enum[symmetric] .
  have c_pos: 0 < ?c
  proof −
    have (c_first :: 'c) ∈ UNIV by simp
    thus ?thesis by (simp add: card_gt_0_iff)
  qed
  have m_le_c: m ≤ ?c using m_le len_xs by simp
  have len_take: length (take m ?xs) = m
    using m_le by simp
  have len_w: length ?w = m using len_take by simp
  have len_lhs: length ?lhs = 1
  proof −
    have length ?lhs = (length ?w + ?c − 1) div ?c
      using length_encode_input[where bl = bl and u = ?w] .
    also have ... = (m + ?c − 1) div ?c using len_w by simp
    also have ... = 1
      using ceil_div_k_plus_c_minus_1[OF c_pos m_pos m_le_c] .
    finally show ?thesis .
  qed
  have zero_lt_lhs: 0 < length ?lhs using len_lhs by simp
  have nth0_at: ⋀x :: 'c. (?lhs ! 0) x = f x
  proof −
    fix x :: 'c
    have idx_lt_xs: c_idx x < length ?xs using c_idx_in_range(1) .
    have nth_step: (?lhs ! 0) x =
                     (let j = 0 * ?c + c_idx x
                        in if j < length ?w then ?w ! j else bl)
      by (rule encode_input_nth[OF zero_lt_lhs])
    show (?lhs ! 0) x = f x
    proof (cases c_idx x < m)
      case True
      hence j_lt_w: c_idx x < length ?w using len_w by simp
      have w_at: ?w ! c_idx x = f x
      proof −
        have take_at: (take m ?xs) ! c_idx x = ?xs ! c_idx x
```

```
      using True by simp
    have map_at: ?w ! c_idx x = f ((take m ?xs) ! c_idx x)
      using True len_take by (simp add: nth_map)
    have ?w ! c_idx x = f (?xs ! c_idx x)
      using map_at take_at by simp
    also have ?xs ! c_idx x = x using c_idx_in_range(2)[of x] .
    finally show ?thesis .
  qed
  show ?thesis using nth_step j_lt_w w_at by (simp add: Let_def)
 next
  case False
  hence m_le_idx: m ≤ c_idx x by simp
  have f_eq_bl: f x = bl using pad m_le_idx by simp
  have body_bl: (let j = 0 * ?c + c_idx x
                 in if j < length ?w then ?w ! j else bl) = bl
    using m_le_idx m_le len_w by (auto simp: Let_def)
  have lhs_bl: (?lhs ! 0) x = bl
    using nth_step body_bl by simp
  show ?thesis using lhs_bl f_eq_bl by simp
 qed
qed
have nth0: ?lhs ! 0 = f using nth0_at by (rule ext)
show ?thesis
proof (rule nth_equalityI)
 show length ?lhs = length ([f] :: ('c ⇒ 'a) list)
  using len_lhs by simp
next
 fix i assume i_lt: i < length ?lhs
 hence i = 0 using len_lhs by simp
 thus ?lhs ! i = ([f] :: ('c ⇒ 'a) list) ! i
  using nth0 by simp
qed
qed
```

The encoder splits over an append whenever the prefix's length is a multiple of c: each block of *encode_input bl* (*u1* @ *u2*) falls entirely within *u1* or entirely within *u2*, so the encoded list is the concatenation of the per-half encodings. Used by the round-trip's *rev_induct*: when peeling off the last block of a well-formed w, the prefix's decode is all-pure (length divisible by c), allowing the encoder to split.

```
lemma encode_input_append_div:
 fixes u1 u2 :: 'a list
 assumes c_div: card (UNIV :: 'c :: enum set) dvd length u1
 shows (encode_input bl (u1 @ u2) :: ('c ⇒ 'a) list)
       = encode_input bl u1 @ encode_input bl u2
proof −
 let ?c = card (UNIV :: 'c set)
 let ?lhs = encode_input bl (u1 @ u2) :: ('c ⇒ 'a) list
 let ?rhs1 = encode_input bl u1 :: ('c ⇒ 'a) list
```

```
let ?rhs2 = encode_input bl u2 :: ('c ⇒ 'a) list
have c_pos: 0 < ?c
proof −
  have (c_first :: 'c) ∈ UNIV by simp
  thus ?thesis by (simp add: card_gt_0_iff)
qed
obtain n where n_eq: length u1 = n ∗ ?c
  using c_div by (auto simp: dvd_def mult.commute)
have len_rhs1: length ?rhs1 = n
proof −
  have length ?rhs1 = (length u1 + ?c − 1) div ?c
    using length_encode_input[where bl = bl and u = u1] .
  also have ... = (n ∗ ?c + ?c − 1) div ?c using n_eq by simp
  also have ... = n using ceil_div_mult_c[OF c_pos] .
  finally show ?thesis .
qed
have len_rhs2: length ?rhs2 = (length u2 + ?c − 1) div ?c
  using length_encode_input[where bl = bl and u = u2] .
have len_lhs: length ?lhs = n + length ?rhs2
proof −
  have rearr: n ∗ ?c + length u2 + ?c − 1
              = (length u2 + ?c − 1) + n ∗ ?c
    using c_pos by arith
  have c_ne: ?c ≠ 0 using c_pos by simp
  have length ?lhs = (length u1 + length u2 + ?c − 1) div ?c
    using length_encode_input[where bl = bl and u = u1 @ u2] by simp
  also have ... = (n ∗ ?c + length u2 + ?c − 1) div ?c
    using n_eq by simp
  also have ... = ((length u2 + ?c − 1) + n ∗ ?c) div ?c
    using rearr by simp
  also have ... = n + (length u2 + ?c − 1) div ?c
    using c_ne by (simp add: div_mult_self1)
  also have ... = n + length ?rhs2 using len_rhs2 by simp
  finally show ?thesis .
qed
have len_eq: length ?lhs = length (?rhs1 @ ?rhs2)
  using len_rhs1 len_lhs by simp
show ?thesis
proof (rule nth_equalityI)
  show length ?lhs = length (?rhs1 @ ?rhs2) using len_eq .
next
  fix i assume i_lt: i < length ?lhs
  have lhs_step: ⋀x :: 'c. (?lhs ! i) x =
                    (let j = i ∗ ?c + c_idx x
                       in if j < length (u1 @ u2)
                            then (u1 @ u2) ! j else bl)
    using encode_input_nth[OF i_lt] by simp
  show ?lhs ! i = (?rhs1 @ ?rhs2) ! i
  proof (cases i < n)
```

```
  case True
  have i_lt_rhs1: i < length ?rhs1 using True len_rhs1 by simp
  have rhs_split: (?rhs1 @ ?rhs2) ! i = ?rhs1 ! i
    using i_lt_rhs1 by (simp add: nth_append)
  have rhs1_step: ⋀x :: 'c. (?rhs1 ! i) x =
                    (let j = i * ?c + c_idx x
                       in if j < length u1 then u1 ! j else bl)
    using encode_input_nth[OF i_lt_rhs1] by simp
  show ?thesis
  proof (rule ext)
    fix x :: 'c
    have idx_lt_c: c_idx x < ?c using c_idx_lt_card .
    have j_lt_u1: i * ?c + c_idx x < length u1
    proof −
      have step1: i + 1 ≤ n using True by simp
      hence step2: (i + 1) * ?c ≤ n * ?c by (rule mult_le_mono1)
      hence step3: i * ?c + ?c ≤ n * ?c
        by (simp add: distrib_right)
      have i * ?c + c_idx x < i * ?c + ?c using idx_lt_c by simp
      also have ... ≤ n * ?c using step3 .
      also have ... = length u1 using n_eq by simp
      finally show ?thesis .
    qed
    have j_lt_lhs: i * ?c + c_idx x < length (u1 @ u2)
      using j_lt_u1 by simp
    have lhs_at: (?lhs ! i) x = u1 ! (i * ?c + c_idx x)
    proof −
      have (?lhs ! i) x = (u1 @ u2) ! (i * ?c + c_idx x)
        using lhs_step[of x] j_lt_lhs by (simp add: Let_def)
      also have ... = u1 ! (i * ?c + c_idx x)
        using j_lt_u1 by (simp add: nth_append)
      finally show ?thesis .
    qed
    have rhs_at: (?rhs1 ! i) x = u1 ! (i * ?c + c_idx x)
      using rhs1_step[of x] j_lt_u1 by (simp add: Let_def)
    show (?lhs ! i) x = ((?rhs1 @ ?rhs2) ! i) x
      using lhs_at rhs_at rhs_split by simp
  qed
next
  case False
  hence n_le_i: n ≤ i by simp
  define i' where i' = i − n
  have i_eq: i = i' + n using n_le_i unfolding i'_def by simp
  have i_mul: i * ?c = i' * ?c + n * ?c
    using i_eq by (simp add: distrib_right)
  have len_rhs1_le: length ?rhs1 ≤ i
    using n_le_i len_rhs1 by simp
  have rhs2_idx: i − length ?rhs1 = i'
    using i'_def len_rhs1 by simp
```

```
    have i'_lt_rhs2: i' < length ?rhs2
      using i_lt len_lhs n_le_i unfolding i'_def by linarith
    have rhs_split: (?rhs1 @ ?rhs2) ! i = ?rhs2 ! i'
      using len_rhs1_le rhs2_idx by (simp add: nth_append)
    have rhs2_step: ⋀x :: 'c. (?rhs2 ! i') x =
                      (let j' = i' * ?c + c_idx x
                         in if j' < length u2 then u2 ! j' else bl)
      using encode_input_nth[OF i'_lt_rhs2] by simp
    show ?thesis
    proof (rule ext)
      fix x :: 'c
      have idx_lt_c: c_idx x < ?c using c_idx_lt_card .
      have j_ge_u1: length u1 ≤ i * ?c + c_idx x
        using i_mul n_eq n_le_i by simp
      have j_diff: i * ?c + c_idx x - length u1 = i' * ?c + c_idx x
        using i_mul n_eq by simp
      have lhs_lt_iff: (i * ?c + c_idx x < length (u1 @ u2))
                         ⟷ (i' * ?c + c_idx x < length u2)
        using i_mul n_eq by simp
      show (?lhs ! i) x = ((?rhs1 @ ?rhs2) ! i) x
      proof (cases i' * ?c + c_idx x < length u2)
        case True
        hence j_lt_lhs: i * ?c + c_idx x < length (u1 @ u2)
          using lhs_lt_iff by simp
        have lhs_at: (?lhs ! i) x = u2 ! (i' * ?c + c_idx x)
        proof -
          have (?lhs ! i) x = (u1 @ u2) ! (i * ?c + c_idx x)
            using lhs_step[of x] j_lt_lhs by (simp add: Let_def)
          also have ... = u2 ! (i * ?c + c_idx x - length u1)
            using j_ge_u1 by (simp add: nth_append)
          also have ... = u2 ! (i' * ?c + c_idx x) using j_diff by simp
          finally show ?thesis .
        qed
        have rhs2_at: (?rhs2 ! i') x = u2 ! (i' * ?c + c_idx x)
          using rhs2_step[of x] True by (simp add: Let_def)
        show ?thesis using lhs_at rhs2_at rhs_split by simp
      next
        case False
        hence j_ge_lhs: ¬ (i * ?c + c_idx x < length (u1 @ u2))
          using lhs_lt_iff by simp
        have lhs_at: (?lhs ! i) x = bl
          using lhs_step[of x] j_ge_lhs by (simp add: Let_def)
        have rhs2_at: (?rhs2 ! i') x = bl
          using rhs2_step[of x] False by (simp add: Let_def)
        show ?thesis using lhs_at rhs2_at rhs_split by simp
      qed
    qed
  qed
qed
```

qed

Encoding the empty input gives the empty block list. Trivial corollary of *length_encode_input*: $(0 + c - 1)$ *div* $c = 0$ regardless of c's value (zero in nat division when $c = 0$; zero by *div_less* when $c > 0$).

```
lemma encode_input_empty:
  (encode_input bl ([] :: 'a list) :: ('c :: enum ⇒ 'a) list) = []
proof −
  have c_pos: 0 < card (UNIV :: 'c set)
  proof −
    have (c_first :: 'c) ∈ UNIV by simp
    thus ?thesis by (simp add: card_gt_0_iff)
  qed
  have div_zero: (card (UNIV :: 'c set) − Suc 0)
                   div card (UNIV :: 'c set) = 0
    using c_pos by (simp add: div_less)
  have length (encode_input bl ([] :: 'a list)
                 :: ('c ⇒ 'a) list)
         = (0 + card (UNIV :: 'c set) − 1)
             div card (UNIV :: 'c set)
    using length_encode_input[where bl = bl and u = [] :: 'a list]
    by simp
  also have ... = 0 using div_zero by simp
  finally have length (encode_input bl ([] :: 'a list)
                         :: ('c ⇒ 'a) list) = 0 .
  thus ?thesis by simp
qed
```

If a snoc-list is well-formed, every cell of the prefix is pure: well-formedness allows the *last* cell to be padded, but the prefix's last cell is no longer last after the snoc, so it must be pure. Bridges the *rev_induct* step in *encode_decode_round_trip*: the IH applies to the prefix only if the prefix is itself well-formed (which follows from per-cell purity).

```
lemma ae_input_well_formed_init_pure:
  fixes xs :: ('c :: enum ⇒ 'a) list
    and y :: 'c ⇒ 'a
  assumes ae_input_well_formed bl (xs @ [y])
  shows ∀ i < length xs. is_pure_block bl (xs ! i)
proof (intro allI impI)
  fix i assume i_lt_xs: i < length xs
  have i_lt: i < length (xs @ [y]) using i_lt_xs by simp
  have not_last: i ≠ length (xs @ [y]) − 1 using i_lt_xs by simp
  have at_i: is_pure_block bl ((xs @ [y]) ! i)
               ∨ (i = length (xs @ [y]) − 1
                  ∧ is_padded_block bl ((xs @ [y]) ! i))
    using assms i_lt unfolding ae_input_well_formed_def by blast
  hence pure: is_pure_block bl ((xs @ [y]) ! i) using not_last by blast
  have nth_eq: (xs @ [y]) ! i = xs ! i
    using i_lt_xs by (simp add: nth_append)
```

```
  show is_pure_block bl (xs ! i) using pure nth_eq by simp
qed
```

Encoder image lies inside *gamma_block* ($\Sigma_M \cup \{bl_M\}$): every cell of every block in *encode_input bl_M u* is either an input symbol from u (in Σ_M) or the blank *bl_M*.

```
lemma encode_input_in_gamma_block:
  fixes M :: ('q, 'a) mttm
    and u :: 'a list
  assumes u_sub: set u ⊆ Sigma_tm M
  shows set (encode_input (bl_tm M) u :: ('c :: enum ⇒ 'a) list)
        ⊆ gamma_block (Sigma_tm M ∪ {bl_tm M})
proof
  fix f :: 'c ⇒ 'a
  assume f_in: f ∈ set (encode_input (bl_tm M) u :: ('c ⇒ 'a) list)
  let ?c = card (UNIV :: 'c set)
  from f_in obtain i where
    i_lt: i < (length u + ?c − 1) div ?c
    and f_eq: f = (λx. let j = i * ?c + c_idx x in
                       if j < length u then u ! j else bl_tm M)
    unfolding encode_input_def Let_def by auto
  have range f ⊆ Sigma_tm M ∪ {bl_tm M}
  proof
    fix v
    assume v ∈ range f
    then obtain x :: 'c where v_eq: v = f x by auto
    show v ∈ Sigma_tm M ∪ {bl_tm M}
    proof (cases i * ?c + c_idx x < length u)
      case True
      hence fx: f x = u ! (i * ?c + c_idx x)
        using f_eq by (simp add: Let_def)
      have u ! (i * ?c + c_idx x) ∈ set u using True by simp
      hence u ! (i * ?c + c_idx x) ∈ Sigma_tm M using u_sub by auto
      thus ?thesis using v_eq fx by auto
    next
      case False
      hence f x = bl_tm M using f_eq by (simp add: Let_def)
      thus ?thesis using v_eq by auto
    qed
  qed
  thus f ∈ gamma_block (Sigma_tm M ∪ {bl_tm M})
    unfolding gamma_block_def by simp
qed
```

The all-blank block never appears in *encode_input bl_M u*: the cell at the canonical zero offset (*c_idx x = 0*) always lies in the input range and so is in Σ_M, not *bl_M*. Discharges the *bl_block bl_M* $\notin$ *set w* hypothesis for upstream call sites that feed an encoded word into the noncanonical / steps-bound lemmas.

```
lemma encode_input_no_bl_block:
  fixes M :: ('q, 'a) mttm
    and u :: 'a list
  assumes vM: valid_mttm M
      and u_sub: set u ⊆ Sigma_tm M
  shows bl_block (bl_tm M)
          ∉ set (encode_input (bl_tm M) u :: ('c :: enum ⇒ 'a) list)
proof
  let ?c = card (UNIV :: 'c set)
  let ?N = (length u + ?c − 1) div ?c
  let ?xs = enum_class.enum :: 'c list
  let ?x0 = ?xs ! 0
  assume bl_in: bl_block (bl_tm M)
                  ∈ set (encode_input (bl_tm M) u :: ('c ⇒ 'a) list)
  from bl_in obtain i where
    i_lt: i < ?N
    and f_eq: (bl_block (bl_tm M) :: 'c ⇒ 'a)
                = (λx. let j = i * ?c + c_idx x in
                       if j < length u then u ! j else bl_tm M)
    unfolding encode_input_def Let_def by auto
  have c_pos: 0 < ?c
  proof −
    have (c_first :: 'c) ∈ UNIV by simp
    thus ?thesis by (simp add: card_gt_0_iff)
  qed
  have N_pos: 0 < ?N using i_lt by simp
  have u_nonempty: 0 < length u
  proof (rule ccontr)
    assume ¬ 0 < length u
    hence length u = 0 by simp
    hence ?N = (?c − 1) div ?c by simp
    also have ... = 0 using c_pos by simp
    finally have ?N = 0 .
    thus False using N_pos by simp
  qed
  have N_times_c_le: ?N * ?c ≤ length u + ?c − 1
    using c_pos by (metis div_times_less_eq_dividend)
  have ic_lt: i * ?c < length u
  proof −
    have suc_le: Suc i ≤ ?N using i_lt by simp
    have Suc i * ?c ≤ ?N * ?c by (rule mult_le_mono1[OF suc_le])
    hence i * ?c + ?c ≤ ?N * ?c by simp
    also have ... ≤ length u + ?c − 1 by (rule N_times_c_le)
    finally have i * ?c + ?c ≤ length u + ?c − 1 .
    thus ?thesis using c_pos u_nonempty by linarith
  qed
  have len_xs_eq: length ?xs = ?c
    using card_eq_length_enum[where 'c = 'c] by simp
  have len_xs: 0 < length ?xs using c_pos len_xs_eq by simp
```

```
  have x0_idx: c_idx ?x0 = 0
    using c_idx_enum_nth[OF len_xs] .
  have eval_f: (λx. let j = i * ?c + c_idx x in
                      if j < length u then u ! j else bl_tm M) ?x0
                = u ! (i * ?c)
    using ic_lt x0_idx by (simp add: Let_def)
  have eval_bl: (bl_block (bl_tm M) :: 'c ⇒ 'a) ?x0 = bl_tm M
    unfolding bl_block_def by simp
  have u ! (i * ?c) ∈ set u using ic_lt by simp
  hence u ! (i * ?c) ∈ Sigma_tm M using u_sub by auto
  hence u ! (i * ?c) ≠ bl_tm M
    using bl_tm_notin_Sigma_tm[OF vM] by auto
  thus False using f_eq eval_f eval_bl by metis
qed
```

Encoder output excludes the LE-block. Mirror of *encode_input_no_bl_block*: the first filled position $i \cdot c$ of any block-index $i < \lceil |u|/c \rceil$ is strictly less than $|u|$, so the block's value at *c_first* equals $u\ !\ (i \cdot c) \in \Sigma_M$; since $le_tm\ M \notin \Sigma_tm\ M$ (substrate's *valid_mttm_LE_not_Sigma*), that value differs from $le_tm\ M$, so the block is not the constant-*le* function. Used by *alphabet_enlarge_language* to show *set* (*encode_input* ...) ⊆ *Sigma_tm* (*alphabet_enlarge M*).

```
lemma encode_input_no_LE_block:
  fixes M :: ('q, 'a) mttm
    and u :: 'a list
  assumes vM: valid_mttm M
      and u_sub: set u ⊆ Sigma_tm M
  shows LE_block (le_tm M)
          ∉ set (encode_input (bl_tm M) u :: ('c :: enum ⇒ 'a) list)
proof
  let ?c = card (UNIV :: 'c set)
  let ?N = (length u + ?c − 1) div ?c
  let ?xs = enum_class.enum :: 'c list
  let ?x0 = ?xs ! 0
  assume LE_in: LE_block (le_tm M)
                  ∈ set (encode_input (bl_tm M) u :: ('c ⇒ 'a) list)
  from LE_in obtain i where
    i_lt: i < ?N
    and f_eq: (LE_block (le_tm M) :: 'c ⇒ 'a)
                = (λx. let j = i * ?c + c_idx x in
                         if j < length u then u ! j else bl_tm M)
    unfolding encode_input_def Let_def by auto
  have c_pos: 0 < ?c
  proof −
    have (c_first :: 'c) ∈ UNIV by simp
    thus ?thesis by (simp add: card_gt_0_iff)
  qed
  have N_pos: 0 < ?N using i_lt by simp
  have u_nonempty: 0 < length u
```

```
  proof (rule ccontr)
    assume ¬ 0 < length u
    hence length u = 0 by simp
    hence ?N = (?c − 1) div ?c by simp
    also have ... = 0 using c_pos by simp
    finally have ?N = 0 .
    thus False using N_pos by simp
  qed
  have N_times_c_le: ?N * ?c ≤ length u + ?c − 1
    using c_pos by (metis div_times_less_eq_dividend)
  have ic_lt: i * ?c < length u
  proof −
    have suc_le: Suc i ≤ ?N using i_lt by simp
    have Suc i * ?c ≤ ?N * ?c by (rule mult_le_mono1[OF suc_le])
    hence i * ?c + ?c ≤ ?N * ?c by simp
    also have ... ≤ length u + ?c − 1 by (rule N_times_c_le)
    finally have i * ?c + ?c ≤ length u + ?c − 1 .
    thus ?thesis using c_pos u_nonempty by linarith
  qed
  have len_xs_eq: length ?xs = ?c
    using card_eq_length_enum[where 'c = 'c] by simp
  have len_xs: 0 < length ?xs using c_pos len_xs_eq by simp
  have x0_idx: c_idx ?x0 = 0
    using c_idx_enum_nth[OF len_xs] .
  have eval_f: (λx. let j = i * ?c + c_idx x in
                    if j < length u then u ! j else bl_tm M) ?x0
              = u ! (i * ?c)
    using ic_lt x0_idx by (simp add: Let_def)
  have eval_LE: (LE_block (le_tm M) :: 'c ⇒ 'a) ?x0 = le_tm M
    unfolding LE_block_def by simp
  have u ! (i * ?c) ∈ set u using ic_lt by simp
  hence u ! (i * ?c) ∈ Sigma_tm M using u_sub by auto
  hence u ! (i * ?c) ≠ le_tm M
    using valid_mttm_LE_not_Sigma[OF vM] by auto
  thus False using f_eq eval_f eval_LE by metis
qed
```

2.18.3 Encoder well-formedness and round-trip

Encoder output is well-formed: every block except possibly the last is pure; the last is either pure (when *length u* divides evenly) or trailing-padded (when it doesn't). Discharges the *ae_input_well_formed* precondition for upstream call sites that feed an encoded word into the steps-bound lemma.

This is the forward arm of the characterisation biconditional *ae_validation_canonical_iff_encoder_image* below: every encoder image is well-formed. The reverse arm of that biconditional every well-formed AE-input is in the encoder image is not consumed by the headline language theorems (which are quantified only over explicit encoder-image inputs) and

is retained there as a structural completeness result.

```
lemma encode_input_well_formed:
  fixes M :: ('q, 'a) mttm
    and u :: 'a list
  assumes vM: valid_mttm M
      and u_sub: set u ⊆ Sigma_tm M
  shows ae_input_well_formed (bl_tm M)
          (encode_input (bl_tm M) u :: ('c :: enum ⇒ 'a) list)
proof −
  let ?c = card (UNIV :: 'c set)
  let ?N = (length u + ?c − 1) div ?c
  let ?xs = enum_class.enum :: 'c list
  let ?w = encode_input (bl_tm M) u :: ('c ⇒ 'a) list
  have len_w: length ?w = ?N by (rule length_encode_input)
  have c_pos: 0 < ?c
  proof −
    have (c_first :: 'c) ∈ UNIV by simp
    thus ?thesis by (simp add: card_gt_0_iff)
  qed
  have len_xs: length ?xs = ?c
    using card_eq_length_enum[where 'c = 'c] by simp
  have bl_notin_Sigma: bl_tm M ∉ Sigma_tm M
    using bl_tm_notin_Sigma_tm[OF vM] .
  show ?thesis
    unfolding ae_input_well_formed_def
  proof (intro allI impI)
    fix s :: nat
    assume s_lt: s < length ?w
    hence s_lt_N: s < ?N using len_w by simp
    have at_s: ?w ! s
                = (λx. let j = s * ?c + c_idx x in
                       if j < length u then u ! j else bl_tm M)
    proof −
      have s_lt_range: s < length [0 ..< ?N] using s_lt_N by simp
      have nth_range: [0 ..< ?N] ! s = s using s_lt_N by simp
      have map_unfold:
        ?w = map (λi. (λx. let j = i * ?c + c_idx x in
                           if j < length u then u ! j
                           else bl_tm M))
                 [0 ..< ?N]
        unfolding encode_input_def Let_def by simp
      show ?thesis
        using nth_map[OF s_lt_range,
                      where f = λi. (λx. let j = i * ?c + c_idx x in
                                         if j < length u then u ! j
                                         else bl_tm M)]
              nth_range map_unfold
        by simp
    qed
```

— For non-last blocks, every cell maps into the input range, hence the block is pure. For the last block, either the input divides evenly (block is pure) or there is a trailing-padded prefix.

consider (*non_last*) *Suc s* < *?N*
| (*last*) *Suc s* = *?N*
using *s_lt_N* **by** *linarith*
thus *is_pure_block* (*bl_tm M*) (*?w* ! *s*)
∨ (*s* = *length ?w* − *1* ∧ *is_padded_block* (*bl_tm M*) (*?w* ! *s*))
proof *cases*
case *non_last*
have *all_pure*: ∀ *x* :: *'c*. (*?w* ! *s*) *x* ≠ *bl_tm M*
proof
fix *x* :: *'c*
have *idx_lt_c*: *c_idx x* < *?c* **by** (*rule c_idx_lt_card*)
— For *Suc s* < *?N*, (*s* + *2*) ∗ *c* ≤ *?N* ∗ *c* ≤ *length u* + *c* − *1*, so (*s* + *1*) ∗ *c* < *length u*, hence *s* ∗ *c* + *c_idx x* < (*s* + *1*) ∗ *c* < *length u*.
have *suc2*: *Suc* (*Suc s*) ≤ *?N* **using** *non_last* **by** *simp*
have *Suc* (*Suc s*) ∗ *?c* ≤ *?N* ∗ *?c*
by (*rule mult_le_mono1*[*OF suc2*])
also have ... ≤ *length u* + *?c* − *1*
using *c_pos* **by** (*metis div_times_less_eq_dividend*)
finally have *suc2_bound*: *Suc* (*Suc s*) ∗ *?c* ≤ *length u* + *?c* − *1* **.**
have *prod_lt*: *s* ∗ *?c* + *c_idx x* < *length u*
using *suc2_bound idx_lt_c c_pos* **by** *simp*
hence (*?w* ! *s*) *x* = *u* ! (*s* ∗ *?c* + *c_idx x*)
using *at_s* **by** (*simp add*: *Let_def*)
moreover have *u* ! (*s* ∗ *?c* + *c_idx x*) ∈ *set u*
using *prod_lt* **by** *simp*
ultimately have (*?w* ! *s*) *x* ∈ *Sigma_tm M* **using** *u_sub* **by** *auto*
thus (*?w* ! *s*) *x* ≠ *bl_tm M* **using** *bl_notin_Sigma* **by** *auto*
qed
hence *is_pure_block* (*bl_tm M*) (*?w* ! *s*)
unfolding *is_pure_block_def* **by** *simp*
thus *?thesis* **..**
next
case *last*
— Last block: split on whether *length u* is a multiple of *?c*. Even ⟶ pure; uneven ⟶ padded with witness *k* = *length u mod ?c*.
have *s_eq*: *s* = *length ?w* − *1*
using *last len_w* **by** *simp*
have *s_eq_pred*: *s* = *?N* − *1* **using** *last* **by** *simp*
have *N_pos*: *1* ≤ *?N* **using** *last* **by** *simp*
have *s_times_c*: *s* ∗ *?c* = *?N* ∗ *?c* − *?c*
proof −
have *s* ∗ *?c* = (*?N* − *1*) ∗ *?c* **using** *s_eq_pred* **by** *simp*
also have ... = *?N* ∗ *?c* − *1* ∗ *?c*
by (*rule diff_mult_distrib*)
also have ... = *?N* ∗ *?c* − *?c* **by** *simp*
finally show *?thesis* **.**

```
    qed
    show ?thesis
    proof (cases length u mod ?c = 0)
      case True
      — Pure case: ?N * ?c = length u, so every cell in block s lands in the input range.
      have N_eq: ?N * ?c = length u
      proof −
        let ?q = length u div ?c
        have c_neq_0: ?c ≠ 0 using c_pos by simp
        have lu_eq: length u = ?q * ?c
          using True
          by (metis div_mult_mod_eq add.right_neutral)
        have c_minus_1_lt: ?c − 1 < ?c using c_pos by simp
        have div_minus: (?c − 1) div ?c = 0
          using c_minus_1_lt by simp
        have add_assoc: length u + ?c − 1 = length u + (?c − 1)
          using c_pos by simp
        have ?N = (length u + (?c − 1)) div ?c
          using add_assoc by simp
        also have ... = (?q * ?c + (?c − 1)) div ?c
          using lu_eq by simp
        also have ... = ?q + (?c − 1) div ?c
          by simp
        also have ... = ?q using div_minus by simp
        finally have ?N = ?q .
        thus ?thesis using lu_eq by simp
      qed
      have all_pure: ∀ x :: 'c. (?w ! s) x ≠ bl_tm M
      proof
        fix x :: 'c
        have idx_lt_c: c_idx x < ?c by (rule c_idx_lt_card)
        have N_pos_times: ?c ≤ ?N * ?c
          using N_pos by simp
        have prod_lt_N: s * ?c + c_idx x < ?N * ?c
          using s_times_c idx_lt_c c_pos N_pos_times by linarith
        hence prod_lt: s * ?c + c_idx x < length u using N_eq by simp
        hence (?w ! s) x = u ! (s * ?c + c_idx x)
          using at_s by (simp add: Let_def)
        moreover have u ! (s * ?c + c_idx x) ∈ set u
          using prod_lt by simp
        ultimately have (?w ! s) x ∈ Sigma_tm M using u_sub by auto
        thus (?w ! s) x ≠ bl_tm M using bl_notin_Sigma by auto
      qed
      hence is_pure_block (bl_tm M) (?w ! s)
        unfolding is_pure_block_def by simp
      thus ?thesis ..
    next
      case False
```

— Padded case: take the witness k = *length u mod ?c*. Decompose *length* $u = q \cdot c + k$ with $1 \le k < c$; then block $s = q$ has $s \cdot c + j$ = *length u* $- k + j$, which lies in the input range iff $j < k$.

let *?k = length u mod ?c*
let *?q = length u div ?c*
have *k_pos*: *1 ≤ ?k* **using** *False* **by** *simp*
have *k_lt_c*: *?k < ?c* **using** *c_pos* **by** *simp*
have *k_lt_xs*: *?k < length ?xs* **using** *k_lt_c len_xs* **by** *simp*
have *lu_eq*: *length u = ?q * ?c + ?k*
by *simp*
have *N_eq*: *?N = ?q + 1*
proof −
have *plus_eq*: *length u + ?c − 1 = ?q * ?c + (?k + ?c − 1)*
using *lu_eq k_pos* **by** *linarith*
have *N_step*: *?N = (?q * ?c + (?k + ?c − 1)) div ?c*
using *plus_eq* **by** *simp*
have *div_step*: *(?q * ?c + (?k + ?c − 1)) div ?c*
= ?q + (?k + ?c − 1) div ?c
by *simp*
have *k_plus_c_minus_1_lt*: *?k + ?c − 1 < 2 * ?c*
using *k_lt_c c_pos* **by** *linarith*
have *k_plus_c_minus_1_ge*: *?c ≤ ?k + ?c − 1*
using *k_pos c_pos* **by** *linarith*
have *div_kpc_eq_1*: *(?k + ?c − 1) div ?c = 1*
proof −
have *rewrite*: *?k + ?c − 1 = (?k − 1) + ?c*
using *k_pos c_pos* **by** *linarith*
have *small*: *(?k − 1) div ?c = 0*
using *k_lt_c* **by** *simp*
have *c_neq_0*: *?c ≠ 0* **using** *c_pos* **by** *simp*
have *((?k − 1) + ?c) div ?c = (?k − 1) div ?c + 1*
by (*rule div_add_self2*[*OF c_neq_0*])
also have *... = 1* **using** *small* **by** *simp*
finally show *?thesis* **using** *rewrite* **by** *simp*
qed
show *?thesis* **using** *N_step div_step div_kpc_eq_1* **by** *simp*
qed
have *s_eq_q*: *s = ?q* **using** *last N_eq* **by** *simp*
have *s_times_c_eq*: *s * ?c = length u − ?k*
proof −
have *q_eq*: *length u div ?c * ?c = length u − length u mod ?c*
by (*metis add_diff_cancel_right' div_mod_decomp*)
show *?thesis* **using** *s_eq_q q_eq* **by** *simp*
qed
have *padded*: *is_padded_block* (*bl_tm M*) (*?w ! s*)
unfolding *is_padded_block_def*
proof (*intro exI*[*of _ ?k*] *conjI*)
show *1 ≤ ?k* **by** (*rule k_pos*)
show *?k < length ?xs* **by** (*rule k_lt_xs*)

```
        show ∀x :: 'c. c_idx x < ?k ⟶ (?w ! s) x ≠ bl_tm M
        proof (intro allI impI)
          fix x :: 'c
          assume idx_lt_k: c_idx x < ?k
          have k_le_lu: ?k ≤ length u
            using lu_eq by linarith
          have prod_lt: s * ?c + c_idx x < length u
            using s_times_c_eq idx_lt_k k_le_lu by linarith
          hence (?w ! s) x = u ! (s * ?c + c_idx x)
            using at_s by (simp add: Let_def)
          moreover have u ! (s * ?c + c_idx x) ∈ set u
            using prod_lt by simp
          ultimately have (?w ! s) x ∈ Sigma_tm M
            using u_sub by auto
          thus (?w ! s) x ≠ bl_tm M using bl_notin_Sigma by auto
        qed
        show ∀x :: 'c. ?k ≤ c_idx x ⟶ (?w ! s) x = bl_tm M
        proof (intro allI impI)
          fix x :: 'c
          assume idx_ge_k: ?k ≤ c_idx x
          have s * ?c + c_idx x ≥ length u
            using s_times_c_eq idx_ge_k by linarith
          hence ¬ s * ?c + c_idx x < length u by simp
          thus (?w ! s) x = bl_tm M
            using at_s by (simp add: Let_def)
        qed
      qed
      thus ?thesis using s_eq by simp
    qed
  qed
 qed
qed
```

Encoder ∘ decoder is the identity on a single canonical block: pure cell → *encode_input bl* (*map x enum*) = [*x*]; padded cell with witness *k* → *encode_input bl* (*map x* (*take k enum*)) = [*x*]. Per-cell identity for the round-trip.

```
lemma encode_decode_block:
  fixes x :: 'c :: enum ⇒ 'a
  assumes is_canonical_block bl x
  shows encode_input bl (ae_decode_block bl x)
        = ([x] :: ('c ⇒ 'a) list)
proof (cases is_pure_block bl x)
  case True
  have decode: ae_decode_block bl x
               = map x (enum_class.enum :: 'c list)
    by (rule ae_decode_block_pure[OF True])
  show ?thesis using decode encode_input_map_enum by simp
next
```

```
  case False
  hence padded: is_padded_block bl x
    using assms unfolding is_canonical_block_def by blast
  obtain k where k_pos: 1 ≤ k
    and k_lt: k < length (enum_class.enum :: 'c list)
    and prefix: ∀ y. c_idx y < k ⟶ x y ≠ bl
    and suffix: ∀ y. c_idx y ≥ k ⟶ x y = bl
    using padded unfolding is_padded_block_def by blast
  have decode: ae_decode_block bl x
              = map x (take k (enum_class.enum :: 'c list))
    by (rule ae_decode_block_padded[OF k_lt prefix suffix])
  have m_pos: 0 < k using k_pos by simp
  have m_le: k ≤ length (enum_class.enum :: 'c list) using k_lt by simp
  have enc: encode_input bl (map x (take k (enum_class.enum :: 'c list)))
            = ([x] :: ('c ⇒ 'a) list)
    using encode_input_map_take_enum[OF m_pos m_le suffix] .
  show ?thesis using decode enc by simp
qed
```

The round-trip: for any well-formed block list w, encoding the decoder's output reproduces w. By *rev_induct*: the empty case is trivial; the snoc step uses *encode_input_append_div* (the prefix decodes to a length divisible by c, since all of w's prefix cells are pure) plus *encode_decode_block* on the last block.

```
lemma encode_decode_round_trip:
  fixes w :: ('c :: enum ⇒ 'a) list
  assumes wf: ae_input_well_formed bl w
  shows encode_input bl (ae_decode_input bl w) = w
proof −
  have main: ae_input_well_formed bl w
             ⟶ encode_input bl (ae_decode_input bl w) = w
  proof (induct w rule: rev_induct)
    case Nil
    show ?case
      by (simp add: ae_decode_input_def encode_input_empty)
  next
    case (snoc x xs)
    show ?case
    proof
      assume wf_snoc: ae_input_well_formed bl (xs @ [x])
      have all_pure: ∀ i < length xs. is_pure_block bl (xs ! i)
        using wf_snoc by (rule ae_input_well_formed_init_pure)
      have wf_xs: ae_input_well_formed bl xs
        unfolding ae_input_well_formed_def using all_pure by blast
      have ih: encode_input bl (ae_decode_input bl xs) = xs
        using snoc.hyps wf_xs by blast
      have x_canonical: is_canonical_block bl x
      proof −
        let ?n = length (xs @ [x]) − 1
```

```
    have x_eq: (xs @ [x]) ! ?n = x by simp
    have n_lt: ?n < length (xs @ [x]) by simp
    have is_pure_block bl ((xs @ [x]) ! ?n)
          ∨ (?n = length (xs @ [x]) − 1
             ∧ is_padded_block bl ((xs @ [x]) ! ?n))
      using wf_snoc n_lt unfolding ae_input_well_formed_def by blast
    hence is_pure_block bl x ∨ is_padded_block bl x
      using x_eq by simp
    thus ?thesis unfolding is_canonical_block_def .
  qed
  have block_eq: encode_input bl (ae_decode_block bl x) = [x]
    by (rule encode_decode_block[OF x_canonical])
  have len_xs_decode:
    card (UNIV :: 'c set) dvd length (ae_decode_input bl xs)
  proof −
    have length (ae_decode_input bl xs)
          = length xs * card (UNIV :: 'c set)
      by (rule length_ae_decode_input_pure[OF all_pure])
    thus ?thesis by simp
  qed
  have decode_split:
    ae_decode_input bl (xs @ [x])
      = ae_decode_input bl xs @ ae_decode_block bl x
    by (simp add: ae_decode_input_def)
  have encode_input bl (ae_decode_input bl (xs @ [x]))
        = encode_input bl
            (ae_decode_input bl xs @ ae_decode_block bl x)
    using decode_split by simp
  also have (encode_input bl
               (ae_decode_input bl xs @ ae_decode_block bl x)
                 :: ('c ⇒ 'a) list)
            = encode_input bl (ae_decode_input bl xs)
               @ encode_input bl (ae_decode_block bl x)
    by (rule encode_input_append_div[OF len_xs_decode])
  also have ... = xs @ encode_input bl (ae_decode_block bl x)
    using ih by simp
  also have ... = xs @ [x] using block_eq by simp
  finally show encode_input bl (ae_decode_input bl (xs @ [x]))
                 = xs @ [x] .
 qed
qed
show ?thesis using main wf by blast
qed

end
theory AlphabetEnlargement_ValidationStep
 imports AlphabetEnlargement_Codec
begin
```

2.19 Validation step machinery and initial config

2.19.1 Validation canonicity equivalence with encoder image

If a block is in *gamma_block* (*Sigma_M* ∪ {*bl_M*}), then the decoder's per-block output stays in *Sigma_M*: the takeWhile-prefix strips off any *bl_M*-symbols, leaving only *Sigma_M* elements. Per-block contribution to the set-containment side of the iff lemma's forward direction.

lemma *ae_decode_block_subset*:
  fixes *c* :: *'c* :: *enum* ⇒ *'a*
    and *Sigma* :: *'a set*
    and *bl* :: *'a*
  assumes *c* ∈ *gamma_block* (*Sigma* ∪ {*bl*})
  shows *set* (*ae_decode_block bl c*) ⊆ *Sigma*
proof
  fix *a* **assume** *a_in*: *a* ∈ *set* (*ae_decode_block bl c*)
  let *?xs* = *map c* (*enum_class.enum* :: *'c list*)
  have *a_in_tW*: *a* ∈ *set* (*takeWhile* (λ*a*. *a* ≠ *bl*) *?xs*)
    using *a_in* **by** (*simp add*: *ae_decode_block_def*)
  have *neq_bl*: ⋀*ys* :: *'a list*. ∀ *x* ∈ *set* (*takeWhile* (λ*a*. *a* ≠ *bl*) *ys*).
                                  x ≠ *bl*
  proof −
    fix *ys* :: *'a list*
    show ∀ *x* ∈ *set* (*takeWhile* (λ*a*. *a* ≠ *bl*) *ys*). *x* ≠ *bl*
      by (*induct ys*) (*auto split*: *if_split_asm*)
  qed
  have *a_neq_bl*: *a* ≠ *bl* **using** *a_in_tW neq_bl* **by** *blast*
  have *a_in_xs*: *a* ∈ *set ?xs* **using** *a_in_tW set_takeWhileD* **by** *metis*
  obtain *x* **where** *a_eq*: *a* = *c x* **using** *a_in_xs* **by** *auto*
  have *c_x_in*: *c x* ∈ *Sigma* ∪ {*bl*}
    using *assms* **unfolding** *gamma_block_def* **by** *auto*
  show *a* ∈ *Sigma* **using** *a_eq a_neq_bl c_x_in* **by** *blast*
qed

The "in encoder image ⟶ well-formed" direction of *ae_validation_canonical_iff_encoder_image*. Stated as a standalone lemma since it doesn't need the alphabet hypothesis (the existential's *set u* ⊆ *Sigma_tm M* suffices) and is cited by *ae_validation_post_state_canonical* for going from "given *u* ∈ *Sigma_tm M**" to "validation passes on *encode_input* (*bl_tm M*) *u*".

lemma *ae_well_formed_of_encoder_image*:
  fixes *M* :: (*'q*, *'a*) *mttm*
    and *u* :: *'a list*
  assumes *vM*: *valid_mttm M*
      and *u_sub*: *set u* ⊆ *Sigma_tm M*
  shows *ae_input_well_formed* (*bl_tm M*)
        (*encode_input* (*bl_tm M*) *u* :: (*'c* :: *enum* ⇒ *'a*) *list*)
proof −
  have *bl_notin*: *bl_tm M* ∉ *Sigma_tm M*

```
    using bl_tm_notin_Sigma_tm[OF vM] .
  show ?thesis
    unfolding ae_input_well_formed_def
  proof (intro allI impI)
    fix s
    assume s_lt: s < length (encode_input (bl_tm M) u :: ('c ⇒ 'a) list)
    show is_pure_block (bl_tm M) ((encode_input (bl_tm M) u :: ('c ⇒ 'a) list)
! s)
          ∨ (s = length (encode_input (bl_tm M) u :: ('c ⇒ 'a) list) − 1
             ∧ is_padded_block (bl_tm M)
                 ((encode_input (bl_tm M) u :: ('c ⇒ 'a) list) ! s))
    proof (cases (s + 1) * card (UNIV :: 'c set) ≤ length u)
      case True
      — Pure block: every slot index s * c + c_idx x is < length u
      have pure: is_pure_block (bl_tm M)
                   ((encode_input (bl_tm M) u :: ('c ⇒ 'a) list) ! s)
        unfolding is_pure_block_def
      proof (rule allI)
        fix x :: 'c
        have c_idx_lt: c_idx x < card (UNIV :: 'c set)
          by (rule c_idx_lt_card)
        have j_lt: s * card (UNIV :: 'c set) + c_idx x < length u
          using True c_idx_lt by (auto simp: algebra_simps)
        have w_s_x: ((encode_input (bl_tm M) u :: ('c ⇒ 'a) list) ! s) x
                      = u ! (s * card (UNIV :: 'c set) + c_idx x)
          using encode_input_nth[OF s_lt, of x] j_lt by (simp add: Let_def)
        have u ! (s * card (UNIV :: 'c set) + c_idx x) ∈ set u
          using j_lt by (rule nth_mem)
        hence u ! (s * card (UNIV :: 'c set) + c_idx x) ∈ Sigma_tm M
          using u_sub by blast
        thus ((encode_input (bl_tm M) u :: ('c ⇒ 'a) list) ! s) x
               ≠ bl_tm M
          using bl_notin w_s_x by auto
      qed
      show ?thesis using pure by (rule disjI1)
    next
      case False
      — Last block, padded. Pick witness k = length u − s * c: cells with c_idx x
< k index into u (hence non-blank); cells with c_idx x ≥ k are out-of-range and
equal bl_tm M.
      let ?c = card (UNIV :: 'c set)
      let ?N = length (encode_input (bl_tm M) u :: ('c ⇒ 'a) list)
      have N_eq: ?N = (length u + ?c − 1) div ?c
        by (rule length_encode_input)
      have c_eq_len_enum: ?c = length (enum_class.enum :: 'c list)
        using enum_class.UNIV_enum enum_class.enum_distinct
        by (metis distinct_card length_remdups_card_conv set_remdups)
      have c_pos: 0 < ?c
      proof −
```

```
  have (c_first :: 'c) ∈ UNIV by simp
  thus ?thesis by (simp add: card_gt_0_iff)
qed
have len_u_pos: 0 < length u
proof (rule ccontr)
  assume ¬ 0 < length u
  hence u_eq: length u = 0 by simp
  have ?N = (?c − 1) div ?c using N_eq u_eq by simp
  also have ... = 0 using c_pos by simp
  finally have ?N = 0 .
  thus False using s_lt by simp
qed
from False have sp1c_gt: length u < (s + 1) * ?c by simp
have N_form: ?N = (length u − 1) div ?c + 1
proof −
  have eq1: length u + ?c − 1 = (length u − 1) + ?c
    using len_u_pos c_pos by arith
  have eq2: ((length u − 1) + ?c) div ?c
            = (length u − 1) div ?c + 1
    using c_pos by (simp add: div_add_self2)
  show ?thesis using N_eq eq1 eq2 by simp
qed
have N_minus_1: ?N − 1 = (length u − 1) div ?c
  using N_form by simp
have s_eq: s = ?N − 1
proof (rule ccontr)
  assume s ≠ ?N − 1
  with s_lt have s_lt_Nm1: s < ?N − 1 by simp
  hence sp1_le: s + 1 ≤ ?N − 1 by simp
  have (s + 1) * ?c ≤ (?N − 1) * ?c
    using sp1_le by (rule mult_le_mono1)
  also have (?N − 1) * ?c = ((length u − 1) div ?c) * ?c
    using N_minus_1 by simp
  also have ... ≤ length u − 1
    by (rule div_times_less_eq_dividend)
  also have ... < length u using len_u_pos by simp
  finally have (s + 1) * ?c < length u .
  thus False using sp1c_gt by simp
qed
have sc_lt: s * ?c < length u
proof −
  have s = (length u − 1) div ?c using s_eq N_minus_1 by simp
  hence s * ?c = ((length u − 1) div ?c) * ?c by simp
  also have ... ≤ length u − 1
    by (rule div_times_less_eq_dividend)
  also have ... < length u using len_u_pos by simp
  finally show ?thesis .
qed
let ?k = length u − s * ?c
```

```
  have k_ge_1: 1 ≤ ?k using sc_lt by simp
  have k_lt_c: ?k < ?c
  proof −
    have length u < s ∗ ?c + ?c
      using sp1c_gt by (simp add: algebra_simps)
    thus ?thesis using sc_lt by simp
  qed
  have k_lt_enum: ?k < length (enum_class.enum :: 'c list)
    using k_lt_c c_eq_len_enum by simp
  have padded: is_padded_block (bl_tm M)
                ((encode_input (bl_tm M) u :: ('c ⇒ 'a) list) ! s)
    unfolding is_padded_block_def
  proof (rule exI[of _ ?k], intro conjI)
    show 1 ≤ ?k by (rule k_ge_1)
    show ?k < length (enum_class.enum :: 'c list) by (rule k_lt_enum)
    show ∀ x. c_idx x < ?k
            ⟶ ((encode_input (bl_tm M) u :: ('c ⇒ 'a) list) ! s) x
                ≠ bl_tm M
    proof (intro allI impI)
      fix x :: 'c
      assume cx_lt: c_idx x < ?k
      hence j_lt: s ∗ ?c + c_idx x < length u by simp
      have w_s_x: ((encode_input (bl_tm M) u :: ('c ⇒ 'a) list) ! s) x
                    = u ! (s ∗ ?c + c_idx x)
        using encode_input_nth[OF s_lt, of x] j_lt by (simp add: Let_def)
      have u ! (s ∗ ?c + c_idx x) ∈ set u
        using j_lt by (rule nth_mem)
      hence u ! (s ∗ ?c + c_idx x) ∈ Sigma_tm M
        using u_sub by blast
      thus ((encode_input (bl_tm M) u :: ('c ⇒ 'a) list) ! s) x
            ≠ bl_tm M
        using bl_notin w_s_x by auto
    qed
    show ∀ x. ?k ≤ c_idx x
            ⟶ ((encode_input (bl_tm M) u :: ('c ⇒ 'a) list) ! s) x
                = bl_tm M
    proof (intro allI impI)
      fix x :: 'c
      assume cx_ge: ?k ≤ c_idx x
      hence j_ge: length u ≤ s ∗ ?c + c_idx x by simp
      hence j_not_lt: ¬ s ∗ ?c + c_idx x < length u by simp
      show ((encode_input (bl_tm M) u :: ('c ⇒ 'a) list) ! s) x
            = bl_tm M
        using encode_input_nth[OF s_lt, of x] j_not_lt
        by (simp add: Let_def)
    qed
  qed
  from s_eq padded show ?thesis by blast
qed
```

```
  qed
qed
```

Validation lemmas structural induction over the input. These do not require simulation infrastructure.

The following biconditional characterises the AE machine's set of well-formed inputs: a string is *ae_input_well_formed* if and only if it is the encoder image of some base-alphabet input. The headline language theorems *alphabet_enlarge_language* and *alphabet_enlarge_language_forward* are quantified only over explicit encoder-image inputs (*set w* ⊆ *Sigma_tm M*, with the AE-side string given as *encode_input* (*bl_tm M*) *w*), so they need only the forward direction provided above by *encode_input_well_formed*; they never appeal to this biconditional's reverse direction (every well-formed AE-input has a preimage under the encoder). The biconditional is retained as a structural completeness result identifying exactly what the AE machine accepts as a legitimate input, and is potentially useful for the nondeterministic-reverse research thread, where extracting an encoder preimage from an AE-validation hypothesis is one of the load-bearing steps.

```
lemma ae_validation_canonical_iff_encoder_image:
  fixes M :: ('q, 'a) mttm
    and w :: ('c :: enum ⇒ 'a) list
  assumes vM: valid_mttm M
      and w_sub: set w ⊆ gamma_block (Sigma_tm M ∪ {bl_tm M})
  shows ae_input_well_formed (bl_tm M) w
        ⟷ (∃ u. set u ⊆ Sigma_tm M
                ∧ w = encode_input (bl_tm M) u)
proof
  assume wf: ae_input_well_formed (bl_tm M) w
  let ?u = ae_decode_input (bl_tm M) w
  have round_trip: encode_input (bl_tm M) ?u = w
    using encode_decode_round_trip[OF wf] .
  have u_sub: set ?u ⊆ Sigma_tm M
  proof
    fix a assume a ∈ set ?u
    hence a ∈ set (concat (map (ae_decode_block (bl_tm M)) w))
      by (simp add: ae_decode_input_def)
    then obtain c where c_in: c ∈ set w
                    and a_in_blk: a ∈ set (ae_decode_block (bl_tm M) c)
      by auto
    have c_in_gamma: c ∈ gamma_block (Sigma_tm M ∪ {bl_tm M})
      using c_in w_sub by blast
    have set (ae_decode_block (bl_tm M) c) ⊆ Sigma_tm M
      by (rule ae_decode_block_subset[OF c_in_gamma])
    thus a ∈ Sigma_tm M using a_in_blk by blast
  qed
  show ∃ u. set u ⊆ Sigma_tm M
            ∧ w = encode_input (bl_tm M) u
```

using *u_sub round_trip*[*symmetric*] **by** *blast*
next
assume ∃ *u*. *set u* ⊆ *Sigma_tm M*
∧ *w* = *encode_input* (*bl_tm M*) *u*
then obtain *u* **where**
u_sub: *set u* ⊆ *Sigma_tm M* **and**
w_eq: *w* = *encode_input* (*bl_tm M*) *u* **by** *blast*
have *ae_input_well_formed* (*bl_tm M*)
(*encode_input* (*bl_tm M*) *u* :: (′*c* ⇒ ′*a*) *list*)
using *ae_well_formed_of_encoder_image*[*OF vM u_sub*] **.**
thus *ae_input_well_formed* (*bl_tm M*) *w* **using** *w_eq* **by** *simp*
qed

2.19.2 Validation step helpers

Step-helper library for the validation phase. Each helper packages one validation substep relation as an *mttm_step*-constructor: given source-state and read-tape constraints satisfying the relation's source pattern, produce a single *mttm_step* (*alphabet_enlarge_delta M*). Used by the three remaining commit-B lemmas (*ae_validation_steps_bound*, *ae_validation_post_state_canonical*, *ae_validation_post_state_noncanonical*) to assemble validation-phase chains by composition rather than re-deriving each step.

lemma *ae_step_make*:
fixes *M* :: (′*q*, ′*a*) *mttm*
and *ts* :: *nat* ⇒ *nat* ⇒ (′*c* :: *enum* ⇒ ′*a*)
and *n* :: *nat* ⇒ *nat*
and *s s*′ :: ′*q* × (′*a*, ′*c*) *ae_stage*
and *a*′ :: *nat* ⇒ (′*c* ⇒ ′*a*)
and *d* :: *nat* ⇒ *dir*
assumes *rel*: (*s*, (λ*k*. *ts k* (*n k*)), *s*′, *a*′, *d*)
∈ *alphabet_enlarge_delta M*
shows (*Config*$_M$ *s ts n*,
Config$_M$ *s*′ (λ*k*. (*ts k*)(*n k* := *a*′ *k*))
(λ*k*. *go_dir* (*d k*) (*n k*)))
∈ *mttm_step* (*alphabet_enlarge_delta M*)
proof (*rule mttm_step.intros*)
show (*s*, (λ*k*. *ts k* (*n k*)), *s*′, *a*′, *d*)
∈ *alphabet_enlarge_delta M* **by** (*rule rel*)
qed

Reflexive frozen-tail for the initial stage. The stage fields *init_offset* / *init_buffer le* / *init_dest* agree with themselves beyond any tape count K, so this discharges the *stage_tail* premise of every validation step-helper at the call sites, where the stage is literally *init_stage* (*le_tm M*) (the validation phases never touch the buffer).

lemma *init_stage_tail*:
(∀ *j*≥*K*. *init_offset j* = *init_offset j*)

```
  ∧ (∀ j≥K. init_buffer le j = init_buffer le j)
  ∧ (∀ j≥K. init_dest j = init_dest j)
 by simp
```

VFwd advance: read *LE_block* or a pure block on tape 0; head moves R on tape 0, N elsewhere; phase stays VFwd; no write change.

```
lemma ae_step_val_fwd_advance:
 fixes M :: ('q, 'a) mttm
   and ts :: nat ⇒ nat ⇒ ('c :: enum ⇒ 'a)
   and n :: nat ⇒ nat
   and q :: 'q
   and ofs :: nat ⇒ 'c
   and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
   and dest :: nat ⇒ ae_dest
 assumes q_in: q ∈ Q_tm M
     and q_neq_t: q ≠ t_tm M
     and q_neq_r: q ≠ r_tm M
     and read: ts (0 :: nat) (n 0) = LE_block (le_tm M)
               ∨ is_pure_block (bl_tm M) (ts 0 (n 0))
     and bt: ∀ j≥k_tm M. ∀ i. ts j i = bl_block (bl_tm M)
     and stage_tail: (∀ j≥k_tm M. ofs j = init_offset j)
                   ∧ (∀ j≥k_tm M. buf j = init_buffer (le_tm M) j)
                   ∧ (∀ j≥k_tm M. dest j = init_dest j)
     and gamma: ∀ k. ts k (n k) ∈ gamma_block (Γ_tm M)
     and buf_gamma: ∀ k. fst (buf k) ∈ gamma_block (Γ_tm M)
                       ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
                       ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
 shows (Config_M (q, ofs, buf, dest, VFwd) ts n,
        Config_M (q, ofs, buf, dest, VFwd) ts
          (λk. go_dir (if k = 0 then dir.R else dir.N) (n k)))
       ∈ mttm_step (alphabet_enlarge_delta M)
proof −
 let ?d = λk :: nat. if k = 0 then dir.R else dir.N
 let ?a = λk. ts k (n k)
 have rel_in: ((q, ofs, buf, dest, VFwd), ?a,
               (q, ofs, buf, dest, VFwd), ?a, ?d)
               ∈ ae_delta_val_fwd_advance M
   unfolding ae_delta_val_fwd_advance_def
   using q_in q_neq_t q_neq_r read bt by auto
 have src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                   (snd ((q, ofs, buf, dest, VFwd)
                         :: 'q × ('a, 'c) ae_stage))
   using buf_gamma stage_tail unfolding ae_valid_stage_def by simp
 have aed_in: ((q, ofs, buf, dest, VFwd), ?a,
               (q, ofs, buf, dest, VFwd), ?a, ?d)
               ∈ alphabet_enlarge_delta M
   unfolding alphabet_enlarge_delta_def
   using rel_in gamma src_valid by auto
 have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts
```

```
    by (rule ext) auto
  have step: (Config_M (q, ofs, buf, dest, VFwd) ts n,
                Config_M (q, ofs, buf, dest, VFwd)
                  (λk. (ts k)(n k := ?a k))
                  (λk. go_dir (?d k) (n k)))
                ∈ mttm_step (alphabet_enlarge_delta M)
  proof (rule ae_step_make)
    show ((q, ofs, buf, dest, VFwd), ?a,
          (q, ofs, buf, dest, VFwd), ?a, ?d)
            ∈ alphabet_enlarge_delta M by (rule aed_in)
  qed
  show ?thesis using step ts_unchanged by simp
qed
```

VFwd → VFwdPad: read a padded block on tape 0; head moves R on tape 0, N elsewhere; phase becomes VFwdPad; no write change.

```
lemma ae_step_val_fwd_to_padded:
  fixes M :: ('q, 'a) mttm
    and ts :: nat ⇒ nat ⇒ ('c :: enum ⇒ 'a)
    and n :: nat ⇒ nat
    and q :: 'q
    and ofs :: nat ⇒ 'c
    and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
    and dest :: nat ⇒ ae_dest
  assumes q_in: q ∈ Q_tm M
      and q_neq_t: q ≠ t_tm M
      and q_neq_r: q ≠ r_tm M
      and read: is_padded_block (bl_tm M) (ts (0 :: nat) (n 0))
      and bt: ∀j≥k_tm M. ∀i. ts j i = bl_block (bl_tm M)
      and stage_tail: (∀j≥k_tm M. ofs j = init_offset j)
                    ∧ (∀j≥k_tm M. buf j = init_buffer (le_tm M) j)
                    ∧ (∀j≥k_tm M. dest j = init_dest j)
      and gamma: ∀k. ts k (n k) ∈ gamma_block (Γ_tm M)
      and buf_gamma: ∀k. fst (buf k) ∈ gamma_block (Γ_tm M)
                       ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
                       ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
  shows (Config_M (q, ofs, buf, dest, VFwd) ts n,
           Config_M (q, ofs, buf, dest, VFwdPad) ts
             (λk. go_dir (if k = 0 then dir.R else dir.N) (n k)))
          ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  let ?d = λk :: nat. if k = 0 then dir.R else dir.N
  let ?a = λk. ts k (n k)
  have rel_in: ((q, ofs, buf, dest, VFwd), ?a,
                (q, ofs, buf, dest, VFwdPad), ?a, ?d)
                ∈ ae_delta_val_fwd_to_padded M
    unfolding ae_delta_val_fwd_to_padded_def
    using q_in q_neq_t q_neq_r read bt by auto
  have src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
```

```
                (snd ((q, ofs, buf, dest, VFwd)
                      :: 'q × ('a, 'c) ae_stage))
    using buf_gamma stage_tail unfolding ae_valid_stage_def by simp
  have dst_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                (snd ((q, ofs, buf, dest, VFwdPad)
                      :: 'q × ('a, 'c) ae_stage))
    using buf_gamma stage_tail unfolding ae_valid_stage_def by simp
  have aed_in: ((q, ofs, buf, dest, VFwd), ?a,
                (q, ofs, buf, dest, VFwdPad), ?a, ?d)
                ∈ alphabet_enlarge_delta M
    unfolding alphabet_enlarge_delta_def
    using rel_in gamma src_valid dst_valid by auto
  have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts
    by (rule ext) auto
  have step: (Config_M (q, ofs, buf, dest, VFwd) ts n,
              Config_M (q, ofs, buf, dest, VFwdPad)
                (λk. (ts k)(n k := ?a k))
                (λk. go_dir (?d k) (n k)))
              ∈ mttm_step (alphabet_enlarge_delta M)
  proof (rule ae_step_make)
    show ((q, ofs, buf, dest, VFwd), ?a,
          (q, ofs, buf, dest, VFwdPad), ?a, ?d)
            ∈ alphabet_enlarge_delta M by (rule aed_in)
  qed
  show ?thesis using step ts_unchanged by simp
qed
```

VFwd → VRet: read *bl_block bl_M* on tape 0 (past the encoded input); N moves uniformly (head stays); phase becomes VRet.

```
lemma ae_step_val_fwd_to_ret:
  fixes M :: ('q, 'a) mttm
    and ts :: nat ⇒ nat ⇒ ('c :: enum ⇒ 'a)
    and n :: nat ⇒ nat
    and q :: 'q
    and ofs :: nat ⇒ 'c
    and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
    and dest :: nat ⇒ ae_dest
  assumes q_in: q ∈ Q_tm M
      and q_neq_t: q ≠ t_tm M
      and q_neq_r: q ≠ r_tm M
      and read: ts (0 :: nat) (n 0) = bl_block (bl_tm M)
      and bt: ∀ j≥k_tm M. ∀ i. ts j i = bl_block (bl_tm M)
      and stage_tail: (∀ j≥k_tm M. ofs j = init_offset j)
                    ∧ (∀ j≥k_tm M. buf j = init_buffer (le_tm M) j)
                    ∧ (∀ j≥k_tm M. dest j = init_dest j)
      and gamma: ∀ k. ts k (n k) ∈ gamma_block (Γ_tm M)
      and buf_gamma: ∀ k. fst (buf k) ∈ gamma_block (Γ_tm M)
                    ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
                    ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
```

```
  shows (Config_M (q, ofs, buf, dest, VFwd) ts n,
         Config_M (q, ofs, buf, dest, VRet) ts n)
        ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  let ?d = λ_ :: nat. dir.N
  let ?a = λk. ts k (n k)
  have rel_in: ((q, ofs, buf, dest, VFwd), ?a,
                (q, ofs, buf, dest, VRet), ?a, ?d)
                ∈ ae_delta_val_fwd_to_ret M
    unfolding ae_delta_val_fwd_to_ret_def
    using q_in q_neq_t q_neq_r read bt by auto
  have src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                    (snd ((q, ofs, buf, dest, VFwd)
                          :: 'q × ('a, 'c) ae_stage))
    using buf_gamma stage_tail unfolding ae_valid_stage_def by simp
  have dst_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                    (snd ((q, ofs, buf, dest, VRet)
                          :: 'q × ('a, 'c) ae_stage))
    using buf_gamma stage_tail unfolding ae_valid_stage_def by simp
  have aed_in: ((q, ofs, buf, dest, VFwd), ?a,
                (q, ofs, buf, dest, VRet), ?a, ?d)
                ∈ alphabet_enlarge_delta M
    unfolding alphabet_enlarge_delta_def
    using rel_in gamma src_valid dst_valid by auto
  have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts
    by (rule ext) auto
  have n_unchanged: (λk :: nat. go_dir (?d k) (n k)) = n
    by (rule ext) simp
  have step: (Config_M (q, ofs, buf, dest, VFwd) ts n,
              Config_M (q, ofs, buf, dest, VRet)
                (λk. (ts k)(n k := ?a k))
                (λk. go_dir (?d k) (n k)))
              ∈ mttm_step (alphabet_enlarge_delta M)
  proof (rule ae_step_make)
    show ((q, ofs, buf, dest, VFwd), ?a,
          (q, ofs, buf, dest, VRet), ?a, ?d)
           ∈ alphabet_enlarge_delta M by (rule aed_in)
  qed
  show ?thesis using step ts_unchanged n_unchanged by simp
qed
```

VFwdPad → VRet: read *bl_block bl_M* on tape 0 (past the trailing-padded block into the blanks); N moves; phase becomes VRet.

```
lemma ae_step_val_pad_to_ret:
  fixes M :: ('q, 'a) mttm
    and ts :: nat ⇒ nat ⇒ ('c :: enum ⇒ 'a)
    and n :: nat ⇒ nat
    and q :: 'q
    and ofs :: nat ⇒ 'c
```

```
    and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
    and dest :: nat ⇒ ae_dest
  assumes q_in: q ∈ Q_tm M
      and read: ts (0 :: nat) (n 0) = bl_block (bl_tm M)
      and bt: ∀ j≥k_tm M. ∀ i. ts j i = bl_block (bl_tm M)
      and stage_tail: (∀ j≥k_tm M. ofs j = init_offset j)
                    ∧ (∀ j≥k_tm M. buf j = init_buffer (le_tm M) j)
                    ∧ (∀ j≥k_tm M. dest j = init_dest j)
      and gamma: ∀ k. ts k (n k) ∈ gamma_block (Γ_tm M)
      and buf_gamma: ∀ k. fst (buf k) ∈ gamma_block (Γ_tm M)
                        ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
                        ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
  shows (Config_M (q, ofs, buf, dest, VFwdPad) ts n,
         Config_M (q, ofs, buf, dest, VRet) ts n)
        ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  let ?d = λ_ :: nat. dir.N
  let ?a = λk. ts k (n k)
  have rel_in: ((q, ofs, buf, dest, VFwdPad), ?a,
                (q, ofs, buf, dest, VRet), ?a, ?d)
                ∈ ae_delta_val_pad_to_ret M
    unfolding ae_delta_val_pad_to_ret_def
    using q_in read bt by auto
  have src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                    (snd ((q, ofs, buf, dest, VFwdPad)
                          :: 'q × ('a, 'c) ae_stage))
    using buf_gamma stage_tail unfolding ae_valid_stage_def by simp
  have dst_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                    (snd ((q, ofs, buf, dest, VRet)
                          :: 'q × ('a, 'c) ae_stage))
    using buf_gamma stage_tail unfolding ae_valid_stage_def by simp
  have aed_in: ((q, ofs, buf, dest, VFwdPad), ?a,
                (q, ofs, buf, dest, VRet), ?a, ?d)
                ∈ alphabet_enlarge_delta M
    unfolding alphabet_enlarge_delta_def
    using rel_in gamma src_valid dst_valid by auto
  have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts
    by (rule ext) auto
  have n_unchanged: (λk :: nat. go_dir (?d k) (n k)) = n
    by (rule ext) simp
  have step: (Config_M (q, ofs, buf, dest, VFwdPad) ts n,
              Config_M (q, ofs, buf, dest, VRet)
                (λk. (ts k)(n k := ?a k))
                (λk. go_dir (?d k) (n k)))
              ∈ mttm_step (alphabet_enlarge_delta M)
  proof (rule ae_step_make)
    show ((q, ofs, buf, dest, VFwdPad), ?a,
          (q, ofs, buf, dest, VRet), ?a, ?d)
            ∈ alphabet_enlarge_delta M by (rule aed_in)
```

```
  qed
  show ?thesis using step ts_unchanged n_unchanged by simp
qed
```

VRet step: read a non-LE block on tape 0 (during the return scan); head moves L on tape 0, N elsewhere; phase stays VRet.

```
lemma ae_step_val_ret_step:
  fixes M :: ('q, 'a) mttm
    and ts :: nat ⇒ nat ⇒ ('c :: enum ⇒ 'a)
    and n :: nat ⇒ nat
    and q :: 'q
    and ofs :: nat ⇒ 'c
    and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
    and dest :: nat ⇒ ae_dest
  assumes q_in: q ∈ Q_tm M
      and read: ts (0 :: nat) (n 0) ≠ LE_block (le_tm M)
      and bt: ∀ j≥k_tm M. ∀ i. ts j i = bl_block (bl_tm M)
      and stage_tail: (∀ j≥k_tm M. ofs j = init_offset j)
                    ∧ (∀ j≥k_tm M. buf j = init_buffer (le_tm M) j)
                    ∧ (∀ j≥k_tm M. dest j = init_dest j)
      and gamma: ∀ k. ts k (n k) ∈ gamma_block (Γ_tm M)
      and buf_gamma: ∀ k. fst (buf k) ∈ gamma_block (Γ_tm M)
                        ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
                        ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
  shows (Config_M (q, ofs, buf, dest, VRet) ts n,
         Config_M (q, ofs, buf, dest, VRet) ts
           (λk. go_dir (if k = 0 then dir.L else dir.N) (n k)))
        ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  let ?d = λk :: nat. if k = 0 then dir.L else dir.N
  let ?a = λk. ts k (n k)
  have rel_in: ((q, ofs, buf, dest, VRet), ?a,
                (q, ofs, buf, dest, VRet), ?a, ?d)
                ∈ ae_delta_val_ret_step M
    unfolding ae_delta_val_ret_step_def
    using q_in read bt by auto
  have src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                    (snd ((q, ofs, buf, dest, VRet)
                          :: 'q × ('a, 'c) ae_stage))
    using buf_gamma stage_tail unfolding ae_valid_stage_def by simp
  have aed_in: ((q, ofs, buf, dest, VRet), ?a,
                (q, ofs, buf, dest, VRet), ?a, ?d)
                ∈ alphabet_enlarge_delta M
    unfolding alphabet_enlarge_delta_def
    using rel_in gamma src_valid by auto
  have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts
    by (rule ext) auto
  have step: (Config_M (q, ofs, buf, dest, VRet) ts n,
              Config_M (q, ofs, buf, dest, VRet)
```

```
                (λk. (ts k)(n k := ?a k))
                (λk. go_dir (?d k) (n k)))
              ∈ mttm_step (alphabet_enlarge_delta M)
  proof (rule ae_step_make)
    show ((q, ofs, buf, dest, VRet), ?a,
          (q, ofs, buf, dest, VRet), ?a, ?d)
           ∈ alphabet_enlarge_delta M by (rule aed_in)
  qed
  show ?thesis using step ts_unchanged by simp
qed
```

VRet → Sim: read *LE_block le_M* on tape 0 (return scan reached position 0); N moves uniformly (head stays); phase becomes Sim with *substep_idx* = SS1.

```
lemma ae_step_val_ret_to_sim:
  fixes M :: ('q, 'a) mttm
    and ts :: nat ⇒ nat ⇒ ('c :: enum ⇒ 'a)
    and n :: nat ⇒ nat
    and q :: 'q
    and ofs :: nat ⇒ 'c
    and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
    and dest :: nat ⇒ ae_dest
  assumes q_in: q ∈ Q_tm M
      and read: ts (0 :: nat) (n 0) = LE_block (le_tm M)
      and bt: ∀ j≥k_tm M. ∀ i. ts j i = bl_block (bl_tm M)
      and stage_tail: (∀ j≥k_tm M. ofs j = init_offset j)
                    ∧ (∀ j≥k_tm M. buf j = init_buffer (le_tm M) j)
                    ∧ (∀ j≥k_tm M. dest j = init_dest j)
      and gamma: ∀ k. ts k (n k) ∈ gamma_block (Γ_tm M)
      and buf_gamma: ∀ k. fst (buf k) ∈ gamma_block (Γ_tm M)
                        ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
                        ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
  shows (Config_M (q, ofs, buf, dest, VRet) ts n,
         Config_M (q, ofs, buf, dest, SS1) ts n)
        ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  let ?d = λ_ :: nat. dir.N
  let ?a = λk. ts k (n k)
  have rel_in: ((q, ofs, buf, dest, VRet), ?a,
                (q, ofs, buf, dest, SS1), ?a, ?d)
                ∈ ae_delta_val_ret_to_sim M
    unfolding ae_delta_val_ret_to_sim_def
    using q_in read bt by auto
  have src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                  (snd ((q, ofs, buf, dest, VRet)
                        :: 'q × ('a, 'c) ae_stage))
    using buf_gamma stage_tail unfolding ae_valid_stage_def by simp
  have dst_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                  (snd ((q, ofs, buf, dest, SS1)
```

```
                                :: 'q × ('a, 'c) ae_stage))
    using buf_gamma stage_tail unfolding ae_valid_stage_def by simp
  have aed_in: ((q, ofs, buf, dest, VRet), ?a,
                (q, ofs, buf, dest, SS1), ?a, ?d)
                ∈ alphabet_enlarge_delta M
    unfolding alphabet_enlarge_delta_def
    using rel_in gamma src_valid dst_valid by auto
  have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts
    by (rule ext) auto
  have n_unchanged: (λk :: nat. go_dir (?d k) (n k)) = n
    by (rule ext) simp
  have step: (Config_M (q, ofs, buf, dest, VRet) ts n,
              Config_M (q, ofs, buf, dest, SS1)
                (λk. (ts k)(n k := ?a k))
                (λk. go_dir (?d k) (n k)))
              ∈ mttm_step (alphabet_enlarge_delta M)
  proof (rule ae_step_make)
    show ((q, ofs, buf, dest, VRet), ?a,
          (q, ofs, buf, dest, SS1), ?a, ?d)
            ∈ alphabet_enlarge_delta M by (rule aed_in)
  qed
  show ?thesis using step ts_unchanged n_unchanged by simp
qed
```

VFwd reject: read a non-canonical block on tape 0 (in Σ' but neither pure nor padded blanks in non-trailing positions); N moves uniformly; transition to (*r_M*, *init_stage le_M*).

```
lemma ae_step_val_fwd_reject:
  fixes M :: ('q, 'a) mttm
    and ts :: nat ⇒ nat ⇒ ('c :: enum ⇒ 'a)
    and n :: nat ⇒ nat
    and q :: 'q
    and ofs :: nat ⇒ 'c
    and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
    and dest :: nat ⇒ ae_dest
  assumes vM: valid_mttm M
      and q_in: q ∈ Q_tm M
      and q_neq_t: q ≠ t_tm M
      and q_neq_r: q ≠ r_tm M
      and read_nle: ts (0 :: nat) (n 0) ≠ LE_block (le_tm M)
      and read_nbl: ts (0 :: nat) (n 0) ≠ bl_block (bl_tm M)
      and read_ncan: ¬ is_canonical_block (bl_tm M) (ts (0 :: nat) (n 0))
      and bt: ∀ j≥k_tm M. ∀ i. ts j i = bl_block (bl_tm M)
      and stage_tail: (∀ j≥k_tm M. ofs j = init_offset j)
                    ∧ (∀ j≥k_tm M. buf j = init_buffer (le_tm M) j)
                    ∧ (∀ j≥k_tm M. dest j = init_dest j)
      and gamma: ∀ k. ts k (n k) ∈ gamma_block (Γ_tm M)
      and buf_gamma: ∀ k. fst (buf k) ∈ gamma_block (Γ_tm M)
                   ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
```

```
                    ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
  shows (Config_M (q, ofs, buf, dest, VFwd) ts n,
         Config_M (r_tm M, init_stage (le_tm M)) ts n)
        ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  let ?d = λ_ :: nat. dir.N
  let ?a = λk. ts k (n k)
  have rel_in: ((q, ofs, buf, dest, VFwd), ?a,
                (r_tm M, init_stage (le_tm M)), ?a, ?d)
                ∈ ae_delta_val_fwd_reject M
    unfolding ae_delta_val_fwd_reject_def
    using q_in q_neq_t q_neq_r read_nle read_nbl read_ncan bt by auto
  have src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                    (snd ((q, ofs, buf, dest, VFwd)
                          :: 'q × ('a, 'c) ae_stage))
    using buf_gamma stage_tail unfolding ae_valid_stage_def by simp
  have dst_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                    (snd ((r_tm M, init_stage (le_tm M))
                          :: 'q × ('a, 'c) ae_stage))
   using ae_valid_stage_init[OF valid_mttm_LE_in_Gamma[OF vM]] by simp
  have aed_in: ((q, ofs, buf, dest, VFwd), ?a,
                (r_tm M, init_stage (le_tm M)), ?a, ?d)
                ∈ alphabet_enlarge_delta M
    unfolding alphabet_enlarge_delta_def
    using rel_in gamma src_valid dst_valid by auto
  have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts
    by (rule ext) auto
  have n_unchanged: (λk :: nat. go_dir (?d k) (n k)) = n
    by (rule ext) simp
  have step: (Config_M (q, ofs, buf, dest, VFwd) ts n,
              Config_M (r_tm M, init_stage (le_tm M))
                (λk. (ts k)(n k := ?a k))
                (λk. go_dir (?d k) (n k)))
              ∈ mttm_step (alphabet_enlarge_delta M)
  proof (rule ae_step_make)
    show ((q, ofs, buf, dest, VFwd), ?a,
          (r_tm M, init_stage (le_tm M)), ?a, ?d)
            ∈ alphabet_enlarge_delta M by (rule aed_in)
  qed
  show ?thesis using step ts_unchanged n_unchanged by simp
qed
```

VFwdPad reject: read anything on tape 0 other than *bl_block bl_M* (a non-blank block after the trailing-padded one); N moves; transition to (*r_M*, *init_stage le_M*).

```
lemma ae_step_val_pad_reject:
  fixes M :: ('q, 'a) mttm
    and ts :: nat ⇒ nat ⇒ ('c :: enum ⇒ 'a)
    and n :: nat ⇒ nat
```

```
    and q :: 'q
    and ofs :: nat ⇒ 'c
    and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
    and dest :: nat ⇒ ae_dest
  assumes vM: valid_mttm M
      and q_in: q ∈ Q_tm M
      and read: ts (0 :: nat) (n 0) ≠ bl_block (bl_tm M)
      and bt: ∀ j≥k_tm M. ∀ i. ts j i = bl_block (bl_tm M)
      and stage_tail: (∀ j≥k_tm M. ofs j = init_offset j)
                    ∧ (∀ j≥k_tm M. buf j = init_buffer (le_tm M) j)
                    ∧ (∀ j≥k_tm M. dest j = init_dest j)
      and gamma: ∀ k. ts k (n k) ∈ gamma_block (Γ_tm M)
      and buf_gamma: ∀ k. fst (buf k) ∈ gamma_block (Γ_tm M)
                        ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
                        ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
  shows (Config_M (q, ofs, buf, dest, VFwdPad) ts n,
         Config_M (r_tm M, init_stage (le_tm M)) ts n)
        ∈ mttm_step (alphabet_enlarge_delta M)
proof −
  let ?d = λ_ :: nat. dir.N
  let ?a = λk. ts k (n k)
  have rel_in: ((q, ofs, buf, dest, VFwdPad), ?a,
                (r_tm M, init_stage (le_tm M)), ?a, ?d)
                ∈ ae_delta_val_pad_reject M
    unfolding ae_delta_val_pad_reject_def
    using q_in read bt by auto
  have src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                    (snd ((q, ofs, buf, dest, VFwdPad)
                          :: 'q × ('a, 'c) ae_stage))
    using buf_gamma stage_tail unfolding ae_valid_stage_def by simp
  have dst_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                    (snd ((r_tm M, init_stage (le_tm M))
                          :: 'q × ('a, 'c) ae_stage))
   using ae_valid_stage_init[OF valid_mttm_LE_in_Gamma[OF vM]] by simp
  have aed_in: ((q, ofs, buf, dest, VFwdPad), ?a,
                (r_tm M, init_stage (le_tm M)), ?a, ?d)
                ∈ alphabet_enlarge_delta M
    unfolding alphabet_enlarge_delta_def
    using rel_in gamma src_valid dst_valid by auto
  have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts
    by (rule ext) auto
  have n_unchanged: (λk :: nat. go_dir (?d k) (n k)) = n
    by (rule ext) simp
  have step: (Config_M (q, ofs, buf, dest, VFwdPad) ts n,
              Config_M (r_tm M, init_stage (le_tm M))
                (λk. (ts k)(n k := ?a k))
                (λk. go_dir (?d k) (n k)))
              ∈ mttm_step (alphabet_enlarge_delta M)
  proof (rule ae_step_make)
```

```
    show ((q, ofs, buf, dest, VFwdPad), ?a,
          (r_tm M, init_stage (le_tm M)), ?a, ?d)
            ∈ alphabet_enlarge_delta M by (rule aed_in)
  qed
  show ?thesis using step ts_unchanged n_unchanged by simp
qed
```

2.19.3 Initial-config shape and gamma-block

Shape lookups for *ae_init_config*. These give the tape contents at named positions LE at position 0, input blocks at positions *1 . . . length w*, blank-block elsewhere and the uniform state / head shape. Used pervasively by the validation chain proofs; trivial unfoldings of *ae_init_config_def*.

```
lemma ae_init_config_state:
  mt_state (ae_init_config M w) = (s_tm M, init_stage (le_tm M))
  unfolding ae_init_config_def by simp

lemma ae_init_config_pos:
  mt_pos (ae_init_config M w) k = 0
  unfolding ae_init_config_def by simp

lemma ae_init_config_tape_le:
  assumes k < k_tm M
  shows mt_tape (ae_init_config M w) k 0 = LE_block (le_tm M)
  using assms unfolding ae_init_config_def by simp

lemma ae_init_config_tape_input:
  assumes 1 ≤ p and p ≤ length w and 0 < k_tm M
  shows mt_tape (ae_init_config M w) 0 p = w ! (p − 1)
  using assms unfolding ae_init_config_def by simp

lemma ae_init_config_tape_blank_after_input:
  assumes p > length w
  shows mt_tape (ae_init_config M w) 0 p = bl_block (bl_tm M)
  using assms unfolding ae_init_config_def by simp

lemma ae_init_config_tape_other:
  assumes k ≠ 0 and p > 0
  shows mt_tape (ae_init_config M w) k p = bl_block (bl_tm M)
  using assms unfolding ae_init_config_def by simp
```

Blank-tail of the initial configuration: every cell of every inactive tape (index $\geq$ *k_tm M*) holds the blank block. This is the value-level support condition the substrate's *init_config_mttm* imposes on *alphabet_enlarge M*; it discharges the $\forall j{\geq}$*k_tm M*. *ts j* (*n j*) = *bl_block* (*bl_tm M*) premise of every validation step-helper (the tape is never written during validation, so *ts* = *mt_tape* (*ae_init_config M w*) throughout).

```
lemma ae_init_config_tape_blank_tail:
```

```
  assumes k_tm M ≤ k
  shows mt_tape (ae_init_config M w) k p = bl_block (bl_tm M)
  using assms unfolding ae_init_config_def by simp
```

Gamma-block membership of every tape cell of the initial configuration. Uniformly discharges the $\forall k.\ ts\ k\ (n\ k) \in gamma_block\ (\Gamma_tm\ M)$ premise of every validation step-helper, regardless of the head position n reached during the chain.

```
lemma ae_init_config_in_gamma_block:
  fixes M :: ('q, 'a) mttm
    and w :: (('c :: enum) ⇒ 'a) list
  assumes vM: valid_mttm M
      and w_sub: set w ⊆ gamma_block (Sigma_tm M ∪ {bl_tm M})
  shows mt_tape (ae_init_config M w) k p ∈ gamma_block (Γ_tm M)
proof −
  have bl_in: bl_tm M ∈ Γ_tm M
    by (rule valid_mttm_blank_in_Gamma[OF vM])
  have le_in: le_tm M ∈ Γ_tm M
    by (rule valid_mttm_LE_in_Gamma[OF vM])
  have Sigma_sub: Sigma_tm M ⊆ Γ_tm M
    by (rule valid_mttm_Sigma_sub_Gamma[OF vM])
  have sub: Sigma_tm M ∪ {bl_tm M} ⊆ Γ_tm M
    using Sigma_sub bl_in by auto
  have w_sub_G: set w ⊆ gamma_block (Γ_tm M)
    using w_sub gamma_block_mono[OF sub] by blast
  have kpos: 0 < k_tm M by (rule valid_mttm_k_pos[OF vM])
  consider
      (zero_act) p = 0 and k < k_tm M
    | (zero_inact) p = 0 and ¬ k < k_tm M
    | (input) p ≠ 0 and k = 0 and p ≤ length w
    | (blank) p ≠ 0 and ¬ (k = 0 ∧ p ≤ length w)
    by blast
  thus ?thesis
  proof cases
    case zero_act
    have mt_tape (ae_init_config M w) k p = LE_block (le_tm M)
      using zero_act ae_init_config_tape_le by simp
    thus ?thesis using LE_block_in_gamma_block[OF le_in] by simp
  next
    case zero_inact
    have mt_tape (ae_init_config M w) k p = bl_block (bl_tm M)
      using zero_inact unfolding ae_init_config_def by simp
    thus ?thesis using bl_block_in_gamma_block[OF bl_in] by simp
  next
    case input
    have mt_tape (ae_init_config M w) k p = w ! (p − 1)
      using input kpos ae_init_config_tape_input[of p w M] by simp
    moreover have w ! (p − 1) ∈ set w
      using input by auto
```

```
    ultimately show ?thesis using w_sub_G by auto
  next
    case blank
    have mt_tape (ae_init_config M w) k p = bl_block (bl_tm M)
      using blank unfolding ae_init_config_def by auto
    thus ?thesis using bl_block_in_gamma_block[OF bl_in] by simp
  qed
qed

end
theory AlphabetEnlargement_ValidationBound
  imports AlphabetEnlargement_ValidationStep
begin
```

2.20 Validation sweeps, post-state, and step bound

2.20.1 Validation sweeps and tape correspondence

Forward-sweep iteration helper. Starting from *ae_init_config M w*, after *Suc k* validation steps along a k-pure prefix of w, the head on tape 0 is at position *Suc k* and the phase is still *VFwd*. The chain is built by induction on k, applying *ae_step_val_fwd_advance* once per step (the LE block at position 0 takes the first step, then each pure block along positions $1 \ldots k$ takes one more).

```
lemma ae_validation_fwd_sweep_pure:
  fixes M :: ('q, 'a) mttm
    and w :: (('c :: enum) ⇒ 'a) list
  assumes vM: valid_mttm M
      and w_sub: set w ⊆ gamma_block (Sigma_tm M ∪ {bl_tm M})
      and s_in_Q: s_tm M ∈ Q_tm M
      and s_neq_t: s_tm M ≠ t_tm M
      and s_neq_r: s_tm M ≠ r_tm M
      and k_bound: k ≤ length w
      and pure: ∀ i < k. is_pure_block (bl_tm M) (w ! i)
  shows (ae_init_config M w,
           Config_M (s_tm M, init_stage (le_tm M))
                    (mt_tape (ae_init_config M w))
                    (λi :: nat. if i = 0 then Suc k else 0))
         ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc k
  using k_bound pure
proof (induction k)
  case 0
  let ?ts = mt_tape (ae_init_config M w)
  let ?n0 = λ_ :: nat. 0 :: nat
  let ?n1 = λi :: nat. if i = 0 then Suc 0 else 0
  have init_eq: ae_init_config M w
                = Config_M (s_tm M, init_stage (le_tm M)) ?ts ?n0
    by (simp add: ae_init_config_def)
  have kpos: 0 < k_tm M by (rule valid_mttm_k_pos[OF vM])
```

have *read*: *?ts 0 (?n0 0)* = *LE_block (le_tm M)*
  ∨ *is_pure_block (bl_tm M) (?ts 0 (?n0 0))*
 using *ae_init_config_tape_le*[*OF kpos*] **by** *auto*
have *bt_all*: ∀ *j*≥*k_tm M*. ∀ *i*. *?ts j i* = *bl_block (bl_tm M)*
 using *ae_init_config_tape_blank_tail* **by** *blast*
have *gamma*: ∀ *kk* :: *nat*. *?ts kk (?n0 kk)* ∈ *gamma_block* (Γ_*tm M*)
 using *ae_init_config_in_gamma_block*[*OF vM w_sub*] **by** *simp*
have *buf_gamma_init*:
  ∀ *kk* :: *nat*.
    fst (init_buffer (le_tm M) kk
      :: (′*c* ⇒ ′*a*) × (′*c* ⇒ ′*a*) × (′*c* ⇒ ′*a*))
     ∈ *gamma_block* (Γ_*tm M*)
    ∧ *fst* (*snd* (*init_buffer* (*le_tm M*) *kk*
      :: (′*c* ⇒ ′*a*) × (′*c* ⇒ ′*a*) × (′*c* ⇒ ′*a*)))
     ∈ *gamma_block* (Γ_*tm M*)
    ∧ *snd* (*snd* (*init_buffer* (*le_tm M*) *kk*
      :: (′*c* ⇒ ′*a*) × (′*c* ⇒ ′*a*) × (′*c* ⇒ ′*a*)))
     ∈ *gamma_block* (Γ_*tm M*)
 by (*rule init_buffer_in_gamma_block_at_M*[*OF vM*])
have *step1*: (*Config*$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
      init_dest, *VFwd*) *?ts ?n0*,
   Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
      init_dest, *VFwd*) *?ts*
    (λ*kk*. *go_dir* (*if kk* = (*0* :: *nat*) *then dir.R else dir.N*)
      (*?n0 kk*)))
   ∈ *mttm_step* (*alphabet_enlarge_delta M*)
 by (*rule ae_step_val_fwd_advance*[**where** *ts* = *?ts* **and** *n* = *?n0*,
      OF s_in_Q s_neq_t s_neq_r read bt_all
       init_stage_tail gamma buf_gamma_init])
have *post_eq*:
 (λ*kk* :: *nat*. *go_dir* (*if kk* = *0 then dir.R else dir.N*) (*?n0 kk*)) = *?n1*
 by (*rule ext*) *simp*
have *step1_n1*: (*Config*$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
      init_dest, *VFwd*) *?ts ?n0*,
   Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
      init_dest, *VFwd*) *?ts ?n1*)
   ∈ *mttm_step* (*alphabet_enlarge_delta M*)
 using *step1 post_eq* **by** *simp*
have *stage_eq*: *init_stage* (*le_tm M*)
   = (*init_offset*, *init_buffer* (*le_tm M*), *init_dest*, *VFwd*)
 unfolding *init_stage_def* **..**
have *step1_init*: (*Config*$_M$ (*s_tm M*, *init_stage* (*le_tm M*)) *?ts ?n0*,
    Config$_M$ (*s_tm M*, *init_stage* (*le_tm M*)) *?ts ?n1*)
   ∈ *mttm_step* (*alphabet_enlarge_delta M*)
 using *step1_n1* **unfolding** *stage_eq* **by** *simp*
have *chain1*: (*ae_init_config M w*,
    Config$_M$ (*s_tm M*, *init_stage* (*le_tm M*)) *?ts ?n1*)
   ∈ *mttm_step* (*alphabet_enlarge_delta M*)
 using *step1_init init_eq* **by** *simp*

```
  show ?case using chain1 by (simp add: relpow_1)
next
  case (Suc k)
  let ?ts = mt_tape (ae_init_config M w)
  let ?n_k = λi :: nat. if i = 0 then Suc k else 0
  let ?n_Sk = λi :: nat. if i = 0 then Suc (Suc k) else 0
  have k_le: k ≤ length w using Suc.prems(1) by simp
  have pure_k: ∀ i < k. is_pure_block (bl_tm M) (w ! i)
    using Suc.prems(2) by simp
  from Suc.IH[OF k_le pure_k]
  have IH: (ae_init_config M w,
            Config_M (s_tm M, init_stage (le_tm M)) ?ts ?n_k)
            ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc k .
  have read_pure: is_pure_block (bl_tm M) (w ! k)
    using Suc.prems(2) by auto
  have Sk_pos: 1 ≤ Suc k by simp
  have Sk_le: Suc k ≤ length w using Suc.prems(1) .
  have kpos: 0 < k_tm M by (rule valid_mttm_k_pos[OF vM])
  have read_eq: ?ts 0 (Suc k) = w ! k
    using ae_init_config_tape_input[OF Sk_pos Sk_le kpos] by simp
  have read: ?ts 0 (?n_k 0) = LE_block (le_tm M)
            ∨ is_pure_block (bl_tm M) (?ts 0 (?n_k 0))
    using read_pure read_eq by simp
  have bt_all: ∀ j≥k_tm M. ∀ i. ?ts j i = bl_block (bl_tm M)
    using ae_init_config_tape_blank_tail by blast
  have gamma: ∀ kk :: nat. ?ts kk (?n_k kk) ∈ gamma_block (Γ_tm M)
    using ae_init_config_in_gamma_block[OF vM w_sub] by simp
  have buf_gamma_init:
     ∀ kk :: nat.
        fst (init_buffer (le_tm M) kk
               :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
           ∈ gamma_block (Γ_tm M)
        ∧ fst (snd (init_buffer (le_tm M) kk
                      :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)))
           ∈ gamma_block (Γ_tm M)
        ∧ snd (snd (init_buffer (le_tm M) kk
                      :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)))
           ∈ gamma_block (Γ_tm M)
    by (rule init_buffer_in_gamma_block_at_M[OF vM])
  have step: (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                        init_dest, VFwd) ?ts ?n_k,
              Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                        init_dest, VFwd) ?ts
                (λkk. go_dir (if kk = (0 :: nat) then dir.R else dir.N)
                        (?n_k kk)))
              ∈ mttm_step (alphabet_enlarge_delta M)
    by (rule ae_step_val_fwd_advance[where ts = ?ts and n = ?n_k,
                                     OF s_in_Q s_neq_t s_neq_r read bt_all
                                       init_stage_tail gamma buf_gamma_init])
```

```
  have post_eq:
    (λkk :: nat. go_dir (if kk = 0 then dir.R else dir.N) (?n_k kk)) = ?n_Sk
    by (rule ext) simp
  have step_n_Sk: (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                             init_dest, VFwd) ?ts ?n_k,
                   Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                             init_dest, VFwd) ?ts ?n_Sk)
                  ∈ mttm_step (alphabet_enlarge_delta M)
    using step post_eq by simp
  have stage_eq: init_stage (le_tm M)
                 = (init_offset, init_buffer (le_tm M), init_dest, VFwd)
    unfolding init_stage_def ..
  have step_init: (Config_M (s_tm M, init_stage (le_tm M)) ?ts ?n_k,
                   Config_M (s_tm M, init_stage (le_tm M)) ?ts ?n_Sk)
                  ∈ mttm_step (alphabet_enlarge_delta M)
    using step_n_Sk unfolding stage_eq by simp
  show ?case by (rule relpow_Suc_I[OF IH step_init])
qed
```

Return-sweep iteration helper. Symmetric counterpart to the forward sweep: starting at *VRet* with the head on tape 0 at position p, retreat to position 0 in p applications of *ae_step_val_ret_step*, then take one more step via *ae_step_val_ret_to_sim* to reach *SS1*. Total: *Suc p* steps. The non-LE constraint on positions $1 \ldots p$ is hoisted into the goal as a $\longrightarrow$-form so the standard induction on p exposes the correct restriction at the IH.

```
lemma ae_validation_ret_sweep:
  fixes M :: ('q, 'a) mttm
    and ts :: nat ⇒ nat ⇒ ('c :: enum ⇒ 'a)
    and q :: 'q
    and ofs :: nat ⇒ 'c
    and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
    and dest :: nat ⇒ ae_dest
    and p :: nat
  assumes q_in: q ∈ Q_tm M
      and tape_le: ts (0 :: nat) 0 = LE_block (le_tm M)
      and gamma: ∀k i. ts k i ∈ gamma_block (Γ_tm M)
      and buf_gamma: ∀k. fst (buf k) ∈ gamma_block (Γ_tm M)
                       ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
                       ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
      and bt: ∀k≥k_tm M. ∀i. ts k i = bl_block (bl_tm M)
      and stage_tail: (∀j≥k_tm M. ofs j = init_offset j)
                    ∧ (∀j≥k_tm M. buf j = init_buffer (le_tm M) j)
                    ∧ (∀j≥k_tm M. dest j = init_dest j)
  shows (∀i. 1 ≤ i ∧ i ≤ p ⟶ ts 0 i ≠ LE_block (le_tm M))
        ⟶ (Config_M (q, ofs, buf, dest, VRet) ts
                    (λi :: nat. if i = 0 then p else 0),
            Config_M (q, ofs, buf, dest, SS1) ts (λ_. 0))
           ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc p
proof (induction p)
```

```
  case 0
  let ?n0 = λ_ :: nat. 0 :: nat
  have read: ts (0 :: nat) (?n0 0) = LE_block (le_tm M)
    using tape_le by simp
  have gamma_at: ∀ k :: nat. ts k (?n0 k) ∈ gamma_block (Γ_tm M)
    using gamma by simp
  have step1: (Config_M (q, ofs, buf, dest, VRet) ts ?n0,
                Config_M (q, ofs, buf, dest, SS1) ts ?n0)
               ∈ mttm_step (alphabet_enlarge_delta M)
    by (rule ae_step_val_ret_to_sim[where ts = ts and n = ?n0,
                                    OF q_in read bt stage_tail gamma_at
                                      buf_gamma])
  have shape_eq: (λi :: nat. if i = 0 then (0 :: nat) else 0) = ?n0
    by (rule ext) simp
  show ?case
  proof
    assume (∀ i. 1 ≤ i ∧ i ≤ 0 ⟶ ts 0 i ≠ LE_block (le_tm M))
    have one_step: (Config_M (q, ofs, buf, dest, VRet) ts
                      (λi :: nat. if i = 0 then 0 else 0),
                    Config_M (q, ofs, buf, dest, SS1) ts (λ_. 0))
                   ∈ mttm_step (alphabet_enlarge_delta M)
      using step1 shape_eq by simp
    thus (Config_M (q, ofs, buf, dest, VRet) ts
            (λi :: nat. if i = 0 then 0 else 0),
          Config_M (q, ofs, buf, dest, SS1) ts (λ_. 0))
          ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc 0
      by (simp add: relpow_1)
  qed
next
  case (Suc p)
  let ?n_Sp = λi :: nat. if i = 0 then Suc p else 0
  let ?n_p = λi :: nat. if i = 0 then p else 0
  show ?case
  proof
    assume tape_non_le_Sp:
      (∀ i. 1 ≤ i ∧ i ≤ Suc p ⟶ ts 0 i ≠ LE_block (le_tm M))
    have read_non_le: ts (0 :: nat) (?n_Sp 0) ≠ LE_block (le_tm M)
      using tape_non_le_Sp[rule_format, of Suc p] by simp
    have gamma_Sp: ∀ k :: nat. ts k (?n_Sp k) ∈ gamma_block (Γ_tm M)
      using gamma by simp
    have step1: (Config_M (q, ofs, buf, dest, VRet) ts ?n_Sp,
                  Config_M (q, ofs, buf, dest, VRet) ts
                    (λkk. go_dir (if kk = (0 :: nat) then dir.L else dir.N)
                            (?n_Sp kk)))
                 ∈ mttm_step (alphabet_enlarge_delta M)
      by (rule ae_step_val_ret_step[where ts = ts and n = ?n_Sp,
                                    OF q_in read_non_le bt stage_tail
                                      gamma_Sp buf_gamma])
    have post_eq:
```

```
    (λkk :: nat. go_dir (if kk = 0 then dir.L else dir.N) (?n_Sp kk))
     = ?n_p
    by (rule ext) simp
  have step1_p: (Config_M (q, ofs, buf, dest, VRet) ts ?n_Sp,
                 Config_M (q, ofs, buf, dest, VRet) ts ?n_p)
                ∈ mttm_step (alphabet_enlarge_delta M)
    using step1 post_eq by simp
  have tape_non_le_p:
   ∀ i. 1 ≤ i ∧ i ≤ p ⟶ ts 0 i ≠ LE_block (le_tm M)
    using tape_non_le_Sp by auto
  have IH_chain: (Config_M (q, ofs, buf, dest, VRet) ts ?n_p,
                  Config_M (q, ofs, buf, dest, SS1) ts (λ_. 0))
                 ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc p
    using Suc.IH tape_non_le_p by blast
  show (Config_M (q, ofs, buf, dest, VRet) ts ?n_Sp,
        Config_M (q, ofs, buf, dest, SS1) ts (λ_. 0))
        ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc (Suc p)
    by (rule relpow_Suc_I2[OF step1_p IH_chain])
 qed
qed
```

Partial return sweep: m leftward *VRet* steps from head p reach head $p - m$, staying in *VRet*, provided every visited cell $1 \ldots p$ is non-*LE* (so the return never short-circuits to *SS1*). This exposes each intermediate validation config as a reachable witness, which the full *ae_validation_ret_sweep* hides behind its composed endpoint. Used by *ae_validation_prefix_markers* to supply return-phase witnesses for the prefix-uniqueness induction.

```
lemma ae_validation_ret_partial:
 fixes M :: ('q, 'a) mttm
   and ts :: nat ⇒ nat ⇒ ('c :: enum ⇒ 'a)
   and q :: 'q
   and ofs :: nat ⇒ 'c
   and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
   and dest :: nat ⇒ ae_dest
   and p :: nat
 assumes q_in: q ∈ Q_tm M
     and gamma: ∀ k i. ts k i ∈ gamma_block (Γ_tm M)
     and buf_gamma: ∀ k. fst (buf k) ∈ gamma_block (Γ_tm M)
                       ∧ fst (snd (buf k)) ∈ gamma_block (Γ_tm M)
                       ∧ snd (snd (buf k)) ∈ gamma_block (Γ_tm M)
     and bt: ∀ k≥k_tm M. ∀ i. ts k i = bl_block (bl_tm M)
     and stage_tail: (∀ j≥k_tm M. ofs j = init_offset j)
                   ∧ (∀ j≥k_tm M. buf j = init_buffer (le_tm M) j)
                   ∧ (∀ j≥k_tm M. dest j = init_dest j)
     and non_le: ∀ i. 1 ≤ i ∧ i ≤ p ⟶ ts 0 i ≠ LE_block (le_tm M)
 shows m ≤ p
       ⟶ (Config_M (q, ofs, buf, dest, VRet) ts
                   (λi :: nat. if i = 0 then p else 0),
          Config_M (q, ofs, buf, dest, VRet) ts
```

```
                (λi :: nat. if i = 0 then p − m else 0))
          ∈ mttm_step (alphabet_enlarge_delta M) ^^ m
proof (induction m)
  case 0
  have eq: p − 0 = p by simp
  show ?case unfolding eq by simp
next
  case (Suc m)
  let ?n_pm = λi :: nat. if i = 0 then p − m else 0
  let ?n_pSm = λi :: nat. if i = 0 then p − Suc m else 0
  show ?case
  proof
    assume Sm_le: Suc m ≤ p
    have m_le: m ≤ p using Sm_le by simp
    have pm_pos: 1 ≤ p − m using Sm_le by simp
    have pm_le: p − m ≤ p by simp
    have read_non_le: ts (0 :: nat) (?n_pm 0) ≠ LE_block (le_tm M)
      using non_le[rule_format, of p − m] pm_pos pm_le by simp
    have gamma_pm: ∀ k :: nat. ts k (?n_pm k) ∈ gamma_block (Γ_tm M)
      using gamma by simp
    have step1: (Config_M (q, ofs, buf, dest, VRet) ts ?n_pm,
                 Config_M (q, ofs, buf, dest, VRet) ts
                   (λkk. go_dir (if kk = (0 :: nat) then dir.L else dir.N)
                     (?n_pm kk)))
                ∈ mttm_step (alphabet_enlarge_delta M)
      by (rule ae_step_val_ret_step[where ts = ts and n = ?n_pm,
                                    OF q_in read_non_le bt stage_tail
                                      gamma_pm buf_gamma])
    have post_eq:
      (λkk :: nat. go_dir (if kk = 0 then dir.L else dir.N) (?n_pm kk))
       = ?n_pSm
      using pm_pos by (rule_tac ext) simp
    have step1_pSm:
      (Config_M (q, ofs, buf, dest, VRet) ts ?n_pm,
         Config_M (q, ofs, buf, dest, VRet) ts ?n_pSm)
        ∈ mttm_step (alphabet_enlarge_delta M)
      using step1 post_eq by simp
    have IH_chain:
      (Config_M (q, ofs, buf, dest, VRet) ts
                (λi :: nat. if i = 0 then p else 0),
        Config_M (q, ofs, buf, dest, VRet) ts ?n_pm)
        ∈ mttm_step (alphabet_enlarge_delta M) ^^ m
      using Suc.IH m_le by blast
    show (Config_M (q, ofs, buf, dest, VRet) ts
                   (λi :: nat. if i = 0 then p else 0),
          Config_M (q, ofs, buf, dest, VRet) ts ?n_pSm)
          ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc m
      by (rule relpow_Suc_I[OF IH_chain step1_pSm])
  qed
```

qed

Tape correspondence at the initial configuration: the substrate's *init_config* tape (raw input *u*) corresponds to the AE-side *ae_init_config* tape (encoded block list *encode_input bl_M u*) under *ae_tape_correspondence*. The three position regions match one-to-one:

- position *0*: substrate has *LE = le_M*; ae has *LE_block* (the correspondence's *tM 0 = le* conjunct);
- tape *0*, position $p = (s-1)\cdot c + c_idx\ i + 1 \leq length\ u$: substrate has *u ! (p−1)*; ae has *encode_input ! (s−1)* at offset *i*, which by *encode_input_nth* is *u ! (p−1)*;
- any position past the input or any tape $k \neq 0$: both sides deliver *bl_M = blank_M*.

```
lemma ae_tape_correspondence_init:
  fixes M :: ('q, 'a) mttm
    and u :: 'a list
  assumes vM: valid_mttm M
      and u_sub: set u ⊆ Sigma_tm M
  shows ∀k<k_tm M. ae_tape_correspondence (le_tm M)
            (mt_tape (init_config_mttm M u) k)
            (mt_tape (ae_init_config M
                      (encode_input (bl_tm M) u
                        :: ('c :: enum ⇒ 'a) list)) k)
proof (intro allI impI)
  fix k :: nat
  assume k_lt: k < k_tm M
  have kpos: 0 < k_tm M using k_lt by linarith
  obtain Q Σ Γ bl le δ sM tM r kM where MTTM:
    M = MTTM Q Σ Γ bl le δ sM tM r kM
    using mttm.exhaust by metis
  let ?w = encode_input (bl_tm M) u :: ('c :: enum ⇒ 'a) list
  let ?c = card (UNIV :: 'c set)
  show ae_tape_correspondence (le_tm M)
         (mt_tape (init_config_mttm M u) k)
         (mt_tape (ae_init_config M ?w) k)
    unfolding ae_tape_correspondence_def
  proof (intro conjI allI impI)
    show mt_tape (init_config_mttm M u) k 0 = le_tm M
      using MTTM k_lt by simp
  next
    fix s :: nat and i :: 'c
    assume s_ge: 1 ≤ s
    let ?p = (s − 1) * ?c + c_idx i + 1
    have p_pos: ?p ≥ 1 by simp
    show mt_tape (init_config_mttm M u) k ?p
```

```
    = mt_tape (ae_init_config M ?w) k s i
proof (cases k = 0)
  case False
  have subst:
    mt_tape (init_config_mttm M u) k ?p = bl_tm M
    using MTTM False p_pos by simp
  have ae:
    mt_tape (ae_init_config M ?w) k s i = bl_tm M
    unfolding ae_init_config_def using s_ge False
    by (simp add: bl_block_def)
  show ?thesis using subst ae by simp
next
  case True
  have k_eq: k = 0 using True .
  show ?thesis
  proof (cases ?p ≤ length u)
    case False
    have p_gt: length u < ?p using False by simp
    have subst_bl:
      mt_tape (init_config_mttm M u) k ?p = bl_tm M
      using MTTM k_eq p_pos p_gt by simp
    have ae_bl:
      mt_tape (ae_init_config M ?w) k s i = bl_tm M
    proof (cases s ≤ length ?w)
      case False
      thus ?thesis
        unfolding ae_init_config_def using s_ge k_eq
        by (simp add: bl_block_def)
    next
      case True
      have s_minus_1_lt: s − 1 < length ?w
        using True s_ge by simp
      have nth_eq:
        (?w ! (s − 1)) i =
            (let j = (s − 1) * ?c + c_idx i in
                if j < length u then u ! j else bl_tm M)
        by (rule encode_input_nth[OF s_minus_1_lt])
      have j_ge: (s − 1) * ?c + c_idx i ≥ length u
        using p_gt by simp
      have ae_val: (?w ! (s − 1)) i = bl_tm M
        using nth_eq j_ge by (simp add: Let_def)
      have ae_lhs:
        mt_tape (ae_init_config M ?w) k s i = (?w ! (s − 1)) i
        unfolding ae_init_config_def using s_ge k_eq True kpos by simp
      show ?thesis using ae_lhs ae_val by simp
    qed
    show ?thesis using subst_bl ae_bl by simp
  next
    case True
```

```
      have p_le: ?p ≤ length u using True .
      have subst_val:
        mt_tape (init_config_mttm M u) k ?p
          = u ! ((s − 1) ∗ ?c + c_idx i)
        using MTTM k_eq p_pos p_le kpos by simp
      have c_pos: ?c > 0 using c_idx_lt_card[where x = i] by linarith
      have len_w: length ?w = (length u + ?c − 1) div ?c
        by (rule length_encode_input)
      have prod_lt: (s − 1) ∗ ?c < length u
        using p_le by linarith
      have s_le_len_w: s ≤ length ?w
      proof −
        have s_times_c_eq: s ∗ ?c = (s − 1) ∗ ?c + ?c
          using s_ge by (auto simp: algebra_simps)
        have step: s ∗ ?c ≤ length u + ?c − 1
          using prod_lt c_pos s_times_c_eq by linarith
        have s_div: s ∗ ?c div ?c = s using c_pos by simp
        have s = s ∗ ?c div ?c using s_div by simp
        also have ... ≤ (length u + ?c − 1) div ?c
          using step by (rule div_le_mono)
        finally show ?thesis using len_w by simp
      qed
      have s_minus_1_lt: s − 1 < length ?w
        using s_le_len_w s_ge by simp
      have nth_eq:
        (?w ! (s − 1)) i =
            (let j = (s − 1) ∗ ?c + c_idx i in
                if j < length u then u ! j else bl_tm M)
        by (rule encode_input_nth[OF s_minus_1_lt])
      have j_lt: (s − 1) ∗ ?c + c_idx i < length u
        using p_le by simp
      have ae_val: (?w ! (s − 1)) i = u ! ((s − 1) ∗ ?c + c_idx i)
        using nth_eq j_lt by (simp add: Let_def)
      have ae_lhs:
        mt_tape (ae_init_config M ?w) k s i = (?w ! (s − 1)) i
        unfolding ae_init_config_def using s_ge k_eq s_le_len_w kpos
        by simp
      show ?thesis using subst_val ae_lhs ae_val by simp
    qed
  qed
 qed
qed
```

2.20.2 Validation post-state and step count

Well-formed inputs (without *bl_block*) drive the validation chain to the explicit SS1 boundary configuration. Extracted from the well-formed branch of *ae_validation_steps_bound* so that *ae_validation_post_state_canonical* can use the same chain to prove the simulation against the substrate's

init_config (the steps-bound lemma's *obtains* form discards the explicit final config).

This is the narrower companion of the general step-count form *ae_validation_phase_step_count* in theory *AlphabetEnlargement*: it gives the exact length $2 \cdot |w| + 4$ with SS1 as the only outcome under the stronger precondition *ae_input_well_formed* (*bl_tm M*) *w*, whereas the general form gives an existential bound $n \leq 2 \cdot |w| + f_v$ with SS1-or-reject outcomes for any input (well-formed or not). The linear-speedup proof uses this canonical form, applied via *encode_input_well_formed* to explicit encoder-image inputs.

```
lemma ae_validation_well_formed_to_SS1:
  fixes M :: ('q, 'a) mttm
    and w :: ('c :: enum ⇒ 'a) list
  assumes vM: valid_mttm M
      and w_sub: set w ⊆ gamma_block (Sigma_tm M ∪ {bl_tm M})
      and wf: ae_input_well_formed (bl_tm M) w
      and s_in_Q: s_tm M ∈ Q_tm M
      and s_neq_t: s_tm M ≠ t_tm M
      and s_neq_r: s_tm M ≠ r_tm M
      and le_neq_bl: le_tm M ≠ bl_tm M
  shows (ae_init_config M w,
         Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                   init_dest, SS1)
                  (mt_tape (ae_init_config M w)) (λ_. 0))
           ∈ mttm_step (alphabet_enlarge_delta M) ^^ (2 * length w + 4)
proof −
  let ?init = ae_init_config M w
  let ?ts = mt_tape ?init
  have kpos: 0 < k_tm M by (rule valid_mttm_k_pos[OF vM])
  have bt_all: ∀ j≥k_tm M. ∀ i. ?ts j i = bl_block (bl_tm M)
    using ae_init_config_tape_blank_tail by blast
  have buf_gamma_init:
     ∀ kk :: nat.
        fst (init_buffer (le_tm M) kk
               :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
           ∈ gamma_block (Γ_tm M)
        ∧ fst (snd (init_buffer (le_tm M) kk
                      :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)))
           ∈ gamma_block (Γ_tm M)
        ∧ snd (snd (init_buffer (le_tm M) kk
                      :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)))
           ∈ gamma_block (Γ_tm M)
    by (rule init_buffer_in_gamma_block_at_M[OF vM])
  have LE_neq_bl_block: LE_block (le_tm M) ≠ bl_block (bl_tm M)
    using le_neq_bl unfolding LE_block_def bl_block_def
    by (metis fun_eq_iff)
  have LE_notin: LE_block (le_tm M) ∉ set w
  proof −
```

```
  have le_notin: le_tm M ∉ Sigma_tm M ∪ {bl_tm M}
    using valid_mttm_LE_not_Sigma[OF vM] le_neq_bl by auto
  have LE_block (le_tm M) ∉ gamma_block (Sigma_tm M ∪ {bl_tm M})
    using le_notin
    unfolding LE_block_def gamma_block_def by auto
  thus ?thesis using w_sub by auto
qed
— Determine whether all w-cells are pure or the last is padded.
have all_pure_or_last_padded:
  (∀ i < length w. is_pure_block (bl_tm M) (w ! i))
   ∨ (0 < length w
        ∧ (∀ i < length w − 1. is_pure_block (bl_tm M) (w ! i))
        ∧ is_padded_block (bl_tm M) (w ! (length w − 1)))
proof (cases length w = 0)
  case True
  thus ?thesis by simp
next
  case False
  hence wpos: 0 < length w by simp
  show ?thesis
  proof (cases is_pure_block (bl_tm M) (w ! (length w − 1)))
    case True
    have ∀ i < length w. is_pure_block (bl_tm M) (w ! i)
    proof (intro allI impI)
      fix i assume i_lt: i < length w
      have is_pure_block (bl_tm M) (w ! i)
           ∨ (i = length w − 1
              ∧ is_padded_block (bl_tm M) (w ! i))
        using wf i_lt unfolding ae_input_well_formed_def by auto
      thus is_pure_block (bl_tm M) (w ! i)
        using True by auto
    qed
    thus ?thesis by simp
  next
    case False
    have last_padded: is_padded_block (bl_tm M) (w ! (length w − 1))
    proof −
      have idx_lt: length w − 1 < length w using wpos by linarith
      have or_form: is_pure_block (bl_tm M) (w ! (length w − 1))
           ∨ (length w − 1 = length w − 1
              ∧ is_padded_block (bl_tm M) (w ! (length w − 1)))
        using wf idx_lt unfolding ae_input_well_formed_def by blast
      thus ?thesis using False by auto
    qed
    have prefix_pure: ∀ i < length w − 1. is_pure_block (bl_tm M) (w ! i)
    proof (intro allI impI)
      fix i assume i_lt: i < length w − 1
      hence i_lt2: i < length w using wpos by linarith
      have is_pure_block (bl_tm M) (w ! i)
```

```
          ∨ (i = length w − 1
            ∧ is_padded_block (bl_tm M) (w ! i))
        using wf i_lt2 unfolding ae_input_well_formed_def by auto
      thus is_pure_block (bl_tm M) (w ! i) using i_lt by auto
    qed
    show ?thesis using wpos prefix_pure last_padded by blast
  qed
qed
have tape_le: ?ts (0 :: nat) 0 = LE_block (le_tm M)
  by (rule ae_init_config_tape_le[OF kpos])
have gamma_all: ∀ k i. ?ts k i ∈ gamma_block (Γ_tm M)
  using ae_init_config_in_gamma_block[OF vM w_sub] by simp
have stage_eq: init_stage (le_tm M)
             = (init_offset, init_buffer (le_tm M), init_dest, VFwd)
  unfolding init_stage_def ..
— Verify positions 1 ... Suc (length w) on tape 0 are non-LE (from LE_notin for
input positions; from LE_neq_bl_block for the blank tail).
have non_le: ∀ i. 1 ≤ i ∧ i ≤ Suc (length w)
                ⟶ ?ts 0 i ≠ LE_block (le_tm M)
proof (intro allI impI)
  fix i assume i_range: 1 ≤ i ∧ i ≤ Suc (length w)
  have i_pos: 1 ≤ i using i_range by simp
  consider (in_input) i ≤ length w | (past) i = Suc (length w)
    using i_range by linarith
  thus ?ts 0 i ≠ LE_block (le_tm M)
  proof cases
    case in_input
    have eq: ?ts 0 i = w ! (i − 1)
      using ae_init_config_tape_input[OF i_pos in_input kpos] by simp
    have w ! (i − 1) ∈ set w using i_pos in_input by auto
    thus ?thesis using LE_notin eq by auto
  next
    case past
    have ineq: length w < i using past by simp
    have eq: ?ts 0 i = bl_block (bl_tm M)
      by (rule ae_init_config_tape_blank_after_input[OF ineq])
    have LE_block (le_tm M) ≠ bl_block (bl_tm M)
      by (rule LE_neq_bl_block)
    thus ?thesis using eq by force
  qed
qed
from all_pure_or_last_padded consider
    (all_pure) ∀ i < length w. is_pure_block (bl_tm M) (w ! i)
  | (last_padded) 0 < length w
                ∀ i < length w − 1. is_pure_block (bl_tm M) (w ! i)
                is_padded_block (bl_tm M) (w ! (length w − 1))
  by blast
thus ?thesis
proof cases
```

case *all_pure*
let *?n_lwSS* = λi :: *nat*. *if* i = *0 then Suc* (*length w*) *else 0*
have *lw_le*: *length w* $\leq$ *length w* **by** *simp*
have *sweep*:
(*?init*,
Config$_M$ (*s_tm M*, *init_stage* (*le_tm M*)) *?ts ?n_lwSS*)
$\in$ *mttm_step* (*alphabet_enlarge_delta M*) $^{\frown\frown}$ *Suc* (*length w*)
by (*rule ae_validation_fwd_sweep_pure*[*OF vM w_sub s_in_Q s_neq_t s_neq_r lw_le all_pure*])
have *lw_gt*: *Suc* (*length w*) > *length w* **by** *simp*
have *read_bl*: *?ts 0* (*?n_lwSS 0*) = *bl_block* (*bl_tm M*)
using *ae_init_config_tape_blank_after_input*[*OF lw_gt*] **by** *simp*
have *gamma_lwSS*:
$\forall$ *kk* :: *nat*. *?ts kk* (*?n_lwSS kk*) $\in$ *gamma_block* (Γ*_tm M*)
using *gamma_all* **by** *simp*
have *step_to_ret*:
(*Config*$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VFwd*) *?ts ?n_lwSS*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_lwSS*)
$\in$ *mttm_step* (*alphabet_enlarge_delta M*)
by (*rule ae_step_val_fwd_to_ret*[**where** *ts* = *?ts* **and** *n* = *?n_lwSS*,
OF s_in_Q s_neq_t s_neq_r read_bl bt_all init_stage_tail gamma_lwSS buf_gamma_init])
have *step_to_ret_init*:
(*Config*$_M$ (*s_tm M*, *init_stage* (*le_tm M*)) *?ts ?n_lwSS*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_lwSS*)
$\in$ *mttm_step* (*alphabet_enlarge_delta M*)
using *step_to_ret* **unfolding** *stage_eq* **by** *simp*
have *post_sweep*:
(*?init*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_lwSS*)
$\in$ *mttm_step* (*alphabet_enlarge_delta M*) $^{\frown\frown}$ *Suc* (*Suc* (*length w*))
by (*rule relpow_Suc_I*[*OF sweep step_to_ret_init*])
have *ret_arrow*:
($\forall$ *i*. 1 $\leq$ *i* $\wedge$ *i* $\leq$ *Suc* (*length w*)
$\longrightarrow$ *?ts 0 i* $\neq$ *LE_block* (*le_tm M*))
$\longrightarrow$ (*Config*$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_lwSS*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *SS1*) *?ts* (λ_. *0*))
$\in$ *mttm_step* (*alphabet_enlarge_delta M*)
$^{\frown\frown}$ *Suc* (*Suc* (*length w*))
by (*rule ae_validation_ret_sweep*[*OF s_in_Q tape_le gamma_all buf_gamma_init bt_all init_stage_tail*])
have *ret_chain*:
(*Config*$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),

```
            init_dest, VRet) ?ts ?n_lwSS,
      Config_M (s_tm M, init_offset, init_buffer (le_tm M),
            init_dest, SS1) ?ts (λ_. 0))
     ∈ mttm_step (alphabet_enlarge_delta M)
        ^^ Suc (Suc (length w))
  using ret_arrow non_le by (rule mp)
 have full_chain_pre:
  (?init,
     Config_M (s_tm M, init_offset, init_buffer (le_tm M),
            init_dest, SS1) ?ts (λ_. 0))
     ∈ mttm_step (alphabet_enlarge_delta M)
        ^^ (Suc (Suc (length w)) + Suc (Suc (length w)))
 proof −
  have comp:
   (?init,
      Config_M (s_tm M, init_offset, init_buffer (le_tm M),
            init_dest, SS1) ?ts (λ_. 0))
     ∈ mttm_step (alphabet_enlarge_delta M)
        ^^ Suc (Suc (length w))
       O mttm_step (alphabet_enlarge_delta M)
        ^^ Suc (Suc (length w))
    using post_sweep ret_chain by (rule relcompI)
  thus ?thesis by (simp only: relpow_add)
 qed
 have total_eq:
  Suc (Suc (length w)) + Suc (Suc (length w)) = 2 * length w + 4
  by simp
 have full_chain:
  (?init,
     Config_M (s_tm M, init_offset, init_buffer (le_tm M),
            init_dest, SS1) ?ts (λ_. 0))
     ∈ mttm_step (alphabet_enlarge_delta M)
        ^^ (2 * length w + 4)
  using full_chain_pre unfolding total_eq .
 show ?thesis using full_chain .
next
 case last_padded
 let ?lw = length w
 let ?lwm1 = length w − 1
 let ?n_lw = λi :: nat. if i = 0 then ?lw else 0
 let ?n_lwSS = λi :: nat. if i = 0 then Suc ?lw else 0
 have wpos: 0 < ?lw using last_padded(1) .
 have m1_le: ?lwm1 ≤ ?lw by simp
 have prefix_pure: ∀ i < ?lwm1. is_pure_block (bl_tm M) (w ! i)
  using last_padded(2) .
 have sweep:
  (?init,
     Config_M (s_tm M, init_stage (le_tm M)) ?ts
           (λi :: nat. if i = 0 then Suc ?lwm1 else 0))
```

```
    ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc ?lwm1
  by (rule ae_validation_fwd_sweep_pure[OF vM w_sub s_in_Q s_neq_t
                                         s_neq_r m1_le prefix_pure])
have suc_m1_eq: Suc ?lwm1 = ?lw using wpos by simp
have sweep_lw:
  (?init,
     Config_M (s_tm M, init_stage (le_tm M)) ?ts ?n_lw)
     ∈ mttm_step (alphabet_enlarge_delta M) ^^ ?lw
  using sweep unfolding suc_m1_eq .
have lw_pos: 1 ≤ ?lw using wpos by linarith
have lw_le: ?lw ≤ ?lw by simp
have read_at_lw_eq: ?ts 0 ?lw = w ! ?lwm1
  using ae_init_config_tape_input[OF lw_pos lw_le kpos] by simp
have read_padded:
  is_padded_block (bl_tm M) (?ts 0 (?n_lw 0))
  using read_at_lw_eq last_padded(3) by simp
have gamma_lw:
  ∀ kk :: nat. ?ts kk (?n_lw kk) ∈ gamma_block (Γ_tm M)
  using gamma_all by simp
have step_to_pad:
  (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
             init_dest, VFwd) ?ts ?n_lw,
    Config_M (s_tm M, init_offset, init_buffer (le_tm M),
             init_dest, VFwdPad) ?ts
          (λkk. go_dir (if kk = (0 :: nat) then dir.R else dir.N)
                  (?n_lw kk)))
     ∈ mttm_step (alphabet_enlarge_delta M)
  by (rule ae_step_val_fwd_to_padded[where ts = ?ts and n = ?n_lw,
       OF s_in_Q s_neq_t s_neq_r read_padded bt_all init_stage_tail
         gamma_lw buf_gamma_init])
have post_eq_pad:
  (λkk :: nat. go_dir (if kk = 0 then dir.R else dir.N) (?n_lw kk))
  = ?n_lwSS
  by (rule ext) simp
have step_to_pad_norm:
  (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
             init_dest, VFwd) ?ts ?n_lw,
    Config_M (s_tm M, init_offset, init_buffer (le_tm M),
             init_dest, VFwdPad) ?ts ?n_lwSS)
     ∈ mttm_step (alphabet_enlarge_delta M)
  using step_to_pad post_eq_pad by simp
have step_to_pad_init:
  (Config_M (s_tm M, init_stage (le_tm M)) ?ts ?n_lw,
    Config_M (s_tm M, init_offset, init_buffer (le_tm M),
             init_dest, VFwdPad) ?ts ?n_lwSS)
     ∈ mttm_step (alphabet_enlarge_delta M)
  using step_to_pad_norm unfolding stage_eq by simp
have lw_gt: Suc ?lw > ?lw by simp
have read_bl_lwSS: ?ts 0 (?n_lwSS 0) = bl_block (bl_tm M)
```

using *ae_init_config_tape_blank_after_input*[*OF lw_gt*] **by** *simp*
have *gamma_lwSS*:
∀ *kk* :: *nat*. *?ts kk* (*?n_lwSS kk*) ∈ *gamma_block* (Γ_*tm M*)
using *gamma_all* **by** *simp*
have *step_pad_to_ret*:
(*Config*$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VFwdPad*) *?ts ?n_lwSS*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_lwSS*)
∈ *mttm_step* (*alphabet_enlarge_delta M*)
by (*rule ae_step_val_pad_to_ret*[**where** *ts* = *?ts* **and** *n* = *?n_lwSS*,
OF s_in_Q read_bl_lwSS bt_all init_stage_tail
gamma_lwSS buf_gamma_init])
have *chain_lw_to_VFwdPad*:
(*?init*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VFwdPad*) *?ts ?n_lwSS*)
∈ *mttm_step* (*alphabet_enlarge_delta M*) ^^ *Suc ?lw*
by (*rule relpow_Suc_I*[*OF sweep_lw step_to_pad_init*])
have *chain_to_VRet*:
(*?init*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_lwSS*)
∈ *mttm_step* (*alphabet_enlarge_delta M*) ^^ *Suc* (*Suc ?lw*)
by (*rule relpow_Suc_I*[*OF chain_lw_to_VFwdPad step_pad_to_ret*])
have *ret_arrow*:
(∀ *i*. 1 ≤ *i* ∧ *i* ≤ *Suc ?lw*
⟶ *?ts* 0 *i* ≠ *LE_block* (*le_tm M*))
⟶ (*Config*$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_lwSS*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *SS1*) *?ts* (λ_. 0))
∈ *mttm_step* (*alphabet_enlarge_delta M*)
^^ *Suc* (*Suc ?lw*)
by (*rule ae_validation_ret_sweep*[*OF s_in_Q tape_le gamma_all*
buf_gamma_init bt_all init_stage_tail])
have *ret_chain*:
(*Config*$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_lwSS*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *SS1*) *?ts* (λ_. 0))
∈ *mttm_step* (*alphabet_enlarge_delta M*)
^^ *Suc* (*Suc ?lw*)
using *ret_arrow non_le* **by** (*rule mp*)
have *full_chain_pre*:
(*?init*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *SS1*) *?ts* (λ_. 0))
∈ *mttm_step* (*alphabet_enlarge_delta M*)

```
          ^^ (Suc (Suc ?lw) + Suc (Suc ?lw))
  proof −
    have comp:
      (?init,
        Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                  init_dest, SS1) ?ts (λ_. 0))
       ∈ mttm_step (alphabet_enlarge_delta M)
           ^^ Suc (Suc ?lw)
         O mttm_step (alphabet_enlarge_delta M)
           ^^ Suc (Suc ?lw)
      using chain_to_VRet ret_chain by (rule relcompI)
    thus ?thesis by (simp only: relpow_add)
  qed
  have total_eq:
    Suc (Suc ?lw) + Suc (Suc ?lw) = 2 * ?lw + 4
    by simp
  have full_chain:
    (?init,
      Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                init_dest, SS1) ?ts (λ_. 0))
     ∈ mttm_step (alphabet_enlarge_delta M)
         ^^ (2 * ?lw + 4)
    using full_chain_pre unfolding total_eq .
  show ?thesis using full_chain .
 qed
qed
```

Every prefix of the validation run carries a *marker* witness: a config reachable in exactly i steps whose stage index is one of *VFwd* / *VFwdPad* / *VRet* (i.e. still inside the validation sweep, not yet at *SS1* nor rejected), for every $i < 2 * length\ w + 4$. Forward witnesses for $i \leq length\ w$ come from *ae_validation_fwd_sweep_pure* at parameter $i - 1$; the boundary slot $i = Suc\ (length\ w)$ is *VFwd* (all-pure) or *VFwdPad* (last-padded); return witnesses come from the partial return sweep. This is the witness half of the prefix-uniqueness argument in *ae_validation_prefix_markers*.

```
lemma ae_validation_prefix_witnesses:
 fixes M :: ('q, 'a) mttm
   and w :: ('c :: enum ⇒ 'a) list
 assumes vM: valid_mttm M
     and w_sub: set w ⊆ gamma_block (Sigma_tm M ∪ {bl_tm M})
     and wf: ae_input_well_formed (bl_tm M) w
     and s_in_Q: s_tm M ∈ Q_tm M
     and s_neq_t: s_tm M ≠ t_tm M
     and s_neq_r: s_tm M ≠ r_tm M
     and le_neq_bl: le_tm M ≠ bl_tm M
 shows ∀ i<2 * length w + 4. ∃ d.
         (ae_init_config M w, d)
           ∈ mttm_step (alphabet_enlarge_delta M) ^^ i
         ∧ snd (snd (snd (snd (mt_state d))))
```

$\in \{VFwd,\ VFwdPad,\ VRet\}$
proof −
let *?init* = *ae_init_config M w*
let *?ts* = *mt_tape ?init*
let *?R* = *mttm_step* (*alphabet_enlarge_delta M*)
let *?lw* = *length w*
let *?vfwd* = $\lambda h :: nat.$ $Config_M$ (*s_tm M*, *init_offset*,
init_buffer (*le_tm M*), *init_dest*, *VFwd*) *?ts*
($\lambda i :: nat.$ *if i = 0 then h else 0*)
let *?vret* = $\lambda h :: nat.$ $Config_M$ (*s_tm M*, *init_offset*,
init_buffer (*le_tm M*), *init_dest*, *VRet*) *?ts*
($\lambda i :: nat.$ *if i = 0 then h else 0*)
let *?vpad* = $\lambda h :: nat.$ $Config_M$ (*s_tm M*, *init_offset*,
init_buffer (*le_tm M*), *init_dest*, *VFwdPad*) *?ts*
($\lambda i :: nat.$ *if i = 0 then h else 0*)
have *kpos*: $0 < k_tm\ M$ **by** (*rule valid_mttm_k_pos*[*OF vM*])
have *bt_all*: $\forall j{\geq}k_tm\ M.\ \forall i.$ *?ts j i* = *bl_block* (*bl_tm M*)
using *ae_init_config_tape_blank_tail* **by** *blast*
have *buf_gamma_init*:
$\forall kk :: nat.$
fst (*init_buffer* (*le_tm M*) *kk*
$:: ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$)
$\in$ *gamma_block* (Γ*_tm M*)
$\wedge$ *fst* (*snd* (*init_buffer* (*le_tm M*) *kk*
$:: ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$))
$\in$ *gamma_block* (Γ*_tm M*)
$\wedge$ *snd* (*snd* (*init_buffer* (*le_tm M*) *kk*
$:: ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$))
$\in$ *gamma_block* (Γ*_tm M*)
by (*rule init_buffer_in_gamma_block_at_M*[*OF vM*])
have *LE_neq_bl_block*: *LE_block* (*le_tm M*) $\neq$ *bl_block* (*bl_tm M*)
using *le_neq_bl* **unfolding** *LE_block_def bl_block_def*
by (*metis fun_eq_iff*)
have *LE_notin*: *LE_block* (*le_tm M*) $\notin$ *set w*
proof −
have *le_notin*: *le_tm M* $\notin$ *Sigma_tm M* $\cup$ {*bl_tm M*}
using *valid_mttm_LE_not_Sigma*[*OF vM*] *le_neq_bl* **by** *auto*
have *LE_block* (*le_tm M*) $\notin$ *gamma_block* (*Sigma_tm M* $\cup$ {*bl_tm M*})
using *le_notin* **unfolding** *LE_block_def gamma_block_def* **by** *auto*
thus *?thesis* **using** *w_sub* **by** *auto*
qed
have *tape_le*: *?ts* (*0* :: *nat*) *0* = *LE_block* (*le_tm M*)
by (*rule ae_init_config_tape_le*[*OF kpos*])
have *gamma_all*: $\forall k\ i.$ *?ts k i* $\in$ *gamma_block* (Γ*_tm M*)
using *ae_init_config_in_gamma_block*[*OF vM w_sub*] **by** *simp*
have *stage_eq*: *init_stage* (*le_tm M*)
= (*init_offset*, *init_buffer* (*le_tm M*), *init_dest*, *VFwd*)
unfolding *init_stage_def* **..**
have *non_le_Slw*: $\forall i.\ 1 \leq i \wedge i \leq$ *Suc ?lw*

```
                ⟶ ?ts 0 i ≠ LE_block (le_tm M)
  proof (intro allI impI)
    fix i assume i_range: 1 ≤ i ∧ i ≤ Suc ?lw
    have i_pos: 1 ≤ i using i_range by simp
    consider (in_input) i ≤ ?lw | (past) i = Suc ?lw
      using i_range by linarith
    thus ?ts 0 i ≠ LE_block (le_tm M)
    proof cases
      case in_input
      have eq: ?ts 0 i = w ! (i − 1)
        using ae_init_config_tape_input[OF i_pos in_input kpos] by simp
      have w ! (i − 1) ∈ set w using i_pos in_input by auto
      thus ?thesis using LE_notin eq by auto
    next
      case past
      have ineq: ?lw < i using past by simp
      have eq: ?ts 0 i = bl_block (bl_tm M)
        by (rule ae_init_config_tape_blank_after_input[OF ineq])
      thus ?thesis using LE_neq_bl_block by force
    qed
  qed
```

— Forward witnesses for $i \le$ *length w*, uniformly (the pure-prefix $\forall j < i-1$ holds for any well-formed input).

```
  have vfwd_wit: ⋀i. i ≤ ?lw ⟹ (?init, ?vfwd i) ∈ ?R ^^ i
  proof −
    fix i assume i_le: i ≤ ?lw
    show (?init, ?vfwd i) ∈ ?R ^^ i
    proof (cases i = 0)
      case True
      have head0: (λi :: nat. if i = 0 then 0 else (0 :: nat)) = (λ_. 0)
        by simp
      have eq0: ?vfwd 0 = ?init
        unfolding ae_init_config_def init_stage_def head0 by simp
      have (?init, ?vfwd 0) ∈ ?R ^^ 0 using eq0 relpow_0_I by metis
      thus ?thesis unfolding True .
    next
      case False
      hence i_pos: 1 ≤ i by simp
      have k_le: i − 1 ≤ ?lw using i_le by simp
      have pure_short: ∀j < i − 1. is_pure_block (bl_tm M) (w ! j)
      proof (intro allI impI)
        fix j assume j_lt: j < i − 1
        have j_lt_lw: j < ?lw using j_lt i_le by linarith
        have j_neq: j ≠ ?lw − 1 using j_lt i_le by linarith
        have is_pure_block (bl_tm M) (w ! j)
              ∨ (j = ?lw − 1 ∧ is_padded_block (bl_tm M) (w ! j))
          using wf j_lt_lw unfolding ae_input_well_formed_def by auto
        thus is_pure_block (bl_tm M) (w ! j) using j_neq by auto
      qed
```

```
    have sweep:
      (?init, Config_M (s_tm M, init_stage (le_tm M)) ?ts
                       (λj :: nat. if j = 0 then Suc (i − 1) else 0))
          ∈ ?R ^^ Suc (i − 1)
      by (rule ae_validation_fwd_sweep_pure[OF vM w_sub s_in_Q s_neq_t
                                             s_neq_r k_le pure_short])
    have suci: Suc (i − 1) = i using i_pos by simp
    show ?thesis using sweep unfolding suci stage_eq by simp
  qed
qed
— Classify the input: all blocks pure, or the last padded.
have all_pure_or_last_padded:
  (∀ i < ?lw. is_pure_block (bl_tm M) (w ! i))
  ∨ (0 < ?lw
       ∧ (∀ i < ?lw − 1. is_pure_block (bl_tm M) (w ! i))
       ∧ is_padded_block (bl_tm M) (w ! (?lw − 1)))
proof (cases ?lw = 0)
  case True thus ?thesis by simp
next
  case False
  hence wpos: 0 < ?lw by simp
  show ?thesis
  proof (cases is_pure_block (bl_tm M) (w ! (?lw − 1)))
    case True
    have ∀ i < ?lw. is_pure_block (bl_tm M) (w ! i)
    proof (intro allI impI)
      fix i assume i_lt: i < ?lw
      have is_pure_block (bl_tm M) (w ! i)
           ∨ (i = ?lw − 1 ∧ is_padded_block (bl_tm M) (w ! i))
        using wf i_lt unfolding ae_input_well_formed_def by auto
      thus is_pure_block (bl_tm M) (w ! i) using True by auto
    qed
    thus ?thesis by simp
  next
    case False
    have last_padded: is_padded_block (bl_tm M) (w ! (?lw − 1))
    proof −
      have idx_lt: ?lw − 1 < ?lw using wpos by linarith
      have is_pure_block (bl_tm M) (w ! (?lw − 1))
           ∨ (?lw − 1 = ?lw − 1
                ∧ is_padded_block (bl_tm M) (w ! (?lw − 1)))
        using wf idx_lt unfolding ae_input_well_formed_def by blast
      thus ?thesis using False by auto
    qed
    have prefix_pure: ∀ i < ?lw − 1. is_pure_block (bl_tm M) (w ! i)
    proof (intro allI impI)
      fix i assume i_lt: i < ?lw − 1
      hence i_lt2: i < ?lw using wpos by linarith
      have is_pure_block (bl_tm M) (w ! i)
```

```
          ∨ (i = ?lw − 1 ∧ is_padded_block (bl_tm M) (w ! i))
      using wf i_lt2 unfolding ae_input_well_formed_def by auto
    thus is_pure_block (bl_tm M) (w ! i) using i_lt by auto
   qed
   show ?thesis using wpos prefix_pure last_padded by blast
  qed
 qed
 — Boundary witness at step Suc (length w): VFwd head Suc lw (all-pure) or
VFwdPad head Suc lw (last-padded).
 have boundary_wit: ∃ d. (?init, d) ∈ ?R ^^ Suc ?lw
       ∧ snd (snd (snd (snd (mt_state d)))) ∈ {VFwd, VFwdPad, VRet}
 proof −
  from all_pure_or_last_padded consider
    (all_pure) ∀ i < ?lw. is_pure_block (bl_tm M) (w ! i)
   | (last_padded) 0 < ?lw
                ∀ i < ?lw − 1. is_pure_block (bl_tm M) (w ! i)
                is_padded_block (bl_tm M) (w ! (?lw − 1))
    by blast
  thus ?thesis
  proof cases
   case all_pure
   have lw_le: ?lw ≤ ?lw by simp
   have sweep:
    (?init, Config_M (s_tm M, init_stage (le_tm M)) ?ts
                (λi :: nat. if i = 0 then Suc ?lw else 0))
      ∈ ?R ^^ Suc ?lw
    by (rule ae_validation_fwd_sweep_pure[OF vM w_sub s_in_Q s_neq_t
                                        s_neq_r lw_le all_pure])
   have reach: (?init, ?vfwd (Suc ?lw)) ∈ ?R ^^ Suc ?lw
    using sweep unfolding stage_eq by simp
   have snd (snd (snd (snd (mt_state (?vfwd (Suc ?lw))))))
        ∈ {VFwd, VFwdPad, VRet} by simp
   thus ?thesis using reach by blast
  next
   case last_padded
   let ?n_lw = λi :: nat. if i = 0 then ?lw else 0
   have wpos: 0 < ?lw using last_padded(1) .
   have m1_le: ?lw − 1 ≤ ?lw by simp
   have sweep:
    (?init, Config_M (s_tm M, init_stage (le_tm M)) ?ts
             (λi :: nat. if i = 0 then Suc (?lw − 1) else 0))
      ∈ ?R ^^ Suc (?lw − 1)
    by (rule ae_validation_fwd_sweep_pure[OF vM w_sub s_in_Q s_neq_t
                                        s_neq_r m1_le last_padded(2)])
   have suc_m1_eq: Suc (?lw − 1) = ?lw using wpos by simp
   have sweep_lw: (?init, ?vfwd ?lw) ∈ ?R ^^ ?lw
    using sweep unfolding suc_m1_eq stage_eq by simp
   have lw_pos: 1 ≤ ?lw using wpos by linarith
   have lw_le: ?lw ≤ ?lw by simp
```

```
    have read_at_lw_eq: ?ts 0 ?lw = w ! (?lw − 1)
      using ae_init_config_tape_input[OF lw_pos lw_le kpos] by simp
    have read_padded: is_padded_block (bl_tm M) (?ts 0 (?n_lw 0))
      using read_at_lw_eq last_padded(3) by simp
    have gamma_lw: ∀ kk :: nat. ?ts kk (?n_lw kk) ∈ gamma_block (Γ_tm M)
      using gamma_all by simp
    have step_to_pad:
      (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                 init_dest, VFwd) ?ts ?n_lw,
       Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                 init_dest, VFwdPad) ?ts
                (λkk. go_dir (if kk = (0 :: nat) then dir.R else dir.N)
                             (?n_lw kk)))
       ∈ ?R
      by (rule ae_step_val_fwd_to_padded[where ts = ?ts and n = ?n_lw,
          OF s_in_Q s_neq_t s_neq_r read_padded bt_all init_stage_tail
             gamma_lw buf_gamma_init])
    have post_eq_pad:
      (λkk :: nat. go_dir (if kk = 0 then dir.R else dir.N) (?n_lw kk))
       = (λi :: nat. if i = 0 then Suc ?lw else 0)
      by (rule ext) simp
    have step_to_pad_norm: (?vfwd ?lw, ?vpad (Suc ?lw)) ∈ ?R
      using step_to_pad post_eq_pad by simp
    have reach: (?init, ?vpad (Suc ?lw)) ∈ ?R ^^ Suc ?lw
      by (rule relpow_Suc_I[OF sweep_lw step_to_pad_norm])
    have snd (snd (snd (snd (mt_state (?vpad (Suc ?lw))))))
          ∈ {VFwd, VFwdPad, VRet} by simp
    thus ?thesis using reach by blast
  qed
qed
— Pivot witness into VRet at head Suc lw, reached in Suc (Suc lw) steps (both classification cases).
have to_VRet: (?init, ?vret (Suc ?lw)) ∈ ?R ^^ Suc (Suc ?lw)
proof −
  from all_pure_or_last_padded consider
      (all_pure) ∀ i < ?lw. is_pure_block (bl_tm M) (w ! i)
    | (last_padded) 0 < ?lw
                ∀ i < ?lw − 1. is_pure_block (bl_tm M) (w ! i)
                is_padded_block (bl_tm M) (w ! (?lw − 1))
    by blast
  thus ?thesis
  proof cases
    case all_pure
    let ?n_Slw = λi :: nat. if i = 0 then Suc ?lw else 0
    have lw_le: ?lw ≤ ?lw by simp
    have sweep: (?init, ?vfwd (Suc ?lw)) ∈ ?R ^^ Suc ?lw
      using ae_validation_fwd_sweep_pure[OF vM w_sub s_in_Q s_neq_t
            s_neq_r lw_le all_pure] unfolding stage_eq by simp
    have lw_gt: Suc ?lw > ?lw by simp
```

```
  have read_bl: ?ts 0 (?n_Slw 0) = bl_block (bl_tm M)
    using ae_init_config_tape_blank_after_input[OF lw_gt] by simp
  have gamma_Slw: ∀ kk :: nat. ?ts kk (?n_Slw kk) ∈ gamma_block (Γ_tm M)
    using gamma_all by simp
  have step_to_ret: (?vfwd (Suc ?lw), ?vret (Suc ?lw)) ∈ ?R
    by (rule ae_step_val_fwd_to_ret[where ts = ?ts and n = ?n_Slw,
          OF s_in_Q s_neq_t s_neq_r read_bl bt_all init_stage_tail
            gamma_Slw buf_gamma_init])
  show ?thesis by (rule relpow_Suc_I[OF sweep step_to_ret])
next
  case last_padded
  let ?n_lw = λi :: nat. if i = 0 then ?lw else 0
  let ?n_Slw = λi :: nat. if i = 0 then Suc ?lw else 0
  have wpos: 0 < ?lw using last_padded(1) .
  have m1_le: ?lw − 1 ≤ ?lw by simp
  have sweep:
    (?init, Config_M (s_tm M, init_stage (le_tm M)) ?ts
            (λi :: nat. if i = 0 then Suc (?lw − 1) else 0))
      ∈ ?R ^^ Suc (?lw − 1)
    by (rule ae_validation_fwd_sweep_pure[OF vM w_sub s_in_Q s_neq_t
                                          s_neq_r m1_le last_padded(2)])
  have suc_m1_eq: Suc (?lw − 1) = ?lw using wpos by simp
  have sweep_lw: (?init, ?vfwd ?lw) ∈ ?R ^^ ?lw
    using sweep unfolding suc_m1_eq stage_eq by simp
  have lw_pos: 1 ≤ ?lw using wpos by linarith
  have lw_le: ?lw ≤ ?lw by simp
  have read_at_lw_eq: ?ts 0 ?lw = w ! (?lw − 1)
    using ae_init_config_tape_input[OF lw_pos lw_le kpos] by simp
  have read_padded: is_padded_block (bl_tm M) (?ts 0 (?n_lw 0))
    using read_at_lw_eq last_padded(3) by simp
  have gamma_lw: ∀ kk :: nat. ?ts kk (?n_lw kk) ∈ gamma_block (Γ_tm M)
    using gamma_all by simp
  have step_to_pad:
    (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
               init_dest, VFwd) ?ts ?n_lw,
      Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                init_dest, VFwdPad) ?ts
               (λkk. go_dir (if kk = (0 :: nat) then dir.R else dir.N)
                            (?n_lw kk)))
      ∈ ?R
    by (rule ae_step_val_fwd_to_padded[where ts = ?ts and n = ?n_lw,
          OF s_in_Q s_neq_t s_neq_r read_padded bt_all init_stage_tail
            gamma_lw buf_gamma_init])
  have post_eq_pad:
    (λkk :: nat. go_dir (if kk = 0 then dir.R else dir.N) (?n_lw kk))
     = ?n_Slw
    by (rule ext) simp
  have step_to_pad_norm: (?vfwd ?lw, ?vpad (Suc ?lw)) ∈ ?R
    using step_to_pad post_eq_pad by simp
```

```
    have chain_lw_to_VFwdPad: (?init, ?vpad (Suc ?lw)) ∈ ?R ^^ Suc ?lw
      by (rule relpow_Suc_I[OF sweep_lw step_to_pad_norm])
    have read_bl_Slw: ?ts 0 (?n_Slw 0) = bl_block (bl_tm M)
      using ae_init_config_tape_blank_after_input[OF lessI] by simp
    have gamma_Slw: ∀ kk :: nat. ?ts kk (?n_Slw kk) ∈ gamma_block (Γ_tm M)
      using gamma_all by simp
    have step_pad_to_ret: (?vpad (Suc ?lw), ?vret (Suc ?lw)) ∈ ?R
      by (rule ae_step_val_pad_to_ret[where ts = ?ts and n = ?n_Slw,
            OF s_in_Q read_bl_Slw bt_all init_stage_tail
              gamma_Slw buf_gamma_init])
    show ?thesis
      by (rule relpow_Suc_I[OF chain_lw_to_VFwdPad step_pad_to_ret])
  qed
qed
```

— Return witnesses: m leftward steps from the pivot reach $VRet$ head $Suc\ lw - m$ in $Suc\ (Suc\ lw) + m$ steps.

```
have ret_reach: ⋀m. m ≤ Suc ?lw
      ⟹ (?init, ?vret (Suc ?lw − m)) ∈ ?R ^^ (Suc (Suc ?lw) + m)
proof −
  fix m assume mle: m ≤ Suc ?lw
  have rp: (?vret (Suc ?lw), ?vret (Suc ?lw − m)) ∈ ?R ^^ m
    by (rule mp[OF ae_validation_ret_partial[OF s_in_Q gamma_all
          buf_gamma_init bt_all init_stage_tail non_le_Slw] mle])
  have (?init, ?vret (Suc ?lw − m)) ∈ ?R ^^ Suc (Suc ?lw) O ?R ^^ m
    using to_VRet rp by (rule relcompI)
  thus (?init, ?vret (Suc ?lw − m)) ∈ ?R ^^ (Suc (Suc ?lw) + m)
    by (simp only: relpow_add)
qed
```

— Assemble: each step $i < 2\ lw + 4$ falls in the forward, boundary, or return region.

```
show ?thesis
proof (intro allI impI)
  fix i :: nat assume i_lt: i < 2 * ?lw + 4
  consider (fwd) i ≤ ?lw | (bd) i = Suc ?lw | (ret) Suc ?lw < i
    by linarith
  thus ∃ d. (?init, d) ∈ ?R ^^ i
          ∧ snd (snd (snd (snd (mt_state d)))) ∈ {VFwd, VFwdPad, VRet}
  proof cases
    case fwd
    have (?init, ?vfwd i) ∈ ?R ^^ i by (rule vfwd_wit[OF fwd])
    moreover have snd (snd (snd (snd (mt_state (?vfwd i)))))
                  ∈ {VFwd, VFwdPad, VRet} by simp
    ultimately show ?thesis by blast
  next
    case bd
    show ?thesis using boundary_wit unfolding bd by blast
  next
    case ret
    define m where m = i − Suc (Suc ?lw)
```

```
    have i_ge: Suc (Suc ?lw) ≤ i using ret by simp
    have m_le: m ≤ Suc ?lw using i_lt m_def by simp
    have i_eq: Suc (Suc ?lw) + m = i using i_ge m_def by simp
    have (?init, ?vret (Suc ?lw − m)) ∈ ?R ^^ i
      using ret_reach[OF m_le] unfolding i_eq .
    moreover have snd (snd (snd (snd (mt_state (?vret (Suc ?lw − m))))))
                  ∈ {VFwd, VFwdPad, VRet} by simp
    ultimately show ?thesis by blast
   qed
  qed
qed
```

Prefix uniqueness: every config reachable from *ae_init_config* in $i < 2 * length\ w + 4$ steps is a validation marker (stage index *VFwd* / *VFwdPad* / *VRet*). This is the invariant the relpow validation-functional needs, and is *det-free*: the validation sweep is functional regardless of M's (non)determinism, since the only nondeterministic substep of *alphabet_enlarge_delta* (the *SS4*-to-*SS5* transition) is never reached before *SS1*. Proof: strong induction on i; the induction hypothesis supplies the relpow-functional's invariant premise, so the arbitrary reachable d is forced equal to the marker witness exhibited by *ae_validation_prefix_witnesses*.

```
lemma ae_validation_prefix_markers:
  fixes M :: ('q, 'a) mttm
    and w :: ('c :: enum ⇒ 'a) list
  assumes vM: valid_mttm M
      and w_sub: set w ⊆ gamma_block (Sigma_tm M ∪ {bl_tm M})
      and wf: ae_input_well_formed (bl_tm M) w
      and s_in_Q: s_tm M ∈ Q_tm M
      and s_neq_t: s_tm M ≠ t_tm M
      and s_neq_r: s_tm M ≠ r_tm M
      and le_neq_bl: le_tm M ≠ bl_tm M
  shows ∀ i<2 * length w + 4. ∀ d.
          (ae_init_config M w, d)
             ∈ mttm_step (alphabet_enlarge_delta M) ^^ i
            ⟶ snd (snd (snd (snd (mt_state d))))
                ∈ {VFwd, VFwdPad, VRet}
proof −
  let ?init = ae_init_config M w
  let ?R = mttm_step (alphabet_enlarge_delta M)
  let ?n_val = 2 * length w + 4
  have wit: ∀ i<?n_val. ∃ d. (?init, d) ∈ ?R ^^ i
            ∧ snd (snd (snd (snd (mt_state d)))) ∈ {VFwd, VFwdPad, VRet}
    by (rule ae_validation_prefix_witnesses[OF vM w_sub wf s_in_Q
                                              s_neq_t s_neq_r le_neq_bl])
  have main: ∀ i0. i0 < ?n_val
               ⟶ (∀ d. (?init, d) ∈ ?R ^^ i0
                     ⟶ snd (snd (snd (snd (mt_state d))))
                         ∈ {VFwd, VFwdPad, VRet})
```

```
  proof (rule allI)
    fix i1 :: nat
    show i1 < ?n_val
          ⟶ (∀ d. (?init, d) ∈ ?R ^^ i1
                ⟶ snd (snd (snd (snd (mt_state d))))
                    ∈ {VFwd, VFwdPad, VRet})
    proof (induction i1 rule: less_induct)
      case (less i)
      show ?case
      proof (rule impI)
        assume i_lt: i < ?n_val
        show ∀ d. (?init, d) ∈ ?R ^^ i
                  ⟶ snd (snd (snd (snd (mt_state d))))
                      ∈ {VFwd, VFwdPad, VRet}
        proof (intro allI impI)
          fix d assume reach: (?init, d) ∈ ?R ^^ i
          have inv: ∀ j<i. ∀ d'. (?init, d') ∈ ?R ^^ j
                    ⟶ snd (snd (snd (snd (mt_state d'))))
                        ∈ {VFwd, VFwdPad, VRet}
          proof (intro allI impI)
            fix j d' assume j_lt: j < i and reach': (?init, d') ∈ ?R ^^ j
            have j < ?n_val using j_lt i_lt by simp
            thus snd (snd (snd (snd (mt_state d')))) ∈ {VFwd, VFwdPad, VRet}
              using less.IH[OF j_lt] reach' by blast
          qed
          obtain wd where wreach: (?init, wd) ∈ ?R ^^ i
            and wmark: snd (snd (snd (snd (mt_state wd)))) ∈ {VFwd, VFwdPad,
VRet}
            using wit i_lt by blast
          have d = wd
            by (rule mttm_step_alphabet_enlarge_val_relpow_functional
                  [OF le_neq_bl reach wreach inv])
          thus snd (snd (snd (snd (mt_state d)))) ∈ {VFwd, VFwdPad, VRet}
            using wmark by simp
        qed
      qed
    qed
    show ?thesis using main by blast
  qed

lemma ae_validation_post_state_canonical:
    fixes M :: ('q, 'a) mttm
      and u :: 'a list
    assumes vM: valid_mttm M
        and u_sub: set u ⊆ Sigma_tm M
        and le_neq_bl: le_tm M ≠ bl_tm M
    obtains n :: nat and c' where
        (ae_init_config M (encode_input (bl_tm M) u), c')
```

```
        ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
  and ae_simulates M
        (init_config_mttm M u)
        (c' :: ('c :: enum ⇒ 'a,
              'q × ('a, 'c) ae_stage) mt_config)
  and ae_buffer_in_gamma_block M c'
  and ∀ kk<k_tm M. mt_tape c' kk 0 = LE_block (le_tm M)
  and ∀ i<n. ∀ d :: ('c :: enum ⇒ 'a,
                  'q × ('a, 'c) ae_stage) mt_config.
        (ae_init_config M (encode_input (bl_tm M) u), d)
          ∈ mttm_step (alphabet_enlarge_delta M) ^^ i
        ⟶ snd (snd (snd (snd (mt_state d))))
            ∈ {VFwd, VFwdPad, VRet}
proof −
  let ?w = encode_input (bl_tm M) u :: ('c :: enum ⇒ 'a) list
  let ?init = ae_init_config M ?w
  let ?subst_init = init_config_mttm M u :: ('a, 'q) mt_config
  have w_sub: set ?w ⊆ gamma_block (Sigma_tm M ∪ {bl_tm M})
    by (rule encode_input_in_gamma_block[OF u_sub])
  have wf: ae_input_well_formed (bl_tm M) ?w
    by (rule encode_input_well_formed[OF vM u_sub])
  have s_in_Q: s_tm M ∈ Q_tm M by (rule s_tm_in_Q_tm[OF vM])
  — Tape correspondence is shared across the s-degenerate and s-nondegenerate cases (the chain doesn't modify the tape).
  have tape_corr:
    ∀ k<k_tm M. ae_tape_correspondence (le_tm M)
          (mt_tape ?subst_init k) (mt_tape ?init k)
    by (rule ae_tape_correspondence_init[OF vM u_sub])
  — Gamma-block invariant on the M'-tape: also shared across both cases since the validation chain in the nondegen case reuses mt_tape ?init verbatim (no writes).
  have gamma_block_init: ae_tape_in_gamma_block M ?init
    unfolding ae_tape_in_gamma_block_def
  proof (intro conjI)
    show ∀ k p. mt_tape ?init k p ∈ gamma_block (Γ_tm M)
      using ae_init_config_in_gamma_block[OF vM w_sub] by simp
  next
    show ∀ j≥k_tm M. ∀ p. mt_tape ?init j p = bl_block (bl_tm M)
      using ae_init_config_tape_blank_tail by blast
  qed
  — Position correspondence: substrate's n_M k = 0; ae's n_M' k = 0; ae_decode_pos 0 _ = 0.
  have ae_pos_zero: ∀ k. mt_pos ?init k = 0
    unfolding ae_init_config_def by simp
  have subst_pos_zero: ∀ k. mt_pos ?subst_init k = 0
    by (cases M) simp
  have pos_corr_init:
    ∀ k. mt_pos ?subst_init k
          = ae_decode_pos (mt_pos ?init k) ((init_offset :: nat ⇒ 'c) k)
    using ae_pos_zero subst_pos_zero
```

unfolding *ae_decode_pos_def* **by** *simp*
consider (*degen*) *s_tm M = t_tm M ∨ s_tm M = r_tm M*
| (*nondegen*) *s_tm M ≠ t_tm M ∧ s_tm M ≠ r_tm M*
by *blast*
thus *?thesis*
proof *cases*
case *degen*
— *n = 0*: chain is reflexive; simulation's halt-branch fires.
have *chain_0*:
(*?init*, *?init*) ∈ *mttm_step* (*alphabet_enlarge_delta M*) ^^ *0*
by *simp*
have *qM_subst*: *mt_state ?subst_init = s_tm M*
by (*cases M*) *simp*
have *init_state*:
mt_state ?init = (*s_tm M*, *init_offset*,
init_buffer (*le_tm M*), *init_dest*, *VFwd*)
unfolding *ae_init_config_def init_stage_def* **by** *simp*
have *simulation*: *ae_simulates M ?subst_init ?init*
unfolding *ae_simulates_def Let_def init_state init_stage_def*
using *qM_subst tape_corr pos_corr_init degen gamma_block_init* **by** *simp*
— *buf_gamma*: *?init*'s state buffer is *init_buffer* (*le_tm M*), which is constant *LE_block* (*le_tm M*) in every slot; *init_buffer_in_gamma_block_at_M* discharges.
have *buf_gamma_init*: *ae_buffer_in_gamma_block M ?init*
unfolding *ae_buffer_in_gamma_block_def init_state init_stage_def*
using *init_buffer_in_gamma_block_at_M*[*OF vM*] **by** *simp*
— *le_anchor*: position 0 of every M'-tape is *LE_block* (*le_tm M*) by *ae_init_config_def*.
have *le_anchor_init*:
∀ *kk*<*k_tm M*. *mt_tape ?init kk 0 = LE_block* (*le_tm M*)
by (*auto intro*: *ae_init_config_tape_le*)
— *n = 0*, so the prefix-marker clause is vacuous.
have *markers_0*:
∀ *i*<(*0* :: *nat*). ∀ *d*.
(*?init*, *d*) ∈ *mttm_step* (*alphabet_enlarge_delta M*) ^^ *i*
⟶ *snd* (*snd* (*snd* (*snd* (*mt_state d*)))) ∈ {*VFwd*, *VFwdPad*, *VRet*}
by *simp*
show *?thesis*
using *chain_0 simulation buf_gamma_init le_anchor_init markers_0*
by (*rule that*)
next
case *nondegen*
— Build SS1-landing chain via helper; simulation's SS1-branch fires.
have *s_neq_t*: *s_tm M ≠ t_tm M* **using** *nondegen* **by** *simp*
have *s_neq_r*: *s_tm M ≠ r_tm M* **using** *nondegen* **by** *simp*
let *?c'* = $Config_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *SS1*)
(*mt_tape ?init*) (λ_ :: *nat*. (*0* :: *nat*))
:: (*'c* ⇒ *'a*, *'q* × (*'a*, *'c*) *ae_stage*) *mt_config*

```
    have chain:
      (?init, ?c') ∈ mttm_step (alphabet_enlarge_delta M)
                    ^^ (2 * length ?w + 4)
      by (rule ae_validation_well_formed_to_SS1[OF vM w_sub wf
           s_in_Q s_neq_t s_neq_r le_neq_bl])
    have qM_subst: mt_state ?subst_init = s_tm M
      by (cases M) simp
    have c'_state:
      mt_state ?c' = (s_tm M, init_offset,
                      init_buffer (le_tm M), init_dest, SS1)
      by simp
    have pos_corr_c':
      ∀k. mt_pos ?subst_init k
            = ae_decode_pos (mt_pos ?c' k)
                            ((init_offset :: nat ⇒ 'c) k)
      using subst_pos_zero by (simp add: ae_decode_pos_def)
    have gamma_block_c': ae_tape_in_gamma_block M ?c'
      unfolding ae_tape_in_gamma_block_def
      using gamma_block_init unfolding ae_tape_in_gamma_block_def
      by simp
    have simulation: ae_simulates M ?subst_init ?c'
      unfolding ae_simulates_def Let_def c'_state
      using qM_subst tape_corr pos_corr_c' gamma_block_c'
            s_neq_t s_neq_r by simp
   — buf_gamma: SS1-landing config still carries the canonical init_buffer (le_tm
M); init_buffer_in_gamma_block_at_M discharges as in the degen case.
    have buf_gamma_c': ae_buffer_in_gamma_block M ?c'
      unfolding ae_buffer_in_gamma_block_def
      using init_buffer_in_gamma_block_at_M[OF vM] by simp
     — le_anchor: the SS1-landing config's tape equals mt_tape ?init verbatim
(validation does not write the M'-tape); position 0 inherits LE_block (le_tm M)
from ae_init_config_tape_le.
    have le_anchor_c':
       ∀kk<k_tm M. mt_tape ?c' kk 0 = LE_block (le_tm M)
      by (auto intro: ae_init_config_tape_le)
    — The validation prefix is uniquely a marker run (det-free).
    have markers:
      ∀i<2 * length ?w + 4. ∀d.
         (?init, d) ∈ mttm_step (alphabet_enlarge_delta M) ^^ i
           ⟶ snd (snd (snd (snd (mt_state d)))) ∈ {VFwd, VFwdPad, VRet}
      by (rule ae_validation_prefix_markers[OF vM w_sub wf s_in_Q
           s_neq_t s_neq_r le_neq_bl])
    show ?thesis
      using chain simulation buf_gamma_c' le_anchor_c' markers
      by (rule that)
  qed
qed
```

Noncanonical inputs reject. The *bl_block bl_M* ∉ *set w* hypothesis matches M''s input alphabet Σ' (Σ' = *gamma_block* (Σ_M ∪ {*bl_M*})

$-$ {*bl_block*, *LE_block*}). Without it, e.g. $w = [bl_block\ bl_M]$ triggers *ae_delta_val_fwd_to_ret* at the first input block, mistaking it for end-of-input validation passes rather than rejecting, falsifying the lemma as previously stated. $LE_block\ le_M \notin set\ w$ follows already from $set\ w \subseteq gamma_block\ (\Sigma_M \cup \{bl_M\})$ plus $le_M \notin \Sigma_M \cup \{bl_M\}$, so it need not be assumed separately.

lemma *ae_validation_post_state_noncanonical*:
  fixes $M :: ('q, 'a)\ mttm$
    and $w :: (('c :: enum) \Rightarrow 'a)\ list$
  assumes *vM*: *valid_mttm M*
      and *w_sub*: $set\ w \subseteq gamma_block\ (Sigma_tm\ M \cup \{bl_tm\ M\})$
      and *bl_notin*: $bl_block\ (bl_tm\ M) \notin set\ w$
      and *w_bad*: $\neg\ ae_input_well_formed\ (bl_tm\ M)\ w$
      and *s_neq_t*: $s_tm\ M \neq t_tm\ M$
      and *s_neq_r*: $s_tm\ M \neq r_tm\ M$
  obtains $n :: nat$ **and** c' **where**
      $n \leq length\ w + 2$
    and $(ae_init_config\ M\ w,\ c')$
          $\in mttm_step\ (alphabet_enlarge_delta\ M) \frown\frown n$
    and *case mt_state c' of* $(qM', _, _, _, _) \Rightarrow qM' = r_tm\ M$
proof $-$
  let *?init* = *ae_init_config M w*
  let *?ts* = *mt_tape ?init*
  have *kpos*: $0 < k_tm\ M$ **by** (*rule valid_mttm_k_pos*[*OF vM*])
  have *bt_all*: $\forall j{\geq}k_tm\ M.\ \forall i.\ ?ts\ j\ i = bl_block\ (bl_tm\ M)$
    using *ae_init_config_tape_blank_tail* **by** *blast*
  have *buf_gamma_init*:
      $\forall kk :: nat.$
        $fst\ (init_buffer\ (le_tm\ M)\ kk$
              $:: ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a))$
          $\in gamma_block\ (\Gamma_tm\ M)$
        $\wedge\ fst\ (snd\ (init_buffer\ (le_tm\ M)\ kk$
                    $:: ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)))$
          $\in gamma_block\ (\Gamma_tm\ M)$
        $\wedge\ snd\ (snd\ (init_buffer\ (le_tm\ M)\ kk$
                    $:: ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)))$
          $\in gamma_block\ (\Gamma_tm\ M)$
    by (*rule init_buffer_in_gamma_block_at_M*[*OF vM*])
  have *s_in_Q*: $s_tm\ M \in Q_tm\ M$ **by** (*rule s_tm_in_Q_tm*[*OF vM*])
  from *w_bad* **have** *failure_exists*:
    $\exists s'.\ s' < length\ w$
        $\wedge\ \neg\ is_pure_block\ (bl_tm\ M)\ (w\ !\ s')$
        $\wedge\ \neg\ (s' = length\ w - 1$
            $\wedge\ is_padded_block\ (bl_tm\ M)\ (w\ !\ s'))$
    unfolding *ae_input_well_formed_def* **by** *auto*
  define *P* **where**
    $P = (\lambda s'.\ s' < length\ w$
            $\wedge\ \neg\ is_pure_block\ (bl_tm\ M)\ (w\ !\ s')$

```
            ∧ ¬ (s' = length w − 1
                ∧ is_padded_block (bl_tm M) (w ! s')))
  define s where s = (LEAST s'. P s')
  have failure_exists_P: ∃ s'. P s'
    using failure_exists unfolding P_def by simp
  from failure_exists_P have P_s: P s
    unfolding s_def by (rule LeastI_ex)
  have s_lt: s < length w using P_s unfolding P_def by simp
  have w_pos: 0 < length w using s_lt by linarith
  have s_not_pure: ¬ is_pure_block (bl_tm M) (w ! s)
    using P_s unfolding P_def by simp
  have s_not_last_padded:
    ¬ (s = length w − 1 ∧ is_padded_block (bl_tm M) (w ! s))
    using P_s unfolding P_def by simp
  have s_min: ∀ s'. P s' ⟶ s ≤ s'
    unfolding s_def using Least_le by metis
  have pure_prefix: ∀ i < s. is_pure_block (bl_tm M) (w ! i)
  proof (intro allI impI)
    fix i assume i_lt_s: i < s
    have i_lt_lw: i < length w using i_lt_s s_lt by simp
    have i_neq_last: i ≠ length w − 1
      using i_lt_s s_lt w_pos by linarith
    have not_P_i: ¬ P i using s_min i_lt_s by force
    hence is_pure_block (bl_tm M) (w ! i)
          ∨ (i = length w − 1
             ∧ is_padded_block (bl_tm M) (w ! i))
      using i_lt_lw unfolding P_def by auto
    thus is_pure_block (bl_tm M) (w ! i)
      using i_neq_last by auto
  qed
  have s_le_lw: s ≤ length w using s_lt by simp
  have sweep:
    (?init,
       Config_M (s_tm M, init_stage (le_tm M)) ?ts
             (λi :: nat. if i = 0 then Suc s else 0))
      ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc s
      by (rule ae_validation_fwd_sweep_pure[OF vM w_sub s_in_Q s_neq_t
s_neq_r
                                            s_le_lw pure_prefix])
  have Sk_pos: 1 ≤ Suc s by simp
  have Sk_le: Suc s ≤ length w using s_lt by simp
  have read_at_Ss: ?ts 0 (Suc s) = w ! s
    using ae_init_config_tape_input[OF Sk_pos Sk_le kpos] by simp
  have w_s_in_set: w ! s ∈ set w using s_lt by auto
  have w_s_neq_bl: w ! s ≠ bl_block (bl_tm M)
    using w_s_in_set bl_notin by auto
  have w_s_neq_le: w ! s ≠ LE_block (le_tm M)
  proof (cases le_tm M = bl_tm M)
    case True
```

```
  have eq: LE_block (le_tm M) = bl_block (bl_tm M)
    using True unfolding LE_block_def bl_block_def by simp
  have w ! s ≠ bl_block (bl_tm M) using w_s_in_set bl_notin by auto
  thus ?thesis using eq by metis
next
  case False
  have le_notin: le_tm M ∉ Sigma_tm M ∪ {bl_tm M}
    using valid_mttm_LE_not_Sigma[OF vM] False by auto
  have LE_block (le_tm M) ∉ gamma_block (Sigma_tm M ∪ {bl_tm M})
    using le_notin
    unfolding LE_block_def gamma_block_def by auto
  moreover have w ! s ∈ gamma_block (Sigma_tm M ∪ {bl_tm M})
    using w_s_in_set w_sub by auto
  ultimately show ?thesis by auto
qed
have case_split: (¬ is_canonical_block (bl_tm M) (w ! s))
                 ∨ (is_padded_block (bl_tm M) (w ! s)
                   ∧ Suc s < length w)
proof (cases is_padded_block (bl_tm M) (w ! s))
  case True
  have s ≠ length w − 1 using s_not_last_padded True by simp
  hence Suc s < length w using s_lt by linarith
  thus ?thesis using True by simp
next
  case False
  hence ¬ is_canonical_block (bl_tm M) (w ! s)
    using s_not_pure unfolding is_canonical_block_def by simp
  thus ?thesis by simp
qed
consider
    (noncan) ¬ is_canonical_block (bl_tm M) (w ! s)
  | (pad_misplaced) is_padded_block (bl_tm M) (w ! s)
                    Suc s < length w
  using case_split by blast
thus ?thesis
proof cases
  case noncan
  let ?n_Ss = λi :: nat. if i = 0 then Suc s else 0
  have read_nle: ?ts 0 (?n_Ss 0) ≠ LE_block (le_tm M)
    using read_at_Ss w_s_neq_le by simp
  have read_nbl: ?ts 0 (?n_Ss 0) ≠ bl_block (bl_tm M)
    using read_at_Ss w_s_neq_bl by simp
  have read_ncan: ¬ is_canonical_block (bl_tm M) (?ts 0 (?n_Ss 0))
    using read_at_Ss noncan by simp
  have gamma: ∀ kk :: nat. ?ts kk (?n_Ss kk) ∈ gamma_block (Γ_tm M)
    using ae_init_config_in_gamma_block[OF vM w_sub] by simp
  have step_rej:
    (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
               init_dest, VFwd) ?ts ?n_Ss,
```

```
      Config_M (r_tm M, init_stage (le_tm M)) ?ts ?n_Ss)
    ∈ mttm_step (alphabet_enlarge_delta M)
  by (rule ae_step_val_fwd_reject[where ts = ?ts and n = ?n_Ss,
      OF vM s_in_Q s_neq_t s_neq_r read_nle read_nbl read_ncan
      bt_all init_stage_tail gamma buf_gamma_init])
 have stage_eq: init_stage (le_tm M)
              = (init_offset, init_buffer (le_tm M), init_dest, VFwd)
  unfolding init_stage_def ..
 have step_rej_init:
  (Config_M (s_tm M, init_stage (le_tm M)) ?ts ?n_Ss,
    Config_M (r_tm M, init_stage (le_tm M)) ?ts ?n_Ss)
    ∈ mttm_step (alphabet_enlarge_delta M)
  using step_rej unfolding stage_eq by simp
 have full_chain:
  (?init, Config_M (r_tm M, init_stage (le_tm M)) ?ts ?n_Ss)
    ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc (Suc s)
  by (rule relpow_Suc_I[OF sweep step_rej_init])
 have bound: Suc (Suc s) ≤ length w + 2 using s_lt by linarith
 have q_eq:
  case mt_state (Config_M (r_tm M, init_stage (le_tm M)) ?ts ?n_Ss)
    of (qM', _, _, _, _) ⇒ qM' = r_tm M
  by (simp add: init_stage_def)
 show ?thesis
  by (rule that[OF bound full_chain q_eq])
next
 case pad_misplaced
 let ?n_Ss = λi :: nat. if i = 0 then Suc s else 0
 let ?n_SSs = λi :: nat. if i = 0 then Suc (Suc s) else 0
 have read_pad: is_padded_block (bl_tm M) (?ts 0 (?n_Ss 0))
  using read_at_Ss pad_misplaced(1) by simp
 have gamma_Ss: ∀ kk :: nat. ?ts kk (?n_Ss kk) ∈ gamma_block (Γ_tm M)
  using ae_init_config_in_gamma_block[OF vM w_sub] by simp
 have step_to_pad:
  (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
             init_dest, VFwd) ?ts ?n_Ss,
    Config_M (s_tm M, init_offset, init_buffer (le_tm M),
             init_dest, VFwdPad) ?ts
           (λkk. go_dir (if kk = (0 :: nat) then dir.R else dir.N)
                        (?n_Ss kk)))
    ∈ mttm_step (alphabet_enlarge_delta M)
  by (rule ae_step_val_fwd_to_padded[where ts = ?ts and n = ?n_Ss,
      OF s_in_Q s_neq_t s_neq_r read_pad bt_all init_stage_tail
        gamma_Ss buf_gamma_init])
 have post_eq:
  (λkk :: nat. go_dir (if kk = 0 then dir.R else dir.N) (?n_Ss kk))
  = ?n_SSs
  by (rule ext) simp
 have step_to_pad_SS:
  (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
```

```
            init_dest, VFwd) ?ts ?n_Ss,
    Config_M (s_tm M, init_offset, init_buffer (le_tm M),
            init_dest, VFwdPad) ?ts ?n_SSs)
   ∈ mttm_step (alphabet_enlarge_delta M)
  using step_to_pad post_eq by simp
have stage_eq: init_stage (le_tm M)
            = (init_offset, init_buffer (le_tm M), init_dest, VFwd)
  unfolding init_stage_def ..
have step_to_pad_init:
  (Config_M (s_tm M, init_stage (le_tm M)) ?ts ?n_Ss,
    Config_M (s_tm M, init_offset, init_buffer (le_tm M),
            init_dest, VFwdPad) ?ts ?n_SSs)
   ∈ mttm_step (alphabet_enlarge_delta M)
  using step_to_pad_SS unfolding stage_eq by simp
have SSs_pos: 1 ≤ Suc (Suc s) by simp
have SSs_le: Suc (Suc s) ≤ length w
  using pad_misplaced(2) by linarith
have read_at_SSs: ?ts 0 (Suc (Suc s)) = w ! Suc s
  using ae_init_config_tape_input[OF SSs_pos SSs_le kpos] by simp
have wSs_in_set: w ! Suc s ∈ set w
  using SSs_le by auto
have wSs_neq_bl: w ! Suc s ≠ bl_block (bl_tm M)
  using wSs_in_set bl_notin by auto
have read_nbl_SS: ?ts 0 (?n_SSs 0) ≠ bl_block (bl_tm M)
  using read_at_SSs wSs_neq_bl by simp
have gamma_SS: ∀ kk :: nat. ?ts kk (?n_SSs kk) ∈ gamma_block (Γ_tm M)
  using ae_init_config_in_gamma_block[OF vM w_sub] by simp
have step_pad_rej:
  (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
            init_dest, VFwdPad) ?ts ?n_SSs,
    Config_M (r_tm M, init_stage (le_tm M)) ?ts ?n_SSs)
   ∈ mttm_step (alphabet_enlarge_delta M)
  by (rule ae_step_val_pad_reject[where ts = ?ts and n = ?n_SSs,
       OF vM s_in_Q read_nbl_SS bt_all init_stage_tail
         gamma_SS buf_gamma_init])
have post_sweep:
  (?init,
    Config_M (s_tm M, init_offset, init_buffer (le_tm M),
            init_dest, VFwdPad) ?ts ?n_SSs)
   ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc (Suc s)
  by (rule relpow_Suc_I[OF sweep step_to_pad_init])
have full_chain:
  (?init, Config_M (r_tm M, init_stage (le_tm M)) ?ts ?n_SSs)
   ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc (Suc (Suc s))
  by (rule relpow_Suc_I[OF post_sweep step_pad_rej])
have bound: Suc (Suc (Suc s)) ≤ length w + 2
  using pad_misplaced(2) by linarith
have q_eq:
  case mt_state (Config_M (r_tm M, init_stage (le_tm M)) ?ts ?n_SSs)
```

```
      of (qM', _, _, _, _) ⇒ qM' = r_tm M
    by (simp add: init_stage_def)
  show ?thesis
    by (rule that[OF bound full_chain q_eq])
 qed
qed

lemma ae_validation_steps_bound:
 fixes M :: ('q, 'a) mttm
   and w :: ('c :: enum ⇒ 'a) list
 assumes vM: valid_mttm M
     and w_sub: set w ⊆ gamma_block (Sigma_tm M ∪ {bl_tm M})
     and s_neq_t: s_tm M ≠ t_tm M
     and s_neq_r: s_tm M ≠ r_tm M
     and le_neq_bl: le_tm M ≠ bl_tm M
 obtains f_v :: nat and n :: nat and c' where
     n ≤ 2 * length w + f_v
   and (ae_init_config M w, c')
         ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
   and case mt_state c' of (qM', _, _, _, idx) ⇒
         idx = SS1 ∨ qM' = r_tm M
proof −
 let ?init = ae_init_config M w
 let ?ts = mt_tape ?init
 have kpos: 0 < k_tm M by (rule valid_mttm_k_pos[OF vM])
 have bt_all: ∀ j≥k_tm M. ∀ i. ?ts j i = bl_block (bl_tm M)
   using ae_init_config_tape_blank_tail by blast
 have buf_gamma_init:
    ∀ kk :: nat.
       fst (init_buffer (le_tm M) kk
             :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
          ∈ gamma_block (Γ_tm M)
       ∧ fst (snd (init_buffer (le_tm M) kk
                   :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)))
          ∈ gamma_block (Γ_tm M)
       ∧ snd (snd (init_buffer (le_tm M) kk
                   :: ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)))
          ∈ gamma_block (Γ_tm M)
  by (rule init_buffer_in_gamma_block_at_M[OF vM])
 have s_in_Q: s_tm M ∈ Q_tm M by (rule s_tm_in_Q_tm[OF vM])
 consider
    (no_bl_wf)
      bl_block (bl_tm M) ∉ set w ae_input_well_formed (bl_tm M) w
  | (no_bl_nwf)
      bl_block (bl_tm M) ∉ set w ¬ ae_input_well_formed (bl_tm M) w
  | (has_bl)
      bl_block (bl_tm M) ∈ set w
  by blast
 thus ?thesis
```

```
  proof cases
    case no_bl_nwf
    obtain n c' where
       bound: n ≤ length w + 2
      and chain: (ae_init_config M w, c')
                 ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
      and state_eq: case mt_state c' of (qM', _, _, _, _) ⇒ qM' = r_tm M
      using ae_validation_post_state_noncanonical[OF vM w_sub
            no_bl_nwf(1) no_bl_nwf(2) s_neq_t s_neq_r] by metis
    have bound2: n ≤ 2 * length w + 4 using bound by linarith
    have state_eq_disj: case mt_state c'
          of (qM', _, _, _, idx) ⇒ idx = SS1 ∨ qM' = r_tm M
      using state_eq by (cases mt_state c') auto
    show ?thesis using bound2 chain state_eq_disj by (rule that)
  next
    case no_bl_wf
    have wf: ae_input_well_formed (bl_tm M) w using no_bl_wf(2) .
    have full_chain:
      (?init,
        Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                  init_dest, SS1) ?ts (λ_. 0))
       ∈ mttm_step (alphabet_enlarge_delta M)
           ^^ (2 * length w + 4)
      by (rule ae_validation_well_formed_to_SS1[OF vM w_sub wf s_in_Q
           s_neq_t s_neq_r le_neq_bl])
    have bound: (2 * length w + 4 :: nat) ≤ 2 * length w + 4 by simp
    have state_eq:
      case mt_state (Config_M (s_tm M, init_offset,
                               init_buffer (le_tm M),
                               init_dest, SS1) ?ts (λ_. 0))
        of (qM', _, _, _, idx) ⇒ idx = SS1 ∨ qM' = r_tm M
      by simp
    show ?thesis by (rule that[OF bound full_chain state_eq])
  next
    case has_bl
    have LE_neq_bl_block: LE_block (le_tm M) ≠ bl_block (bl_tm M)
      using le_neq_bl unfolding LE_block_def bl_block_def
      by (metis fun_eq_iff)
    have tape_le: ?ts (0 :: nat) 0 = LE_block (le_tm M)
      by (rule ae_init_config_tape_le[OF kpos])
    have gamma_all: ∀ k i. ?ts k i ∈ gamma_block (Γ_tm M)
      using ae_init_config_in_gamma_block[OF vM w_sub] by simp
    have stage_eq: init_stage (le_tm M)
                   = (init_offset, init_buffer (le_tm M), init_dest, VFwd)
      unfolding init_stage_def ..
    — Show LE_block le_M ∉ set w: in the le ≠ bl regime, le ∉ Σ ∪ {bl} forces
LE_block out of gamma_block.
    have LE_notin: LE_block (le_tm M) ∉ set w
    proof −
```

```
    have le_notin: le_tm M ∉ Sigma_tm M ∪ {bl_tm M}
      using valid_mttm_LE_not_Sigma[OF vM] le_neq_bl by auto
    have LE_block (le_tm M) ∉ gamma_block (Sigma_tm M ∪ {bl_tm M})
      using le_notin
      unfolding LE_block_def gamma_block_def by auto
    thus ?thesis using w_sub by auto
  qed
  — Find the first non-pure index  well-defined because bl_block is non-pure and present somewhere in w.
  define P where
    P = (λi. i < length w ∧ ¬ is_pure_block (bl_tm M) (w ! i))
  have ex_P: ∃ i. P i
  proof −
    from has_bl obtain j where j_lt: j < length w and j_eq: w ! j = bl_block (bl_tm M)
      by (auto simp: in_set_conv_nth)
    have ¬ is_pure_block (bl_tm M) (bl_block (bl_tm M))
      unfolding is_pure_block_def bl_block_def by auto
    hence ¬ is_pure_block (bl_tm M) (w ! j) using j_eq by simp
    thus ?thesis unfolding P_def using j_lt by auto
  qed
  define k where k = (LEAST i. P i)
  from ex_P have P_k: P k unfolding k_def by (rule LeastI_ex)
  have k_lt: k < length w using P_k unfolding P_def by simp
  have k_not_pure: ¬ is_pure_block (bl_tm M) (w ! k)
    using P_k unfolding P_def by simp
  have k_min: ∀ j. P j ⟶ k ≤ j
    unfolding k_def using Least_le by metis
  have pure_prefix: ∀ i < k. is_pure_block (bl_tm M) (w ! i)
  proof (intro allI impI)
    fix i assume i_lt_k: i < k
    have i_lt_w: i < length w using i_lt_k k_lt by simp
    have ¬ P i using k_min i_lt_k by force
    thus is_pure_block (bl_tm M) (w ! i) unfolding P_def using i_lt_w by auto
  qed
  have k_le_lw: k ≤ length w using k_lt by simp
  have sweep:
    (?init,
      Config_M (s_tm M, init_stage (le_tm M)) ?ts
            (λi :: nat. if i = 0 then Suc k else 0))
      ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc k
    by (rule ae_validation_fwd_sweep_pure[OF vM w_sub s_in_Q s_neq_t
                                          s_neq_r k_le_lw pure_prefix])
  let ?n_Sk = λi :: nat. if i = 0 then Suc k else 0
  have Sk_pos: 1 ≤ Suc k by simp
  have Sk_le: Suc k ≤ length w using k_lt by simp
  have read_at_Sk: ?ts 0 (Suc k) = w ! k
    using ae_init_config_tape_input[OF Sk_pos Sk_le kpos] by simp
```

have *w_k_in_set*: *w* ! *k* ∈ *set w* **using** *k_lt* **by** *auto*
have *w_k_neq_le*: *w* ! *k* ≠ *LE_block* (*le_tm M*)
using *w_k_in_set LE_notin* **by** *auto*
— Case-split on *w*!*k*: *bl_block* (passes), padded (sub-case on next), or non-canonical (rejects).
consider
(*case_bl*) *w* ! *k* = *bl_block* (*bl_tm M*)
| (*case_pad*) *is_padded_block* (*bl_tm M*) (*w* ! *k*)
| (*case_noncan*) ¬ *is_canonical_block* (*bl_tm M*) (*w* ! *k*)
w ! *k* ≠ *bl_block* (*bl_tm M*)
using *k_not_pure* **unfolding** *is_canonical_block_def* **by** *blast*
thus *?thesis*
proof *cases*
case *case_bl*
— Forward sweep + *fwd_to_ret* + return sweep, ending at SS1.
have *read_bl*: *?ts 0* (*?n_Sk 0*) = *bl_block* (*bl_tm M*)
using *read_at_Sk case_bl* **by** *simp*
have *gamma_Sk*:
∀ *kk* :: *nat*. *?ts kk* (*?n_Sk kk*) ∈ *gamma_block* (Γ_*tm M*)
using *gamma_all* **by** *simp*
have *step_to_ret*:
(*Config*$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VFwd*) *?ts ?n_Sk*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_Sk*)
∈ *mttm_step* (*alphabet_enlarge_delta M*)
by (*rule ae_step_val_fwd_to_ret*[**where** *ts* = *?ts* **and** *n* = *?n_Sk*,
OF s_in_Q s_neq_t s_neq_r read_bl bt_all init_stage_tail
gamma_Sk buf_gamma_init])
have *step_to_ret_init*:
(*Config*$_M$ (*s_tm M*, *init_stage* (*le_tm M*)) *?ts ?n_Sk*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_Sk*)
∈ *mttm_step* (*alphabet_enlarge_delta M*)
using *step_to_ret* **unfolding** *stage_eq* **by** *simp*
have *post_sweep*:
(*?init*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_Sk*)
∈ *mttm_step* (*alphabet_enlarge_delta M*) ^^ *Suc* (*Suc k*)
by (*rule relpow_Suc_I*[*OF sweep step_to_ret_init*])
have *non_le_Sk*: ∀ *i*. 1 ≤ *i* ∧ *i* ≤ *Suc k*
⟶ *?ts 0 i* ≠ *LE_block* (*le_tm M*)
proof (*intro allI impI*)
fix *i* **assume** *i_range*: 1 ≤ *i* ∧ *i* ≤ *Suc k*
have *i_pos*: 1 ≤ *i* **using** *i_range* **by** *simp*
have *i_le_lw*: *i* ≤ *length w* **using** *i_range Sk_le* **by** *linarith*
have *eq*: *?ts 0 i* = *w* ! (*i* − 1)
using *ae_init_config_tape_input*[*OF i_pos i_le_lw kpos*] **by** *simp*

```
  have w ! (i − 1) ∈ set w using i_pos i_le_lw by auto
  thus ?ts 0 i ≠ LE_block (le_tm M) using LE_notin eq by auto
qed
have ret_arrow:
  (∀ i. 1 ≤ i ∧ i ≤ Suc k
        ⟶ ?ts 0 i ≠ LE_block (le_tm M))
   ⟶ (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                  init_dest, VRet) ?ts ?n_Sk,
       Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                  init_dest, SS1) ?ts (λ_. 0))
       ∈ mttm_step (alphabet_enlarge_delta M)
           ^^ Suc (Suc k)
  by (rule ae_validation_ret_sweep[OF s_in_Q tape_le gamma_all
                                      buf_gamma_init bt_all init_stage_tail])
have ret_chain:
  (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
              init_dest, VRet) ?ts ?n_Sk,
     Config_M (s_tm M, init_offset, init_buffer (le_tm M),
              init_dest, SS1) ?ts (λ_. 0))
     ∈ mttm_step (alphabet_enlarge_delta M)
         ^^ Suc (Suc k)
  using ret_arrow non_le_Sk by (rule mp)
have full_chain_pre:
  (?init,
     Config_M (s_tm M, init_offset, init_buffer (le_tm M),
              init_dest, SS1) ?ts (λ_. 0))
     ∈ mttm_step (alphabet_enlarge_delta M)
         ^^ (Suc (Suc k) + Suc (Suc k))
proof −
  have comp:
    (?init,
       Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                  init_dest, SS1) ?ts (λ_. 0))
       ∈ mttm_step (alphabet_enlarge_delta M)
           ^^ Suc (Suc k)
         O mttm_step (alphabet_enlarge_delta M)
           ^^ Suc (Suc k)
    using post_sweep ret_chain by (rule relcompI)
  thus ?thesis by (simp only: relpow_add)
qed
have bound: Suc (Suc k) + Suc (Suc k) ≤ 2 * length w + 4
  using k_lt by linarith
have state_eq:
  case mt_state (Config_M (s_tm M, init_offset,
                           init_buffer (le_tm M),
                           init_dest, SS1) ?ts (λ_. 0))
       of (qM', _, _, _, idx) ⇒ idx = SS1 ∨ qM' = r_tm M
  by simp
show ?thesis by (rule that[OF bound full_chain_pre state_eq])
```

```
  next
    case case_pad
      — In has_bl + first-non-pure-padded, the bl_block must appear at some
position j > k, hence Suc k < length w.
    have Sk_lt: Suc k < length w
    proof −
      from has_bl obtain j where j_lt: j < length w
        and j_eq: w ! j = bl_block (bl_tm M) by (auto simp: in_set_conv_nth)
      have j ≥ k
      proof (rule ccontr)
        assume ¬ j ≥ k
        hence j < k by simp
        hence is_pure_block (bl_tm M) (w ! j) using pure_prefix by simp
        moreover have ¬ is_pure_block (bl_tm M) (bl_block (bl_tm M))
          unfolding is_pure_block_def bl_block_def by auto
        ultimately show False using j_eq by auto
      qed
      moreover have j ≠ k
      proof
        assume j = k
        hence wk_bl: w ! k = bl_block (bl_tm M) using j_eq by simp
        have not_padded:
          ¬ is_padded_block (bl_tm M) ((bl_block (bl_tm M)) :: 'c ⇒ 'a)
        proof
          assume is_padded_block (bl_tm M)
                 ((bl_block (bl_tm M)) :: 'c ⇒ 'a)
          then obtain k_pad where
            k_pad_pos: k_pad ≥ 1
            and k_pad_lt: k_pad < length (enum_class.enum :: 'c list)
            and prefix_cond:
              ∀ x. c_idx x < k_pad
                   ⟶ ((bl_block (bl_tm M)) :: 'c ⇒ 'a) x ≠ bl_tm M
            unfolding is_padded_block_def by blast
          have len_pos: 0 < length (enum_class.enum :: 'c list)
            using k_pad_lt k_pad_pos by linarith
          have c_idx ((enum_class.enum :: 'c list) ! 0) = 0
            by (rule c_idx_enum_nth[OF len_pos])
          hence c_first_idx: c_idx (c_first :: 'c) = 0
            unfolding c_first_def .
          hence c_idx (c_first :: 'c) < k_pad
            using k_pad_pos by simp
          hence (bl_block (bl_tm M) :: 'c ⇒ 'a) c_first ≠ bl_tm M
            using prefix_cond by blast
          thus False unfolding bl_block_def by simp
        qed
        show False using case_pad wk_bl not_padded by simp
      qed
      ultimately have j > k by simp
      thus ?thesis using j_lt by linarith
```

qed
let *?n_SSk* = λi :: *nat*. *if i* = *0 then Suc* (*Suc k*) *else 0*
have *read_pad*: *is_padded_block* (*bl_tm M*) (*?ts 0* (*?n_Sk 0*))
using *read_at_Sk case_pad* **by** *simp*
have *gamma_Sk*:
$\forall$ *kk* :: *nat*. *?ts kk* (*?n_Sk kk*) $\in$ *gamma_block* (Γ*_tm M*)
using *gamma_all* **by** *simp*
have *step_to_pad*:
(*Config*$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VFwd*) *?ts ?n_Sk*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VFwdPad*) *?ts*
(λ*kk*. *go_dir* (*if kk* = (*0* :: *nat*) *then dir.R else dir.N*)
(*?n_Sk kk*)))
$\in$ *mttm_step* (*alphabet_enlarge_delta M*)
by (*rule ae_step_val_fwd_to_padded*[**where** *ts* = *?ts* **and** *n* = *?n_Sk*,
OF s_in_Q s_neq_t s_neq_r read_pad bt_all init_stage_tail
gamma_Sk buf_gamma_init])
have *post_eq_pad*:
(λ*kk* :: *nat*. *go_dir* (*if kk* = *0 then dir.R else dir.N*) (*?n_Sk kk*))
= *?n_SSk*
by (*rule ext*) *simp*
have *step_to_pad_norm*:
(*Config*$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VFwd*) *?ts ?n_Sk*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VFwdPad*) *?ts ?n_SSk*)
$\in$ *mttm_step* (*alphabet_enlarge_delta M*)
using *step_to_pad post_eq_pad* **by** *simp*
have *step_to_pad_init*:
(*Config*$_M$ (*s_tm M*, *init_stage* (*le_tm M*)) *?ts ?n_Sk*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VFwdPad*) *?ts ?n_SSk*)
$\in$ *mttm_step* (*alphabet_enlarge_delta M*)
using *step_to_pad_norm* **unfolding** *stage_eq* **by** *simp*
have *post_sweep_VFwdPad*:
(*?init*,
Config$_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VFwdPad*) *?ts ?n_SSk*)
$\in$ *mttm_step* (*alphabet_enlarge_delta M*) ^^ *Suc* (*Suc k*)
by (*rule relpow_Suc_I*[*OF sweep step_to_pad_init*])
have *SSk_pos*: *1* $\leq$ *Suc* (*Suc k*) **by** *simp*
have *SSk_le*: *Suc* (*Suc k*) $\leq$ *length w* **using** *Sk_lt* **by** *linarith*
have *read_at_SSk*: *?ts 0* (*Suc* (*Suc k*)) = *w* ! *Suc k*
using *ae_init_config_tape_input*[*OF SSk_pos SSk_le kpos*] **by** *simp*
have *wSk_in_set*: *w* ! *Suc k* $\in$ *set w* **using** *SSk_le* **by** *auto*
have *gamma_SSk*:
$\forall$ *kk* :: *nat*. *?ts kk* (*?n_SSk kk*) $\in$ *gamma_block* (Γ*_tm M*)
using *gamma_all* **by** *simp*

— Sub-case on $w\,!\,Suc\ k$: *bl_block* (passes via *pad_to_ret*) or non-*bl_block* (rejects via *pad_reject*).

consider
(*pad_pass*) $w\,!\,Suc\ k = bl_block\ (bl_tm\ M)$
| (*pad_reject*) $w\,!\,Suc\ k \neq bl_block\ (bl_tm\ M)$
by *blast*
thus *?thesis*
proof *cases*
case *pad_pass*
have *read_bl_SSk*: *?ts 0* (*?n_SSk 0*) = *bl_block* (*bl_tm M*)
using *read_at_SSk pad_pass* **by** *simp*
have *step_pad_to_ret*:
($Config_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VFwdPad*) *?ts ?n_SSk*,
$Config_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_SSk*)
$\in$ *mttm_step* (*alphabet_enlarge_delta M*)
by (*rule ae_step_val_pad_to_ret*[**where** *ts* = *?ts* **and** *n* = *?n_SSk*,
OF s_in_Q read_bl_SSk bt_all init_stage_tail
gamma_SSk buf_gamma_init])
have *post_sweep_VRet*:
(*?init*,
$Config_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_SSk*)
$\in$ *mttm_step* (*alphabet_enlarge_delta M*) $^{\frown\frown}$ *Suc* (*Suc* (*Suc k*))
by (*rule relpow_Suc_I*[*OF post_sweep_VFwdPad step_pad_to_ret*])
have *non_le_SSk*: $\forall i.\ 1 \leq i \wedge i \leq Suc\ (Suc\ k) \longrightarrow ?ts\ 0\ i \neq LE_block\ (le_tm\ M)$
proof (*intro allI impI*)
fix *i* **assume** *i_range*: $1 \leq i \wedge i \leq Suc\ (Suc\ k)$
have *i_pos*: $1 \leq i$ **using** *i_range* **by** *simp*
have *i_le_lw*: $i \leq length\ w$ **using** *i_range SSk_le* **by** *linarith*
have *eq*: *?ts 0 i* = *w* ! ($i - 1$)
using *ae_init_config_tape_input*[*OF i_pos i_le_lw kpos*] **by** *simp*
have *w* ! ($i - 1$) $\in$ *set w* **using** *i_pos i_le_lw* **by** *auto*
thus *?ts 0 i* $\neq$ *LE_block* (*le_tm M*) **using** *LE_notin eq* **by** *auto*
qed
have *ret_arrow*:
($\forall i.\ 1 \leq i \wedge i \leq Suc\ (Suc\ k) \longrightarrow ?ts\ 0\ i \neq LE_block\ (le_tm\ M)$)
$\longrightarrow$ ($Config_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *VRet*) *?ts ?n_SSk*,
$Config_M$ (*s_tm M*, *init_offset*, *init_buffer* (*le_tm M*),
init_dest, *SS1*) *?ts* ($\lambda_.\ 0$))
$\in$ *mttm_step* (*alphabet_enlarge_delta M*)
$^{\frown\frown}$ *Suc* (*Suc* (*Suc k*))
by (*rule ae_validation_ret_sweep*[*OF s_in_Q tape_le gamma_all*
buf_gamma_init bt_all init_stage_tail])
have *ret_chain*:

```
    (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                init_dest, VRet) ?ts ?n_SSk,
       Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                init_dest, SS1) ?ts (λ_. 0))
      ∈ mttm_step (alphabet_enlarge_delta M)
          ^^ Suc (Suc (Suc k))
    using ret_arrow non_le_SSk by (rule mp)
  have full_chain:
    (?init,
       Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                init_dest, SS1) ?ts (λ_. 0))
      ∈ mttm_step (alphabet_enlarge_delta M)
          ^^ (Suc (Suc (Suc k)) + Suc (Suc (Suc k)))
  proof −
    have comp:
      (?init,
         Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                  init_dest, SS1) ?ts (λ_. 0))
        ∈ mttm_step (alphabet_enlarge_delta M)
            ^^ Suc (Suc (Suc k))
          O mttm_step (alphabet_enlarge_delta M)
            ^^ Suc (Suc (Suc k))
      using post_sweep_VRet ret_chain by (rule relcompI)
    thus ?thesis by (simp only: relpow_add)
  qed
  have bound:
    Suc (Suc (Suc k)) + Suc (Suc (Suc k)) ≤ 2 * length w + 4
    using Sk_lt by linarith
  have state_eq:
    case mt_state (Config_M (s_tm M, init_offset,
                             init_buffer (le_tm M),
                             init_dest, SS1) ?ts (λ_. 0))
      of (qM', _, _, _, idx) ⇒ idx = SS1 ∨ qM' = r_tm M
    by simp
  show ?thesis by (rule that[OF bound full_chain state_eq])
next
  case pad_reject
  have read_nbl_SSk: ?ts 0 (?n_SSk 0) ≠ bl_block (bl_tm M)
    using read_at_SSk pad_reject by simp
  have step_pad_rej:
    (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
                init_dest, VFwdPad) ?ts ?n_SSk,
       Config_M (r_tm M, init_stage (le_tm M)) ?ts ?n_SSk)
      ∈ mttm_step (alphabet_enlarge_delta M)
    by (rule ae_step_val_pad_reject[where ts = ?ts and n = ?n_SSk,
          OF vM s_in_Q read_nbl_SSk bt_all init_stage_tail
            gamma_SSk buf_gamma_init])
  have full_chain:
    (?init, Config_M (r_tm M, init_stage (le_tm M)) ?ts ?n_SSk)
```

```
        ∈ mttm_step (alphabet_enlarge_delta M)
            ^^ Suc (Suc (Suc k))
      by (rule relpow_Suc_I[OF post_sweep_VFwdPad step_pad_rej])
    have bound: Suc (Suc (Suc k)) ≤ 2 * length w + 4
      using Sk_lt by linarith
    have state_eq:
      case mt_state (Config_M (r_tm M, init_stage (le_tm M))
                              ?ts ?n_SSk)
        of (qM', _, _, _, idx) ⇒ idx = SS1 ∨ qM' = r_tm M
      by (simp add: init_stage_def)
    show ?thesis by (rule that[OF bound full_chain state_eq])
  qed
next
  case case_noncan
  have read_nle: ?ts 0 (?n_Sk 0) ≠ LE_block (le_tm M)
    using read_at_Sk w_k_neq_le by simp
  have read_nbl: ?ts 0 (?n_Sk 0) ≠ bl_block (bl_tm M)
    using read_at_Sk case_noncan(2) by simp
  have read_ncan: ¬ is_canonical_block (bl_tm M) (?ts 0 (?n_Sk 0))
    using read_at_Sk case_noncan(1) by simp
  have gamma_Sk:
    ∀ kk :: nat. ?ts kk (?n_Sk kk) ∈ gamma_block (Γ_tm M)
    using gamma_all by simp
  have step_rej:
    (Config_M (s_tm M, init_offset, init_buffer (le_tm M),
               init_dest, VFwd) ?ts ?n_Sk,
      Config_M (r_tm M, init_stage (le_tm M)) ?ts ?n_Sk)
      ∈ mttm_step (alphabet_enlarge_delta M)
    by (rule ae_step_val_fwd_reject[where ts = ?ts and n = ?n_Sk,
        OF vM s_in_Q s_neq_t s_neq_r read_nle read_nbl read_ncan
        bt_all init_stage_tail gamma_Sk buf_gamma_init])
  have step_rej_init:
    (Config_M (s_tm M, init_stage (le_tm M)) ?ts ?n_Sk,
      Config_M (r_tm M, init_stage (le_tm M)) ?ts ?n_Sk)
      ∈ mttm_step (alphabet_enlarge_delta M)
    using step_rej unfolding stage_eq by simp
  have full_chain:
    (?init, Config_M (r_tm M, init_stage (le_tm M)) ?ts ?n_Sk)
      ∈ mttm_step (alphabet_enlarge_delta M) ^^ Suc (Suc k)
    by (rule relpow_Suc_I[OF sweep step_rej_init])
  have bound: Suc (Suc k) ≤ 2 * length w + 4 using k_lt by linarith
  have state_eq:
    case mt_state (Config_M (r_tm M, init_stage (le_tm M)) ?ts ?n_Sk)
      of (qM', _, _, _, idx) ⇒ idx = SS1 ∨ qM' = r_tm M
    by (simp add: init_stage_def)
  show ?thesis by (rule that[OF bound full_chain state_eq])
 qed
 qed
qed
```

end
theory *AlphabetEnlargement_ComputeCorrect*
imports *AlphabetEnlargement_ValidationBound*
begin

Entry point of the *alphabet_enlarge* forward-simulation chain (the combinator is defined in *AlphabetEnlargement_Simulation*). The chain establishes that $M' = alphabet_enlarge\ M$ simulates M, and culminates in the three top-level theorems characterising the combinator well-formedness preservation (*alphabet_enlarge_wf*), forward language preservation modulo input encoding (*alphabet_enlarge_language_forward*), and the linear time bound (*alphabet_enlarge_time*) which are proved in theory *AlphabetEnlargement* at the end of the chain. The construction follows the linear-speedup theorem of Hartmanis and Stearns [4, Theorem 2], modernised in Hopcroft and Ullman [6, Theorem 12.3].

The forward chain is split across theories in dependency order, each importing the previous:

- *AlphabetEnlargement_ComputeCorrect* (this theory): per-substep state shape and invariant preservation, the *mttm_step* lift, and buffered c-fold compute correctness.
- *AlphabetEnlargement_SS4*: SS4 trace-existence and the buffer characterisations at SS4 entry.
- *AlphabetEnlargement_OutputWF*: home classification and output well-formedness.
- *AlphabetEnlargement_ForwardStage*: the per-tape unified forward stage *ae_simulates_forward_stage_general*.
- *AlphabetEnlargement_Acceptance*: acceptance correspondence and the step-count / chunked simulation engine.
- *AlphabetEnlargement*: the three top-level theorems.

2.21 Forward simulation chain

2.21.1 Per-substep state shape and invariant preservation

Per-substep mid-stage invariant preservation lemmas. Eight in total, one per simulation substep transition SSn→SS$n{+}1$ (with SS9 ≡ SS1 by wrap-around).

Common skeleton: each per-substep delta is a set of tuples whose post-state component fixes *idx* at the target substep and constrains

q via $q \in Q_tm\ M$. This helper packages the *mttm_step*-elimination plus the set-comprehension destructuring once, reducing each per-substep proof to a one-line shape obligation discharged by *auto* on the relevant *ae_delta_ss<N>_ss<N+1>_def*.

The helper does not apply to *ae_step_ss8_ss1_invariant*: that substep's halting branch lands at $idx = VFwd$, not $idx = SS1$, so the post-state's idx is not uniformly fixed.

```
lemma ae_substep_state_shape:
  fixes M :: ('q, 'a) mttm
    and δ :: (('q × ('a, 'c :: enum) ae_stage)
            × (nat ⇒ ('c ⇒ 'a))
            × ('q × ('a, 'c) ae_stage)
            × (nat ⇒ ('c ⇒ 'a))
            × (nat ⇒ dir)) set
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
    and tgt :: substep_idx
  assumes step: (c', c'') ∈ mttm_step δ
      and shape: ⋀s a s' a' d.
                   (s, a, s', a', d) ∈ δ
                     ⟹ ∃ q ofs buf dest.
                          s' = (q, ofs, buf, dest, tgt)
                          ∧ q ∈ Q_tm M
  shows case mt_state c'' of (qM', _, _, _, idx) ⇒
          idx = tgt ∧ qM' ∈ Q_tm M
proof −
  from step obtain s ts n s' a' d where
      c''_eq: c'' = Config_M s' (λk. (ts k)(n k := a' k))
                                (λk. go_dir (d k) (n k))
      and rel: (s, (λk. ts k (n k)), s', a', d) ∈ δ
    by (auto elim: mttm_step.cases)
  from shape[OF rel] obtain q ofs buf dest where
      s'_eq: s' = (q, ofs, buf, dest, tgt)
      and q_in: q ∈ Q_tm M
    by blast
  show ?thesis using c''_eq s'_eq q_in by simp
qed

lemma ae_step_ss1_ss2_invariant:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes valid_mttm M
      and ae_inv_ss1 M c'
      and (c', c'') ∈ mttm_step (ae_delta_ss1_ss2 M)
    shows ae_inv_ss2 M c''
proof −
  have shape: ⋀s a s' a' d.
```

```
                (s, a, s', a', d) ∈ ae_delta_ss1_ss2 M
                  ⟹ ∃ q ofs buf dest.
                        s' = (q, ofs, buf, dest, SS2)
                        ∧ q ∈ Q_tm M
    by (force simp: ae_delta_ss1_ss2_def)
  show ae_inv_ss2 M c''
    unfolding ae_inv_ss2_def
    using ae_substep_state_shape[OF assms(3) shape] .
qed

lemma ae_step_ss2_ss3_invariant:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes valid_mttm M
      and ae_inv_ss2 M c'
      and (c', c'') ∈ mttm_step (ae_delta_ss2_ss3 M)
    shows ae_inv_ss3 M c''
proof −
  have shape: ⋀s a s' a' d.
                (s, a, s', a', d) ∈ ae_delta_ss2_ss3 M
                  ⟹ ∃ q ofs buf dest.
                        s' = (q, ofs, buf, dest, SS3)
                        ∧ q ∈ Q_tm M
    by (force simp: ae_delta_ss2_ss3_def)
  show ae_inv_ss3 M c''
    unfolding ae_inv_ss3_def
    using ae_substep_state_shape[OF assms(3) shape] .
qed

lemma ae_step_ss3_ss4_invariant:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes valid_mttm M
      and ae_inv_ss3 M c'
      and (c', c'') ∈ mttm_step (ae_delta_ss3_ss4 M)
    shows ae_inv_ss4 M c''
proof −
  have shape: ⋀s a s' a' d.
                (s, a, s', a', d) ∈ ae_delta_ss3_ss4 M
                  ⟹ ∃ q ofs buf dest.
                        s' = (q, ofs, buf, dest, SS4)
                        ∧ q ∈ Q_tm M
    by (force simp: ae_delta_ss3_ss4_def)
  show ae_inv_ss4 M c''
    unfolding ae_inv_ss4_def
    using ae_substep_state_shape[OF assms(3) shape] .
qed
```

lemma *ae_step_ss4_ss5_invariant*:
  fixes M :: ('q, 'a) *mttm*
    and c' c'' :: ('c :: *enum* ⇒ 'a,
                'q × ('a, 'c) *ae_stage*) *mt_config*
  assumes *valM*: *valid_mttm M*
      and *ae_inv_ss4 M c'*
      and *step*: $(c', c'') \in$ *mttm_step* (*ae_delta_ss4_ss5 M*)
    shows *ae_inv_ss5 M c''*
proof −
  — Unlike the template substeps, ss4 → ss5 advances the q-component via *m_steps_buffered*; *Q_tm*-membership is preserved through the chain by *m_steps_buffered_state_preservation*. The *ae_substep_state_shape* helper does not apply directly because its uniform shape obligation can't pull in *valM* at the right scope; we instead inline the *mttm_step*-elimination.
  from *step* **obtain** *s ts n s' a' d* **where**
      c''_eq: $c'' = \mathit{Config}_M$ *s'* (λ*k*. (*ts k*)(*n k* := *a' k*))
                          (λ*k*. *go_dir* (*d k*) (*n k*))
      and *rel*: (*s*, (λ*k*. *ts k* (*n k*)), *s'*, *a'*, *d*) ∈ *ae_delta_ss4_ss5 M*
    by (*auto elim*: *mttm_step.cases*)
  from *rel*[*unfolded ae_delta_ss4_ss5_def mem_Collect_eq*]
  obtain *q ofs buf dest_old q' ofs' buf' dest'*
        buf_full end_pos bufC
    where *s_eq*: *s* = (*q*, *ofs*, *buf*, *dest_old*, *SS4*)
      and *s'_eq*: *s'* = (*q'*, *ofs'*, *buf'*, *dest'*, *SS5*)
      and *q_Q*: *q* ∈ *Q_tm M*
      and *m_steps*: ((*q*, *buf_full*, λ*k*. (*AE_Home*, *ofs k*)),
                  (*q'*, *bufC*, *end_pos*))
                    ∈ *m_steps_buffered M*
    by *auto*
  from *m_steps_buffered_state_preservation*[*OF valM m_steps q_Q*]
  have *q'_in_Q*: *q'* ∈ *Q_tm M* **.**
  show *ae_inv_ss5 M c''*
    unfolding *ae_inv_ss5_def* **using** *c''_eq s'_eq q'_in_Q* **by** *simp*
qed

lemma *ae_step_ss5_ss6_invariant*:
  fixes M :: ('q, 'a) *mttm*
    and c' c'' :: ('c :: *enum* ⇒ 'a,
                'q × ('a, 'c) *ae_stage*) *mt_config*
  assumes *valid_mttm M*
      and *ae_inv_ss5 M c'*
      and $(c', c'') \in$ *mttm_step* (*ae_delta_ss5_ss6 M*)
    shows *ae_inv_ss6 M c''*
proof −
  have *shape*: ⋀*s a s' a' d*.
                (*s*, *a*, *s'*, *a'*, *d*) ∈ *ae_delta_ss5_ss6 M*
                  ⟹ ∃ *q ofs buf dest*.
                      s' = (*q*, *ofs*, *buf*, *dest*, *SS6*)

```
                    ∧ q ∈ Q_tm M
    by (force simp: ae_delta_ss5_ss6_def)
  show ae_inv_ss6 M c''
    unfolding ae_inv_ss6_def
    using ae_substep_state_shape[OF assms(3) shape] .
qed

lemma ae_step_ss6_ss7_invariant:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes valid_mttm M
      and ae_inv_ss6 M c'
      and (c', c'') ∈ mttm_step (ae_delta_ss6_ss7 M)
    shows ae_inv_ss7 M c''
proof −
  have shape: ⋀s a s' a' d.
                (s, a, s', a', d) ∈ ae_delta_ss6_ss7 M
                  ⟹ ∃ q ofs buf dest.
                       s' = (q, ofs, buf, dest, SS7)
                       ∧ q ∈ Q_tm M
    by (force simp: ae_delta_ss6_ss7_def)
  show ae_inv_ss7 M c''
    unfolding ae_inv_ss7_def
    using ae_substep_state_shape[OF assms(3) shape] .
qed

lemma ae_step_ss7_ss8_invariant:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes valid_mttm M
      and ae_inv_ss7 M c'
      and (c', c'') ∈ mttm_step (ae_delta_ss7_ss8 M)
    shows ae_inv_ss8 M c''
proof −
  have shape: ⋀s a s' a' d.
                (s, a, s', a', d) ∈ ae_delta_ss7_ss8 M
                  ⟹ ∃ q ofs buf dest.
                       s' = (q, ofs, buf, dest, SS8)
                       ∧ q ∈ Q_tm M
    by (force simp: ae_delta_ss7_ss8_def)
  show ae_inv_ss8 M c''
    unfolding ae_inv_ss8_def
    using ae_substep_state_shape[OF assms(3) shape] .
qed

lemma ae_step_ss8_ss1_invariant:
  fixes M :: ('q, 'a) mttm
```

```
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes valid_mttm M
      and ae_inv_ss8 M c'
      and non_halt: case mt_state c' of (qM', _, _, _, _) ⇒
                      qM' ≠ t_tm M ∧ qM' ≠ r_tm M
      and step: (c', c'') ∈ mttm_step (ae_delta_ss8_ss1 M)
    shows ae_inv_ss1 M c''
proof −
  — The cycle-closure substep has a halting branch (q ∈ {t_tm M, r_tm M}) that lands at idx = VFwd, not idx = SS1; the non_halt assumption forces the steady-state branch.
  from step obtain s ts n s' a' d where
      c'_eq: c' = Config_M s ts n
      and c''_eq: c'' = Config_M s' (λk. (ts k)(n k := a' k))
                                    (λk. go_dir (d k) (n k))
      and rel: (s, (λk. ts k (n k)), s', a', d) ∈ ae_delta_ss8_ss1 M
    by (auto elim: mttm_step.cases)
  from rel obtain q ofs buf dest stage' where
      s_eq: s = (q, ofs, buf, dest, SS8)
      and s'_eq: s' = (q, stage')
      and q_in: q ∈ Q_tm M
      and stage'_eq: stage' = (if q ∈ {t_tm M, r_tm M}
                               then init_stage (le_tm M)
                               else (ofs, buf, init_dest, SS1))
    by (auto simp: ae_delta_ss8_ss1_def)
  from non_halt c'_eq s_eq have q_non_halt: q ≠ t_tm M ∧ q ≠ r_tm M
    by simp
  with stage'_eq have stage'_resolved:
      stage' = (ofs, buf, init_dest, SS1) by simp
  show ae_inv_ss1 M c''
    unfolding ae_inv_ss1_def
    using c''_eq s'_eq stage'_resolved q_in by simp
qed
```

Halt-branch invariant for SS8→SS1: when M's simulated state at SS8 is halting ($qM' \in \{t_tm\ M, r_tm\ M\}$), the SS8→SS1 step routes to *init_stage le* (post-idx is *VFwd*, not *SS1*) and preserves the halt-state q. Companion to *ae_step_ss8_ss1_invariant* (non-halt branch). The chain proof in *ae_simulates_forward_stage* case-splits on whether M's state is halting at SS8 and applies one of these two lemmas accordingly; the halt branch lands in the simulation's halt disjunct ($qM' \in \{t_tm\ M, r_tm\ M\}$), the non-halt branch in the SS1 disjunct.

```
lemma ae_step_ss8_ss1_invariant_halt:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes inv: ae_inv_ss8 M c'
      and halt: case mt_state c' of (qM', _, _, _, _) ⇒
```

```
                qM' ∈ {t_tm M, r_tm M}
    and step: (c', c'') ∈ mttm_step (ae_delta_ss8_ss1 M)
  shows (case mt_state c'' of (qM', _, _, _, idx) ⇒
           qM' ∈ {t_tm M, r_tm M} ∧ idx = VFwd)
proof −
  from step obtain s ts n s' a' d where
    c'_eq: c' = Config_M s ts n
    and c''_eq: c'' = Config_M s' (λk. (ts k)(n k := a' k))
                                  (λk. go_dir (d k) (n k))
    and rel: (s, (λk. ts k (n k)), s', a', d) ∈ ae_delta_ss8_ss1 M
    by (auto elim: mttm_step.cases)
  from rel obtain q ofs buf dest stage' where
    s_eq: s = (q, ofs, buf, dest, SS8)
    and s'_eq: s' = (q, stage')
    and q_in: q ∈ Q_tm M
    and stage'_eq: stage' = (if q ∈ {t_tm M, r_tm M}
                             then init_stage (le_tm M)
                             else (ofs, buf, init_dest, SS1))
    by (auto simp: ae_delta_ss8_ss1_def)
  from halt c'_eq s_eq have q_halt: q ∈ {t_tm M, r_tm M}
    by simp
  with stage'_eq have stage'_resolved:
      stage' = init_stage (le_tm M) by simp
  show ?thesis
    using c''_eq s'_eq stage'_resolved q_halt
    unfolding init_stage_def by simp
qed
```

2.21.2 *mttm_step* lifts of cross-phase exclusions

mttm_step-level lifts of the cross-phase exclusion lemmas *ae_delta_ssN_only*: when a substrate-level step in the alphabet-enlarged combinator's union has source *substep_idx SSN*, the step is in the canonical *ae_delta_ssN_ssM* substep relation. Used by the reverse arm to commit each peeled δ'-step to its canonical substep before invoking the per-substep invariant propagation.

```
lemma mttm_step_ae_delta_ss1_only:
  fixes M :: ('q, 'a) mttm
    and c c' :: ('c :: enum ⇒ 'a,
                 'q × ('a, 'c) ae_stage) mt_config
  assumes step: (c, c') ∈ mttm_step (alphabet_enlarge_delta M)
      and ss1: snd (snd (snd (snd (mt_state c)))) = SS1
    shows (c, c') ∈ mttm_step (ae_delta_ss1_ss2 M)
proof −
  from step obtain s ts n s' a' d where
    c_eq: c = Config_M s ts n
    and c'_eq: c' = Config_M s' (λk. (ts k)(n k := a' k))
                                (λk. go_dir (d k) (n k))
```

```
    and rel: (s, (λk. ts k (n k)), s', a', d)
             ∈ alphabet_enlarge_delta M
  by (auto elim: mttm_step.cases)
 from ss1 c_eq have s_ss1: snd (snd (snd s))) = SS1
  by simp
 have rel_sub: (s, (λk. ts k (n k)), s', a', d)
               ∈ ae_delta_ss1_ss2 M
  by (rule ae_delta_ss1_only[OF rel s_ss1])
 show ?thesis
  unfolding c_eq c'_eq
  by (rule mttm_step.step[where ts = ts and n = n and a = a' and dir = d,
                          OF rel_sub])
qed

lemma mttm_step_ae_delta_ss2_only:
 fixes M :: ('q, 'a) mttm
  and c c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
 assumes step: (c, c') ∈ mttm_step (alphabet_enlarge_delta M)
    and ss2: snd (snd (snd (snd (mt_state c)))) = SS2
  shows (c, c') ∈ mttm_step (ae_delta_ss2_ss3 M)
proof −
 from step obtain s ts n s' a' d where
    c_eq: c = Config_M s ts n
    and c'_eq: c' = Config_M s' (λk. (ts k)(n k := a' k))
                                (λk. go_dir (d k) (n k))
    and rel: (s, (λk. ts k (n k)), s', a', d)
             ∈ alphabet_enlarge_delta M
  by (auto elim: mttm_step.cases)
 from ss2 c_eq have s_ss2: snd (snd (snd s))) = SS2
  by simp
 have rel_sub: (s, (λk. ts k (n k)), s', a', d)
               ∈ ae_delta_ss2_ss3 M
  by (rule ae_delta_ss2_only[OF rel s_ss2])
 show ?thesis
  unfolding c_eq c'_eq
  by (rule mttm_step.step[where ts = ts and n = n and a = a' and dir = d,
                          OF rel_sub])
qed

lemma mttm_step_ae_delta_ss3_only:
 fixes M :: ('q, 'a) mttm
  and c c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
 assumes step: (c, c') ∈ mttm_step (alphabet_enlarge_delta M)
    and ss3: snd (snd (snd (snd (mt_state c)))) = SS3
  shows (c, c') ∈ mttm_step (ae_delta_ss3_ss4 M)
proof −
 from step obtain s ts n s' a' d where
```

```
    c_eq: c = Config_M s ts n
    and c'_eq: c' = Config_M s' (λk. (ts k)(n k := a' k))
                                (λk. go_dir (d k) (n k))
    and rel: (s, (λk. ts k (n k)), s', a', d)
             ∈ alphabet_enlarge_delta M
  by (auto elim: mttm_step.cases)
from ss3 c_eq have s_ss3: snd (snd (snd (snd s))) = SS3
  by simp
have rel_sub: (s, (λk. ts k (n k)), s', a', d)
              ∈ ae_delta_ss3_ss4 M
  by (rule ae_delta_ss3_only[OF rel s_ss3])
show ?thesis
  unfolding c_eq c'_eq
  by (rule mttm_step.step[where ts = ts and n = n and a = a' and dir = d,
                          OF rel_sub])
qed

lemma mttm_step_ae_delta_ss4_only:
  fixes M :: ('q, 'a) mttm
    and c c' :: ('c :: enum ⇒ 'a,
                 'q × ('a, 'c) ae_stage) mt_config
  assumes step: (c, c') ∈ mttm_step (alphabet_enlarge_delta M)
      and ss4: snd (snd (snd (snd (mt_state c)))) = SS4
    shows (c, c') ∈ mttm_step (ae_delta_ss4_ss5 M)
proof −
  from step obtain s ts n s' a' d where
    c_eq: c = Config_M s ts n
    and c'_eq: c' = Config_M s' (λk. (ts k)(n k := a' k))
                                (λk. go_dir (d k) (n k))
    and rel: (s, (λk. ts k (n k)), s', a', d)
             ∈ alphabet_enlarge_delta M
  by (auto elim: mttm_step.cases)
from ss4 c_eq have s_ss4: snd (snd (snd (snd s))) = SS4
  by simp
have rel_sub: (s, (λk. ts k (n k)), s', a', d)
              ∈ ae_delta_ss4_ss5 M
  by (rule ae_delta_ss4_only[OF rel s_ss4])
show ?thesis
  unfolding c_eq c'_eq
  by (rule mttm_step.step[where ts = ts and n = n and a = a' and dir = d,
                          OF rel_sub])
qed

lemma mttm_step_ae_delta_ss5_only:
  fixes M :: ('q, 'a) mttm
    and c c' :: ('c :: enum ⇒ 'a,
                 'q × ('a, 'c) ae_stage) mt_config
  assumes step: (c, c') ∈ mttm_step (alphabet_enlarge_delta M)
      and ss5: snd (snd (snd (snd (mt_state c)))) = SS5
```

```
    shows (c, c') ∈ mttm_step (ae_delta_ss5_ss6 M)
proof −
  from step obtain s ts n s' a' d where
      c_eq: c = Config_M s ts n
      and c'_eq: c' = Config_M s' (λk. (ts k)(n k := a' k))
                                  (λk. go_dir (d k) (n k))
      and rel: (s, (λk. ts k (n k)), s', a', d)
               ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  from ss5 c_eq have s_ss5: snd (snd (snd (snd s))) = SS5
    by simp
  have rel_sub: (s, (λk. ts k (n k)), s', a', d)
                ∈ ae_delta_ss5_ss6 M
    by (rule ae_delta_ss5_only[OF rel s_ss5])
  show ?thesis
    unfolding c_eq c'_eq
    by (rule mttm_step.step[where ts = ts and n = n and a = a' and dir = d,
                            OF rel_sub])
qed

lemma mttm_step_ae_delta_ss6_only:
  fixes M :: ('q, 'a) mttm
    and c c' :: ('c :: enum ⇒ 'a,
                 'q × ('a, 'c) ae_stage) mt_config
  assumes step: (c, c') ∈ mttm_step (alphabet_enlarge_delta M)
      and ss6: snd (snd (snd (snd (mt_state c)))) = SS6
    shows (c, c') ∈ mttm_step (ae_delta_ss6_ss7 M)
proof −
  from step obtain s ts n s' a' d where
      c_eq: c = Config_M s ts n
      and c'_eq: c' = Config_M s' (λk. (ts k)(n k := a' k))
                                  (λk. go_dir (d k) (n k))
      and rel: (s, (λk. ts k (n k)), s', a', d)
               ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  from ss6 c_eq have s_ss6: snd (snd (snd (snd s))) = SS6
    by simp
  have rel_sub: (s, (λk. ts k (n k)), s', a', d)
                ∈ ae_delta_ss6_ss7 M
    by (rule ae_delta_ss6_only[OF rel s_ss6])
  show ?thesis
    unfolding c_eq c'_eq
    by (rule mttm_step.step[where ts = ts and n = n and a = a' and dir = d,
                            OF rel_sub])
qed

lemma mttm_step_ae_delta_ss7_only:
  fixes M :: ('q, 'a) mttm
    and c c' :: ('c :: enum ⇒ 'a,
```

```
                'q × ('a, 'c) ae_stage) mt_config
  assumes step: (c, c') ∈ mttm_step (alphabet_enlarge_delta M)
      and ss7: snd (snd (snd (snd (mt_state c)))) = SS7
    shows (c, c') ∈ mttm_step (ae_delta_ss7_ss8 M)
proof −
  from step obtain s ts n s' a' d where
      c_eq: c = Config_M s ts n
      and c'_eq: c' = Config_M s' (λk. (ts k)(n k := a' k))
                                  (λk. go_dir (d k) (n k))
      and rel: (s, (λk. ts k (n k)), s', a', d)
               ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  from ss7 c_eq have s_ss7: snd (snd (snd (snd s))) = SS7
    by simp
  have rel_sub: (s, (λk. ts k (n k)), s', a', d)
                ∈ ae_delta_ss7_ss8 M
    by (rule ae_delta_ss7_only[OF rel s_ss7])
  show ?thesis
    unfolding c_eq c'_eq
    by (rule mttm_step.step[where ts = ts and n = n and a = a' and dir = d,
                            OF rel_sub])
qed

lemma mttm_step_ae_delta_ss8_only:
  fixes M :: ('q, 'a) mttm
    and c c' :: ('c :: enum ⇒ 'a,
                 'q × ('a, 'c) ae_stage) mt_config
  assumes step: (c, c') ∈ mttm_step (alphabet_enlarge_delta M)
      and ss8: snd (snd (snd (snd (mt_state c)))) = SS8
    shows (c, c') ∈ mttm_step (ae_delta_ss8_ss1 M)
proof −
  from step obtain s ts n s' a' d where
      c_eq: c = Config_M s ts n
      and c'_eq: c' = Config_M s' (λk. (ts k)(n k := a' k))
                                  (λk. go_dir (d k) (n k))
      and rel: (s, (λk. ts k (n k)), s', a', d)
               ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  from ss8 c_eq have s_ss8: snd (snd (snd (snd s))) = SS8
    by simp
  have rel_sub: (s, (λk. ts k (n k)), s', a', d)
                ∈ ae_delta_ss8_ss1 M
    by (rule ae_delta_ss8_only[OF rel s_ss8])
  show ?thesis
    unfolding c_eq c'_eq
    by (rule mttm_step.step[where ts = ts and n = n and a = a' and dir = d,
                            OF rel_sub])
qed
```

2.21.3 Buffered c-fold compute correctness

Algebraic correctness of the buffered c-fold compute: starting from a 3-block buffer matching a 3c-cell window of the M-tape, with M's head at the home block and within the non-LE region, the buffered compute simulates M's actual c-step trace. The conclusion exhibits a buffered-compute trajectory and a matching M-trace, with the post-state buffer / head still satisfying the window invariant.

Proof skeleton:

Induction on a step counter $i \in [0, c]$.

Base case ($i = 0$): pre-state matches itself by the *window* hypothesis.

Step case ($i \leq c$): assume the IH holds at step i. Either M halts at i (early-stop branch fires; $q_i \in \{t, r\}$; done), or M takes an $(i + 1)$-st step. In the second sub-case:

1. Read symbol from the buffer at IH's bp_i via *read_bp*; the window invariant gives this equals tsM_i (nM_i). 2. Apply M's δ (total on non-halting states by *valid_mttm*) to get ($q_\{i+1\}$, a', d). 3. Write a' k back to the buffer via *write_bp*. 4. Advance bp_i via *bp_advance_le*. The load-bearing claim is that this returns *Some* $bp_\{i+1\}$ (i.e., the head doesn't walk off the buffer). This holds because displacement after $i + 1$ steps is at most $i + 1 \leq c$, and the buffer covers 3c positions with the head starting at home (linearised positions $[c, 2c-1]$), so the head stays within $[1, 3c-2] \subset [0, 3c-1]$. 5. Verify the post-state still satisfies the window invariant: same *p_start*; new $bp_\{i+1\}$; buffer's linearised reading at position *bp_linear* $bp_\{i+1\}$ agrees with $tsM_\{i+1\}$ at $nM_\{i+1\}$.

The step case's load-bearing arithmetic (item 4) is a *bp_advance_le*-vs-tape-position commutation lemma: *bp_linear* of the advanced *bp* equals the linearised tape position relative to *p_start*. This sub-lemma is non-trivial but standalone it doesn't depend on M's δ, only on the encoding's arithmetic.

The *delta_total* precondition rules out the stuck-non-halt case: the substrate's *δ_set* axiom permits non-halting states with no δ-successor ($\delta \subseteq (Q - \{t,r\}) \times ...$ is a subset, not an equality). Without this hypothesis the conclusion is unprovable: a stuck qM has no buffered run and is non-halt, so neither disjunct of the final claim can hold. Hartmanis and Stearns's Theorem 2 implicitly assumes δ is total on $Q - \{t,r\}$; a caller whose machines are total by construction discharges this precondition trivially.

Per-tape unified companion of *ae_coupled_run_aux*, *_le0*, and *_le1*. Takes a per-tape regime selector *pos* :: $nat \Rightarrow nat$ (= *mt_pos* c' k at the SS4 entry) and the per-tape hybrid window-invariant predicate *ae_window_invariant_general*, which dispatches internally on each tape's regime. The buffered run is the same simultaneous *m_step_buffered M* relation as the three siblings the regime selector is frozen per tape during the run (*m_step_buffered* has no c' in scope), so per-tape regime case-splits

commute with the induction on n.

The no-LE hypothesis is per-tape regime-aware: width c for $pos = 0$, $2c$ for $pos = 1$, $3c$ for $pos \geq 2$. Uniform $p_start = (pos - 2) * c + 1$ simplifies to 1 on $pos \in \{0, 1\}$ by nat arithmetic, matching the three siblings' $Suc\ i$ shape.

The post-compute left-slot guard ($fst\ (buf'\ k) \neq LE_block$) is preserved on $pos = 0$ tapes (the generalisation of the le0 chain strengthening); le1 tapes have $fst\ (buf\ k) = LE_block$ on entry, and steady tapes do not need this property at the SS4→SS5 boundary.

It sits alongside *ae_coupled_run_aux_le0* and *_le1* as load-bearing inductive helpers. Consumer: *ae_m_steps_buffered_correct_trace_general*, feeding *ae_step_ss4_ss5_exists_general_trace*, which closes the SS4→SS5 trace in *ae_simulates_forward_stage_general*'s body.

lemma *ae_coupled_run_aux_general*:
  fixes $M :: ('q, 'a)\ mttm$
    and $qM :: 'q$
    and $tsM :: nat \Rightarrow nat \Rightarrow 'a$
    and $nM :: nat \Rightarrow nat$
    and $ofs :: nat \Rightarrow ('c :: enum)$
    and $buf_full :: nat \Rightarrow (('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a))$
    and $pos :: nat \Rightarrow nat$
    and $n :: nat$
    and $cM_n :: ('a, 'q)\ mt_config$
  assumes vM: $valid_mttm\ M$
      and lu: $le_unique\ M$
      and qM_in: $qM \in Q_tm\ M$
      and $window$: $\forall k{<}k_tm\ M.\ ae_window_invariant_general\ (tsM\ k)\ (nM\ k)$
        $(AE_Home, ofs\ k)\ (buf_full\ k)\ (pos\ k)\ (le_tm\ M)$
      and pad_home: $\forall k{\geq}k_tm\ M.\ fst\ (snd\ (buf_full\ k)) = bl_block\ (bl_tm\ M)$
      and $trace$: $(Config_M\ qM\ tsM\ nM, cM_n)$
        $\in mttm_step\ (delta_tm\ M)^{\frown\frown} n$
      and n_bound: $n \leq card\ (UNIV :: 'c\ set)$
      and no_le:
        $\forall k.\ (pos\ k = 0$
          $\longrightarrow (\forall i.\ i < card\ (UNIV :: 'c\ set)$
            $\longrightarrow tsM\ k\ (Suc\ i) \neq le_tm\ M))$
          $\wedge\ (pos\ k = 1$
            $\longrightarrow (\forall i.\ i < 2 * card\ (UNIV :: 'c\ set)$
              $\longrightarrow tsM\ k\ (Suc\ i) \neq le_tm\ M))$
          $\wedge\ (pos\ k \geq 2$
            $\longrightarrow (\forall i.\ i < 3 * card\ (UNIV :: 'c\ set)$
              $\longrightarrow tsM\ k$
                $((pos\ k - 2) * card\ (UNIV :: 'c\ set)$
                  $+ 1 + i)$
                $\neq le_tm\ M))$
      and $left_not_le_pos0$:
        $\forall k{<}k_tm\ M.\ pos\ k = 0 \longrightarrow fst\ (buf_full\ k) \neq LE_block\ (le_tm\ M)$

shows $\exists\, buf'\ end_pos.$
$((qM,\ buf_full,\ \lambda k.\ (AE_Home,\ ofs\ k)),$
$(mt_state\ cM_n,\ buf',\ end_pos))$
$\in (m_step_buffered\ M)\ ^{\frown\frown}\ n$
$\wedge\ (\forall\, k{<}k_tm\ M.\ ae_window_invariant_general$
$(mt_tape\ cM_n\ k)\ (mt_pos\ cM_n\ k)$
$(end_pos\ k)\ (buf'\ k)\ (pos\ k)\ (le_tm\ M))$
$\wedge\ (\forall\, k.\ (pos\ k = 0$
$\longrightarrow\ (\forall\, i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow\ mt_tape\ cM_n\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k = 1$
$\longrightarrow\ (\forall\, i.\ i < 2 * card\ (UNIV :: 'c\ set)$
$\longrightarrow\ mt_tape\ cM_n\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k \geq 2$
$\longrightarrow\ (\forall\, i.\ i < 3 * card\ (UNIV :: 'c\ set)$
$\longrightarrow\ mt_tape\ cM_n\ k$
$((pos\ k - 2) * card\ (UNIV :: 'c\ set)$
$+ 1 + i)$
$\neq le_tm\ M)))$
$\wedge\ (\forall\, k{<}k_tm\ M.\ pos\ k = 0$
$\longrightarrow\ fst\ (buf'\ k) \neq LE_block\ (le_tm\ M))$
$\wedge\ (\forall\, k.\ card\ (UNIV :: 'c\ set) \leq bp_linear\ (end_pos\ k) + n$
$\wedge\ bp_linear\ (end_pos\ k)$
$< 2 * card\ (UNIV :: 'c\ set) + n)$
$\wedge\ (\forall\, k{\geq}k_tm\ M.\ fst\ (snd\ (buf'\ k)) = bl_block\ (bl_tm\ M)$
$\wedge\ end_pos\ k = (AE_Home,\ ofs\ k))$
using *trace n_bound*
proof (*induction n arbitrary*: *cM_n*)
case *0*
— Base case: zero M-steps means cM_n is the initial config, the buffered run is empty, and the four output conjuncts hold by direct lifting of the lemma's input hypotheses (*window*, *no_le*, *left_not_le_pos0*) at the unchanged buffered state. Witnesses: $buf' = buf_full$, $end_pos = \lambda k.\ (AE_Home,\ ofs\ k)$.
from ‹$(Config_M\ qM\ tsM\ nM,\ cM_n)$
$\in mttm_step\ (delta_tm\ M)\ ^{\frown\frown}\ 0$›
have *cM_eq*: $cM_n = Config_M\ qM\ tsM\ nM$ **by** *simp*
hence *st*: $mt_state\ cM_n = qM$
and *tp*: $mt_tape\ cM_n = tsM$
and *ps*: $mt_pos\ cM_n = nM$ **by** *auto*
show *?case*
proof (*intro exI* [**where** $x = buf_full$]
exI [**where** $x = \lambda k.\ (AE_Home,\ ofs\ k)$]
conjI)
show $((qM,\ buf_full,\ \lambda k.\ (AE_Home,\ ofs\ k)),$
$(mt_state\ cM_n,\ buf_full,\ \lambda k.\ (AE_Home,\ ofs\ k)))$
$\in (m_step_buffered\ M)\ ^{\frown\frown}\ 0$
using *st* **by** *simp*
show $\forall\, k{<}k_tm\ M.\ ae_window_invariant_general$
$(mt_tape\ cM_n\ k)\ (mt_pos\ cM_n\ k)$

```
            ((λk. (AE_Home, ofs k)) k)
            (buf_full k) (pos k) (le_tm M)
    using window tp ps by simp
  show ∀k. (pos k = 0
            ⟶ (∀i. i < card (UNIV :: 'c set)
                    ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M))
         ∧ (pos k = 1
            ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
                    ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M))
         ∧ (pos k ≥ 2
            ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
                    ⟶ mt_tape cM_n k
                        ((pos k − 2) * card (UNIV :: 'c set)
                          + 1 + i)
                      ≠ le_tm M))
    using no_le tp by simp
  show ∀k<k_tm M. pos k = 0
            ⟶ fst (buf_full k) ≠ LE_block (le_tm M)
    using left_not_le_pos0 .
  show ∀k. card (UNIV :: 'c set)
            ≤ bp_linear ((λk. (AE_Home, ofs k)) k) + 0
         ∧ bp_linear ((λk. (AE_Home, ofs k)) k)
             < 2 * card (UNIV :: 'c set) + 0
  proof (intro allI conjI)
    fix k
    have lin: bp_linear ((λk. (AE_Home, ofs k)) k)
                = card (UNIV :: 'c set) + c_idx (ofs k)
      unfolding bp_linear_def by simp
    show card (UNIV :: 'c set)
          ≤ bp_linear ((λk. (AE_Home, ofs k)) k) + 0
      using lin by simp
    show bp_linear ((λk. (AE_Home, ofs k)) k)
          < 2 * card (UNIV :: 'c set) + 0
      using lin c_idx_lt_card[where x = ofs k] by simp
  qed
  show ∀k≥k_tm M. fst (snd (buf_full k)) = bl_block (bl_tm M)
            ∧ (λk. (AE_Home, ofs k)) k = (AE_Home, ofs k)
    using pad_home by simp
 qed
next
 case (Suc n')
```

— Inductive step: peel the last M-side step via *relpow_Suc_E* to obtain an intermediate config cM_n'; apply the IH at n' to obtain a coupled buffered run of length n'; extend it by one *m_step_buffered* step mirroring the M-side transition; re-establish the four output conjuncts at $n = Suc\ n'$.

The four ingredients are: (i) buffered read matches M-side read via window-invariant buf-linearisation; (ii) same δ-tuple drives both sides; (iii) *bp_advance_le* is total at every reached head position (*no_le* rules out the only None case); (iv) window invariant preserved by parallel writes and head advances.

```
  have n'_bound: n' ≤ card (UNIV :: 'c set)
    using Suc.prems(2) by simp
  from Suc.prems(1) obtain cM_n' where
      trace_n': (Config_M qM tsM nM, cM_n')
                  ∈ mttm_step (delta_tm M) ^^ n'
    and last_step: (cM_n', cM_n) ∈ mttm_step (delta_tm M)
    by (rule relpow_Suc_E)
  from Suc.IH[OF trace_n' n'_bound]
  obtain buf_n' end_pos_n' where
      coupled_n':
        ((qM, buf_full, λk. (AE_Home, ofs k)),
          (mt_state cM_n', buf_n', end_pos_n'))
            ∈ (m_step_buffered M) ^^ n'
    and window_n':
      ∀ k<k_tm M. ae_window_invariant_general
            (mt_tape cM_n' k) (mt_pos cM_n' k)
            (end_pos_n' k) (buf_n' k) (pos k) (le_tm M)
    and no_le_n':
      ∀ k. (pos k = 0
            ⟶ (∀ i. i < card (UNIV :: 'c set)
                  ⟶ mt_tape cM_n' k (Suc i)
                      ≠ le_tm M))
          ∧ (pos k = 1
              ⟶ (∀ i. i < 2 * card (UNIV :: 'c set)
                    ⟶ mt_tape cM_n' k (Suc i)
                        ≠ le_tm M))
          ∧ (pos k ≥ 2
              ⟶ (∀ i. i < 3 * card (UNIV :: 'c set)
                    ⟶ mt_tape cM_n' k
                        ((pos k − 2)
                           * card (UNIV :: 'c set)
                         + 1 + i)
                        ≠ le_tm M))
    and left_n':
      ∀ k<k_tm M. pos k = 0
          ⟶ fst (buf_n' k) ≠ LE_block (le_tm M)
    and bp_bound_n':
      ∀ k. card (UNIV :: 'c set) ≤ bp_linear (end_pos_n' k) + n'
          ∧ bp_linear (end_pos_n' k)
              < 2 * card (UNIV :: 'c set) + n'
    and pad_n':
      ∀ k≥k_tm M. fst (snd (buf_n' k)) = bl_block (bl_tm M)
          ∧ end_pos_n' k = (AE_Home, ofs k)
    by blast
```

— Unpack *last_step* via *mttm_step.cases* into its δ-tuple ingredients: a state shape $cM_n' = Config\ q_pre\ ts_pre\ n_pre$, the resulting cM_n as the M-side write + head-advance, and the $(q_pre, read, q_post, a_step, dir_step)$ δ-membership.

```
  from last_step obtain q_pre ts_pre n_pre q_post a_step dir_step where
      cM_n'_eq: cM_n' = Config_M q_pre ts_pre n_pre
```

```
  and cM_n_eq:
    cM_n = Config_M q_post
            (λk. (ts_pre k)(n_pre k := a_step k))
            (λk. go_dir (dir_step k) (n_pre k))
  and delta_mem:
    (q_pre, λk. ts_pre k (n_pre k),
      q_post, a_step, dir_step) ∈ delta_tm M
  by (rule mttm_step.cases)
 — Derived closed forms on cM_n' / cM_n's components  convenient handles for the buffered-side construction in the next increment.
 have st_n': mt_state cM_n' = q_pre
  and tp_n': mt_tape cM_n' = ts_pre
  and ps_n': mt_pos cM_n' = n_pre
  using cM_n'_eq by auto
 have st_n: mt_state cM_n = q_post
  and tp_n: ∀k. mt_tape cM_n k
              = (ts_pre k)(n_pre k := a_step k)
  and ps_n: ∀k. mt_pos cM_n k
              = go_dir (dir_step k) (n_pre k)
  using cM_n_eq by auto
 — At this point we have: (a) the coupled buffered run of length n' ending at (q_pre, buf_n', end_pos_n'); (b) the four invariants at cM_n'; (c) the M-side δ-tuple (q_pre, read, q_post, a_step, dir_step); (d) closed forms for cM_n's state / tape / pos.
 — Ingredient (i) of the inductive step: per-tape read-match. The buffered read at end_pos_n' k equals the M-side read at n_pre k, via the unified read_bp_via_window_general.
 have read_match:
   ∀k. read_bp (buf_n' k) (end_pos_n' k) = ts_pre k (n_pre k)
 proof (intro allI)
  fix k
  show read_bp (buf_n' k) (end_pos_n' k) = ts_pre k (n_pre k)
  proof (cases k < k_tm M)
   case True
    — Active tape: the read-match is the window invariant's buf-linearisation clause projected at the head offset.
    from window_n'[rule_format, OF True] have wi_k:
      ae_window_invariant_general
        (mt_tape cM_n' k) (mt_pos cM_n' k)
        (end_pos_n' k) (buf_n' k) (pos k) (le_tm M) .
    hence wi_k':
      ae_window_invariant_general
        (ts_pre k) (n_pre k)
        (end_pos_n' k) (buf_n' k) (pos k) (le_tm M)
     using tp_n' ps_n' by simp
    show ?thesis
     by (rule read_bp_via_window_general[OF wi_k'])
  next
   case False
```

— Padding tape $k \geq k_tm\ M$: M reads blank (*valid_mttm_delta_support*) and the buffered home block is blank with the head parked at home (*pad_n'*), so both sides read *bl_tm M*.

hence *kge*: $k_tm\ M \leq k$ **by** *simp*
from *pad_n'*[*rule_format*, *OF kge*]
have *ph*: $fst\ (snd\ (buf_n'\ k)) = bl_block\ (bl_tm\ M)$
and *ep*: $end_pos_n'\ k = (AE_Home, ofs\ k)$ **by** *simp_all*
have *rd*: $ts_pre\ k\ (n_pre\ k) = bl_tm\ M$
using *valid_mttm_delta_support*[*OF vM delta_mem kge*] **by** *simp*
obtain *l h r* **where** *blk*: $buf_n'\ k = (l, h, r)$
by (*cases* $buf_n'\ k$) *auto*
from *ph blk* **have** *h_eq*: $h = bl_block\ (bl_tm\ M)$ **by** *simp*
have $read_bp\ (buf_n'\ k)\ (end_pos_n'\ k) = bl_tm\ M$
using *blk ep h_eq* **by** (*simp add*: *read_bp_def bl_block_def*)
thus *?thesis* **using** *rd* **by** *simp*
qed
qed

— Per-tape position bounds at step *n'*: the fifth output conjunct of the IH (*bp_bound_n'*) plus the bound $Suc\ n' \leq c$ (giving $n' < c$) yields strict bounds $bp_linear < 3c - 1$ and $0 < bp_linear$ per tape, excluding both None configurations of *bp_advance*.

have *c_pos*: $1 \leq card\ (UNIV :: 'c\ set)$
using *Suc.prems*(*2*) **by** *linarith*
have *bp_lt_max*:
$\forall k.\ bp_linear\ (end_pos_n'\ k)$
$< 3 * card\ (UNIV :: 'c\ set) - 1$
proof (*intro allI*)
fix *k*
have *lin_lt*: $bp_linear\ (end_pos_n'\ k) < 2 * card\ (UNIV :: 'c\ set) + n'$
using *bp_bound_n'* **by** *blast*
show $bp_linear\ (end_pos_n'\ k) < 3 * card\ (UNIV :: 'c\ set) - 1$
using *lin_lt Suc.prems*(*2*) *c_pos* **by** *linarith*
qed
have *bp_gt_zero*:
$\forall k.\ 0 < bp_linear\ (end_pos_n'\ k)$
proof (*intro allI*)
fix *k*
have *low*: $card\ (UNIV :: 'c\ set) \leq bp_linear\ (end_pos_n'\ k) + n'$
using *bp_bound_n'* **by** *blast*
show $0 < bp_linear\ (end_pos_n'\ k)$
using *low Suc.prems*(*2*) *c_pos* **by** *linarith*
qed

— Ingredient (iii): per-tape *bp_advance_le* totality at step *n'*. Apply *bp_advance_le_total* per tape with *bp_lt_max*, *bp_gt_zero*; lift across tapes via choice to obtain the new per-tape head position function $end_pos_n :: nat \Rightarrow 'c\ bp$.

have *per_tape_total*:
$\forall k.\ \exists p'.\ bp_advance_le\ (le_tm\ M)$
$(ts_pre\ k\ (n_pre\ k))$

```
            (end_pos_n' k) (dir_step k)
        = Some p'
  proof (intro allI)
    fix k
    show ∃p'. bp_advance_le (le_tm M)
            (ts_pre k (n_pre k))
            (end_pos_n' k) (dir_step k)
        = Some p'
      by (rule bp_advance_le_total[OF bp_lt_max[rule_format]
                                      bp_gt_zero[rule_format]])
  qed
  from per_tape_total
  have ∃end_pos_n.
       ∀k. bp_advance_le (le_tm M)
             (ts_pre k (n_pre k))
             (end_pos_n' k) (dir_step k)
           = Some (end_pos_n k)
    by (rule choice)
  then obtain end_pos_n where
      end_pos_n_eq:
       ∀k. bp_advance_le (le_tm M)
             (ts_pre k (n_pre k))
             (end_pos_n' k) (dir_step k)
           = Some (end_pos_n k)
    by blast
```

— Ingredient (ii): assemble the buffered last step's membership in *m_step_buffered M*. Define the new buffer *buf_n* as the per-tape write-back of *buf_n'* at *end_pos_n'* with *a_step*; apply the intro rule *m_step_bufferedI* with the four ingredients (*delta_mem*, *read_match*'s reverse, *buf_n*'s definitional read-back, and *end_pos_n_eq*).

```
  define buf_n where
    buf_n = (λk. write_bp (buf_n' k) (end_pos_n' k) (a_step k))
  have read_match_sym:
      ∀k. ts_pre k (n_pre k) = read_bp (buf_n' k) (end_pos_n' k)
    using read_match by auto
  have buf_n_def_forall:
      ∀k. buf_n k = write_bp (buf_n' k) (end_pos_n' k) (a_step k)
    using buf_n_def by simp
  have m_step_last:
      ((q_pre, buf_n', end_pos_n'),
        (q_post, buf_n, end_pos_n))
          ∈ m_step_buffered M
    by (rule m_step_bufferedI[OF delta_mem read_match_sym
                                  buf_n_def_forall end_pos_n_eq])
```

— Chain extension: bridge *coupled_n''*s endpoint (*mt_state cM_n'*, *buf_n'*, *end_pos_n'*) through *st_n'* (*mt_state cM_n' = q_pre*) to match *m_step_last*'s start, then compose via *relpow_Suc_I* for a *Suc n'*-step buffered run ending at (*q_post*, *buf_n*, *end_pos_n*).

```
  have coupled_n_pre:
```

```
    ((qM, buf_full, λk. (AE_Home, ofs k)),
      (q_pre, buf_n', end_pos_n'))
        ∈ (m_step_buffered M) ^^ n'
  using coupled_n' st_n' by simp
 have coupled_n:
    ((qM, buf_full, λk. (AE_Home, ofs k)),
      (q_post, buf_n, end_pos_n))
        ∈ (m_step_buffered M) ^^ Suc n'
  using coupled_n_pre m_step_last by (rule relpow_Suc_I)
```

— Conjunct 5 (*bp_linear* bound at step Suc n'): apply the helper *bp_advance_le_lin_bounded* per tape. Given the IH's conjunct (*bp_bound_n'*) plus the position bounds (*bp_gt_zero*) plus the advance *end_pos_n_eq*, the helper closes both bound components.

```
 have bp_bound_n:
    ∀k. card (UNIV :: 'c set) ≤ bp_linear (end_pos_n k) + Suc n'
        ∧ bp_linear (end_pos_n k)
            < 2 * card (UNIV :: 'c set) + Suc n'
 proof (intro allI)
  fix k
  show card (UNIV :: 'c set) ≤ bp_linear (end_pos_n k) + Suc n'
      ∧ bp_linear (end_pos_n k)
          < 2 * card (UNIV :: 'c set) + Suc n'
   by (rule bp_advance_le_lin_bounded
            [OF end_pos_n_eq[rule_format, of k]
                bp_gt_zero[rule_format, of k]
                conjunct2[OF bp_bound_n'[rule_format, of k]]
                conjunct1[OF bp_bound_n'[rule_format, of k]]])
 qed
```

— Conjunct 4 (*left_not_le_pos0* at step Suc n'): in *pos k = 0* regime, the window invariant excludes *fst (end_pos_n' k) = AE_Left*, so *write_bp*'s update doesn't touch the left slot. Hence *fst (buf_n k) = fst (buf_n' k)*, and the IH's *left_n'* closes the goal.

```
 have left_n:
    ∀k<k_tm M. pos k = 0
        ⟶ fst (buf_n k) ≠ LE_block (le_tm M)
 proof (intro allI impI)
  fix k
  assume klt: k < k_tm M
  assume pos_0: pos k = 0
  from window_n'[rule_format, OF klt] have wi_k:
    ae_window_invariant_general (mt_tape cM_n' k) (mt_pos cM_n' k)
      (end_pos_n' k) (buf_n' k) (pos k) (le_tm M) .
  with pos_0 have wi_le0:
    ae_window_invariant_le0 (mt_tape cM_n' k) (mt_pos cM_n' k)
      (end_pos_n' k) (buf_n' k) (le_tm M)
   unfolding ae_window_invariant_general_def by simp
  have fst_bp_not_left: fst (end_pos_n' k) ≠ AE_Left
  proof (rule ccontr)
   assume ¬ fst (end_pos_n' k) ≠ AE_Left
```

```
    hence fst (end_pos_n' k) = AE_Left by simp
    with wi_le0 show False
      unfolding ae_window_invariant_le0_def by simp
  qed
  have fst_buf_n: fst (buf_n k) = fst (buf_n' k)
  proof −
    obtain l h r where blocks_eq: buf_n' k = (l, h, r)
      by (cases buf_n' k) auto
    obtain b off where bp_eq: end_pos_n' k = (b, off)
      by (cases end_pos_n' k) auto
    from fst_bp_not_left bp_eq have b ≠ AE_Left by simp
    thus ?thesis
      unfolding buf_n_def
      using blocks_eq bp_eq
      by (cases b) (auto simp: write_bp_def)
  qed
  have fst (buf_n' k) ≠ LE_block (le_tm M)
    using left_n' pos_0 klt by blast
  thus fst (buf_n k) ≠ LE_block (le_tm M)
    using fst_buf_n by simp
qed
```

— Conjunct 1 (coupled run with *mt_state cM_n* witness at step Suc n'): re-express *coupled_n*'s endpoint state from *q_post* to *mt_state cM_n* via *st_n*.

```
have coupled_chain:
    ((qM, buf_full, λk. (AE_Home, ofs k)),
      (mt_state cM_n, buf_n, end_pos_n))
        ∈ (m_step_buffered M) ^^ Suc n'
  using coupled_n st_n by simp
```

— Contrapositive of the substrate axiom *valid_mttm_deltaLE_no_write*: if M reads non-LE on tape k cell *n_pre k*, then it does not write LE on tape k. This is what powers the $j = n_pre\ k$ case of conjunct 3: the new tape value at the updated cell is *a_step k*, and we need to show *a_step k* ≠ *le* when the read value was in a no-LE window.

```
have no_le_contra:
    ∀k. ts_pre k (n_pre k) ≠ le_tm M
          ⟶ a_step k ≠ le_tm M
proof (intro allI impI)
  fix k
  assume rd_not_le: ts_pre k (n_pre k) ≠ le_tm M
  show a_step k ≠ le_tm M
  proof
    assume a_le: a_step k = le_tm M
    have (λk. ts_pre k (n_pre k)) k = le_tm M
      by (rule valid_mttm_deltaLE_no_write[OF lu delta_mem a_le])
    hence ts_pre k (n_pre k) = le_tm M by simp
    with rd_not_le show False ..
  qed
qed
```

— Conjunct 3 regime *pos k = 0* (no LE in cells *Suc 0 .. card UNIV*): for each

cell *Suc i* in the window, case-split on whether the M-step's update site *n_pre k* coincides with *Suc i*.
· If not, the fun-update leaves *mt_tape cM_n k* (*Suc i*) equal to *ts_pre k* (*Suc i*) = *mt_tape cM_n′ k* (*Suc i*); the IH (*no_le_n′*) closes the goal.
· If yes, the new value is *a_step k*; the IH gives *ts_pre k* (*n_pre k*) = *ts_pre k* (*Suc i*) ≠ *le_tm M*; the contrapositive *no_le_contra* gives *a_step k* ≠ *le_tm M*.

```
  have no_le0_n:
     ∀k. pos k = 0
           ⟶ (∀i. i < card (UNIV :: 'c set)
                 ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M)
  proof (intro allI impI allI impI)
    fix k i
    assume pos_0: pos k = 0
       and i_bd: i < card (UNIV :: 'c set)
    have ih_at: ts_pre k (Suc i) ≠ le_tm M
    proof −
      from no_le_n′ pos_0 i_bd
      have mt_tape cM_n′ k (Suc i) ≠ le_tm M by blast
      thus ?thesis using tp_n′ by simp
    qed
    show mt_tape cM_n k (Suc i) ≠ le_tm M
    proof (cases Suc i = n_pre k)
      case False
      hence mt_tape cM_n k (Suc i) = ts_pre k (Suc i)
        using tp_n by simp
      thus ?thesis using ih_at by simp
    next
      case True
      hence tape_eq: mt_tape cM_n k (Suc i) = a_step k
        using tp_n by simp
      from True have ts_pre k (n_pre k) = ts_pre k (Suc i)
        by simp
      with ih_at have ts_pre k (n_pre k) ≠ le_tm M by simp
      hence a_step k ≠ le_tm M using no_le_contra by blast
      thus ?thesis using tape_eq by simp
    qed
  qed
```

— Conjunct 3 regime *pos k = 1*: same window shape *Suc i* as *pos = 0*, but the bound is doubled to *2 ∗ card UNIV*. The proof shape is identical; only the bound on *i* changes.

```
  have no_le1_n:
     ∀k. pos k = 1
           ⟶ (∀i. i < 2 ∗ card (UNIV :: 'c set)
                 ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M)
  proof (intro allI impI allI impI)
    fix k i
    assume pos_1: pos k = 1
       and i_bd: i < 2 ∗ card (UNIV :: 'c set)
    have ih_at: ts_pre k (Suc i) ≠ le_tm M
```

proof −
from *no_le_n′ pos_1 i_bd*
have *mt_tape cM_n′ k (Suc i) ≠ le_tm M* **by** *blast*
thus *?thesis* **using** *tp_n′* **by** *simp*
qed
show *mt_tape cM_n k (Suc i) ≠ le_tm M*
proof (*cases Suc i = n_pre k*)
case *False*
hence *mt_tape cM_n k (Suc i) = ts_pre k (Suc i)*
using *tp_n* **by** *simp*
thus *?thesis* **using** *ih_at* **by** *simp*
next
case *True*
hence *tape_eq*: *mt_tape cM_n k (Suc i) = a_step k*
using *tp_n* **by** *simp*
from *True* **have** *ts_pre k (n_pre k) = ts_pre k (Suc i)*
by *simp*
with *ih_at* **have** *ts_pre k (n_pre k) ≠ le_tm M* **by** *simp*
hence *a_step k ≠ le_tm M* **using** *no_le_contra* **by** *blast*
thus *?thesis* **using** *tape_eq* **by** *simp*
qed
qed

— Conjunct 3 regime *pos k ≥ 2* (steady regime): the no-LE window spans *3 ∗ card UNIV* cells starting at offset *(pos k − 2) ∗ card UNIV + 1*. The proof shape is identical to the *pos = 0*/*pos = 1* cases; only the cell expression changes.

have *no_le2_n*:
∀ *k*. *pos k ≥ 2*
⟶ (∀ *i*. *i < 3 ∗ card (UNIV :: ′c set)*
⟶ *mt_tape cM_n k*
((pos k − 2) ∗ card (UNIV :: ′c set) + 1 + i)
≠ le_tm M)
proof (*intro allI impI allI impI*)
fix *k i*
assume *pos_ge*: *pos k ≥ 2*
and *i_bd*: *i < 3 ∗ card (UNIV :: ′c set)*
let *?j = (pos k − 2) ∗ card (UNIV :: ′c set) + 1 + i*
have *ih_at*: *ts_pre k ?j ≠ le_tm M*
proof −
from *no_le_n′ pos_ge i_bd*
have *mt_tape cM_n′ k ?j ≠ le_tm M* **by** *blast*
thus *?thesis* **using** *tp_n′* **by** *simp*
qed
show *mt_tape cM_n k ?j ≠ le_tm M*
proof (*cases ?j = n_pre k*)
case *False*
hence *mt_tape cM_n k ?j = ts_pre k ?j*
using *tp_n* **by** *simp*
thus *?thesis* **using** *ih_at* **by** *simp*
next

```
    case True
    hence tape_eq: mt_tape cM_n k ?j = a_step k
      using tp_n by simp
    from True have ts_pre k (n_pre k) = ts_pre k ?j
      by simp
    with ih_at have ts_pre k (n_pre k) ≠ le_tm M by simp
    hence a_step k ≠ le_tm M using no_le_contra by blast
    thus ?thesis using tape_eq by simp
  qed
qed
```

— Conjunct 3 (no-LE invariant at *cM_n*): assemble the three regime sub-lemmas *no_le0_n*, *no_le1_n*, *no_le2_n* into the dispatch-on-*pos* shape that matches the lemma statement.

```
have no_le_n:
  ∀k. (pos k = 0
        ⟶ (∀i. i < card (UNIV :: 'c set)
              ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M))
      ∧ (pos k = 1
        ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
              ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M))
      ∧ (pos k ≥ 2
        ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
              ⟶ mt_tape cM_n k
                  ((pos k − 2) * card (UNIV :: 'c set)
                    + 1 + i)
                ≠ le_tm M))
  using no_le0_n no_le1_n no_le2_n by blast
```

— Conjunct 2 regime *pos k ≥ 2* (steady regime): apply *ae_window_invariant_step* per tape. Five ingredients: the IH-side invariant from *window_n'*; the *bp_advance_le* step from *end_pos_n_eq*; the position bounds from *bp_lt_max* and *bp_gt_zero*; and the no-LE-at-head fact derived from the IH's *no_le_n'* instantiated at the buffered-head's linearised offset (which lies inside the regime's no-LE window).

```
have win2_n:
  ∀k<k_tm M. pos k ≥ 2
        ⟶ ae_window_invariant
            (mt_tape cM_n k) (mt_pos cM_n k)
            (end_pos_n k) (buf_n k)
            ((pos k − 2) * card (UNIV :: 'c set) + 1)
proof (intro allI impI)
  fix k
  assume klt: k < k_tm M
  assume pos_ge: pos k ≥ 2
  from window_n'[rule_format, OF klt] pos_ge have wi_k:
    ae_window_invariant (mt_tape cM_n' k) (mt_pos cM_n' k)
        (end_pos_n' k) (buf_n' k)
        ((pos k − 2) * card (UNIV :: 'c set) + 1)
    unfolding ae_window_invariant_general_def by simp
  hence wi_k': ae_window_invariant (ts_pre k) (n_pre k)
```

```
                (end_pos_n' k) (buf_n' k)
                ((pos k − 2) ∗ card (UNIV :: 'c set) + 1)
    using tp_n' ps_n' by simp
  have adv_k: bp_advance_le (le_tm M) (ts_pre k (n_pre k))
                (end_pos_n' k) (dir_step k) = Some (end_pos_n k)
    using end_pos_n_eq by blast
  have bp_lt_k: bp_linear (end_pos_n' k)
                  < 3 ∗ card (UNIV :: 'c set) − 1
    using bp_lt_max by blast
  have bp_pos_k: 0 < bp_linear (end_pos_n' k)
    using bp_gt_zero by blast
  — No-LE-at-head: n_pre k = (pos k − 2)∗c + 1 + j where j = bp_linear
(end_pos_n' k) < 3c; instantiate the IH's no-LE window at this j.
  have no_le_k: ts_pre k (n_pre k) ≠ le_tm M
  proof −
    let ?j = bp_linear (end_pos_n' k)
    from wi_k' have head_corr:
       n_pre k = (pos k − 2) ∗ card (UNIV :: 'c set) + 1 + ?j
      unfolding ae_window_invariant_def by simp
    have j_lt: ?j < 3 ∗ card (UNIV :: 'c set)
      by (rule bp_linear_lt_3c)
    from no_le_n' pos_ge j_lt have
       mt_tape cM_n' k
          ((pos k − 2) ∗ card (UNIV :: 'c set) + 1 + ?j)
        ≠ le_tm M by blast
    hence ts_pre k
          ((pos k − 2) ∗ card (UNIV :: 'c set) + 1 + ?j)
        ≠ le_tm M using tp_n' by simp
    thus ?thesis using head_corr by simp
  qed
  have wi_step:
     ae_window_invariant
          ((ts_pre k)(n_pre k := a_step k))
          (go_dir (dir_step k) (n_pre k))
          (end_pos_n k)
          (write_bp (buf_n' k) (end_pos_n' k) (a_step k))
          ((pos k − 2) ∗ card (UNIV :: 'c set) + 1)
    by (rule ae_window_invariant_step
          [OF wi_k' adv_k bp_lt_k bp_pos_k no_le_k])
  show ae_window_invariant
          (mt_tape cM_n k) (mt_pos cM_n k)
          (end_pos_n k) (buf_n k)
          ((pos k − 2) ∗ card (UNIV :: 'c set) + 1)
    using wi_step tp_n ps_n buf_n_def by simp
qed
```

— Conjunct 2 regime *pos k = 1* (LE-edge *le1*): apply *ae_window_invariant_le1_step* per tape. The two LE-aware preconditions:
· *a_le_k* (*tM nM = le* ⟹ *a_step k = le* ∧ *dir_step k* ∈ {*N*, *R*}): derived from *valid_mttm_deltaLE* applied to *delta_mem* at tape *k*.

· *a_not_le_k* (*tM nM* ≠ *le* ⟹ *a_step k* ≠ *le*): a per-tape instance of *no_le_contra* already in scope.
The *no_le_win_k* precondition (no LE in cells *1..2c*) follows from *no_le_n′*'s *pos = 1* branch via *tp_n′*.

```
  have win1_n:
     ∀ k<k_tm M. pos k = 1
           ⟶ ae_window_invariant_le1
                (mt_tape cM_n k) (mt_pos cM_n k)
                (end_pos_n k) (buf_n k) (le_tm M)
  proof (intro allI impI)
    fix k
    assume klt: k < k_tm M
    assume pos_1: pos k = 1
    from window_n′[rule_format, OF klt] pos_1 have wi_k:
       ae_window_invariant_le1 (mt_tape cM_n′ k) (mt_pos cM_n′ k)
            (end_pos_n′ k) (buf_n′ k) (le_tm M)
      unfolding ae_window_invariant_general_def by simp
    hence wi_k′: ae_window_invariant_le1 (ts_pre k) (n_pre k)
                  (end_pos_n′ k) (buf_n′ k) (le_tm M)
      using tp_n′ ps_n′ by simp
    have adv_k: bp_advance_le (le_tm M) (ts_pre k (n_pre k))
                  (end_pos_n′ k) (dir_step k) = Some (end_pos_n k)
      using end_pos_n_eq by blast
    have bp_lt_k: bp_linear (end_pos_n′ k)
                    < 3 * card (UNIV :: 'c set) - 1
      using bp_lt_max by blast
    have bp_pos_k: 0 < bp_linear (end_pos_n′ k)
      using bp_gt_zero by blast
    have a_le_k:
       ts_pre k (n_pre k) = le_tm M
          ⟹ a_step k = le_tm M ∧ dir_step k ∈ {dir.N, dir.R}
    proof −
      assume rd_le: ts_pre k (n_pre k) = le_tm M
      have a_step k = le_tm M ∧ dir_step k ∈ {dir.N, dir.R}
        by (rule valid_mttm_deltaLE[OF vM delta_mem, of k, OF rd_le])
      thus a_step k = le_tm M ∧ dir_step k ∈ {dir.N, dir.R} .
    qed
    have a_not_le_k:
       ts_pre k (n_pre k) ≠ le_tm M ⟹ a_step k ≠ le_tm M
      using no_le_contra by blast
    have no_le_win_k:
       ∀ i. 0 < i ∧ i ≤ 2 * card (UNIV :: 'c set)
             ⟶ ts_pre k i ≠ le_tm M
    proof (intro allI impI)
      fix i
      assume i_bd: 0 < i ∧ i ≤ 2 * card (UNIV :: 'c set)
      obtain j where j_eq: i = Suc j using i_bd by (cases i) auto
      have j_lt: j < 2 * card (UNIV :: 'c set)
        using i_bd j_eq by simp
```

from *no_le_n' pos_1 j_lt*
have *mt_tape cM_n' k (Suc j) ≠ le_tm M* **by** *blast*
thus *ts_pre k i ≠ le_tm M* **using** *tp_n' j_eq* **by** *simp*
qed
have *wi_step*:
ae_window_invariant_le1
((ts_pre k)(n_pre k := a_step k))
(go_dir (dir_step k) (n_pre k))
(end_pos_n k)
(write_bp (buf_n' k) (end_pos_n' k) (a_step k))
(le_tm M)
by (*rule ae_window_invariant_le1_step*
[*OF wi_k' adv_k bp_lt_k bp_pos_k*
a_le_k a_not_le_k no_le_win_k])
show *ae_window_invariant_le1*
(mt_tape cM_n k) (mt_pos cM_n k)
(end_pos_n k) (buf_n k) (le_tm M)
using *wi_step tp_n ps_n buf_n_def* **by** *simp*
qed

— Conjunct 2 regime *pos k = 0* (LE-edge *le0*): apply *ae_window_invariant_le0_step* per tape. Pattern identical to the *pos = 1* call; only the lemma and the no-LE window range (cells *1..c*) differ.

have *win0_n*:
∀ *k*<*k_tm M*. *pos k = 0*
⟶ *ae_window_invariant_le0*
(mt_tape cM_n k) (mt_pos cM_n k)
(end_pos_n k) (buf_n k) (le_tm M)
proof (*intro allI impI*)
fix *k*
assume *klt*: *k < k_tm M*
assume *pos_0*: *pos k = 0*
from *window_n'*[*rule_format*, *OF klt*] *pos_0* **have** *wi_k*:
ae_window_invariant_le0 (mt_tape cM_n' k) (mt_pos cM_n' k)
(end_pos_n' k) (buf_n' k) (le_tm M)
unfolding *ae_window_invariant_general_def* **by** *simp*
hence *wi_k'*: *ae_window_invariant_le0 (ts_pre k) (n_pre k)*
(end_pos_n' k) (buf_n' k) (le_tm M)
using *tp_n' ps_n'* **by** *simp*
have *adv_k*: *bp_advance_le (le_tm M) (ts_pre k (n_pre k))*
(end_pos_n' k) (dir_step k) = Some (end_pos_n k)
using *end_pos_n_eq* **by** *blast*
have *bp_lt_k*: *bp_linear (end_pos_n' k)*
*< 3 * card (UNIV :: 'c set) − 1*
using *bp_lt_max* **by** *blast*
have *bp_pos_k*: *0 < bp_linear (end_pos_n' k)*
using *bp_gt_zero* **by** *blast*
have *a_le_k*:
ts_pre k (n_pre k) = le_tm M
⟹ *a_step k = le_tm M ∧ dir_step k ∈ {dir.N, dir.R}*

```
    proof −
      assume rd_le: ts_pre k (n_pre k) = le_tm M
      show a_step k = le_tm M ∧ dir_step k ∈ {dir.N, dir.R}
        by (rule valid_mttm_deltaLE[OF vM delta_mem, of k, OF rd_le])
    qed
    have a_not_le_k:
       ts_pre k (n_pre k) ≠ le_tm M ⟹ a_step k ≠ le_tm M
      using no_le_contra by blast
    have no_le_win_k:
       ∀ i. 0 < i ∧ i ≤ card (UNIV :: 'c set)
              ⟶ ts_pre k i ≠ le_tm M
    proof (intro allI impI)
      fix i
      assume i_bd: 0 < i ∧ i ≤ card (UNIV :: 'c set)
      obtain j where j_eq: i = Suc j using i_bd by (cases i) auto
      have j_lt: j < card (UNIV :: 'c set)
        using i_bd j_eq by simp
      from no_le_n' pos_0 j_lt
      have mt_tape cM_n' k (Suc j) ≠ le_tm M by blast
      thus ts_pre k i ≠ le_tm M using tp_n' j_eq by simp
    qed
    have wi_step:
       ae_window_invariant_le0
            ((ts_pre k)(n_pre k := a_step k))
            (go_dir (dir_step k) (n_pre k))
            (end_pos_n k)
            (write_bp (buf_n' k) (end_pos_n' k) (a_step k))
            (le_tm M)
      by (rule ae_window_invariant_le0_step
              [OF wi_k' adv_k bp_lt_k bp_pos_k
                  a_le_k a_not_le_k no_le_win_k])
    show ae_window_invariant_le0
            (mt_tape cM_n k) (mt_pos cM_n k)
            (end_pos_n k) (buf_n k) (le_tm M)
      using wi_step tp_n ps_n buf_n_def by simp
  qed
```

— Conjunct 2 (window invariant at *cM_n*): assemble the three regime sub-lemmas *win0_n*, *win1_n*, *win2_n* into the dispatch-on-*pos* shape that matches the lemma statement (*ae_window_invariant_general*).

```
  have window_n:
     ∀ k<k_tm M. ae_window_invariant_general
            (mt_tape cM_n k) (mt_pos cM_n k)
            (end_pos_n k) (buf_n k) (pos k) (le_tm M)
  proof (intro allI impI)
    fix k
    assume klt: k < k_tm M
    show ae_window_invariant_general
          (mt_tape cM_n k) (mt_pos cM_n k)
          (end_pos_n k) (buf_n k) (pos k) (le_tm M)
```

unfolding *ae_window_invariant_general_def*
using *win0_n win1_n win2_n klt* **by** *blast*
qed
— Padding shadow invariant at step *Suc n′*: empty tapes $k \geq k_tm\ M$ keep a blank home block and a head parked at (*AE_Home*, *ofs k*). Preserved from the IH's *pad_n′* by the M-step's δ-support (*read* $= a' = bl$, $d = N$ past tape count): the head stays put under *bp_advance* of a *dir.N* step, and writing *bl* into the blank home is idempotent.
have *pad_n*:
$\forall k{\geq}k_tm\ M.\ fst\ (snd\ (buf_n\ k)) = bl_block\ (bl_tm\ M)$
$\wedge\ end_pos_n\ k = (AE_Home,\ ofs\ k)$
proof (*intro allI impI*)
fix *k*
assume *kge*: $k_tm\ M \leq k$
from *pad_n′*[*rule_format*, *OF kge*]
have *ph*: $fst\ (snd\ (buf_n'\ k)) = bl_block\ (bl_tm\ M)$
and *ep*: $end_pos_n'\ k = (AE_Home,\ ofs\ k)$ **by** *simp_all*
from *valid_mttm_delta_support*[*OF vM delta_mem kge*]
have *rd*: $ts_pre\ k\ (n_pre\ k) = bl_tm\ M$
and *wr*: $a_step\ k = bl_tm\ M$
and *dr*: $dir_step\ k = dir.N$ **by** *simp_all*
have *ep_n*: $end_pos_n\ k = (AE_Home,\ ofs\ k)$
proof −
have *adv*: $bp_advance_le\ (le_tm\ M)\ (ts_pre\ k\ (n_pre\ k))$
$(end_pos_n'\ k)\ (dir_step\ k) = Some\ (end_pos_n\ k)$
using *end_pos_n_eq* **by** *blast*
have $bp_advance_le\ (le_tm\ M)\ (bl_tm\ M)\ (AE_Home,\ ofs\ k)\ dir.N$
$= Some\ (AE_Home,\ ofs\ k)$
by (*simp add*: *bp_advance_le_def bp_advance_def*)
thus *?thesis* **using** *adv rd ep dr* **by** *simp*
qed
have *buf_n_k*: $fst\ (snd\ (buf_n\ k)) = bl_block\ (bl_tm\ M)$
proof −
obtain *l h r* **where** *blk*: $buf_n'\ k = (l,\ h,\ r)$
by (*cases* $buf_n'\ k$) *auto*
from *ph blk* **have** *h_eq*: $h = bl_block\ (bl_tm\ M)$ **by** *simp*
have $buf_n\ k = (l,\ h(ofs\ k := bl_tm\ M),\ r)$
using *blk ep wr* **by** (*simp add*: *buf_n_def write_bp_def*)
hence $fst\ (snd\ (buf_n\ k)) = h(ofs\ k := bl_tm\ M)$ **by** *simp*
also have $\ldots = bl_block\ (bl_tm\ M)$
using *h_eq* **by** (*simp add*: *bl_block_def fun_eq_iff*)
finally show *?thesis* **.**
qed
show $fst\ (snd\ (buf_n\ k)) = bl_block\ (bl_tm\ M)$
$\wedge\ end_pos_n\ k = (AE_Home,\ ofs\ k)$
using *buf_n_k ep_n* **by** *simp*
qed
— Final *show ?case* assembly: package the five discharged conjuncts (*coupled_chain*, *window_n*, *no_le_n*, *left_n*, *bp_bound_n*) into the existential output

```
via buf_n and end_pos_n.
  show ?case
  proof (intro exI [where x = buf_n] exI [where x = end_pos_n] conjI)
    show ((qM, buf_full, λk. (AE_Home, ofs k)),
          (mt_state cM_n, buf_n, end_pos_n))
            ∈ (m_step_buffered M) ^^ Suc n'
      by (rule coupled_chain)
    show ∀ k<k_tm M. ae_window_invariant_general
              (mt_tape cM_n k) (mt_pos cM_n k)
              (end_pos_n k) (buf_n k) (pos k) (le_tm M)
      by (rule window_n)
    show ∀ k. (pos k = 0
              ⟶ (∀ i. i < card (UNIV :: 'c set)
                    ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M))
           ∧ (pos k = 1
                ⟶ (∀ i. i < 2 * card (UNIV :: 'c set)
                      ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M))
           ∧ (pos k ≥ 2
                ⟶ (∀ i. i < 3 * card (UNIV :: 'c set)
                      ⟶ mt_tape cM_n k
                          ((pos k − 2) * card (UNIV :: 'c set)
                            + 1 + i)
                        ≠ le_tm M))
      by (rule no_le_n)
    show ∀ k<k_tm M. pos k = 0
              ⟶ fst (buf_n k) ≠ LE_block (le_tm M)
      by (rule left_n)
    show ∀ k. card (UNIV :: 'c set) ≤ bp_linear (end_pos_n k) + Suc n'
           ∧ bp_linear (end_pos_n k)
               < 2 * card (UNIV :: 'c set) + Suc n'
      by (rule bp_bound_n)
    show ∀ k≥k_tm M. fst (snd (buf_n k)) = bl_block (bl_tm M)
              ∧ end_pos_n k = (AE_Home, ofs k)
      by (rule pad_n)
  qed
qed
```

Per-tape unified companion of *ae_m_steps_buffered_correct_trace*, *_le0*, and *_le1*. A wrapper over *ae_coupled_run_aux_general*: builds the *m_steps_buffered* step (= relpow plus $kM \leq c$ and end-or-halt) and repackages the aux's existential into an *obtains*-style witness.

The hypothesis shape matches the three siblings' wrapper pattern: a per-tape regime selector *pos*, the per-tape hybrid window invariant, the per-tape regime-guarded *no_le*, and the *pos = 0*-conditional left-slot guard. Output conjuncts mirror the input, with the buffered config existentially bound.

```
lemma ae_m_steps_buffered_correct_trace_general:
  fixes M :: ('q, 'a) mttm
    and qM :: 'q
    and tsM :: nat ⇒ nat ⇒ 'a
```

and $nM :: nat \Rightarrow nat$
and $ofs :: nat \Rightarrow ('c :: enum)$
and $buf_full :: nat \Rightarrow (('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a))$
and $pos :: nat \Rightarrow nat$
and $kM :: nat$
and $cM_k :: ('a, 'q)\ mt_config$
assumes vM: $valid_mttm\ M$
and lu: $le_unique\ M$
and qM_in: $qM \in Q_tm\ M$
and $window$: $\forall k{<}k_tm\ M.\ ae_window_invariant_general\ (tsM\ k)\ (nM\ k)$
$(AE_Home, ofs\ k)\ (buf_full\ k)\ (pos\ k)\ (le_tm\ M)$
and pad_home: $\forall k{\geq}k_tm\ M.\ fst\ (snd\ (buf_full\ k)) = bl_block\ (bl_tm\ M)$
and no_le:
$\forall k.\ (pos\ k = 0$
$\longrightarrow (\forall i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow tsM\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k = 1$
$\longrightarrow (\forall i.\ i < 2 * card\ (UNIV :: 'c\ set)$
$\longrightarrow tsM\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k \geq 2$
$\longrightarrow (\forall i.\ i < 3 * card\ (UNIV :: 'c\ set)$
$\longrightarrow tsM\ k$
$((pos\ k - 2) * card\ (UNIV :: 'c\ set)$
$+ 1 + i)$
$\neq le_tm\ M))$
and $trace$: $(Config_M\ qM\ tsM\ nM, cM_k)$
$\in mttm_step\ (delta_tm\ M)\ ^{\frown\frown}\ kM$
and kM_le: $kM \leq card\ (UNIV :: 'c\ set)$
and end_or_halt:
$kM = card\ (UNIV :: 'c\ set)$
$\vee\ mt_state\ cM_k \in \{t_tm\ M, r_tm\ M\}$
and $left_not_le_pos0$:
$\forall k{<}k_tm\ M.\ pos\ k = 0 \longrightarrow fst\ (buf_full\ k) \neq LE_block\ (le_tm\ M)$
obtains $q_out\ buf'\ end_pos$ **where**
$((qM, buf_full, \lambda k.\ (AE_Home, ofs\ k)),$
$(q_out, buf', end_pos)) \in m_steps_buffered\ M$
and $q_out = mt_state\ cM_k$
and $\forall k{<}k_tm\ M.\ ae_window_invariant_general$
$(mt_tape\ cM_k\ k)\ (mt_pos\ cM_k\ k)$
$(end_pos\ k)\ (buf'\ k)\ (pos\ k)\ (le_tm\ M)$
and $\forall k.\ (pos\ k = 0$
$\longrightarrow (\forall i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow mt_tape\ cM_k\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k = 1$
$\longrightarrow (\forall i.\ i < 2 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt_tape\ cM_k\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k \geq 2$
$\longrightarrow (\forall i.\ i < 3 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt_tape\ cM_k\ k$

```
                                ((pos k − 2) ∗ card (UNIV :: 'c set)
                                  + 1 + i)
                               ≠ le_tm M))
   and ∀ k<k_tm M. pos k = 0
         ⟶ fst (buf' k) ≠ LE_block (le_tm M)
proof −
  have ae_result_general:
    ∃ buf' end_pos.
       ((qM, buf_full, λk. (AE_Home, ofs k)),
        (mt_state cM_k, buf', end_pos))
          ∈ (m_step_buffered M) ^^ kM
      ∧ (∀ k<k_tm M. ae_window_invariant_general
            (mt_tape cM_k k) (mt_pos cM_k k)
            (end_pos k) (buf' k) (pos k) (le_tm M))
      ∧ (∀ k. (pos k = 0
            ⟶ (∀ i. i < card (UNIV :: 'c set)
                  ⟶ mt_tape cM_k k (Suc i) ≠ le_tm M))
          ∧ (pos k = 1
              ⟶ (∀ i. i < 2 ∗ card (UNIV :: 'c set)
                    ⟶ mt_tape cM_k k (Suc i) ≠ le_tm M))
          ∧ (pos k ≥ 2
              ⟶ (∀ i. i < 3 ∗ card (UNIV :: 'c set)
                    ⟶ mt_tape cM_k k
                        ((pos k − 2) ∗ card (UNIV :: 'c set)
                          + 1 + i)
                       ≠ le_tm M)))
      ∧ (∀ k<k_tm M. pos k = 0
            ⟶ fst (buf' k) ≠ LE_block (le_tm M))
      ∧ (∀ k. card (UNIV :: 'c set) ≤ bp_linear (end_pos k) + kM
          ∧ bp_linear (end_pos k)
              < 2 ∗ card (UNIV :: 'c set) + kM)
      ∧ (∀ k≥k_tm M. fst (snd (buf' k)) = bl_block (bl_tm M)
          ∧ end_pos k = (AE_Home, ofs k))
    by (rule ae_coupled_run_aux_general[OF vM lu qM_in window pad_home
trace kM_le
                                        no_le left_not_le_pos0])
  obtain buf' end_pos where
    coupled: ((qM, buf_full, λk. (AE_Home, ofs k)),
              (mt_state cM_k, buf', end_pos))
                ∈ (m_step_buffered M) ^^ kM
   and new_window:
     ∀ k<k_tm M. ae_window_invariant_general
           (mt_tape cM_k k) (mt_pos cM_k k)
           (end_pos k) (buf' k) (pos k) (le_tm M)
   and post_no_le:
     ∀ k. (pos k = 0
           ⟶ (∀ i. i < card (UNIV :: 'c set)
                 ⟶ mt_tape cM_k k (Suc i) ≠ le_tm M))
         ∧ (pos k = 1
```

```
            ⟶ (∀ i. i < 2 * card (UNIV :: 'c set)
                  ⟶ mt_tape cM_k k (Suc i) ≠ le_tm M))
        ∧ (pos k ≥ 2
            ⟶ (∀ i. i < 3 * card (UNIV :: 'c set)
                  ⟶ mt_tape cM_k k
                      ((pos k − 2) * card (UNIV :: 'c set)
                        + 1 + i)
                    ≠ le_tm M))
  and post_left_not_le_pos0:
     ∀ k<k_tm M. pos k = 0
          ⟶ fst (buf' k) ≠ LE_block (le_tm M)
  and bp_bound:
     ∀ k. card (UNIV :: 'c set) ≤ bp_linear (end_pos k) + kM
          ∧ bp_linear (end_pos k)
              < 2 * card (UNIV :: 'c set) + kM
  using ae_result_general by blast
have buffered_full:
   ((qM, buf_full, λk. (AE_Home, ofs k)),
     (mt_state cM_k, buf', end_pos)) ∈ m_steps_buffered M
  unfolding m_steps_buffered_def
  using coupled kM_le end_or_halt by auto
show thesis
proof (rule that[where q_out = mt_state cM_k
                  and buf' = buf' and end_pos = end_pos])
  show ((qM, buf_full, λk. (AE_Home, ofs k)),
        (mt_state cM_k, buf', end_pos)) ∈ m_steps_buffered M
    using buffered_full .
  show mt_state cM_k = mt_state cM_k by (rule refl)
  show ∀ k<k_tm M. ae_window_invariant_general
          (mt_tape cM_k k) (mt_pos cM_k k)
          (end_pos k) (buf' k) (pos k) (le_tm M)
    using new_window .
  show ∀ k. (pos k = 0
            ⟶ (∀ i. i < card (UNIV :: 'c set)
                  ⟶ mt_tape cM_k k (Suc i) ≠ le_tm M))
        ∧ (pos k = 1
            ⟶ (∀ i. i < 2 * card (UNIV :: 'c set)
                  ⟶ mt_tape cM_k k (Suc i) ≠ le_tm M))
        ∧ (pos k ≥ 2
            ⟶ (∀ i. i < 3 * card (UNIV :: 'c set)
                  ⟶ mt_tape cM_k k
                      ((pos k − 2) * card (UNIV :: 'c set)
                        + 1 + i)
                    ≠ le_tm M))
    using post_no_le .
  show ∀ k<k_tm M. pos k = 0
            ⟶ fst (buf' k) ≠ LE_block (le_tm M)
    using post_left_not_le_pos0 .
qed
```

qed

end
theory *AlphabetEnlargement_SS4*
imports *AlphabetEnlargement_ComputeCorrect*
begin

2.22 SS4 trace-existence and buffer characterisations

2.22.1 SS4→SS5 trace-existence

Trace-driven SS4→SS5 step existence. Takes the external M-trace prefix $(Config\ qM\ tsM\ nM,\ cM_k) \in delta_tm^{\frown\frown} kM$ with its no-pre-halt and end-or-full structure, and produces a SS4→SS5 step whose buffered-run choice aligns with the external trace's cM_k endpoint. Used by the chain proof in *ae_simulates_forward_stage* to keep the constructed M'-trace aligned with the externally-given M-trace; the conclusion exposes $q_out = mt_state\ cM_k$ at the SS5 boundary so the chain proof carries the state correspondence forward.

Shared substep-tuple construction for the SS4→SS5 trace-existence lemmas. Given a buffered *m_steps_buffered* result over the SS4-entry buffer (with the right slot holding the freshly-read tape value at the substrate head) and the arm-uniform preconditions $q \in Q_tm\ M$, non-halt, and gamma-block, produces an SS5-stage config c'' with both step relations (substep-specific and full alphabet-enlarge), the expected state shape, and unchanged substrate tape (SS4→SS5 is a no-write substep on the substrate). Consumed by the three *ae_step_ss4_ss5_exists_*_trace* variants: the arm-specific difference lives only in the buffer non-LE claim and the arm-specific post-trace window predicate, both of which the callers derive separately.

lemma *ae_step_ss4_ss5_construct_from_buffered*:
fixes $M :: ('q, 'a)\ mttm$
and $c' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae_stage)\ mt_config$
and $q :: 'q$
and $ofs :: nat \Rightarrow 'c$
and $buf :: nat \Rightarrow ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$
and $dest :: nat \Rightarrow ae_dest$
and $ts :: nat \Rightarrow nat \Rightarrow ('c \Rightarrow 'a)$
and $n :: nat \Rightarrow nat$
and $q_out :: 'q$
and $buf' :: nat \Rightarrow ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$
and $end_pos :: nat \Rightarrow 'c\ bp$
assumes vM: $valid_mttm\ M$
and c'_eq: $c' = Config_M\ (q, ofs, buf, dest, SS4)\ ts\ n$
and q_in: $q \in Q_tm\ M$
and $nhalt$: $q \neq t_tm\ M \wedge q \neq r_tm\ M$
and $gamma$: $ae_tape_in_gamma_block\ M\ c'$

```
    and bufv:   ae_buffer_in_gamma_block M c'
    and mst:    ((q, λk. (fst (buf k),
                          if k < k_tm M then fst (snd (buf k)) else ts k (n k),
                          ts k (n k)),
                  λk. (AE_Home, ofs k)),
                 (q_out, buf', end_pos)) ∈ m_steps_buffered M
  obtains c'' where
    (c', c'') ∈ mttm_step (ae_delta_ss4_ss5 M)
   and (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
   and mt_state c'' = (q_out,
        λk. if k < k_tm M then snd (end_pos k) else init_offset k,
        λk. if k < k_tm M then buf' k else init_buffer (le_tm M) k,
        λk. if k < k_tm M then fst (end_pos k) else init_dest k, SS5)
   and mt_tape c'' = ts
proof −
  let ?a = λk. ts k (n k)
  have a_support: ∀j≥k_tm M. ?a j = bl_block (bl_tm M)
    using gamma c'_eq unfolding ae_tape_in_gamma_block_def by auto
  let ?d = λk. if k < k_tm M
               then (if ?a k = LE_block (le_tm M) then dir.N else dir.L)
               else dir.N
  let ?ofs' = λk. if k < k_tm M then snd (end_pos k) else init_offset k
  let ?buf'g = λk. if k < k_tm M then buf' k else init_buffer (le_tm M) k
  let ?dest' = λk. if k < k_tm M then fst (end_pos k) else init_dest k
  let ?c'' = Config_M (q_out, ?ofs', ?buf'g, ?dest', SS5) ts
                      (λk. go_dir (?d k) (n k))
  have rel_in: ((q, ofs, buf, dest, SS4), ?a,
                (q_out, ?ofs', ?buf'g, ?dest', SS5), ?a, ?d)
                ∈ ae_delta_ss4_ss5 M
    unfolding ae_delta_ss4_ss5_def using q_in nhalt mst a_support by blast
  have gamma_a: ∀k. ?a k ∈ gamma_block (Γ_tm M)
    using gamma c'_eq unfolding ae_tape_in_gamma_block_def by simp
  have src_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                    (snd ((q, ofs, buf, dest, SS4)
                          :: 'q × ('a, 'c) ae_stage))
    using bufv c'_eq
    unfolding ae_buffer_in_gamma_block_def ae_valid_stage_def by simp
  have dst_valid: ae_valid_stage (Γ_tm M) (le_tm M) (k_tm M)
                    (snd ((q_out, ?ofs', ?buf'g, ?dest', SS5)
                          :: 'q × ('a, 'c) ae_stage))
    by (rule ae_delta_ss4_ss5_dest_valid[OF vM rel_in gamma_a src_valid])
  have aed_in: ((q, ofs, buf, dest, SS4), ?a,
                (q_out, ?ofs', ?buf'g, ?dest', SS5), ?a, ?d)
                ∈ alphabet_enlarge_delta M
    unfolding alphabet_enlarge_delta_def
    using rel_in gamma_a src_valid dst_valid by auto
  have ts_unchanged: (λk. (ts k)(n k := ?a k)) = ts
    by (rule ext) auto
  have step_sub_raw:
```

```
    (Config_M (q, ofs, buf, dest, SS4) ts n,
       Config_M (q_out, ?ofs', ?buf'g, ?dest', SS5)
         (λk. (ts k)(n k := ?a k))
         (λk. go_dir (?d k) (n k)))
       ∈ mttm_step (ae_delta_ss4_ss5 M)
    by (rule mttm_step.intros) (rule rel_in)
  have step_sub: (c', ?c'') ∈ mttm_step (ae_delta_ss4_ss5 M)
    using step_sub_raw c'_eq ts_unchanged by simp
  have step_full_raw:
    (Config_M (q, ofs, buf, dest, SS4) ts n,
       Config_M (q_out, ?ofs', ?buf'g, ?dest', SS5)
         (λk. (ts k)(n k := ?a k))
         (λk. go_dir (?d k) (n k)))
       ∈ mttm_step (alphabet_enlarge_delta M)
    by (rule mttm_step.intros) (rule aed_in)
  have step_full: (c', ?c'') ∈ mttm_step (alphabet_enlarge_delta M)
    using step_full_raw c'_eq ts_unchanged by simp
  show thesis
  proof (rule that[where c''=?c''])
    show (c', ?c'') ∈ mttm_step (ae_delta_ss4_ss5 M) using step_sub .
    show (c', ?c'') ∈ mttm_step (alphabet_enlarge_delta M)
      using step_full .
    show mt_state ?c'' = (q_out, ?ofs', ?buf'g, ?dest', SS5) by simp
    show mt_tape ?c'' = ts by simp
  qed
qed
```

Per-tape unified companion of *ae_step_ss4_ss5_exists_trace*, *_le0_trace*, and *_le1_trace*. A trace-driven SS4→SS5 step-existence wrapper. Takes a per-tape regime selector $pos = mt_pos\ c'\ k$, the per-tape hybrid window invariant, the per-tape regime-guarded *no_le*, and the $pos = 0$-conditional left-slot guard. Invokes *ae_m_steps_buffered_correct_trace_general* and then *ae_step_ss4_ss5_construct_from_buffered* to package the substep transition and the per-tape regime-aware buffer slot non-LE-ness in an *obtains*-style witness.

Per-tape regime-aware output: each tape's buffer slot non-LE-ness depends on its regime $pos = 0$ tapes have left and right slots non-LE (home is *LE_block* by le0 window invariant); $pos = 1$ tapes have home and right slots non-LE (left is *LE_block* by le1 window invariant); $pos \geq 2$ tapes have all three slots non-LE.

Consumer: *ae_simulates_forward_stage_general*'s body.

```
lemma ae_step_ss4_ss5_exists_general_trace:
  fixes M :: ('q, 'a) mttm
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
    and tsM :: nat ⇒ nat ⇒ 'a
    and nM :: nat ⇒ nat
```

and $pos :: nat \Rightarrow nat$
and $kM :: nat$
and $cM_k :: ('a, 'q)\ mt_config$
assumes vM: $valid_mttm\ M$
and lu: $le_unique\ M$
and inv: $ae_inv_ss4\ M\ c'$
and $gamma$: $ae_tape_in_gamma_block\ M\ c'$
and $bufv$: $ae_buffer_in_gamma_block\ M\ c'$
and q_neq_t: $fst\ (mt_state\ c') \neq t_tm\ M$
and q_neq_r: $fst\ (mt_state\ c') \neq r_tm\ M$
and $window$:
$\forall k{<}k_tm\ M.\ ae_window_invariant_general\ (tsM\ k)\ (nM\ k)$
$(case\ mt_state\ c'\ of\ (_, ofs, _, _, _)$
$\Rightarrow (AE_Home, ofs\ k))$
$(case\ mt_state\ c'\ of\ (_, _, buf, _, _)$
$\Rightarrow (fst\ (buf\ k), fst\ (snd\ (buf\ k)),$
$mt_tape\ c'\ k\ (mt_pos\ c'\ k)))$
$(pos\ k)\ (le_tm\ M)$
and no_le:
$\forall k.\ (pos\ k = 0$
$\longrightarrow (\forall i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow tsM\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k = 1$
$\longrightarrow (\forall i.\ i < 2 * card\ (UNIV :: 'c\ set)$
$\longrightarrow tsM\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k \geq 2$
$\longrightarrow (\forall i.\ i < 3 * card\ (UNIV :: 'c\ set)$
$\longrightarrow tsM\ k$
$((pos\ k - 2) * card\ (UNIV :: 'c\ set)$
$+ 1 + i)$
$\neq le_tm\ M))$
and $trace$:
$(Config_M\ (fst\ (mt_state\ c'))\ tsM\ nM, cM_k)$
$\in mttm_step\ (delta_tm\ M)\ \widehat{}\ kM$
and kM_le: $kM \leq card\ (UNIV :: 'c\ set)$
and end_or_halt:
$kM = card\ (UNIV :: 'c\ set)$
$\vee\ mt_state\ cM_k \in \{t_tm\ M, r_tm\ M\}$
and $left_not_le_pos0$:
$case\ mt_state\ c'\ of\ (_, _, buf, _, _) \Rightarrow$
$\forall k{<}k_tm\ M.\ pos\ k = 0 \longrightarrow fst\ (buf\ k) \neq LE_block\ (le_tm\ M)$
obtains c'' **where**
$(c', c'') \in mttm_step\ (ae_delta_ss4_ss5\ M)$
and $(c', c'') \in mttm_step\ (alphabet_enlarge_delta\ M)$
and $case\ mt_state\ c''\ of\ (q_out, _, _, _, _)$
$\Rightarrow q_out = mt_state\ cM_k$
and $case\ mt_state\ c''\ of\ (_, _, buf', _, _) \Rightarrow$
$\forall k{<}k_tm\ M.\ (pos\ k = 0$
$\longrightarrow fst\ (buf'\ k) \neq LE_block\ (le_tm\ M)$

$\wedge$ *snd* (*snd* (*buf'* *k*)) $\neq$ *LE_block* (*le_tm M*))
$\wedge$ (*pos k* = *1*
$\longrightarrow$ *fst* (*snd* (*buf'* *k*)) $\neq$ *LE_block* (*le_tm M*)
$\wedge$ *snd* (*snd* (*buf'* *k*)) $\neq$ *LE_block* (*le_tm M*))
$\wedge$ (*pos k* $\geq$ *2*
$\longrightarrow$ *fst* (*buf'* *k*) $\neq$ *LE_block* (*le_tm M*)
$\wedge$ *fst* (*snd* (*buf'* *k*)) $\neq$ *LE_block* (*le_tm M*)
$\wedge$ *snd* (*snd* (*buf'* *k*)) $\neq$ *LE_block* (*le_tm M*))
and *case mt_state c''* *of* (_, *ofs'*, *buf'*, *dest'*, _) $\Rightarrow$
$\forall$ *k*<*k_tm M*. *ae_window_invariant_general*
(*mt_tape cM_k k*) (*mt_pos cM_k k*)
(*dest'* *k*, *ofs'* *k*) (*buf'* *k*) (*pos k*) (*le_tm M*)
proof −
obtain *q ofs buf dest* **where**
state_eq: *mt_state c'* = (*q*, *ofs*, *buf*, *dest*, *SS4*)
and *q_in*: *q* $\in$ *Q_tm M*
using *inv* **unfolding** *ae_inv_ss4_def*
by (*cases mt_state c'*) *auto*
obtain *ts n* **where** *c'_eq*: *c'* = *Config*$_M$ (*q*, *ofs*, *buf*, *dest*, *SS4*) *ts n*
using *state_eq* **by** (*cases c'*) *auto*
let *?a* = λ*k*. *ts k* (*n k*)
let *?d* = λ*k*. *if ?a k* = *LE_block* (*le_tm M*) *then dir.N else dir.L*
let *?buf_full* = λ*k*. (*fst* (*buf k*),
if k < *k_tm M then fst* (*snd* (*buf k*)) *else ?a k*,
?a k)
have *nhalt*: *q* $\neq$ *t_tm M* $\wedge$ *q* $\neq$ *r_tm M*
using *q_neq_t q_neq_r state_eq* **by** *simp*
have *window'*:
$\forall$ *k*<*k_tm M*. *ae_window_invariant_general* (*tsM k*) (*nM k*) (*AE_Home*, *ofs k*)
(*?buf_full k*) (*pos k*) (*le_tm M*)
using *window c'_eq* **by** *simp*
have *pad_home'*:
$\forall$ *k*$\geq$*k_tm M*. *fst* (*snd* (*?buf_full k*)) = *bl_block* (*bl_tm M*)
proof (*intro allI impI*)
fix *k* **assume** *kge*: *k_tm M* $\leq$ *k*
have *ts k* (*n k*) = *bl_block* (*bl_tm M*)
using *gamma c'_eq kge* **unfolding** *ae_tape_in_gamma_block_def* **by** *simp*
thus *fst* (*snd* (*?buf_full k*)) = *bl_block* (*bl_tm M*) **using** *kge* **by** *simp*
qed
have *q_eq*: *fst* (*mt_state c'*) = *q* **using** *state_eq* **by** *simp*
have *trace'*: (*Config*$_M$ *q tsM nM*, *cM_k*)
$\in$ *mttm_step* (*delta_tm M*) $^\frown\!\!^\frown$ *kM*
using *trace q_eq* **by** *simp*
have *left_not_le_buf_full*:
$\forall$ *k*<*k_tm M*. *pos k* = *0* $\longrightarrow$ *fst* (*?buf_full k*) $\neq$ *LE_block* (*le_tm M*)
using *left_not_le_pos0 state_eq* **by** *simp*
obtain *q_out buf' end_pos* **where**
mst_in: ((*q*, *?buf_full*, λ*k*. (*AE_Home*, *ofs k*)),

```
      (q_out, buf', end_pos)) ∈ m_steps_buffered M
  and q_out_eq: q_out = mt_state cM_k
  and new_window:
    ∀ k<k_tm M. ae_window_invariant_general
        (mt_tape cM_k k) (mt_pos cM_k k)
        (end_pos k) (buf' k) (pos k) (le_tm M)
  and post_no_le:
    ∀ k. (pos k = 0
          ⟶ (∀ i. i < card (UNIV :: 'c set)
                ⟶ mt_tape cM_k k (Suc i) ≠ le_tm M))
       ∧ (pos k = 1
          ⟶ (∀ i. i < 2 * card (UNIV :: 'c set)
                ⟶ mt_tape cM_k k (Suc i) ≠ le_tm M))
       ∧ (pos k ≥ 2
          ⟶ (∀ i. i < 3 * card (UNIV :: 'c set)
                ⟶ mt_tape cM_k k
                    ((pos k − 2) * card (UNIV :: 'c set)
                      + 1 + i)
                  ≠ le_tm M))
  and post_left_not_le_pos0:
    ∀ k<k_tm M. pos k = 0
        ⟶ fst (buf' k) ≠ LE_block (le_tm M)
  by (rule ae_m_steps_buffered_correct_trace_general[OF vM lu q_in window'
                                                     pad_home' no_le trace'
                                                     kM_le end_or_halt
                                                     left_not_le_buf_full])
```

— Per-tape regime-aware derivation of buffer-slot non-LE-ness. Each branch reads off the relevant buf-linearisation slot via the corresponding sibling window invariant, combined with the *post_no_le* hypothesis at the slot's M-tape index.

```
  have c_ge_1: 1 ≤ card (UNIV :: 'c set)
    using c_idx_lt_card[of SOME x :: 'c. True] by simp
  let ?c = card (UNIV :: 'c set)
  have buf_per_tape_not_le:
    ∀ k<k_tm M. (pos k = 0
          ⟶ fst (buf' k) ≠ LE_block (le_tm M)
            ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M))
       ∧ (pos k = 1
          ⟶ fst (snd (buf' k)) ≠ LE_block (le_tm M)
            ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M))
       ∧ (pos k ≥ 2
          ⟶ fst (buf' k) ≠ LE_block (le_tm M)
            ∧ fst (snd (buf' k)) ≠ LE_block (le_tm M)
            ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M))
  proof (intro allI impI)
    fix k
    assume klt: k < k_tm M
    obtain l h r where buf_k: buf' k = (l, h, r)
      by (cases buf' k) auto
    show (pos k = 0
```

```
      ⟶ fst (buf' k) ≠ LE_block (le_tm M)
        ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M))
    ∧ (pos k = 1
        ⟶ fst (snd (buf' k)) ≠ LE_block (le_tm M)
          ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M))
    ∧ (pos k ≥ 2
        ⟶ fst (buf' k) ≠ LE_block (le_tm M)
          ∧ fst (snd (buf' k)) ≠ LE_block (le_tm M)
          ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M))
proof (intro conjI)
  — Branch pos = 0 ::: left and right slots non-LE.
  show pos k = 0
      ⟶ fst (buf' k) ≠ LE_block (le_tm M)
        ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M)
proof
  assume hpos: pos k = 0
  have win_k_le0:
      ae_window_invariant_le0 (mt_tape cM_k k) (mt_pos cM_k k)
          (end_pos k) (buf' k) (le_tm M)
    using new_window[rule_format, OF klt] hpos
    unfolding ae_window_invariant_general_def by blast
  have post_no_le_k:
      ∀ i. i < ?c ⟶ mt_tape cM_k k (Suc i) ≠ le_tm M
    using post_no_le hpos by blast
  have lt_0c: (0 :: nat) < ?c using c_ge_1 by simp
  have eq_1:
      mt_tape cM_k k (Suc 0) = buf_lin_at (buf' k) (2 * ?c + 0)
    using win_k_le0 lt_0c
    unfolding ae_window_invariant_le0_def by blast
  have not_lt_c: ¬ (2 * ?c) < ?c by simp
  have not_lt_2c: ¬ (2 * ?c) < 2 * ?c by simp
  have buf_at_2c:
      buf_lin_at (buf' k) (2 * ?c) = r ((enum_class.enum :: 'c list) ! 0)
    using buf_k not_lt_c not_lt_2c
    unfolding buf_lin_at_def Let_def by simp
  have post_at_1: mt_tape cM_k k (Suc 0) ≠ le_tm M
    using post_no_le_k lt_0c by blast
  have r_at_0_ne: r ((enum_class.enum :: 'c list) ! 0) ≠ le_tm M
    using eq_1 buf_at_2c post_at_1 by simp
  have r_ne_le_fn: r ≠ (λ_. le_tm M)
  proof
    assume r = (λ_. le_tm M)
    hence r ((enum_class.enum :: 'c list) ! 0) = le_tm M by simp
    with r_at_0_ne show False by simp
  qed
  have l_not_le: fst (buf' k) ≠ LE_block (le_tm M)
    using post_left_not_le_pos0 hpos klt by blast
  have r_not_le: snd (snd (buf' k)) ≠ LE_block (le_tm M)
    using buf_k r_ne_le_fn unfolding LE_block_def by simp
```

```
  show fst (buf' k) ≠ LE_block (le_tm M)
       ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M)
    using l_not_le r_not_le by blast
qed
— Branch pos = 1 ::: home and right slots non-LE.
show pos k = 1
       ⟶ fst (snd (buf' k)) ≠ LE_block (le_tm M)
         ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M)
proof
  assume hpos: pos k = 1
  have win_k_le1:
     ae_window_invariant_le1 (mt_tape cM_k k) (mt_pos cM_k k)
          (end_pos k) (buf' k) (le_tm M)
    using new_window[rule_format, OF klt] hpos
    unfolding ae_window_invariant_general_def by blast
  have post_no_le_k:
     ∀ i. i < 2 * ?c ⟶ mt_tape cM_k k (Suc i) ≠ le_tm M
    using post_no_le hpos by blast
  — h-slot: i = 0, buf index = c + 0 = c.
  have lt_0_2c: (0 :: nat) < 2 * ?c using c_ge_1 by simp
  have eq_1_h:
     mt_tape cM_k k (Suc 0) = buf_lin_at (buf' k) (?c + 0)
    using win_k_le1 lt_0_2c
    unfolding ae_window_invariant_le1_def by blast
  have not_lt_c: ¬ ?c < ?c by simp
  have c_lt_2c: ?c < 2 * ?c using c_ge_1 by simp
  have buf_at_c:
     buf_lin_at (buf' k) ?c = h ((enum_class.enum :: 'c list) ! 0)
    using buf_k not_lt_c c_lt_2c
    unfolding buf_lin_at_def Let_def by simp
  have post_at_1: mt_tape cM_k k (Suc 0) ≠ le_tm M
    using post_no_le_k lt_0_2c by blast
  have h_at_0_ne: h ((enum_class.enum :: 'c list) ! 0) ≠ le_tm M
    using eq_1_h buf_at_c post_at_1 by simp
  have h_ne_le_fn: h ≠ (λ_. le_tm M)
  proof
    assume h = (λ_. le_tm M)
    hence h ((enum_class.enum :: 'c list) ! 0) = le_tm M by simp
    with h_at_0_ne show False by simp
  qed
  have h_not_le: fst (snd (buf' k)) ≠ LE_block (le_tm M)
    using buf_k h_ne_le_fn unfolding LE_block_def by simp
  — r-slot: i = c, buf index = c + c = 2c.
  have lt_c_2c: ?c < 2 * ?c using c_ge_1 by simp
  have eq_cp1:
     mt_tape cM_k k (Suc ?c) = buf_lin_at (buf' k) (?c + ?c)
    using win_k_le1 lt_c_2c
    unfolding ae_window_invariant_le1_def by blast
  have cc_eq_2c: ?c + ?c = 2 * ?c by simp
```

```
  have not_lt_c2: ¬ (2 * ?c) < ?c by simp
  have not_lt_2c2: ¬ (2 * ?c) < 2 * ?c by simp
  have buf_at_2c:
      buf_lin_at (buf' k) (2 * ?c) = r ((enum_class.enum :: 'c list) ! 0)
    using buf_k not_lt_c2 not_lt_2c2
    unfolding buf_lin_at_def Let_def by simp
  have post_at_cp1: mt_tape cM_k k (Suc ?c) ≠ le_tm M
    using post_no_le_k lt_c_2c by blast
  have r_at_0_ne: r ((enum_class.enum :: 'c list) ! 0) ≠ le_tm M
    using eq_cp1 cc_eq_2c buf_at_2c post_at_cp1 by simp
  have r_ne_le_fn: r ≠ (λ_. le_tm M)
  proof
    assume r = (λ_. le_tm M)
    hence r ((enum_class.enum :: 'c list) ! 0) = le_tm M by simp
    with r_at_0_ne show False by simp
  qed
  have r_not_le: snd (snd (buf' k)) ≠ LE_block (le_tm M)
    using buf_k r_ne_le_fn unfolding LE_block_def by simp
  show fst (snd (buf' k)) ≠ LE_block (le_tm M)
       ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M)
    using h_not_le r_not_le by blast
qed
— Branch pos >= 2 ::: all three slots non-LE.
show pos k ≥ 2
       ⟶ fst (buf' k) ≠ LE_block (le_tm M)
         ∧ fst (snd (buf' k)) ≠ LE_block (le_tm M)
         ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M)
proof
  assume hpos: pos k ≥ 2
  let ?p_start = (pos k − 2) * ?c + 1
  have win_k_steady:
      ae_window_invariant (mt_tape cM_k k) (mt_pos cM_k k)
           (end_pos k) (buf' k) ?p_start
    using new_window[rule_format, OF klt] hpos
    unfolding ae_window_invariant_general_def by blast
  have post_no_le_k:
      ∀ i. i < 3 * ?c
           ⟶ mt_tape cM_k k (?p_start + i) ≠ le_tm M
    using post_no_le hpos by blast
  — l-slot: i = 0, buf index = 0.
  have lt_0: (0 :: nat) < 3 * ?c using c_ge_1 by simp
  have eq_0:
      mt_tape cM_k k (?p_start + 0) = buf_lin_at (buf' k) 0
    using win_k_steady lt_0
    unfolding ae_window_invariant_def by blast
  have buf_at_0:
      buf_lin_at (buf' k) 0 = l ((enum_class.enum :: 'c list) ! 0)
    using buf_k c_ge_1 unfolding buf_lin_at_def Let_def by simp
  have post_at_0: mt_tape cM_k k (?p_start + 0) ≠ le_tm M
```

```
  using post_no_le_k lt_0 by blast
have l_at_0_ne: l ((enum_class.enum :: 'c list) ! 0) ≠ le_tm M
  using eq_0 buf_at_0 post_at_0 by simp
have l_ne_le_fn: l ≠ (λ_. le_tm M)
proof
  assume l = (λ_. le_tm M)
  hence l ((enum_class.enum :: 'c list) ! 0) = le_tm M by simp
  with l_at_0_ne show False by simp
qed
have l_not_le: fst (buf' k) ≠ LE_block (le_tm M)
  using buf_k l_ne_le_fn unfolding LE_block_def by simp
— h-slot: i = c, buf index = c.
have lt_c: ?c < 3 * ?c using c_ge_1 by simp
have eq_c:
    mt_tape cM_k k (?p_start + ?c) = buf_lin_at (buf' k) ?c
  using win_k_steady lt_c
  unfolding ae_window_invariant_def by blast
have c_not_lt_c: ¬ ?c < ?c by simp
have c_lt_2c: ?c < 2 * ?c using c_ge_1 by simp
have buf_at_c:
    buf_lin_at (buf' k) ?c = h ((enum_class.enum :: 'c list) ! 0)
  using buf_k c_not_lt_c c_lt_2c
  unfolding buf_lin_at_def Let_def by simp
have post_at_c: mt_tape cM_k k (?p_start + ?c) ≠ le_tm M
  using post_no_le_k lt_c by blast
have h_at_0_ne: h ((enum_class.enum :: 'c list) ! 0) ≠ le_tm M
  using eq_c buf_at_c post_at_c by simp
have h_ne_le_fn: h ≠ (λ_. le_tm M)
proof
  assume h = (λ_. le_tm M)
  hence h ((enum_class.enum :: 'c list) ! 0) = le_tm M by simp
  with h_at_0_ne show False by simp
qed
have h_not_le: fst (snd (buf' k)) ≠ LE_block (le_tm M)
  using buf_k h_ne_le_fn unfolding LE_block_def by simp
— r-slot: i = 2c, buf index = 2c.
have lt_2c: 2 * ?c < 3 * ?c using c_ge_1 by simp
have eq_2c:
    mt_tape cM_k k (?p_start + 2 * ?c) = buf_lin_at (buf' k) (2 * ?c)
  using win_k_steady lt_2c
  unfolding ae_window_invariant_def by blast
have not_lt_c2: ¬ (2 * ?c) < ?c by simp
have not_lt_2c2: ¬ (2 * ?c) < 2 * ?c by simp
have buf_at_2c:
    buf_lin_at (buf' k) (2 * ?c) = r ((enum_class.enum :: 'c list) ! 0)
  using buf_k not_lt_c2 not_lt_2c2
  unfolding buf_lin_at_def Let_def by simp
have post_at_2c: mt_tape cM_k k (?p_start + 2 * ?c) ≠ le_tm M
  using post_no_le_k lt_2c by blast
```

```
    have r_at_0_ne: r ((enum_class.enum :: 'c list) ! 0) ≠ le_tm M
      using eq_2c buf_at_2c post_at_2c by simp
    have r_ne_le_fn: r ≠ (λ_. le_tm M)
    proof
      assume r = (λ_. le_tm M)
      hence r ((enum_class.enum :: 'c list) ! 0) = le_tm M by simp
      with r_at_0_ne show False by simp
    qed
    have r_not_le: snd (snd (buf' k)) ≠ LE_block (le_tm M)
      using buf_k r_ne_le_fn unfolding LE_block_def by simp
    show fst (buf' k) ≠ LE_block (le_tm M)
         ∧ fst (snd (buf' k)) ≠ LE_block (le_tm M)
         ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M)
      using l_not_le h_not_le r_not_le by blast
  qed
  qed
qed
— Substep tuple via the arm-uniform construction helper.
obtain c'' where
    step_sub: (c', c'') ∈ mttm_step (ae_delta_ss4_ss5 M)
  and step_full: (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
  and c''_state: mt_state c'' = (q_out,
        λk. if k < k_tm M then snd (end_pos k) else init_offset k,
        λk. if k < k_tm M then buf' k else init_buffer (le_tm M) k,
        λk. if k < k_tm M then fst (end_pos k) else init_dest k, SS5)
  and c''_tape: mt_tape c'' = ts
  by (rule ae_step_ss4_ss5_construct_from_buffered[OF vM c'_eq q_in nhalt
gamma bufv mst_in])
show thesis
proof (rule that[where c''=c''])
  show (c', c'') ∈ mttm_step (ae_delta_ss4_ss5 M) using step_sub .
  show (c', c'') ∈ mttm_step (alphabet_enlarge_delta M) using step_full .
  show case mt_state c'' of (q_out', _, _, _, _)
       ⇒ q_out' = mt_state cM_k
    unfolding c''_state using q_out_eq by simp
  show case mt_state c'' of (_, _, buf', _, _) ⇒
       ∀k<k_tm M. (pos k = 0
             ⟶ fst (buf' k) ≠ LE_block (le_tm M)
               ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M))
           ∧ (pos k = 1
               ⟶ fst (snd (buf' k)) ≠ LE_block (le_tm M)
                 ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M))
           ∧ (pos k ≥ 2
               ⟶ fst (buf' k) ≠ LE_block (le_tm M)
                 ∧ fst (snd (buf' k)) ≠ LE_block (le_tm M)
                 ∧ snd (snd (buf' k)) ≠ LE_block (le_tm M))
    unfolding c''_state using buf_per_tape_not_le by simp
  show case mt_state c'' of (_, ofs', buf', dest', _) ⇒
       ∀k<k_tm M. ae_window_invariant_general
```

```
      (mt_tape cM_k k) (mt_pos cM_k k)
      (dest' k, ofs' k) (buf' k) (pos k) (le_tm M)
  unfolding c''_state using new_window by simp
 qed
qed
```

2.22.2 Buffer characterisations at SS4 entry

Per-tape unified variant of the SS4-entry buffer characterisation, threading per-tape regime via the *home_class* hypothesis (parallel to the *home_classification* fact established in the body of *ae_simulates_forward_stage_general*). Each tape's home-cell content classifies it as either an LE-arm tape ($mt_pos\ c'\ k = 0$, home = *LE_block*) or a steady-arm tape ($mt_pos\ c'\ k \geq 1$, home $\neq$ *LE_block*); per-tape dispatch is needed because the *forward_stage_general* caller doesn't guarantee uniform regime across tapes.

Position closed-forms remain uniform thanks to *nat* arithmetic: $pos_c1 = n - 1$, $pos_c2 = n$, $pos_c3 = n + 1$. Each holds in both regimes because in the LE arm $n = 0$, so $0 - 1 = 0$, $go_dir\ N\ 0 = 0$, etc., agree with the steady-arm formulae evaluated at $n = 0$.

The SS4-entry buffer is per-tape: tapes at $pos = 0$ get (*bl_block*, *LE_block*, *snd* (*snd* (*buf k*))); tapes at $pos \geq 1$ get ($mt_tape\ c'\ k\ (pos - 1)$, $mt_tape\ c'\ k\ pos$, *snd* (*snd* (*buf k*))).

```
lemma ae_ss1_to_ss4_buffer_chars_general:
 fixes M :: ('q, 'a) mttm
   and c' c1 c2 c3 :: ('c :: enum ⇒ 'a,
                       'q × ('a, 'c) ae_stage) mt_config
   and qM' :: 'q
   and ofs :: nat ⇒ 'c
   and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
   and dest :: nat ⇒ ae_dest
 assumes c'_state:    mt_state c' = (qM', ofs, buf, dest, SS1)
    and step12:       (c', c1) ∈ mttm_step (ae_delta_ss1_ss2 M)
    and step23:       (c1, c2) ∈ mttm_step (ae_delta_ss2_ss3 M)
    and step34:       (c2, c3) ∈ mttm_step (ae_delta_ss3_ss4 M)
    and home_class:
        ∀k<k_tm M. (mt_pos c' k = 0
                    ⟶ mt_tape c' k (mt_pos c' k) = LE_block (le_tm M))
                 ∧ (mt_pos c' k ≥ 1
                    ⟶ mt_tape c' k (mt_pos c' k) ≠ LE_block (le_tm M))
 shows mt_state c3
       = (qM', ofs,
          (λk. if k < k_tm M
               then (if mt_pos c' k = 0
                     then (bl_block (bl_tm M),
                           LE_block (le_tm M),
                           snd (snd (buf k)))
```

```
                    else (mt_tape c' k (mt_pos c' k − 1),
                          mt_tape c' k (mt_pos c' k),
                          snd (snd (buf k))))
              else init_buffer (le_tm M) k),
        dest, SS4)
  and mt_tape c3 = mt_tape c'
  and ⋀k. k < k_tm M ⟹ mt_pos c3 k = mt_pos c' k + 1
proof −
 — Destructure c'; introduce ?a for the home read.
 obtain ts n where c'_eq:
    c' = Config_M (qM', ofs, buf, dest, SS1) ts n
  using c'_state by (cases c') auto
 let ?a = λk :: nat. ts k (n k)

 — Per-tape regime equivalence ?a k = LE_block ⟷ n k = 0 on active tapes (k
< k_tm M); past tape count the value-level builders freeze every per-tape field, so
no regime fact is needed there.
 have a_le_iff: ⋀k. k < k_tm M
                  ⟹ (?a k = LE_block (le_tm M)) ⟷ n k = 0
 proof −
  fix k assume klt: k < k_tm M
  have a_eq: ?a k = mt_tape c' k (mt_pos c' k) using c'_eq by simp
  have pos_eq: mt_pos c' k = n k using c'_eq by simp
  show (?a k = LE_block (le_tm M)) ⟷ n k = 0
  proof
   assume H: ?a k = LE_block (le_tm M)
   show n k = 0
   proof (rule ccontr)
    assume n k ≠ 0
    hence mt_pos c' k ≥ 1 using pos_eq by simp
    hence mt_tape c' k (mt_pos c' k) ≠ LE_block (le_tm M)
     using home_class klt by blast
    thus False using H a_eq by simp
   qed
  next
   assume n k = 0
   hence mt_pos c' k = 0 using pos_eq by simp
   hence mt_tape c' k (mt_pos c' k) = LE_block (le_tm M)
    using home_class klt by blast
   thus ?a k = LE_block (le_tm M) using a_eq by simp
  qed
 qed

 — SS1→SS2 unfolding. The value-level builder guards every per-tape field: on
active tapes (k < k_tm M) the buffer takes the home read and the head moves L
(steady) or N (LE-stage, n k = 0); past tape count the buffer freezes to init_buffer
(le_tm M) and the head does not move (direction N).
 let ?buf1 = λk. if k < k_tm M
                 then (fst (buf k), ?a k, snd (snd (buf k)))
```

```
                  else init_buffer (le_tm M) k
let ?n1 = λk :: nat. if k < k_tm M then n k − 1 else n k
have c1_struct:
   c1 = Config_M (qM′, ofs, ?buf1, dest, SS2) ts ?n1
proof −
  from step12 c′_eq obtain a buf′ d where
     in_delta: ((qM′, ofs, buf, dest, SS1), a,
                (qM′, ofs, buf′, dest, SS2), a, d)
                  ∈ ae_delta_ss1_ss2 M
    and a_eq: a = ?a
    and c1_eq: c1 = Config_M (qM′, ofs, buf′, dest, SS2)
                     (λk. (ts k)(n k := a k))
                     (λk. go_dir (d k) (n k))
    by (auto elim!: mttm_step.cases simp: ae_delta_ss1_ss2_def)
  have buf′_eq: buf′ = ?buf1
    using in_delta a_eq unfolding ae_delta_ss1_ss2_def by auto
  have d_eq: ⋀k. d k = (if k < k_tm M
                        then (if a k = LE_block (le_tm M)
                                then dir.N else dir.L)
                        else dir.N)
    using in_delta unfolding ae_delta_ss1_ss2_def by auto
  have ts_unchanged: ⋀k. (ts k)(n k := a k) = ts k
    using a_eq by (intro ext) auto
  have pos_uniform: ⋀k. go_dir (d k) (n k)
                      = (if k < k_tm M then n k − 1 else n k)
  proof −
    fix k
    show go_dir (d k) (n k) = (if k < k_tm M then n k − 1 else n k)
    proof (cases k < k_tm M)
      case True
      hence dk: d k = (if a k = LE_block (le_tm M)
                       then dir.N else dir.L)
        using d_eq by simp
      consider (le) a k = LE_block (le_tm M)
           | (nle) a k ≠ LE_block (le_tm M) by blast
      thus ?thesis
      proof cases
        case le
        hence d k = dir.N using dk by simp
        moreover have n k = 0 using a_le_iff[OF True] le a_eq by simp
        ultimately show ?thesis using True by simp
      next
        case nle
        hence d k = dir.L using dk by simp
        thus ?thesis using True by simp
      qed
    next
      case False
      hence d k = dir.N using d_eq by simp
```

```
    thus ?thesis using False by simp
  qed
 qed
 have pos_fun: (λk. go_dir (d k) (n k)) = ?n1
  using pos_uniform by (intro ext) auto
 have tape_fun: (λk. (ts k)(n k := a k)) = ts
  using ts_unchanged by (intro ext) auto
 show ?thesis
  using c1_eq buf'_eq pos_fun tape_fun by simp
qed
```

— SS2→SS3 unfolding. On active tapes the buffer becomes *?buf2*: home slot stays *?a k*; left slot is *bl_block* in the LE arm or *ts k (n k − 1)* in the steady arm. Position returns to *n k* in every regime (incl. past tape count, where the head never left).

```
let ?a' = λk :: nat. ts k (n k − 1)
let ?buf2 = λk. if k < k_tm M
                then (if ?a k = LE_block (le_tm M)
                        then bl_block (bl_tm M)
                        else ?a' k,
                      ?a k,
                      snd (snd (buf k)))
                else init_buffer (le_tm M) k
let ?n2 = λk :: nat. n k
have c2_struct:
   c2 = Config_M (qM', ofs, ?buf2, dest, SS3) ts ?n2
proof −
 from step23 c1_struct obtain a2 buf'' d' where
    in_delta2: ((qM', ofs, ?buf1, dest, SS2), a2,
                (qM', ofs, buf'', dest, SS3), a2, d')
                  ∈ ae_delta_ss2_ss3 M
   and a2_eq: a2 = (λk. ts k (?n1 k))
   and c2_eq: c2 = Config_M (qM', ofs, buf'', dest, SS3)
                    (λk. (ts k)(?n1 k := a2 k))
                    (λk. go_dir (d' k) (?n1 k))
   by (auto elim!: mttm_step.cases simp: ae_delta_ss2_ss3_def)
 have buf''_eq:
    buf'' = (λk. if k < k_tm M
                 then (let (l, h, r) = ?buf1 k in
                       (if h = LE_block (le_tm M)
                          then bl_block (bl_tm M) else a2 k,
                        h, r))
                 else init_buffer (le_tm M) k)
   and d'_eq: ⋀k. d' k = (if k < k_tm M
                          then (if fst (snd (?buf1 k)) = LE_block (le_tm M)
                                  then dir.N else dir.R)
                          else dir.N)
   using in_delta2 unfolding ae_delta_ss2_ss3_def by auto
 have h_buf1: ⋀k. k < k_tm M ⟹ fst (snd (?buf1 k)) = ?a k
```

```
  by simp
have buf''_simp: buf'' = ?buf2
proof (intro ext)
  fix k
  show buf'' k = ?buf2 k
  proof (cases k < k_tm M)
    case True
    have a2k: a2 k = ?a' k using a2_eq True by simp
    have b1: ?buf1 k = (fst (buf k), ?a k, snd (snd (buf k)))
      using True by simp
    have buf'' k = (let (l, h, r) = ?buf1 k in
                    (if h = LE_block (le_tm M)
                       then bl_block (bl_tm M) else a2 k, h, r))
      using buf''_eq True by simp
    also have ... = (if ?a k = LE_block (le_tm M)
                       then bl_block (bl_tm M) else a2 k,
                     ?a k, snd (snd (buf k)))
      using b1 by (simp add: case_prod_beta')
    also have ... = ?buf2 k using True a2k by simp
    finally show ?thesis .
  next
    case False
    have buf'' k = init_buffer (le_tm M) k
      using buf''_eq False by simp
    also have ... = ?buf2 k using False by simp
    finally show ?thesis .
  qed
qed
have ts_unchanged2: ⋀k. (ts k)(?n1 k := a2 k) = ts k
  using a2_eq by (intro ext) auto
have pos_uniform2: ⋀k. go_dir (d' k) (?n1 k) = n k
proof −
  fix k
  show go_dir (d' k) (?n1 k) = n k
  proof (cases k < k_tm M)
    case True
    hence d'k: d' k = (if ?a k = LE_block (le_tm M)
                         then dir.N else dir.R)
      using d'_eq h_buf1[OF True] by simp
    consider (le) ?a k = LE_block (le_tm M)
           | (nle) ?a k ≠ LE_block (le_tm M) by blast
    thus ?thesis
    proof cases
      case le
      hence d' k = dir.N using d'k by simp
      moreover have n k = 0 using a_le_iff[OF True] le by simp
      ultimately show ?thesis using True by simp
    next
      case nle
```

```
      hence d' k = dir.R using d'k by simp
      moreover have n k ≥ 1 using a_le_iff[OF True] nle by simp
      ultimately show ?thesis using True by simp
    qed
  next
    case False
    hence d' k = dir.N using d'_eq by simp
    thus ?thesis using False by simp
  qed
 qed
 have pos_fun2: (λk. go_dir (d' k) (?n1 k)) = ?n2
  using pos_uniform2 by (intro ext) auto
 have tape_fun2: (λk. (ts k)(?n1 k := a2 k)) = ts
  using ts_unchanged2 by (intro ext) auto
 show ?thesis
  using c2_eq buf''_simp pos_fun2 tape_fun2 by simp
qed
have c2_state: mt_state c2 = (qM', ofs, ?buf2, dest, SS3)
 using c2_struct by simp
have c2_tape: mt_tape c2 = ts using c2_struct by simp
have c2_pos: ⋀k. mt_pos c2 k = n k using c2_struct by simp

— SS3→SS4: active tapes move R (n k + 1), inactive tapes are frozen (N); no buffer update.
let ?n3 = λk :: nat. if k < k_tm M then n k + 1 else n k
have c3_struct:
   c3 = Config_M (qM', ofs, ?buf2, dest, SS4) ts ?n3
 using step34 c2_state c2_tape c2_pos
 by (auto simp: ae_delta_ss3_ss4_def
          elim!: mttm_step.cases
          intro!: ext mt_config.expand
          split: if_splits)

— Re-express the SS4-entry buffer on the c'-side (guarded: active tapes carry the regime-split triple, past tape count the frozen init_buffer).
have tape_c'_at: ⋀k. mt_tape c' k = ts k using c'_eq by simp
have pos_c'_at: ⋀k. mt_pos c' k = n k using c'_eq by simp
have buf2_to_concl: ?buf2 =
   (λk. if k < k_tm M
         then (if mt_pos c' k = 0
                then (bl_block (bl_tm M),
                      LE_block (le_tm M),
                      snd (snd (buf k)))
                else (mt_tape c' k (mt_pos c' k − 1),
                      mt_tape c' k (mt_pos c' k),
                      snd (snd (buf k))))
         else init_buffer (le_tm M) k)
proof (intro ext)
 fix k
```

```
  show ?buf2 k
       = (if k < k_tm M
            then (if mt_pos c' k = 0
                    then (bl_block (bl_tm M),
                          LE_block (le_tm M),
                          snd (snd (buf k)))
                    else (mt_tape c' k (mt_pos c' k − 1),
                          mt_tape c' k (mt_pos c' k),
                          snd (snd (buf k))))
            else init_buffer (le_tm M) k)
  proof (cases k < k_tm M)
    case True
    consider (le) ?a k = LE_block (le_tm M)
         | (nle) ?a k ≠ LE_block (le_tm M) by blast
    thus ?thesis
    proof cases
      case le
      have n0: n k = 0 using a_le_iff[OF True] le by simp
      have pos0: mt_pos c' k = 0 using pos_c'_at n0 by simp
      have lhs: ?buf2 k = (bl_block (bl_tm M), ?a k, snd (snd (buf k)))
        using le True by simp
      have a_is_le: ?a k = LE_block (le_tm M) using le .
      show ?thesis using lhs pos0 a_is_le True by simp
    next
      case nle
      have npos: n k ≥ 1 using a_le_iff[OF True] nle by simp
      have pos_ge1: mt_pos c' k ≥ 1 using pos_c'_at npos by simp
      have pos_ne0: mt_pos c' k ≠ 0 using pos_ge1 by simp
      have a'_tape: ?a' k = mt_tape c' k (mt_pos c' k − 1)
        using tape_c'_at pos_c'_at by simp
      have a_tape: ?a k = mt_tape c' k (mt_pos c' k)
        using tape_c'_at pos_c'_at by simp
      have lhs: ?buf2 k = (?a' k, ?a k, snd (snd (buf k)))
        using nle True by simp
      show ?thesis using lhs pos_ne0 a'_tape a_tape True by simp
    qed
  next
    case False
    thus ?thesis by simp
  qed
qed

show mt_state c3
       = (qM', ofs,
          (λk. if k < k_tm M
                 then (if mt_pos c' k = 0
                         then (bl_block (bl_tm M),
                               LE_block (le_tm M),
                               snd (snd (buf k)))
```

```
                    else (mt_tape c' k (mt_pos c' k − 1),
                          mt_tape c' k (mt_pos c' k),
                          snd (snd (buf k))))
                else init_buffer (le_tm M) k),
          dest, SS4)
  using c3_struct buf2_to_concl by simp
 show mt_tape c3 = mt_tape c'
  using c3_struct c'_eq by simp
 show ⋀k. k < k_tm M ⟹ mt_pos c3 k = mt_pos c' k + 1
 proof −
  fix k assume klt: k < k_tm M
  have mt_pos c3 k = (if k < k_tm M then n k + 1 else n k)
   using c3_struct by simp
  thus mt_pos c3 k = mt_pos c' k + 1
   using klt pos_c'_at by simp
 qed
qed
```

Per-tape unified window-from-correspondence helper. Establishes the per-tape *ae_window_invariant_general* from the *forward_stage_general* precondition shapes: simulation invariant (for *tape_corr* and *pos_corr*), LE-anchor (for tape cell 0 = LE), and the three-prong *no_le_per_tape* hypothesis (per-tape, per-regime non-LE guarantees on the M-window). The buffer triple is the SS4-entry per-tape buffer with the right slot replaced by the freshly-read block at *mt_pos c' k + 1* matching the buffer shape that *ae_delta_ss4_ss5* feeds into *m_step_buffered*.

The proof case-splits per-tape on *mt_pos c' k*'s regime (*0* / *1* / *≥ 2*) and dispatches each branch to the corresponding existing per-arm helper's conclusion shape.

```
lemma ae_ss4_window_from_correspondence_general:
 fixes M :: ('q, 'a) mttm
   and cM :: ('a, 'q) mt_config
   and c' :: ('c :: enum ⇒ 'a,
              'q × ('a, 'c) ae_stage) mt_config
   and qM' :: 'q
   and ofs :: nat ⇒ 'c
   and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
   and dest :: nat ⇒ ae_dest
 assumes sim:          ae_simulates M cM c'
     and c'_state:     mt_state c' = (qM', ofs, buf, dest, SS1)
     and le_anchor:    ∀k<k_tm M. mt_tape c' k 0 = LE_block (le_tm M)
     and no_le_per_tape:
         ∀k. (mt_pos c' k ≥ 2
               ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
                     ⟶ mt_tape cM k
                         ((mt_pos c' k − 2)
                           * card (UNIV :: 'c set) + 1 + i)
                       ≠ le_tm M))
```

```
                ∧ (mt_pos c' k = 1
                    ⟶ (∀ i. i < 2 * card (UNIV :: 'c set)
                            ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
                ∧ (mt_pos c' k = 0
                    ⟶ (∀ i. i < card (UNIV :: 'c set)
                            ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
  shows ∀ k<k_tm M. ae_window_invariant_general
              (mt_tape cM k) (mt_pos cM k)
              (AE_Home, ofs k)
              (if mt_pos c' k = 0
                 then (bl_block (bl_tm M),
                       LE_block (le_tm M),
                       mt_tape c' k 1)
               else if mt_pos c' k = 1
                 then (mt_tape c' k 0,
                       mt_tape c' k 1,
                       mt_tape c' k 2)
               else (mt_tape c' k (mt_pos c' k − 1),
                     mt_tape c' k (mt_pos c' k),
                     mt_tape c' k (mt_pos c' k + 1)))
              (mt_pos c' k)
              (le_tm M)
proof −
  let ?c = card (UNIV :: 'c set)
  let ?enum_c = enum_class.enum :: 'c list
  have c_eq_len: ?c = length ?enum_c
    using enum_class.UNIV_enum enum_class.enum_distinct
    by (metis distinct_card length_remdups_card_conv set_remdups)

  — Extract the simulation invariant's tape-correspondence and position-decoding
facts uniformly; both will be projected per-tape inside each regime branch.
  have tape_corr: ∀ k<k_tm M. ae_tape_correspondence (le_tm M)
                    (mt_tape cM k) (mt_tape c' k)
    using sim c'_state unfolding ae_simulates_def by auto
  have pos_corr: ∀ k<k_tm M. mt_pos cM k
                    = ae_decode_pos (mt_pos c' k) (ofs k)
    using sim c'_state unfolding ae_simulates_def by auto

  show ?thesis
  proof (intro allI impI)
    fix k
    assume k_lt: k < k_tm M
    have tape_corr_k: ae_tape_correspondence (le_tm M)
                    (mt_tape cM k) (mt_tape c' k)
      using tape_corr k_lt by blast
    have pos_corr_k: mt_pos cM k
                    = ae_decode_pos (mt_pos c' k) (ofs k)
      using pos_corr k_lt by simp
    have anchor_k: mt_tape c' k 0 = LE_block (le_tm M)
```

```
  using le_anchor k_lt by blast
have cM_zero: mt_tape cM k 0 = le_tm M
  using tape_corr_k unfolding ae_tape_correspondence_def by simp
have tc_app: ⋀s' i'. s' ≥ 1
          ⟹ mt_tape cM k ((s' − 1) * ?c + c_idx i' + 1)
             = mt_tape c' k s' i'
  using tape_corr_k unfolding ae_tape_correspondence_def by simp

consider (le0) mt_pos c' k = 0
       | (le1) mt_pos c' k = 1
       | (steady) mt_pos c' k ≥ 2
  by linarith
thus ae_window_invariant_general
         (mt_tape cM k) (mt_pos cM k)
         (AE_Home, ofs k)
         (if mt_pos c' k = 0
            then (bl_block (bl_tm M),
                  LE_block (le_tm M),
                  mt_tape c' k 1)
          else if mt_pos c' k = 1
            then (mt_tape c' k 0,
                  mt_tape c' k 1,
                  mt_tape c' k 2)
          else (mt_tape c' k (mt_pos c' k − 1),
                mt_tape c' k (mt_pos c' k),
                mt_tape c' k (mt_pos c' k + 1)))
         (mt_pos c' k)
         (le_tm M)
proof cases
  case le0
  have decode_k: mt_pos cM k = 0
  proof −
    have ae_decode_pos 0 (ofs k) = 0
      unfolding ae_decode_pos_def by simp
    thus ?thesis using pos_corr_k le0 by simp
  qed
  have part_1: fst (snd (bl_block (bl_tm M),
                         LE_block (le_tm M),
                         mt_tape c' k 1))
                = LE_block (le_tm M)
    by simp
  have part_2: mt_tape cM k 0 = le_tm M by (rule cM_zero)
  have part_3:
     (case fst (AE_Home, ofs k) of
        AE_Home ⇒ mt_pos cM k = 0
      | AE_Right ⇒ mt_pos cM k
                   = Suc (c_idx (snd (AE_Home, ofs k)))
      | AE_Left ⇒ False)
    using decode_k by simp
```

```
  have part_4: ∀ i. i < ?c
                 ⟶ mt_tape cM k (Suc i)
                    = buf_lin_at
                        (bl_block (bl_tm M),
                         LE_block (le_tm M),
                         mt_tape c' k 1) (2 * ?c + i)
  proof (intro allI impI)
    fix i :: nat
    assume i_lt: i < ?c
    let ?x = ?enum_c ! i
    have i_lt_len: i < length ?enum_c using i_lt c_eq_len by simp
    have cidx_x: c_idx ?x = i using c_idx_enum_nth[OF i_lt_len] .
    have addr_eq: Suc i = (1 - 1) * ?c + c_idx ?x + 1
      using cidx_x by simp
    have one_ge1: (1 :: nat) ≥ 1 by simp
    have mt_tape cM k (Suc i) = mt_tape c' k 1 ?x
      using tc_app[OF one_ge1, of ?x] addr_eq by simp
    also have ... = buf_lin_at
                      (bl_block (bl_tm M),
                       LE_block (le_tm M),
                       mt_tape c' k 1) (2 * ?c + i)
      using i_lt by (simp add: buf_lin_at_def Let_def)
    finally show mt_tape cM k (Suc i)
                 = buf_lin_at
                     (bl_block (bl_tm M),
                      LE_block (le_tm M),
                      mt_tape c' k 1) (2 * ?c + i) .
  qed
  have inv_le0:
     ae_window_invariant_le0 (mt_tape cM k) (mt_pos cM k)
        (AE_Home, ofs k)
        (bl_block (bl_tm M),
         LE_block (le_tm M),
         mt_tape c' k 1)
        (le_tm M)
    unfolding ae_window_invariant_le0_def
    using part_1 part_2 part_3 part_4 by blast
  show ?thesis
    unfolding ae_window_invariant_general_def
    using inv_le0 le0 by simp
next
  case le1
  have decode_k: mt_pos cM k = Suc (c_idx (ofs k))
  proof −
    have step: ae_decode_pos 1 (ofs k)
                 = (1 - 1) * ?c + c_idx (ofs k) + 1
      unfolding ae_decode_pos_def by simp
    hence ae_decode_pos 1 (ofs k) = Suc (c_idx (ofs k)) by simp
    thus ?thesis using pos_corr_k le1 by simp
```

```
qed
have part_1: fst (mt_tape c' k 0, mt_tape c' k 1, mt_tape c' k 2)
            = LE_block (le_tm M)
  using anchor_k by simp
have part_2: mt_tape cM k 0 = le_tm M by (rule cM_zero)
have part_3:
    (case fst (AE_Home, ofs k) of
       AE_Left  ⇒ snd (AE_Home, ofs k) = c_last
                  ∧ mt_pos cM k = 0
     | AE_Home  ⇒ mt_pos cM k
                  = Suc (c_idx (snd (AE_Home, ofs k)))
     | AE_Right ⇒ mt_pos cM k
                  = Suc (?c + c_idx (snd (AE_Home, ofs k))))
  using decode_k by simp
have part_4: ∀ i. i < 2 * ?c
               ⟶ mt_tape cM k (Suc i)
                 = buf_lin_at
                     (mt_tape c' k 0,
                      mt_tape c' k 1,
                      mt_tape c' k 2) (?c + i)
proof (intro allI impI)
  fix i :: nat
  assume i_lt: i < 2 * ?c
  consider (H) i < ?c | (R) ?c ≤ i ∧ i < 2 * ?c
    using i_lt by linarith
  thus mt_tape cM k (Suc i)
       = buf_lin_at
           (mt_tape c' k 0,
            mt_tape c' k 1,
            mt_tape c' k 2) (?c + i)
  proof cases
    case H
    let ?x = ?enum_c ! i
    have i_lt_len: i < length ?enum_c using H c_eq_len by simp
    have cidx_x: c_idx ?x = i using c_idx_enum_nth[OF i_lt_len] .
    have addr_eq: Suc i = (1 - 1) * ?c + c_idx ?x + 1
      using cidx_x by simp
    have one_ge1: (1 :: nat) ≥ 1 by simp
    have mt_tape cM k (Suc i) = mt_tape c' k 1 ?x
      using tc_app[OF one_ge1, of ?x] addr_eq by simp
    also have ... = buf_lin_at
                      (mt_tape c' k 0,
                       mt_tape c' k 1,
                       mt_tape c' k 2) (?c + i)
      using H by (simp add: buf_lin_at_def Let_def)
    finally show ?thesis .
  next
    case R
    let ?x = ?enum_c ! (i - ?c)
```

```
      have im_lt: i − ?c < ?c using R by linarith
      have im_lt_len: i − ?c < length ?enum_c
        using im_lt c_eq_len by simp
      have cidx_x: c_idx ?x = i − ?c
        using c_idx_enum_nth[OF im_lt_len] .
      have shift: Suc i = (2 − 1) * ?c + (i − ?c) + 1
        using R by simp
      have addr_eq: Suc i = (2 − 1) * ?c + c_idx ?x + 1
        using shift cidx_x by simp
      have two_ge1: (2 :: nat) ≥ 1 by simp
      have mt_tape cM k (Suc i) = mt_tape c' k 2 ?x
        using tc_app[OF two_ge1, of ?x] addr_eq by simp
      also have ... = buf_lin_at
                        (mt_tape c' k 0,
                         mt_tape c' k 1,
                         mt_tape c' k 2) (?c + i)
        using R by (simp add: buf_lin_at_def Let_def)
      finally show ?thesis .
    qed
  qed
  have inv_le1:
      ae_window_invariant_le1 (mt_tape cM k) (mt_pos cM k)
        (AE_Home, ofs k)
        (mt_tape c' k 0,
         mt_tape c' k 1,
         mt_tape c' k 2)
        (le_tm M)
    unfolding ae_window_invariant_le1_def
    using part_1 part_2 part_3 part_4 by blast
  show ?thesis
    unfolding ae_window_invariant_general_def
    using inv_le1 le1 by simp
next
  case steady
  let ?p_start = (mt_pos c' k − 2) * ?c + 1
  have s_ge2: mt_pos c' k ≥ 2 using steady .
  have s_ge1: mt_pos c' k ≥ 1 using s_ge2 by simp
  have decode_k: mt_pos cM k
                   = (mt_pos c' k − 1) * ?c + c_idx (ofs k) + 1
  proof −
    have mt_pos c' k ≠ 0 using s_ge1 by simp
    thus ?thesis
      using pos_corr_k unfolding ae_decode_pos_def by simp
  qed
  have part_a: ?p_start ≥ 1 by simp
  have part_b: mt_pos cM k = ?p_start + bp_linear (AE_Home, ofs k)
  proof −
    have bp: bp_linear (AE_Home, ofs k) = ?c + c_idx (ofs k)
      unfolding bp_linear_def by simp
```

```
  have step_sub: (mt_pos c' k − 2) + 1 = mt_pos c' k − 1
    using s_ge2 by arith
  have ?p_start + bp_linear (AE_Home, ofs k)
        = (mt_pos c' k − 2) * ?c + 1 + (?c + c_idx (ofs k))
    using bp by simp
  also have ... = (mt_pos c' k − 2) * ?c + ?c
                  + c_idx (ofs k) + 1
    by simp
  also have ... = ((mt_pos c' k − 2) + 1) * ?c
                  + c_idx (ofs k) + 1
    by (simp add: algebra_simps)
  also have ... = (mt_pos c' k − 1) * ?c + c_idx (ofs k) + 1
    using step_sub by simp
  also have ... = mt_pos cM k using decode_k by simp
  finally show ?thesis by simp
qed
have part_c: ∀ i. i < 3 * ?c
               ⟶ mt_tape cM k (?p_start + i)
                  = buf_lin_at
                      (mt_tape c' k (mt_pos c' k − 1),
                       mt_tape c' k (mt_pos c' k),
                       mt_tape c' k (mt_pos c' k + 1)) i
proof (intro allI impI)
  fix i :: nat
  assume i_lt: i < 3 * ?c
  consider (L) i < ?c | (H) ?c ≤ i ∧ i < 2 * ?c
                      | (R) 2 * ?c ≤ i ∧ i < 3 * ?c
    using i_lt by linarith
  thus mt_tape cM k (?p_start + i)
        = buf_lin_at
            (mt_tape c' k (mt_pos c' k − 1),
             mt_tape c' k (mt_pos c' k),
             mt_tape c' k (mt_pos c' k + 1)) i
  proof cases
    case L
    let ?x = ?enum_c ! i
    have i_lt_len: i < length ?enum_c using L c_eq_len by simp
    have cidx_x: c_idx ?x = i using c_idx_enum_nth[OF i_lt_len] .
    have sub_lhs: (mt_pos c' k − 1) − 1 = mt_pos c' k − 2
      using s_ge2 by arith
    have addr_eq: ?p_start + i
                    = ((mt_pos c' k − 1) − 1) * ?c
                      + c_idx ?x + 1
      using cidx_x sub_lhs by simp
    have s_minus_1_ge1: mt_pos c' k − 1 ≥ 1 using s_ge2 by arith
    have mt_tape cM k (?p_start + i)
          = mt_tape c' k (mt_pos c' k − 1) ?x
      using tc_app[OF s_minus_1_ge1, of ?x] addr_eq by simp
    also have ... = buf_lin_at
```

```
                (mt_tape c' k (mt_pos c' k − 1),
                 mt_tape c' k (mt_pos c' k),
                 mt_tape c' k (mt_pos c' k + 1)) i
    using L by (simp add: buf_lin_at_def Let_def)
  finally show ?thesis .
next
  case H
  let ?x = ?enum_c ! (i − ?c)
  have im_lt: i − ?c < ?c using H by linarith
  have im_lt_len: i − ?c < length ?enum_c
    using im_lt c_eq_len by simp
  have cidx_x: c_idx ?x = i − ?c
    using c_idx_enum_nth[OF im_lt_len] .
  have shift: (mt_pos c' k − 2) * ?c + 1 + i
              = (mt_pos c' k − 1) * ?c + (i − ?c) + 1
  proof −
    have (mt_pos c' k − 1) * ?c
          = (mt_pos c' k − 2) * ?c + ?c
      using s_ge2 by (simp add: algebra_simps)
    thus ?thesis using H by simp
  qed
  have addr_eq: ?p_start + i
                = (mt_pos c' k − 1) * ?c + c_idx ?x + 1
    using shift cidx_x by simp
  have mt_tape cM k (?p_start + i)
       = mt_tape c' k (mt_pos c' k) ?x
    using tc_app[OF s_ge1, of ?x] addr_eq by simp
  also have ... = buf_lin_at
                (mt_tape c' k (mt_pos c' k − 1),
                 mt_tape c' k (mt_pos c' k),
                 mt_tape c' k (mt_pos c' k + 1)) i
    using H by (simp add: buf_lin_at_def Let_def)
  finally show ?thesis .
next
  case R
  let ?x = ?enum_c ! (i − 2 * ?c)
  have ir_lt: i − 2 * ?c < ?c using R by linarith
  have ir_lt_len: i − 2 * ?c < length ?enum_c
    using ir_lt c_eq_len by simp
  have cidx_x: c_idx ?x = i − 2 * ?c
    using c_idx_enum_nth[OF ir_lt_len] .
  have shift: (mt_pos c' k − 2) * ?c + 1 + i
              = mt_pos c' k * ?c + (i − 2 * ?c) + 1
  proof −
    have mt_pos c' k * ?c
          = (mt_pos c' k − 2) * ?c + 2 * ?c
      using s_ge2 by (simp add: algebra_simps)
    thus ?thesis using R by simp
  qed
```

```
      have splus_minus: (mt_pos c' k + 1) - 1 = mt_pos c' k
        by simp
      have addr_eq: ?p_start + i
                    = ((mt_pos c' k + 1) - 1) * ?c
                      + c_idx ?x + 1
        using shift cidx_x splus_minus by simp
      have splus_ge1: mt_pos c' k + 1 ≥ 1 by simp
      have mt_tape cM k (?p_start + i)
             = mt_tape c' k (mt_pos c' k + 1) ?x
        using tc_app[OF splus_ge1, of ?x] addr_eq by simp
      also have ... = buf_lin_at
                      (mt_tape c' k (mt_pos c' k - 1),
                       mt_tape c' k (mt_pos c' k),
                       mt_tape c' k (mt_pos c' k + 1)) i
        using R by (simp add: buf_lin_at_def Let_def)
      finally show ?thesis .
    qed
  qed
  have inv_steady:
    ae_window_invariant (mt_tape cM k) (mt_pos cM k)
      (AE_Home, ofs k)
      (mt_tape c' k (mt_pos c' k - 1),
       mt_tape c' k (mt_pos c' k),
       mt_tape c' k (mt_pos c' k + 1))
      ?p_start
    unfolding ae_window_invariant_def
    using part_a part_b part_c by blast
  show ?thesis
    unfolding ae_window_invariant_general_def
    using inv_steady steady by simp
 qed
qed
qed

end
theory AlphabetEnlargement_OutputWF
 imports AlphabetEnlargement_SS4
begin
```

2.23 Home classification and output well-formedness

2.23.1 Home classification, stage unpack, load chain

Per-tape home-cell LE-classification at SS1. Given the M↔M' tape correspondence (extracted from *ae_simulates*), the LE anchor on tape 0, and the no-LE-window invariant on *cM*'s tape, the home block of *c'*'s tape *k* equals *LE_block* (*le_tm M*) iff *mt_pos c' k = 0*. Direction-agnostic; consumed by both arms of the SS4-stage setup (the forward arm uses it to drive *ae_ss1_to_ss4_buffer_chars_general*'s *home_class* hypothesis; the

reverse arm uses it for the symmetric extraction).

```
lemma ae_home_classification:
  fixes M :: ('q, 'a) mttm
    and cM :: ('a, 'q) mt_config
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
  assumes le_anchor:  ∀k<k_tm M. mt_tape c' k 0 = LE_block (le_tm M)
      and no_le_per_tape:
          ∀k. (mt_pos c' k ≥ 2
                ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
                       ⟶ mt_tape cM k
                           ((mt_pos c' k − 2)
                              * card (UNIV :: 'c set) + 1 + i)
                         ≠ le_tm M))
             ∧ (mt_pos c' k = 1
                ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
                       ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
             ∧ (mt_pos c' k = 0
                ⟶ (∀i. i < card (UNIV :: 'c set)
                       ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
      and tape_corr:
          ∀k<k_tm M. ae_tape_correspondence (le_tm M)
                (mt_tape cM k) (mt_tape c' k)
    shows ∀k<k_tm M. (mt_pos c' k = 0
                ⟶ mt_tape c' k (mt_pos c' k) = LE_block (le_tm M))
            ∧ (mt_pos c' k ≥ 1
                ⟶ mt_tape c' k (mt_pos c' k) ≠ LE_block (le_tm M))
proof (intro allI impI conjI)
  fix k :: nat
  assume klt: k < k_tm M and pos0: mt_pos c' k = 0
  have mt_tape c' k 0 = LE_block (le_tm M) using le_anchor klt by blast
  thus mt_tape c' k (mt_pos c' k) = LE_block (le_tm M)
    using pos0 by simp
next
  fix k :: nat
  assume klt: k < k_tm M and pos_ge1: 1 ≤ mt_pos c' k
  have c_ge_1: 1 ≤ card (UNIV :: 'c set)
    using c_idx_lt_card[of SOME x :: 'c. True] by simp
  have c_eq_len:
      card (UNIV :: 'c set) = length (enum_class.enum :: 'c list)
    using enum_class.UNIV_enum enum_class.enum_distinct
    by (metis distinct_card length_remdups_card_conv set_remdups)
  let ?x = (enum_class.enum :: 'c list) ! 0
  have zero_lt_len: 0 < length (enum_class.enum :: 'c list)
    using c_ge_1 c_eq_len by linarith
  have x_idx0: c_idx ?x = 0
    using c_idx_enum_nth[OF zero_lt_len] .
  have tc_k: ae_tape_correspondence (le_tm M)
                (mt_tape cM k) (mt_tape c' k)
```

```
    using tape_corr klt by blast
  have addr_lit:
      mt_tape cM k
        ((mt_pos c' k − 1) ∗ card (UNIV :: 'c set) + c_idx ?x + 1)
       = mt_tape c' k (mt_pos c' k) ?x
    using tc_k[unfolded ae_tape_correspondence_def, THEN conjunct2,
          rule_format, OF pos_ge1, of ?x] .
  have addr_via_tc:
      mt_tape cM k ((mt_pos c' k − 1) ∗ card (UNIV :: 'c set) + 1)
       = mt_tape c' k (mt_pos c' k) ?x
    using addr_lit x_idx0 by simp
  consider (eq1) mt_pos c' k = 1 | (ge2) mt_pos c' k ≥ 2
    using pos_ge1 by linarith
  hence mt_tape c' k (mt_pos c' k) ?x ≠ le_tm M
  proof cases
    case eq1
    have idx_lt: (0 :: nat) < 2 ∗ card (UNIV :: 'c set)
      using c_ge_1 by simp
    have addr_at_1:
        mt_tape cM k (Suc 0) = mt_tape c' k (mt_pos c' k) ?x
      using addr_via_tc eq1 by simp
    have nle_at_1: mt_tape cM k (Suc 0) ≠ le_tm M
      using no_le_per_tape eq1 idx_lt by blast
    thus ?thesis using addr_at_1 by simp
  next
    case ge2
    let ?j = card (UNIV :: 'c set)
    have j_lt_3c: ?j < 3 ∗ card (UNIV :: 'c set)
      using c_ge_1 by simp
    have addr_eq:
        (mt_pos c' k − 1) ∗ card (UNIV :: 'c set) + 1
         = (mt_pos c' k − 2) ∗ card (UNIV :: 'c set) + 1 + ?j
      using ge2 by (simp add: algebra_simps diff_mult_distrib)
    have nle_at_addr:
        mt_tape cM k ((mt_pos c' k − 2) ∗ card (UNIV :: 'c set) + 1 + ?j)
         ≠ le_tm M
      using no_le_per_tape ge2 j_lt_3c by blast
    hence mt_tape cM k ((mt_pos c' k − 1) ∗ card (UNIV :: 'c set) + 1)
          ≠ le_tm M
      using addr_eq by simp
    thus ?thesis using addr_via_tc by simp
  qed
  thus mt_tape c' k (mt_pos c' k) ≠ LE_block (le_tm M)
    unfolding LE_block_def by auto
qed
```

Auxiliary: unpack the *ae_simulates* invariant at SS1 entry into the structural data forward-stage proofs need. Extracts M''s state-tuple shape (with $idx = SS1$ pinned by the M-state not-halted hypothesis), the M-state equal-

ity, the M↔M' tape and position correspondences, the gamma-block sideband, the SS1 substrate invariant, and the (vacuous-at-SS1) position-link. All three forward-stage variants steady-state and the two head-LE arms *le0*, *le1* open with this same unpacking; factoring it out keeps the variants from cloning 47 lines of preamble apiece.

```
lemma ae_forward_stage_unpack_sim:
  fixes M :: ('q, 'a) mttm
    and cM :: ('a, 'q) mt_config
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
  assumes sim:       ae_simulates M cM c'
      and qM_in_Q:   mt_state cM ∈ Q_tm M
      and q_neq_t:   mt_state cM ≠ t_tm M
      and q_neq_r:   mt_state cM ≠ r_tm M
  obtains qM' ofs buf dest where
      mt_state c' = (qM', ofs, buf, dest, SS1)
    and mt_state cM = qM'
    and qM' ∈ Q_tm M
    and qM' ≠ t_tm M
    and qM' ≠ r_tm M
    and ∀k<k_tm M. ae_tape_correspondence (le_tm M)
          (mt_tape cM k) (mt_tape c' k)
    and ∀k<k_tm M. mt_pos cM k = ae_decode_pos (mt_pos c' k) (ofs k)
    and ae_tape_in_gamma_block M c'
    and ae_inv_ss1 M c'
    and ae_position_link M c'
proof −
  obtain qM' ofs buf dest idx_init where
      state_comp: mt_state c' = (qM', ofs, buf, dest, idx_init)
    by (cases mt_state c')
  have sim_body:
      ((idx_init = SS1 ∧ qM' ∉ {t_tm M, r_tm M})
         ∨ (qM' ∈ {t_tm M, r_tm M}
              ∧ (ofs, buf, dest, idx_init) = init_stage (le_tm M)))
       ∧ mt_state cM = qM'
       ∧ (∀k<k_tm M. ae_tape_correspondence (le_tm M)
              (mt_tape cM k) (mt_tape c' k))
       ∧ (idx_init = SS1
            ⟶ (∀k<k_tm M. mt_pos cM k = ae_decode_pos (mt_pos c' k) (ofs
k)))
       ∧ ae_tape_in_gamma_block M c'
    using sim state_comp unfolding ae_simulates_def by simp
  have qM_eq: mt_state cM = qM' using sim_body by simp
  have qM'_neq_t: qM' ≠ t_tm M using qM_eq q_neq_t by simp
  have qM'_neq_r: qM' ≠ r_tm M using qM_eq q_neq_r by simp
  have idx_is_ss1: idx_init = SS1
    using sim_body qM'_neq_t qM'_neq_r by auto
  have c'_state: mt_state c' = (qM', ofs, buf, dest, SS1)
```

```
  using state_comp idx_is_ss1 by simp
 have tape_corr:
    ∀k<k_tm M. ae_tape_correspondence (le_tm M)
          (mt_tape cM k) (mt_tape c' k)
  using sim_body by simp
 have pos_corr:
    ∀k<k_tm M. mt_pos cM k = ae_decode_pos (mt_pos c' k) (ofs k)
  using sim_body idx_is_ss1 by simp
 have gamma_c': ae_tape_in_gamma_block M c'
  using sim_body by simp
 have qM'_in_Q: qM' ∈ Q_tm M
  using qM_in_Q qM_eq by simp
 have inv_ss1: ae_inv_ss1 M c'
  unfolding ae_inv_ss1_def using c'_state qM'_in_Q by simp
 have pos_link_c': ae_position_link M c'
  using c'_state unfolding ae_position_link_def by simp
 show ?thesis
  by (rule that[OF c'_state qM_eq qM'_in_Q qM'_neq_t qM'_neq_r
                tape_corr pos_corr gamma_c' inv_ss1 pos_link_c'])
qed
```

Auxiliary: the three buffer-load substeps SS1 → SS2 → SS3 → SS4 as a single chained existence lemma. Threads invariant preservation, gamma preservation (tape + buffer), M-state propagation, and position-link propagation across the three substeps; produces the SS4-entry configuration *c3* with all the structural facts forward-stage proofs need before reaching the c-fold compute substep.

The block is arm-uniform: it depends only on the SS1 invariant and the gamma side-band, not on the head's block position or the no-LE window structure. All three forward-stage variants steady-state and the head-LE arms *le0*, *le1* share this load chain verbatim; factoring it out keeps the variants from cloning 122 lines of preservation plumbing apiece.

Position-link propagation through the load chain rests on the pre-state's idx being SS1, SS2, SS3 respectively at each substep, which keeps all three aux-void lemmas (*ae_pos_link_aux_void_ss<N>_ss<M>*) vacuously satisfied.

```
lemma ae_forward_stage_load_chain:
 fixes M :: ('q, 'a) mttm
   and c' :: ('c :: enum ⇒ 'a,
              'q × ('a, 'c) ae_stage) mt_config
   and qM' :: 'q
   and ofs :: nat ⇒ 'c
   and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
   and dest :: nat ⇒ ae_dest
 assumes vM:          valid_mttm M
     and c'_state:    mt_state c' = (qM', ofs, buf, dest, SS1)
     and inv_ss1:     ae_inv_ss1 M c'
     and gamma_c':    ae_tape_in_gamma_block M c'
```

```
    and buf_gamma:   ae_buffer_in_gamma_block M c'
    and pos_link_c': ae_position_link M c'
    and qM'_neq_t:   qM' ≠ t_tm M
    and qM'_neq_r:   qM' ≠ r_tm M
  obtains c1 c2 c3 where
    (c', c1) ∈ mttm_step (ae_delta_ss1_ss2 M)
   and (c', c1) ∈ mttm_step (alphabet_enlarge_delta M)
   and (c1, c2) ∈ mttm_step (ae_delta_ss2_ss3 M)
   and (c1, c2) ∈ mttm_step (alphabet_enlarge_delta M)
   and (c2, c3) ∈ mttm_step (ae_delta_ss3_ss4 M)
   and (c2, c3) ∈ mttm_step (alphabet_enlarge_delta M)
   and ae_inv_ss4 M c3
   and ae_tape_in_gamma_block M c3
   and ae_buffer_in_gamma_block M c3
   and fst (mt_state c3) = qM'
   and ae_position_link M c3
proof −
  have c'_fst_neq_t: fst (mt_state c') ≠ t_tm M
    using c'_state qM'_neq_t by simp
  have c'_fst_neq_r: fst (mt_state c') ≠ r_tm M
    using c'_state qM'_neq_r by simp
  — Substep 1: SS1→SS2.
  obtain c1 where
    step1_sub: (c', c1) ∈ mttm_step (ae_delta_ss1_ss2 M)
   and step1: (c', c1) ∈ mttm_step (alphabet_enlarge_delta M)
    using ae_step_ss1_ss2_exists[OF inv_ss1 gamma_c' buf_gamma
                                    c'_fst_neq_t c'_fst_neq_r] by blast
  have inv_ss2: ae_inv_ss2 M c1
    using ae_step_ss1_ss2_invariant[OF vM inv_ss1 step1_sub] .
  have gamma_c1: ae_tape_in_gamma_block M c1
    using ae_step_alphabet_enlarge_gamma_preserve[OF gamma_c' step1] .
  have buf_gamma_c1: ae_buffer_in_gamma_block M c1
   using ae_step_alphabet_enlarge_buffer_gamma_preserve[OF vM buf_gamma
                                                          gamma_c' step1] .
  have c1_fst_eq: fst (mt_state c1) = qM'
    using step1_sub c'_state
    by (auto simp: ae_delta_ss1_ss2_def elim: mttm_step.cases)
  have c1_fst_neq_t: fst (mt_state c1) ≠ t_tm M
    using c1_fst_eq qM'_neq_t by simp
  have c1_fst_neq_r: fst (mt_state c1) ≠ r_tm M
    using c1_fst_eq qM'_neq_r by simp
  have c'_idx_ss1: snd (snd (snd (snd (mt_state c')))) = SS1
    using c'_state by simp
  have aux_45_at_c': (c', c1) ∈ mttm_step (ae_delta_ss4_ss5 M)
                      ⟹ ae_position_link M c1
    using ae_pos_link_aux_void_ss4_ss5[where M = M and c_pre = c' and
c_post = c1]
          c'_idx_ss1 by simp
  have aux_56_at_c': (c', c1) ∈ mttm_step (ae_delta_ss5_ss6 M)
```

$\Longrightarrow$ *ae_position_link M c1*
using *ae_pos_link_aux_void_ss5_ss6*[**where** $M = M$ **and** *c_pre* $= c'$ **and** *c_post* $=$ *c1*]
c'_idx_ss1 **by** *simp*
have *aux_78_at_c'*: $(c', c1) \in$ *mttm_step* (*ae_delta_ss7_ss8 M*)
$\Longrightarrow$ *ae_position_link M c1*
using *ae_pos_link_aux_void_ss7_ss8*[**where** $M = M$ **and** *c_pre* $= c'$ **and** *c_post* $=$ *c1*]
c'_idx_ss1 **by** *simp*
have *pos_link_c1*: *ae_position_link M c1*
using *ae_step_alphabet_enlarge_position_link_preserve*[*OF pos_link_c' step1 aux_45_at_c' aux_56_at_c' aux_78_at_c'*] **.**
— Substep 2: SS2→SS3.
obtain *c2* **where**
step2_sub: $(c1, c2) \in$ *mttm_step* (*ae_delta_ss2_ss3 M*)
and *step2*: $(c1, c2) \in$ *mttm_step* (*alphabet_enlarge_delta M*)
using *ae_step_ss2_ss3_exists*[*OF vM inv_ss2 gamma_c1 buf_gamma_c1 c1_fst_neq_t c1_fst_neq_r*] **by** *blast*
have *inv_ss3*: *ae_inv_ss3 M c2*
using *ae_step_ss2_ss3_invariant*[*OF vM inv_ss2 step2_sub*] **.**
have *gamma_c2*: *ae_tape_in_gamma_block M c2*
using *ae_step_alphabet_enlarge_gamma_preserve*[*OF gamma_c1 step2*] **.**
have *buf_gamma_c2*: *ae_buffer_in_gamma_block M c2*
using *ae_step_alphabet_enlarge_buffer_gamma_preserve*[*OF vM buf_gamma_c1 gamma_c1 step2*] **.**
have *c2_fst_eq*: *fst* (*mt_state c2*) $= qM'$
using *step2_sub c1_fst_eq*
by (*cases mt_state c1*)
(*auto simp*: *ae_delta_ss2_ss3_def elim*: *mttm_step.cases*)
have *c2_fst_neq_t*: *fst* (*mt_state c2*) $\neq$ *t_tm M*
using *c2_fst_eq qM'_neq_t* **by** *simp*
have *c2_fst_neq_r*: *fst* (*mt_state c2*) $\neq$ *r_tm M*
using *c2_fst_eq qM'_neq_r* **by** *simp*
have *c1_idx_ss2*: *snd* (*snd* (*snd* (*snd* (*mt_state c1*)))) $=$ *SS2*
using *inv_ss2* **unfolding** *ae_inv_ss2_def* **by** (*cases mt_state c1*) *auto*
have *aux_45_at_c1*: $(c1, c2) \in$ *mttm_step* (*ae_delta_ss4_ss5 M*)
$\Longrightarrow$ *ae_position_link M c2*
using *ae_pos_link_aux_void_ss4_ss5*[**where** $M = M$ **and** *c_pre* $=$ *c1* **and** *c_post* $=$ *c2*]
c1_idx_ss2 **by** *simp*
have *aux_56_at_c1*: $(c1, c2) \in$ *mttm_step* (*ae_delta_ss5_ss6 M*)
$\Longrightarrow$ *ae_position_link M c2*
using *ae_pos_link_aux_void_ss5_ss6*[**where** $M = M$ **and** *c_pre* $=$ *c1* **and** *c_post* $=$ *c2*]
c1_idx_ss2 **by** *simp*
have *aux_78_at_c1*: $(c1, c2) \in$ *mttm_step* (*ae_delta_ss7_ss8 M*)
$\Longrightarrow$ *ae_position_link M c2*
using *ae_pos_link_aux_void_ss7_ss8*[**where** $M = M$ **and** *c_pre* $=$ *c1* **and**

```
c_post = c2]
        c1_idx_ss2 by simp
  have pos_link_c2: ae_position_link M c2
    using ae_step_alphabet_enlarge_position_link_preserve[OF pos_link_c1
          step2 aux_45_at_c1 aux_56_at_c1 aux_78_at_c1] .
  — Substep 3: SS3→SS4.
  obtain c3 where
      step3_sub: (c2, c3) ∈ mttm_step (ae_delta_ss3_ss4 M)
    and step3: (c2, c3) ∈ mttm_step (alphabet_enlarge_delta M)
    using ae_step_ss3_ss4_exists[OF inv_ss3 gamma_c2 buf_gamma_c2
                                    c2_fst_neq_t c2_fst_neq_r] by blast
  have inv_ss4: ae_inv_ss4 M c3
    using ae_step_ss3_ss4_invariant[OF vM inv_ss3 step3_sub] .
  have gamma_c3: ae_tape_in_gamma_block M c3
    using ae_step_alphabet_enlarge_gamma_preserve[OF gamma_c2 step3] .
  have buf_gamma_c3: ae_buffer_in_gamma_block M c3
          using ae_step_alphabet_enlarge_buffer_gamma_preserve[OF vM
buf_gamma_c2
                                                gamma_c2 step3] .
  have c3_fst_eq: fst (mt_state c3) = qM'
    using step3_sub c2_fst_eq
    by (cases mt_state c2)
       (auto simp: ae_delta_ss3_ss4_def elim: mttm_step.cases)
  have c2_idx_ss3: snd (snd (snd (snd (mt_state c2)))) = SS3
    using inv_ss3 unfolding ae_inv_ss3_def by (cases mt_state c2) auto
  have aux_45_at_c2: (c2, c3) ∈ mttm_step (ae_delta_ss4_ss5 M)
                    ⟹ ae_position_link M c3
    using ae_pos_link_aux_void_ss4_ss5[where M = M and c_pre = c2 and
c_post = c3]
        c2_idx_ss3 by simp
  have aux_56_at_c2: (c2, c3) ∈ mttm_step (ae_delta_ss5_ss6 M)
                    ⟹ ae_position_link M c3
    using ae_pos_link_aux_void_ss5_ss6[where M = M and c_pre = c2 and
c_post = c3]
        c2_idx_ss3 by simp
  have aux_78_at_c2: (c2, c3) ∈ mttm_step (ae_delta_ss7_ss8 M)
                    ⟹ ae_position_link M c3
    using ae_pos_link_aux_void_ss7_ss8[where M = M and c_pre = c2 and
c_post = c3]
        c2_idx_ss3 by simp
  have pos_link_c3: ae_position_link M c3
    using ae_step_alphabet_enlarge_position_link_preserve[OF pos_link_c2
          step3 aux_45_at_c2 aux_56_at_c2 aux_78_at_c2] .
  show ?thesis
    by (rule that[OF step1_sub step1 step2_sub step2 step3_sub step3
                  inv_ss4 gamma_c3 buf_gamma_c3 c3_fst_eq pos_link_c3])
qed
```

2.23.2 Output well-formedness theorem

Output well-formedness: *alphabet_enlarge* maps a valid substrate machine to a valid one, i.e. *valid_mttm M* $\Longrightarrow$ *valid_mttm* (*alphabet_enlarge M*). Every component of *alphabet_enlarge M* is derived from *M*'s: the state set is *Q_tm M* paired with the finite set of valid simulation stages, the tape alphabet is the block alphabet *gamma_block* (Γ*_tm M*), and the input alphabet, endmarkers, and start/accept/reject states are the corresponding block-encoded images. The proof discharges each *valid_mttm* conjunct (finiteness of the state and alphabet sets, $\Sigma \subseteq \Gamma$, blank- and endmarker-membership, and the LE-discipline on *alphabet_enlarge_delta*) from *M*'s.

This is the structural precondition the headline results build on: *alphabet_enlarge_time* and the language theorems (*alphabet_enlarge_language_forward* here, the biconditional *alphabet_enlarge_language* in *AlphabetEnlargement_Reverse.thy*) all reason about runs of *alphabet_enlarge M*, which first requires it to be a well-formed substrate object.

State-shape projection of *alphabet_enlarge_delta*: every transition's source and destination carry an *M*-state in *Q_tm M*, and the source is neither the lifted accept nor the lifted reject state (*t_tm M*, *init_stage le*) / (*r_tm M*, *init_stage le*). Read off the 16 substep builders by their explicit $q \in$ *Q_tm M* / substep-index discipline (*m_steps_buffered_state_preservation* supplies the SS4→SS5 destination q'). Crucially the proof touches only the discrete state components, never the guarded per-tape lambdas, so no *split*: *if_splits* is needed that split, applied to the *if* $k <$ *k_tm M* tape count guards across all 16 unfolded builders, is what made the monolithic δ-shape *auto* in *alphabet_enlarge_wf* loop. The gamma-codomain and *ae_valid_stage* halves of that δ-shape conjunct come straight off *alphabet_enlarge_delta*'s intersection guards instead.

lemma *alphabet_enlarge_delta_state_shape*:
  fixes *M* :: ('*q*, '*a*) *mttm*
  assumes *valM*: *valid_mttm M*
    and *mem*: (*s*, *a*, *s'*, *a'*, *d*) ∈ *alphabet_enlarge_delta M*
  shows *fst s* ∈ *Q_tm M* ∧ *fst s'* ∈ *Q_tm M*
      ∧ *s* ≠ (*t_tm M*, *init_stage* (*le_tm M*))
      ∧ *s* ≠ (*r_tm M*, *init_stage* (*le_tm M*))
proof −
  from *mem* **have** *mem_u*:
    (*s*, *a*, *s'*, *a'*, *d*) ∈
      ae_delta_val_fwd_advance M ∪ *ae_delta_val_fwd_to_padded M*
      ∪ *ae_delta_val_fwd_reject M* ∪ *ae_delta_val_fwd_to_ret M*
      ∪ *ae_delta_val_pad_to_ret M* ∪ *ae_delta_val_pad_reject M*
      ∪ *ae_delta_val_ret_step M* ∪ *ae_delta_val_ret_to_sim M*
      ∪ *ae_delta_ss1_ss2 M* ∪ *ae_delta_ss2_ss3 M*
      ∪ *ae_delta_ss3_ss4 M* ∪ *ae_delta_ss4_ss5 M*

```
      ∪ ae_delta_ss5_ss6 M ∪ ae_delta_ss6_ss7 M
      ∪ ae_delta_ss7_ss8 M ∪ ae_delta_ss8_ss1 M
    unfolding alphabet_enlarge_delta_def by blast
  show ?thesis
    using mem_u
    unfolding ae_delta_val_fwd_advance_def ae_delta_val_fwd_to_padded_def
              ae_delta_val_fwd_reject_def ae_delta_val_fwd_to_ret_def
              ae_delta_val_pad_to_ret_def ae_delta_val_pad_reject_def
              ae_delta_val_ret_step_def ae_delta_val_ret_to_sim_def
              ae_delta_ss1_ss2_def ae_delta_ss2_ss3_def
              ae_delta_ss3_ss4_def ae_delta_ss4_ss5_def
              ae_delta_ss5_ss6_def ae_delta_ss6_ss7_def
              ae_delta_ss7_ss8_def ae_delta_ss8_ss1_def
              init_stage_def
    using m_steps_buffered_state_preservation[OF valM]
          valid_mttm_r_in_Q[OF valM] valid_mttm_t_in_Q[OF valM]
    by auto
qed

theorem alphabet_enlarge_wf:
  fixes M :: ('q, 'a) mttm
  assumes valM: valid_mttm M
  shows valid_mttm
          (alphabet_enlarge M
            :: ('q × ('a, ('c :: enum)) ae_stage, 'c ⇒ 'a) mttm)
proof −
  obtain Q_M Σ_M Γ_M bl_M le_M δ_M s_M t_M r_M k_M where
    M_eq: M = MTTM Q_M Σ_M Γ_M bl_M le_M δ_M s_M t_M r_M k_M
    using mttm.exhaust by metis

  have fin_Gamma: finite Γ_M
    using valid_mttm_finite_Gamma[OF valM] M_eq by simp
  have le_in_Gamma: le_M ∈ Γ_M
    using valid_mttm_LE_in_Gamma[OF valM] M_eq by simp

  have ae_eq: (alphabet_enlarge M
                :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm) =
              MTTM (Q_M × {stg. ae_valid_stage Γ_M le_M k_M stg})
                   (gamma_block (Σ_M ∪ {bl_M})
                     − {bl_block bl_M, LE_block le_M})
                   (gamma_block Γ_M)
                   (bl_block bl_M)
                   (LE_block le_M)
                   (alphabet_enlarge_delta M)
                   (s_M, init_stage le_M)
                   (t_M, init_stage le_M)
                   (r_M, init_stage le_M)
                   k_M
    unfolding M_eq alphabet_enlarge_def by simp
```

```
  have init_valid: ae_valid_stage Γ_M le_M k_M (init_stage le_M)
    by (rule ae_valid_stage_init[OF le_in_Gamma])

  show valid_mttm
         (alphabet_enlarge M
           :: ('q × ('a, ('c :: enum)) ae_stage, 'c ⇒ 'a) mttm)
    unfolding ae_eq valid_mttm.simps
  proof (intro conjI)
    — (0) 0 < k: enlargement keeps M's tape count.
    show 0 < k_M
      using valid_mttm_k_pos[OF valM] M_eq by simp

    — (1) finite Q'
    show finite (Q_M ×
                 ({stg. ae_valid_stage Γ_M le_M k_M stg}
                   :: ('a, 'c) ae_stage set))
      using valid_mttm_finite_Q[OF valM] M_eq
            finite_ae_valid_stages[OF fin_Gamma]
      by (auto intro: finite_cartesian_product)

    — (2) finite Γ'
    show finite (gamma_block Γ_M :: ('c ⇒ 'a) set)
      using valid_mttm_finite_Gamma[OF valM] M_eq
      by (auto intro: finite_gamma_block)

    — (3) Σ' ⊆ Γ'
    show gamma_block (Σ_M ∪ {bl_M}) − {bl_block bl_M, LE_block le_M}
           ⊆ gamma_block Γ_M
    proof −
      have Σ_M ∪ {bl_M} ⊆ Γ_M
                          using valid_mttm_Sigma_sub_Gamma[OF valM]
valid_mttm_blank_in_Gamma[OF valM] M_eq
        by auto
      hence gamma_block (Σ_M ∪ {bl_M}) ⊆ gamma_block Γ_M
        by (rule gamma_block_mono)
      thus ?thesis by blast
    qed

    — (4) start state in Q'
    show (s_M, init_stage le_M)
           ∈ Q_M × {stg. ae_valid_stage Γ_M le_M k_M stg}
      using valid_mttm_s_in_Q[OF valM] M_eq init_valid by simp

    — (5) accept state in Q'
    show (t_M, init_stage le_M)
           ∈ Q_M × {stg. ae_valid_stage Γ_M le_M k_M stg}
      using valid_mttm_t_in_Q[OF valM] M_eq init_valid by simp
```

— (6) reject state in Q'
show $(r_M,\ init_stage\ le_M)$
$\in Q_M \times \{stg.\ ae_valid_stage\ \Gamma_M\ le_M\ k_M\ stg\}$
using *valid_mttm_r_in_Q*[*OF valM*] *M_eq init_valid* **by** *simp*

— (7a) blank in Γ'
show *bl_block* $bl_M \in$ *gamma_block* Γ_M
using *bl_block_in_gamma_block*[*OF valid_mttm_blank_in_Gamma*[*OF valM*]] *M_eq*
by *simp*

— (7b) blank not in Σ'
show *bl_block* bl_M
$\notin$ *gamma_block* $(\Sigma_M \cup \{bl_M\}) - \{bl_block\ bl_M,\ LE_block\ le_M\}$
by *simp*

— (8a) LE in Γ'
show *LE_block* $le_M \in$ *gamma_block* Γ_M
using *LE_block_in_gamma_block*[*OF valid_mttm_LE_in_Gamma*[*OF valM*]] *M_eq*
by *simp*

— (8b) LE not in Σ'
show *LE_block* le_M
$\notin$ *gamma_block* $(\Sigma_M \cup \{bl_M\}) - \{bl_block\ bl_M,\ LE_block\ le_M\}$
by *simp*

— (9) accept $\neq$ reject
show $(t_M,\ init_stage\ le_M) \neq (r_M,\ init_stage\ le_M)$
using *valid_mttm_t_neq_r*[*OF valM*] *M_eq* **by** *simp*

— (10) *δ_set* shape (range conditions on read / write blocks; source state in $Q' - \{t',\ r'\}$; dest state in Q'). Decomposed by intersection structure: the read/write codomains and the source/dest stage validity come straight off *alphabet_enlarge_delta*'s three intersection guards (no builder unfold), and the M-state shape from *alphabet_enlarge_delta_state_shape* (discrete state components only). Neither half needs *split*: *if_splits* over the 16 guarded builders the trap that made the old monolithic *auto* loop.
show *alphabet_enlarge_delta M*
$\subseteq ((Q_M \times \{stg.\ ae_valid_stage\ \Gamma_M\ le_M\ k_M\ stg\})$
$- \{(t_M,\ init_stage\ le_M),\ (r_M,\ init_stage\ le_M)\})$
$\times\ (UNIV \to gamma_block\ \Gamma_M)$
$\times (Q_M \times \{stg.\ ae_valid_stage\ \Gamma_M\ le_M\ k_M\ stg\})$
$\times\ (UNIV \to gamma_block\ \Gamma_M)$
$\times\ (UNIV \to UNIV)$
proof (*rule subsetI*)
fix *x* **assume** *xin*: $x \in$ *alphabet_enlarge_delta M*
obtain *s rest* **where** *x1*: $x = (s,\ rest)$ **by** (*cases x*)
obtain *a rest2* **where** *x2*: $rest = (a,\ rest2)$ **by** (*cases rest*)

obtain s' *rest3* **where** *x3*: *rest2* $= (s',$ *rest3*$)$ **by** (*cases rest2*)
obtain a' d **where** *x4*: *rest3* $= (a',$ $d)$ **by** (*cases rest3*)
from *x1 x2 x3 x4* **have** *x_eq*: $x = (s,$ $a,$ $s',$ $a',$ $d)$ **by** *simp*
from *xin* **have** *mem_t*: $(s,$ $a,$ $s',$ $a',$ $d) \in$ *alphabet_enlarge_delta M*
unfolding *x_eq* **by** *simp*
— Read/write codomain + source/dest stage validity: off the intersection guards, no builder unfold, no *if_splits*.
from *mem_t* **have** *inter*:
$(\forall k.$ a $k \in$ *gamma_block* $(\Gamma$*_tm M*$))$
$\wedge$ $(\forall k.$ a' $k \in$ *gamma_block* $(\Gamma$*_tm M*$))$
$\wedge$ *ae_valid_stage* $(\Gamma$*_tm M*$)$ $(le$*_tm M*$)$ $(k$*_tm M*$)$ $(snd$ $s)$
$\wedge$ *ae_valid_stage* $(\Gamma$*_tm M*$)$ $(le$*_tm M*$)$ $(k$*_tm M*$)$ $(snd$ $s')$
unfolding *alphabet_enlarge_delta_def* **by** *simp*
— State shape from the substep builders discrete components only.
have *shape*: *fst* $s \in$ *Q_tm M* $\wedge$ *fst* $s' \in$ *Q_tm M*
$\wedge$ $s \neq (t$*_tm M*$,$ *init_stage* $(le$*_tm M*$))$
$\wedge$ $s \neq (r$*_tm M*$,$ *init_stage* $(le$*_tm M*$))$
by (*rule alphabet_enlarge_delta_state_shape*[*OF valM mem_t*])
have *s_in*: $s \in (Q_M \times \{stg.$ *ae_valid_stage* Γ_M le_M k_M $stg\}$
$- \{(t_M,$ *init_stage* $le_M),$ $(r_M,$ *init_stage* $le_M)\}$
using *shape inter M_eq* **by** (*cases s*) *auto*
have *s'_in*: $s' \in Q_M \times \{stg.$ *ae_valid_stage* Γ_M le_M k_M $stg\}$
using *shape inter M_eq* **by** (*cases s'*) *auto*
have *a_in*: $a \in$ *UNIV* $\to$ *gamma_block* Γ_M
using *inter M_eq* **by** (*auto simp*: *Pi_iff*)
have *a'_in*: $a' \in$ *UNIV* $\to$ *gamma_block* Γ_M
using *inter M_eq* **by** (*auto simp*: *Pi_iff*)
show $x \in ((Q_M \times \{stg.$ *ae_valid_stage* Γ_M le_M k_M $stg\}$
$- \{(t_M,$ *init_stage* $le_M),$ $(r_M,$ *init_stage* $le_M)\})$
$\times$ $(UNIV \to$ *gamma_block* $\Gamma_M)$
$\times$ $(Q_M \times \{stg.$ *ae_valid_stage* Γ_M le_M k_M $stg\})$
$\times$ $(UNIV \to$ *gamma_block* $\Gamma_M)$
$\times$ $(UNIV \to UNIV)$
unfolding *x_eq* **using** *s_in s'_in a_in a'_in*
by (*auto simp*: *mem_Times_iff Pi_iff*)
qed

— (11) δLE preservation: case-analysis across the 8 substep relations. Each branch's d k is δLE-safe by construction (LE-guard prefix on action helpers; conditional move on SS1→SS2 and SS4→SS5).
show $\forall q$ a q' a' d k.
$(q,$ $a,$ $q',$ $a',$ $d) \in$ *alphabet_enlarge_delta M* $\longrightarrow$
a $k =$ *LE_block* le_M $\longrightarrow$
a' $k =$ *LE_block* le_M $\wedge$ d $k \in \{dir.N,$ $dir.R\}$
unfolding *alphabet_enlarge_delta_def*
ae_delta_val_fwd_advance_def ae_delta_val_fwd_to_padded_def
ae_delta_val_fwd_reject_def ae_delta_val_fwd_to_ret_def
ae_delta_val_pad_to_ret_def ae_delta_val_pad_reject_def
ae_delta_val_ret_step_def ae_delta_val_ret_to_sim_def

ae_delta_ss1_ss2_def ae_delta_ss2_ss3_def
ae_delta_ss3_ss4_def ae_delta_ss4_ss5_def
ae_delta_ss5_ss6_def ae_delta_ss6_ss7_def
ae_delta_ss7_ss8_def ae_delta_ss8_ss1_def
init_stage_def
using *M_eq*
by (*auto split*: *if_splits prod.splits*)

— (12) δ-support past tape count: every transition leaves tapes $\geq k_M$ blank for both read and write and stationary (*dir.N*). Each builder carries the read-support conjunct $\forall\, j{\geq}k_tm\ M.\ a\ j = bl_block\ (bl_tm\ M)$ (giving $a\ j$, and $a'\ j$ for the $a' = a$ builders); the four write-back builders guard a' to *bl_block* (*bl_tm M*) past tape count; and every *d*-guard's else-branch is *dir.N* (the validation builders' *if* $k = 0$ direction gives *dir.N* for $j \geq k_M > 0$, via $0 < k_M$). Same case-analysis shape as (11); the else-branch selection means the action helpers are never evaluated, so no action unfold is needed.

show $\forall\, q\ a\ q'\ a'\ d.$
$(q,\ a,\ q',\ a',\ d) \in alphabet_enlarge_delta\ M \longrightarrow$
$(\forall\, j{\geq}k_M.\ a\ j = bl_block\ bl_M$
$\wedge\ a'\ j = bl_block\ bl_M$
$\wedge\ d\ j = dir.N)$
unfolding *alphabet_enlarge_delta_def*
ae_delta_val_fwd_advance_def ae_delta_val_fwd_to_padded_def
ae_delta_val_fwd_reject_def ae_delta_val_fwd_to_ret_def
ae_delta_val_pad_to_ret_def ae_delta_val_pad_reject_def
ae_delta_val_ret_step_def ae_delta_val_ret_to_sim_def
ae_delta_ss1_ss2_def ae_delta_ss2_ss3_def
ae_delta_ss3_ss4_def ae_delta_ss4_ss5_def
ae_delta_ss5_ss6_def ae_delta_ss6_ss7_def
ae_delta_ss7_ss8_def ae_delta_ss8_ss1_def
using *M_eq valid_mttm_k_pos*[*OF valM*]
by (*auto split*: *if_splits prod.splits*)
qed
qed

end
theory *AlphabetEnlargement_ForwardCells*
imports *AlphabetEnlargement_OutputWF*
begin

2.24 Forward-stage per-tape reconstruction leaves

Base of the forward-stage chain *ForwardCells* → *ForwardStep* → *ForwardStage*. The unified forward-stage lemma *ae_simulates_forward_stage_general* (theory *AlphabetEnlargement_ForwardStage*) advances the simulation by one *M*-step through eight substeps (*SS1* through *SS8*); the *SS5–SS8* half is packaged as the super-step *ae_forward_stage_step_chain* (theory *AlphabetEnlargement_ForwardStep*). From the configs *c5–c8* that super-step exposes, the

lemma then reconstructs, tape by tape, where each simulated head sits and what its in-window blocks hold. The nine lemmas in this theory are those per-tape reconstruction facts.

Each branches on the tape's block position *mt_pos c′ kk* into the three regimes (steady $\geq$ *2*, le1 = *1*, le0 = *0*) and reads off the *c5*–*c8* intermediate configs the super-step exposes. They group into the head-position trajectories (*ae_fwd_c8_pos_for_∗*, *ae_fwd_c6_pos_for_∗*, *ae_fwd_pos_decode_c8*), the in-window cell values (*ae_fwd_c8_at_∗*), and the per-tape tape-correspondence (*ae_fwd_tape_corr_c8*). Each was extracted from the unified lemma's proof, so its assumption interface is wide: the assumptions are exactly the data-flow the fact consumed when it was an inline block.

Reconstruction leaf of *ae_simulates_forward_stage_general* (the pos-at-least-2 in-window *c8* value at block pos minus 1).

lemma *ae_fwd_c8_at_pos_minus_1_ge2*:
fixes *M* :: (′*q*, ′*a*) *mttm*
and *c′ c4 c5 c6 c7 c8* :: (′*c* :: *enum* ⇒ ′*a*, ′*q* × (′*a*, ′*c*) *ae_stage*) *mt_config*
and *q5 q6* :: ′*q*
and *ofs5 ofs6* :: *nat* ⇒ ′*c*
and *buf5 buf6* :: *nat* ⇒ (′*c* ⇒ ′*a*) × (′*c* ⇒ ′*a*) × (′*c* ⇒ ′*a*)
and *dest5 dest6* :: *nat* ⇒ *ae_dest*
assumes *step5*: (*c4*, *c5*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *c5_state*: *mt_state c5* = (*q5*, *ofs5*, *buf5*, *dest5*, *SS6*)
and *tape_c4_eq_c′*: *mt_tape c4* = *mt_tape c′*
and *buf5_not_le_per_tape*:
∀ *k*<*k_tm M*. (*mt_pos c′ k* = *0*
⟶ *fst* (*buf5 k*) ≠ *LE_block* (*le_tm M*)
∧ *snd* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*))
∧ (*mt_pos c′ k* = *1*
⟶ *fst* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*)
∧ *snd* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*))
∧ (*mt_pos c′ k* ≥ *2*
⟶ *fst* (*buf5 k*) ≠ *LE_block* (*le_tm M*)
∧ *fst* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*)
∧ *snd* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*))
and *c4_pos*: ⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c4 kk* = *mt_pos c′ kk*
and *step6_sub*: (*c5*, *c6*) ∈ *mttm_step* (*ae_delta_ss6_ss7 M*)
and *step6*: (*c5*, *c6*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *step7*: (*c6*, *c7*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *c7_state*: *mt_state c7* = (*q6*, *ofs6*, *buf6*, *dest6*, *SS8*)
and *buf6_eq_buf5*: *buf6* = *buf5*
and *dest6_eq_dest5*: *dest6* = *dest5*
and *step8_sub*: (*c7*, *c8*) ∈ *mttm_step* (*ae_delta_ss8_ss1 M*)
and *step8*: (*c7*, *c8*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *c5_pos_for_pos_ge2*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c′ kk* ≥ *2* ⟹ *dest5 kk* ≠ *AE_Left*

```
        ⟹ mt_pos c5 kk = mt_pos c' kk − 1
    and c5_pos_for_pos_ge2_dest_left:
      ⋀kk. kk < k_tm M ⟹ mt_pos c' kk ≥ 2 ⟹ dest5 kk = AE_Left
        ⟹ mt_pos c5 kk = mt_pos c' kk + 1
    and left_not_le_c'_steady:
      ∀k. mt_pos c' k ≥ 2
        ⟶ mt_tape c' k (mt_pos c' k − 1) ≠ LE_block (le_tm M)
    and c6_pos_for_pos_ge2:
      ⋀kk. kk < k_tm M ⟹ mt_pos c' kk ≥ 2 ⟹ mt_pos c6 kk = mt_pos
c' kk
    and c7_pos_for_pos_ge2:
      ⋀kk. kk < k_tm M ⟹ mt_pos c' kk ≥ 2
        ⟹ mt_pos c7 kk = (if dest5 kk = AE_Left
                          then mt_pos c' kk − 1
                          else mt_pos c' kk + 1)
  shows ⋀kk. kk < k_tm M ⟹ mt_pos c' kk ≥ 2
      ⟹ mt_tape c8 kk (mt_pos c' kk − 1) = fst (buf5 kk)
proof −
  fix kk
  assume kklt: kk < k_tm M
  assume hge2: mt_pos c' kk ≥ 2
  obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
    using prod.exhaust by metis
  have buf6_kk: buf6 kk = (ll, hh, rr)
    using buf5_kk buf6_eq_buf5 by simp
  have h_ne_LE: hh ≠ LE_block (le_tm M)
  proof −
    have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
      using buf5_not_le_per_tape[rule_format, OF kklt] hge2 by blast
    thus ?thesis using buf5_kk by simp
  qed
  have c4_kk_eq: mt_pos c4 kk = mt_pos c' kk using c4_pos kklt by simp
  have pm1_ne_c4: (mt_pos c' kk − 1) ≠ mt_pos c4 kk
    using c4_kk_eq hge2 by linarith
  have c4_at_pm1_ne_LE:
     mt_tape c4 kk (mt_pos c' kk − 1) ≠ LE_block (le_tm M)
  proof −
   have mt_tape c4 kk (mt_pos c' kk − 1) = mt_tape c' kk (mt_pos c' kk − 1)
     using tape_c4_eq_c' by simp
   moreover have mt_tape c' kk (mt_pos c' kk − 1) ≠ LE_block (le_tm M)
     using left_not_le_c'_steady hge2 by blast
   ultimately show ?thesis by simp
  qed
  show mt_tape c8 kk (mt_pos c' kk − 1) = fst (buf5 kk)
  proof (cases dest5 kk)
    case AE_Left
    — SS5/SS6/SS7 off pos-1; SS8 writes l at c7_pos = pos − 1.
    have c5_pos_kk: mt_pos c5 kk = mt_pos c' kk + 1
      using c5_pos_for_pos_ge2_dest_left[OF kklt hge2 AE_Left] .
```

```
    have pm1_ne_c5: (mt_pos c' kk − 1) ≠ mt_pos c5 kk
      using c5_pos_kk hge2 by linarith
    have c6_pos_kk: mt_pos c6 kk = mt_pos c' kk
      using c6_pos_for_pos_ge2[OF kklt hge2] .
    have pm1_ne_c6: (mt_pos c' kk − 1) ≠ mt_pos c6 kk
      using c6_pos_kk hge2 by linarith
    have c7_at_pm1_eq_c4:
       mt_tape c7 kk (mt_pos c' kk − 1) = mt_tape c4 kk (mt_pos c' kk − 1)
    proof −
      have mt_tape c7 kk (mt_pos c' kk − 1)
            = mt_tape c6 kk (mt_pos c' kk − 1)
        using mttm_step_tape_off_head[OF step7 pm1_ne_c6] .
      also have ... = mt_tape c5 kk (mt_pos c' kk − 1)
        using mttm_step_tape_off_head[OF step6 pm1_ne_c5] .
      also have ... = mt_tape c4 kk (mt_pos c' kk − 1)
        using mttm_step_tape_off_head[OF step5 pm1_ne_c4] .
      finally show ?thesis .
    qed
    have c7_pos_kk: mt_pos c7 kk = mt_pos c' kk − 1
      using c7_pos_for_pos_ge2[OF kklt hge2] AE_Left by simp
    obtain qq tts nn qq' aa8 dr8 where
       c7_eq8: c7 = Config_M qq tts nn
      and c8_eq: c8 = Config_M qq'
                      (λk. (tts k)(nn k := aa8 k))
                      (λk. go_dir (dr8 k) (nn k))
      and tr_in: (qq, λk. tts k (nn k), qq', aa8, dr8)
                 ∈ ae_delta_ss8_ss1 M
      using step8_sub by (auto elim: mttm_step.cases)
    obtain q ofs buf dest where
       qq_eq: qq = (q, ofs, buf, dest, SS8)
      and aa8_eq:
         aa8 = (λk. if k < k_tm M then fst (ae_ss8_action (le_tm M)
                        (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
      using tr_in by (auto simp: ae_delta_ss8_ss1_def)
    have qq_state: qq = (q6, ofs6, buf6, dest6, SS8)
      using c7_state c7_eq8 by simp
    have buf_eq: buf = buf6 using qq_eq qq_state by simp
    have dest_eq: dest = dest6 using qq_eq qq_state by simp
    have nn_kk: nn kk = mt_pos c7 kk using c7_eq8 by simp
     have nn_kk_val: nn kk = mt_pos c' kk − 1 using nn_kk c7_pos_kk by
simp
    have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
    proof −
      have tts kk (nn kk) = mt_tape c7 kk (mt_pos c7 kk)
        using c7_eq8 nn_kk by simp
       also have ... = mt_tape c7 kk (mt_pos c' kk − 1) using c7_pos_kk by
simp
    also have ... = mt_tape c4 kk (mt_pos c' kk − 1) using c7_at_pm1_eq_c4
.
```

```
      finally have tts kk (nn kk) = mt_tape c4 kk (mt_pos c' kk − 1) .
      thus ?thesis using c4_at_pm1_ne_LE by simp
    qed
    have dest6_AE_Left: dest6 kk = AE_Left
      using AE_Left dest6_eq_dest5 by simp
    have aa8_kk: aa8 kk = ll
          using aa8_eq buf_eq dest_eq dest6_AE_Left buf6_kk h_ne_LE
read_ne_LE kklt
      by simp
    have mt_tape c8 kk (mt_pos c' kk − 1) = aa8 kk
      using c8_eq nn_kk_val by simp
    also have ... = ll using aa8_kk .
    also have ... = fst (buf5 kk) using buf5_kk by simp
    finally show ?thesis .
  next
    case AE_Home
    — SS6 writes l at c5_pos = pos − 1; SS7/SS8 off pos-1.
    have dest_ne_left: dest5 kk ≠ AE_Left using AE_Home by simp
    have c5_pos_kk: mt_pos c5 kk = mt_pos c' kk − 1
      using c5_pos_for_pos_ge2[OF kklt hge2 dest_ne_left] .
    obtain qq tts nn qq' aa6 dr6 where
      c5_eq6: c5 = Config_M qq tts nn
      and c6_eq: c6 = Config_M qq'
                      (λk. (tts k)(nn k := aa6 k))
                      (λk. go_dir (dr6 k) (nn k))
      and tr_in: (qq, λk. tts k (nn k), qq', aa6, dr6)
                 ∈ ae_delta_ss6_ss7 M
      using step6_sub by (auto elim: mttm_step.cases)
    obtain q ofs buf dest where
      qq_eq: qq = (q, ofs, buf, dest, SS6)
      and aa6_eq:
        aa6 = (λk. if k < k_tm M then fst (ae_ss6_action (le_tm M)
                        (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
      using tr_in by (auto simp: ae_delta_ss6_ss7_def)
    have qq_state: qq = (q5, ofs5, buf5, dest5, SS6)
      using c5_state c5_eq6 by simp
    have buf_eq: buf = buf5 using qq_eq qq_state by simp
    have dest_eq: dest = dest5 using qq_eq qq_state by simp
    have nn_kk: nn kk = mt_pos c5 kk using c5_eq6 by simp
     have nn_kk_val: nn kk = mt_pos c' kk − 1 using nn_kk c5_pos_kk by
simp
    have c5_at_pm1_eq_c4:
      mt_tape c5 kk (mt_pos c' kk − 1) = mt_tape c4 kk (mt_pos c' kk − 1)
      using mttm_step_tape_off_head[OF step5 pm1_ne_c4] .
    have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
    proof −
      have tts kk (nn kk) = mt_tape c5 kk (mt_pos c5 kk)
        using c5_eq6 nn_kk by simp
      also have ... = mt_tape c5 kk (mt_pos c' kk − 1) using c5_pos_kk by
```

```
simp
    also have ... = mt_tape c4 kk (mt_pos c' kk − 1) using c5_at_pm1_eq_c4
.
      finally have tts kk (nn kk) = mt_tape c4 kk (mt_pos c' kk − 1) .
      thus ?thesis using c4_at_pm1_ne_LE by simp
    qed
    have aa6_kk: aa6 kk = ll
      using aa6_eq buf_eq dest_eq AE_Home buf5_kk h_ne_LE read_ne_LE
kklt by simp
    have c6_at_pm1: mt_tape c6 kk (mt_pos c' kk − 1) = ll
    proof −
      have mt_tape c6 kk (mt_pos c' kk − 1) = aa6 kk
        using c6_eq nn_kk_val by simp
      thus ?thesis using aa6_kk by simp
    qed
    have c6_pos_kk: mt_pos c6 kk = mt_pos c' kk
      using c6_pos_for_pos_ge2[OF kklt hge2] .
    have pm1_ne_c6: (mt_pos c' kk − 1) ≠ mt_pos c6 kk
      using c6_pos_kk hge2 by linarith
    have c7_pos_kk: mt_pos c7 kk = mt_pos c' kk + 1
      using c7_pos_for_pos_ge2[OF kklt hge2] AE_Home by simp
    have pm1_ne_c7: (mt_pos c' kk − 1) ≠ mt_pos c7 kk
      using c7_pos_kk by linarith
    have mt_tape c8 kk (mt_pos c' kk − 1) = mt_tape c7 kk (mt_pos c' kk −
1)
      using mttm_step_tape_off_head[OF step8 pm1_ne_c7] .
    also have ... = mt_tape c6 kk (mt_pos c' kk − 1)
      using mttm_step_tape_off_head[OF step7 pm1_ne_c6] .
    also have ... = ll using c6_at_pm1 .
    also have ... = fst (buf5 kk) using buf5_kk by simp
    finally show ?thesis .
  next
    case AE_Right
    — SS6 writes l at c5_pos = pos − 1; SS7/SS8 off pos-1.
    have dest_ne_left: dest5 kk ≠ AE_Left using AE_Right by simp
    have c5_pos_kk: mt_pos c5 kk = mt_pos c' kk − 1
      using c5_pos_for_pos_ge2[OF kklt hge2 dest_ne_left] .
    obtain qq tts nn qq' aa6 dr6 where
      c5_eq6: c5 = Config_M qq tts nn
      and c6_eq: c6 = Config_M qq'
                      (λk. (tts k)(nn k := aa6 k))
                      (λk. go_dir (dr6 k) (nn k))
      and tr_in: (qq, λk. tts k (nn k), qq', aa6, dr6)
                 ∈ ae_delta_ss6_ss7 M
      using step6_sub by (auto elim: mttm_step.cases)
    obtain q ofs buf dest where
      qq_eq: qq = (q, ofs, buf, dest, SS6)
      and aa6_eq:
        aa6 = (λk. if k < k_tm M then fst (ae_ss6_action (le_tm M)
```

```
                    (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
    using tr_in by (auto simp: ae_delta_ss6_ss7_def)
  have qq_state: qq = (q5, ofs5, buf5, dest5, SS6)
    using c5_state c5_eq6 by simp
  have buf_eq: buf = buf5 using qq_eq qq_state by simp
  have dest_eq: dest = dest5 using qq_eq qq_state by simp
  have nn_kk: nn kk = mt_pos c5 kk using c5_eq6 by simp
   have nn_kk_val: nn kk = mt_pos c' kk − 1 using nn_kk c5_pos_kk by
simp
  have c5_at_pm1_eq_c4:
     mt_tape c5 kk (mt_pos c' kk − 1) = mt_tape c4 kk (mt_pos c' kk − 1)
    using mttm_step_tape_off_head[OF step5 pm1_ne_c4] .
  have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
  proof −
    have tts kk (nn kk) = mt_tape c5 kk (mt_pos c5 kk)
      using c5_eq6 nn_kk by simp
    also have ... = mt_tape c5 kk (mt_pos c' kk − 1) using c5_pos_kk by
simp
  also have ... = mt_tape c4 kk (mt_pos c' kk − 1) using c5_at_pm1_eq_c4
.
    finally have tts kk (nn kk) = mt_tape c4 kk (mt_pos c' kk − 1) .
    thus ?thesis using c4_at_pm1_ne_LE by simp
  qed
  have aa6_kk: aa6 kk = ll
     using aa6_eq buf_eq dest_eq AE_Right buf5_kk h_ne_LE read_ne_LE
kklt by simp
  have c6_at_pm1: mt_tape c6 kk (mt_pos c' kk − 1) = ll
  proof −
    have mt_tape c6 kk (mt_pos c' kk − 1) = aa6 kk
      using c6_eq nn_kk_val by simp
    thus ?thesis using aa6_kk by simp
  qed
  have c6_pos_kk: mt_pos c6 kk = mt_pos c' kk
    using c6_pos_for_pos_ge2[OF kklt hge2] .
  have pm1_ne_c6: (mt_pos c' kk − 1) ≠ mt_pos c6 kk
    using c6_pos_kk hge2 by linarith
  have c7_pos_kk: mt_pos c7 kk = mt_pos c' kk + 1
    using c7_pos_for_pos_ge2[OF kklt hge2] AE_Right by simp
  have pm1_ne_c7: (mt_pos c' kk − 1) ≠ mt_pos c7 kk
    using c7_pos_kk by linarith
   have mt_tape c8 kk (mt_pos c' kk − 1) = mt_tape c7 kk (mt_pos c' kk −
1)
    using mttm_step_tape_off_head[OF step8 pm1_ne_c7] .
  also have ... = mt_tape c6 kk (mt_pos c' kk − 1)
    using mttm_step_tape_off_head[OF step7 pm1_ne_c6] .
  also have ... = ll using c6_at_pm1 .
  also have ... = fst (buf5 kk) using buf5_kk by simp
  finally show ?thesis .
 qed
```

qed

Reconstruction leaf of *ae_simulates_forward_stage_general*: the per-tape regime-aware *ae_tape_correspondence* between the simulated M-config *cM_k* and the post-chain M'-config *c8*. Per tape it branches on *mt_pos c' kk* (steady / le1 / le0) and within each regime on the block *s*: an in-window cell matches via the buf-lin transfer coinciding with the *c8* in-window value on the *buf5* slot, an off-window cell via *cM_k = cM = c'* (M-side) and *c8 = c'* (M'-side).

lemma *ae_fwd_tape_corr_c8*:
fixes *M* :: ('q, 'a) *mttm*
and *cM cM_k* :: ('a, 'q) *mt_config*
and *c' c4 c5 c6 c7 c8* :: ('c :: *enum* ⇒ 'a,
'q × ('a, 'c) *ae_stage*) *mt_config*
and *buf5* :: *nat* ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
and *dest5* :: *nat* ⇒ *ae_dest*
assumes *tape_corr*: ∀ *k*<*k_tm M*. *ae_tape_correspondence* (*le_tm M*)
(*mt_tape cM k*) (*mt_tape c' k*)
and *c_ge_1*: *1* ≤ *card* (*UNIV* :: 'c *set*)
and *c4_pos*: ⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c4 kk* = *mt_pos c' kk*
and *c5_pos_for_pos1*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* = *1* ⟹ *dest5 kk* ≠ *AE_Left*
⟹ *mt_pos c5 kk* = *0*
and *c5_pos_for_pos1_dest_left*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* = *1* ⟹ *dest5 kk* = *AE_Left*
⟹ *mt_pos c5 kk* = *2*
and *c7_pos_for_pos1_dest_left*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* = *1* ⟹ *dest5 kk* = *AE_Left*
⟹ *mt_pos c7 kk* = *0*
and *cM_k_zero*: ⋀*kk*. *kk* < *k_tm M* ⟹ *mt_tape cM_k kk 0* = *le_tm M*
and *m_tape_off_window_le0*:
⋀*kk p*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* = *0*
⟹ *p* > *card* (*UNIV* :: 'c *set*)
⟹ *mt_tape cM_k kk p* = *mt_tape cM kk p*
and *m_tape_off_window_le1*:
⋀*kk p*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* = *1*
⟹ *p* > *2* ∗ *card* (*UNIV* :: 'c *set*)
⟹ *mt_tape cM_k kk p* = *mt_tape cM kk p*
and *m_tape_off_window_steady*:
⋀*kk p*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* ≥ *2*
⟹ *p* < (*mt_pos c' kk* − *2*) ∗ *card* (*UNIV* :: 'c *set*) + *1*
∨ *p* ≥ (*mt_pos c' kk* − *2*) ∗ *card* (*UNIV* :: 'c *set*) + *1* + *3* ∗ *card*
(*UNIV* :: 'c *set*)
⟹ *mt_tape cM_k kk p* = *mt_tape cM kk p*
and *c5_pos_for_pos0*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* = *0* ⟹ *mt_pos c5 kk* = *0*
and *c5_pos_for_pos_ge2*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* ≥ *2* ⟹ *dest5 kk* ≠ *AE_Left*
⟹ *mt_pos c5 kk* = *mt_pos c' kk* − *1*

and *c5_pos_for_pos_ge2_dest_left*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk \geq 2 \Longrightarrow dest5\ kk = AE_Left$
$\Longrightarrow mt_pos\ c5\ kk = mt_pos\ c'\ kk + 1$
and *c6_pos_for_pos0*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 0 \Longrightarrow mt_pos\ c6\ kk = 1$
and *c6_pos_for_pos_ge2*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk \geq 2 \Longrightarrow mt_pos\ c6\ kk = mt_pos\ c'\ kk$
and *c7_pos_for_pos0*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 0$
$\Longrightarrow mt_pos\ c7\ kk = (if\ dest5\ kk = AE_Right\ then\ 1\ else\ 0)$
and *c7_pos_for_pos_ge2*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk \geq 2$
$\Longrightarrow mt_pos\ c7\ kk = (if\ dest5\ kk = AE_Left$
$then\ mt_pos\ c'\ kk - 1$
$else\ mt_pos\ c'\ kk + 1)$
and *c6_pos_for_pos1*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 1 \Longrightarrow mt_pos\ c6\ kk = 1$
and *c7_pos_for_pos1*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 1$
$\Longrightarrow mt_pos\ c7\ kk = (if\ dest5\ kk = AE_Left\ then\ 0\ else\ 2)$
and *c8_tape_off_window*:
$\bigwedge kk\ s.\ s \neq mt_pos\ c4\ kk \Longrightarrow s \neq mt_pos\ c5\ kk$
$\Longrightarrow s \neq mt_pos\ c6\ kk \Longrightarrow s \neq mt_pos\ c7\ kk$
$\Longrightarrow mt_tape\ c8\ kk\ s = mt_tape\ c'\ kk\ s$
and *c8_tape_at_one_for_pos0*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 0$
$\Longrightarrow mt_tape\ c8\ kk\ 1 = snd\ (snd\ (buf5\ kk))$
and *c8_tape_at_one_for_pos1*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 1$
$\Longrightarrow mt_tape\ c8\ kk\ 1 = fst\ (snd\ (buf5\ kk))$
and *c8_tape_at_two_for_pos1*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 1$
$\Longrightarrow mt_tape\ c8\ kk\ 2 = snd\ (snd\ (buf5\ kk))$
and *c8_tape_at_pos_for_pos_ge2*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk \geq 2$
$\Longrightarrow mt_tape\ c8\ kk\ (mt_pos\ c'\ kk) = fst\ (snd\ (buf5\ kk))$
and *c8_tape_at_pos_minus_1_for_pos_ge2*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk \geq 2$
$\Longrightarrow mt_tape\ c8\ kk\ (mt_pos\ c'\ kk - 1) = fst\ (buf5\ kk)$
and *c8_tape_at_pos_plus_1_for_pos_ge2*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk \geq 2$
$\Longrightarrow mt_tape\ c8\ kk\ (mt_pos\ c'\ kk + 1) = snd\ (snd\ (buf5\ kk))$
and *tape_cM_k_at_one_for_pos0*:
$\bigwedge kk\ i.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 0$
$\Longrightarrow mt_tape\ cM_k\ kk\ (Suc\ (c_idx\ (i :: 'c)))$
$= (snd\ (snd\ (buf5\ kk)))\ i$
and *tape_cM_k_at_one_for_pos1*:
$\bigwedge kk\ i.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 1$

```
        ⟹ mt_tape cM_k kk (Suc (c_idx (i :: 'c)))
        = (fst (snd (buf5 kk))) i
    and tape_cM_k_at_two_for_pos1:
      ⋀kk i. kk < k_tm M ⟹ mt_pos c' kk = 1
        ⟹ mt_tape cM_k kk (Suc (card (UNIV :: 'c set) + c_idx (i ::
'c)))
        = (snd (snd (buf5 kk))) i
    and tape_cM_k_at_pos_minus_1_for_pos_ge2:
      ⋀kk i. kk < k_tm M ⟹ mt_pos c' kk ≥ 2
        ⟹ mt_tape cM_k kk ((mt_pos c' kk − 2) * card (UNIV :: 'c set)
        + c_idx (i :: 'c) + 1)
        = (fst (buf5 kk)) i
    and tape_cM_k_at_pos_for_pos_ge2:
      ⋀kk i. kk < k_tm M ⟹ mt_pos c' kk ≥ 2
        ⟹ mt_tape cM_k kk ((mt_pos c' kk − 1) * card (UNIV :: 'c set)
        + c_idx (i :: 'c) + 1)
        = (fst (snd (buf5 kk))) i
    and tape_cM_k_at_pos_plus_1_for_pos_ge2:
      ⋀kk i. kk < k_tm M ⟹ mt_pos c' kk ≥ 2
        ⟹ mt_tape cM_k kk (mt_pos c' kk * card (UNIV :: 'c set)
        + c_idx (i :: 'c) + 1)
        = (snd (snd (buf5 kk))) i
  shows ∀kk<k_tm M. ae_tape_correspondence (le_tm M)
      (mt_tape cM_k kk) (mt_tape c8 kk)
proof −
  let ?c = card (UNIV :: 'c set)
  show ?thesis
  proof (intro allI impI)
    fix kk
    assume kklt: kk < k_tm M
    show ae_tape_correspondence (le_tm M)
        (mt_tape cM_k kk) (mt_tape c8 kk)
      unfolding ae_tape_correspondence_def
    proof (intro conjI allI impI)
      show mt_tape cM_k kk 0 = le_tm M using cM_k_zero[OF kklt] by simp
    next
      fix s :: nat and i :: 'c
      assume s_ge_1: s ≥ 1
      let ?p = (s − 1) * ?c + c_idx i + 1
      have c_idx_lt: c_idx i < ?c by (rule c_idx_lt_card)
      have c4_kk_eq: mt_pos c4 kk = mt_pos c' kk using c4_pos kklt by simp
      consider (le0) mt_pos c' kk = 0
          | (le1) mt_pos c' kk = 1
          | (steady) mt_pos c' kk ≥ 2
        by linarith
      thus mt_tape cM_k kk ?p = mt_tape c8 kk s i
      proof cases
        case le0
        — pos=0: s=1 is in-window (buf-lin r-slot transfer matches c8@1 = rr); s ≥
```

2 is off-window on both sides.
show *?thesis*
proof (*cases s = 1*)
case *True*
have *p_eq*: *?p = Suc (c_idx i)* **using** *True* **by** *simp*
have *lhs*: *mt_tape cM_k kk ?p = (snd (snd (buf5 kk))) i*
using *p_eq tape_cM_k_at_one_for_pos0*[*OF kklt le0*] **by** *simp*
have *c8_at_s*: *mt_tape c8 kk s = mt_tape c8 kk 1* **using** *True* **by** *simp*
have *c8_at_1*: *mt_tape c8 kk 1 = snd (snd (buf5 kk))*
using *c8_tape_at_one_for_pos0*[*OF kklt le0*] **.**
show *?thesis* **using** *lhs c8_at_s c8_at_1* **by** *simp*
next
case *False*
have *s_ge_2*: *s ≥ 2* **using** *s_ge_1 False* **by** *linarith*
have *p_gt_c*: *?p > ?c*
proof −
have *s_sub_ge*: *s − 1 ≥ 1* **using** *s_ge_2* **by** *linarith*
have *mult_ge*: *(s − 1) ∗ ?c ≥ ?c*
using *s_sub_ge c_ge_1*
by (*metis mult.commute mult_le_mono2 mult_numeral_1_right*
mult_le_cancel2 nat_mult_1_right)
show *?thesis* **using** *mult_ge* **by** *linarith*
qed
— M-side off-window: *cM_k@p = cM@p = c′@s i*.
have *cM_k_at_p*: *mt_tape cM_k kk ?p = mt_tape cM kk ?p*
using *m_tape_off_window_le0*[*OF kklt le0 p_gt_c*] **.**
have *tc_at*: *mt_tape cM kk ?p = mt_tape c′ kk s i*
using *tape_corr*[*rule_format*, *OF kklt*] *s_ge_1*
unfolding *ae_tape_correspondence_def* **by** *simp*
— M'-side off-window: *c4_pos = 0*, *c5_pos = 0*, *c6_pos = 1*, *c7_pos ∈* {*0*, *1*}; for *s ≥ 2* all four positions *≠ s*.
have *s_ne_c4*: *s ≠ mt_pos c4 kk*
using *c4_kk_eq le0 s_ge_2* **by** *linarith*
have *s_ne_c5*: *s ≠ mt_pos c5 kk*
using *c5_pos_for_pos0*[*OF kklt le0*] *s_ge_2* **by** *simp*
have *s_ne_c6*: *s ≠ mt_pos c6 kk*
using *c6_pos_for_pos0*[*OF kklt le0*] *s_ge_2* **by** *simp*
have *s_ne_c7*: *s ≠ mt_pos c7 kk*
proof (*cases dest5 kk = AE_Right*)
case *True*
have *mt_pos c7 kk = 1*
using *c7_pos_for_pos0*[*OF kklt le0*] *True* **by** *simp*
thus *?thesis* **using** *s_ge_2* **by** *simp*
next
case *False*
have *mt_pos c7 kk = 0*
using *c7_pos_for_pos0*[*OF kklt le0*] *False* **by** *simp*
thus *?thesis* **using** *s_ge_2* **by** *simp*
qed

```
    have c8_at_s: mt_tape c8 kk s = mt_tape c' kk s
     using c8_tape_off_window[OF s_ne_c4 s_ne_c5 s_ne_c6 s_ne_c7] .
    show ?thesis using cM_k_at_p tc_at c8_at_s by simp
  qed
next
  case le1
  — pos=1: s ∈ {1, 2} in-window, s ≥ 3 off-window.
  show ?thesis
  proof (cases s = 1)
    case True
    have p_eq: ?p = Suc (c_idx i) using True by simp
    have lhs: mt_tape cM_k kk ?p = (fst (snd (buf5 kk))) i
      using p_eq tape_cM_k_at_one_for_pos1[OF kklt le1] by simp
    have c8_at_s: mt_tape c8 kk s = mt_tape c8 kk 1 using True by simp
    have c8_at_1: mt_tape c8 kk 1 = fst (snd (buf5 kk))
      using c8_tape_at_one_for_pos1[OF kklt le1] .
    show ?thesis using lhs c8_at_s c8_at_1 by simp
  next
    case False
    have s_ge_2: s ≥ 2 using s_ge_1 False by linarith
    show ?thesis
    proof (cases s = 2)
      case True
      have p_eq: ?p = Suc (?c + c_idx i) using True by simp
      have lhs: mt_tape cM_k kk ?p = (snd (snd (buf5 kk))) i
        using p_eq tape_cM_k_at_two_for_pos1[OF kklt le1] by simp
     have c8_at_s: mt_tape c8 kk s = mt_tape c8 kk 2 using True by simp
      have c8_at_2: mt_tape c8 kk 2 = snd (snd (buf5 kk))
        using c8_tape_at_two_for_pos1[OF kklt le1] .
      show ?thesis using lhs c8_at_s c8_at_2 by simp
    next
      case False
      have s_ge_3: s ≥ 3 using s_ge_2 False by linarith
      have p_gt_2c: ?p > 2 * ?c
      proof −
        have s_sub_ge: s − 1 ≥ 2 using s_ge_3 by linarith
        have mult_ge: (s − 1) * ?c ≥ 2 * ?c
          using s_sub_ge c_ge_1
          by (metis mult.commute mult_le_mono2)
        show ?thesis using mult_ge by linarith
      qed
      have cM_k_at_p: mt_tape cM_k kk ?p = mt_tape cM kk ?p
        using m_tape_off_window_le1[OF kklt le1 p_gt_2c] .
      have tc_at: mt_tape cM kk ?p = mt_tape c' kk s i
        using tape_corr[rule_format, OF kklt] s_ge_1
        unfolding ae_tape_correspondence_def by simp
      have s_ne_c4: s ≠ mt_pos c4 kk
        using c4_kk_eq le1 s_ge_3 by linarith
      have s_ne_c5: s ≠ mt_pos c5 kk
```

```
          proof (cases dest5 kk = AE_Left)
            case True
            have mt_pos c5 kk = 2
              using c5_pos_for_pos1_dest_left[OF kklt le1 True] .
            thus ?thesis using s_ge_3 by simp
          next
            case False
            have mt_pos c5 kk = 0
              using c5_pos_for_pos1[OF kklt le1 False] .
            thus ?thesis using s_ge_3 by simp
          qed
          have s_ne_c6: s ≠ mt_pos c6 kk
            using c6_pos_for_pos1[OF kklt le1] s_ge_3 by simp
          have s_ne_c7: s ≠ mt_pos c7 kk
          proof (cases dest5 kk = AE_Left)
            case True
            have mt_pos c7 kk = 0
              using c7_pos_for_pos1_dest_left[OF kklt le1 True] .
            thus ?thesis using s_ge_3 by simp
          next
            case False
            have mt_pos c7 kk = 2
              using c7_pos_for_pos1[OF kklt le1] False by simp
            thus ?thesis using s_ge_3 by simp
          qed
          have c8_at_s: mt_tape c8 kk s = mt_tape c' kk s
           using c8_tape_off_window[OF s_ne_c4 s_ne_c5 s_ne_c6 s_ne_c7]
.
          show ?thesis using cM_k_at_p tc_at c8_at_s by simp
        qed
      qed
    next
      case steady
      — pos≥2: s ∈ {pos−1, pos, pos+1} in-window; other s off-window.
      show ?thesis
      proof (cases s = mt_pos c' kk − 1)
        case True
        have p_eq: ?p = (mt_pos c' kk − 2) * ?c + c_idx i + 1
        proof −
          have s − 1 = mt_pos c' kk − 2
            using True steady by linarith
          thus ?thesis by simp
        qed
        have lhs: mt_tape cM_k kk ?p = (fst (buf5 kk)) i
         using p_eq tape_cM_k_at_pos_minus_1_for_pos_ge2[OF kklt steady]
by simp
        have c8_at_s: mt_tape c8 kk s = mt_tape c8 kk (mt_pos c' kk − 1)
          using True by simp
        have c8_at_addr: mt_tape c8 kk (mt_pos c' kk − 1) = fst (buf5 kk)
```

```
    using c8_tape_at_pos_minus_1_for_pos_ge2[OF kklt steady] .
  show ?thesis using lhs c8_at_s c8_at_addr by simp
next
  case ne_pm1: False
  show ?thesis
  proof (cases s = mt_pos c' kk)
    case True
    have p_eq: ?p = (mt_pos c' kk - 1) * ?c + c_idx i + 1
      using True steady by simp
    have lhs: mt_tape cM_k kk ?p = (fst (snd (buf5 kk))) i
     using p_eq tape_cM_k_at_pos_for_pos_ge2[OF kklt steady] by simp
    have c8_at_s: mt_tape c8 kk s = mt_tape c8 kk (mt_pos c' kk)
      using True by simp
    have c8_at_addr: mt_tape c8 kk (mt_pos c' kk) = fst (snd (buf5 kk))
      using c8_tape_at_pos_for_pos_ge2[OF kklt steady] .
    show ?thesis using lhs c8_at_s c8_at_addr by simp
  next
    case ne_pos: False
    show ?thesis
    proof (cases s = mt_pos c' kk + 1)
      case True
      have p_eq: ?p = mt_pos c' kk * ?c + c_idx i + 1
        using True by simp
      have lhs: mt_tape cM_k kk ?p = (snd (snd (buf5 kk))) i
      using p_eq tape_cM_k_at_pos_plus_1_for_pos_ge2[OF kklt steady]
        by simp
      have c8_at_s: mt_tape c8 kk s = mt_tape c8 kk (mt_pos c' kk + 1)
        using True by simp
      have c8_at_addr:
          mt_tape c8 kk (mt_pos c' kk + 1) = snd (snd (buf5 kk))
        using c8_tape_at_pos_plus_1_for_pos_ge2[OF kklt steady] .
      show ?thesis using lhs c8_at_s c8_at_addr by simp
    next
      case ne_pp1: False
      — s off-window for steady: s ∉ {pos−1, pos, pos+1}.
      have s_outside: s + 1 < mt_pos c' kk ∨ s > mt_pos c' kk + 1
        using ne_pm1 ne_pos ne_pp1 steady by linarith
      have p_outside:
          ?p < (mt_pos c' kk - 2) * ?c + 1
            ∨ ?p ≥ (mt_pos c' kk - 2) * ?c + 1 + 3 * ?c
      proof -
        from s_outside show ?thesis
        proof
          assume s_lt: s + 1 < mt_pos c' kk
          have pos_ge3: mt_pos c' kk ≥ 3 using s_lt s_ge_1 by linarith
          have s_sub_le: s - 1 ≤ mt_pos c' kk - 3
            using s_lt s_ge_1 by linarith
          have (s - 1) * ?c ≤ (mt_pos c' kk - 3) * ?c
            using s_sub_le by (simp add: mult_le_mono1)
```

```
    hence p_le: ?p ≤ (mt_pos c' kk − 3) * ?c + c_idx i + 1
      by linarith
    have c_split: (mt_pos c' kk − 2) * ?c
                = (mt_pos c' kk − 3) * ?c + ?c
      using pos_ge3 by (simp add: algebra_simps diff_mult_distrib)
    have ?p < (mt_pos c' kk − 2) * ?c + 1
      using p_le c_split c_idx_lt by linarith
    thus ?thesis ..
  next
    assume s_gt: s > mt_pos c' kk + 1
    have s_sub_ge: s − 1 ≥ mt_pos c' kk + 1
      using s_gt by linarith
    have (s − 1) * ?c ≥ (mt_pos c' kk + 1) * ?c
      using s_sub_ge by (rule mult_le_mono1)
    hence p_ge: ?p ≥ (mt_pos c' kk + 1) * ?c + 1
      by linarith
    have c_split: (mt_pos c' kk + 1) * ?c
                = (mt_pos c' kk − 2) * ?c + 3 * ?c
      using steady by (simp add: algebra_simps diff_mult_distrib)
    have ?p ≥ (mt_pos c' kk − 2) * ?c + 1 + 3 * ?c
      using p_ge c_split by linarith
    thus ?thesis ..
  qed
qed
have cM_k_at_p: mt_tape cM_k kk ?p = mt_tape cM kk ?p
  using m_tape_off_window_steady[OF kklt steady p_outside] .
have tc_at: mt_tape cM kk ?p = mt_tape c' kk s i
  using tape_corr[rule_format, OF kklt] s_ge_1
  unfolding ae_tape_correspondence_def by simp
have s_ne_c4: s ≠ mt_pos c4 kk
  using c4_kk_eq ne_pos by simp
have s_ne_c5: s ≠ mt_pos c5 kk
proof (cases dest5 kk = AE_Left)
  case True
  have mt_pos c5 kk = mt_pos c' kk + 1
    using c5_pos_for_pos_ge2_dest_left[OF kklt steady True] .
  thus ?thesis using ne_pp1 by simp
next
  case False
  have mt_pos c5 kk = mt_pos c' kk − 1
    using c5_pos_for_pos_ge2[OF kklt steady False] .
  thus ?thesis using ne_pm1 by simp
qed
have s_ne_c6: s ≠ mt_pos c6 kk
  using c6_pos_for_pos_ge2[OF kklt steady] ne_pos by simp
have s_ne_c7: s ≠ mt_pos c7 kk
proof (cases dest5 kk = AE_Left)
  case True
  have mt_pos c7 kk = mt_pos c' kk − 1
```

```
            using c7_pos_for_pos_ge2[OF kklt steady] True by simp
          thus ?thesis using ne_pm1 by simp
        next
          case False
          have mt_pos c7 kk = mt_pos c' kk + 1
            using c7_pos_for_pos_ge2[OF kklt steady] False by simp
          thus ?thesis using ne_pp1 by simp
        qed
        have c8_at_s: mt_tape c8 kk s = mt_tape c' kk s
        using c8_tape_off_window[OF s_ne_c4 s_ne_c5 s_ne_c6 s_ne_c7]
.
          show ?thesis using cM_k_at_p tc_at c8_at_s by simp
      qed
    qed
  qed
 qed
qed
```

Reconstruction leaf of *ae_simulates_forward_stage_general*: the in-window *c8* value at block 2 for an le1 tape (*mt_pos c′ kk = 1*). Uniformly *c8@2 = rr* (the right buffer slot). Three-way *dest5* case-split: for *AE_Left* SS6 writes *r* at *c5_pos = 2*; for *AE_Home*/*AE_Right* SS8 writes *r* at *c7_pos = 2*; the other substeps are off cell 2.

```
lemma ae_fwd_c8_at_two_for_pos1:
 fixes M :: ('q, 'a) mttm
   and c' c4 c5 c6 c7 c8 :: ('c :: enum ⇒ 'a,
          'q × ('a, 'c) ae_stage) mt_config
   and q5 q6 :: 'q
   and ofs5 ofs6 :: nat ⇒ 'c
   and buf5 buf6 :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
   and dest5 dest6 :: nat ⇒ ae_dest
 assumes step5: (c4, c5) ∈ mttm_step (alphabet_enlarge_delta M)
     and c5_state: mt_state c5 = (q5, ofs5, buf5, dest5, SS6)
     and tape_c4_eq_c': mt_tape c4 = mt_tape c'
     and buf5_not_le_per_tape:
         ∀k<k_tm M. (mt_pos c' k = 0
               ⟶ fst (buf5 k) ≠ LE_block (le_tm M)
               ∧ snd (snd (buf5 k)) ≠ LE_block (le_tm M))
             ∧ (mt_pos c' k = 1
               ⟶ fst (snd (buf5 k)) ≠ LE_block (le_tm M)
               ∧ snd (snd (buf5 k)) ≠ LE_block (le_tm M))
             ∧ (mt_pos c' k ≥ 2
               ⟶ fst (buf5 k) ≠ LE_block (le_tm M)
               ∧ fst (snd (buf5 k)) ≠ LE_block (le_tm M)
               ∧ snd (snd (buf5 k)) ≠ LE_block (le_tm M))
     and right_not_le_c':
         ∀k. mt_tape c' k (mt_pos c' k + 1) ≠ LE_block (le_tm M)
```

```
    and c4_pos: ⋀kk. kk < k_tm M ⟹ mt_pos c4 kk = mt_pos c' kk
    and c5_pos_for_pos1:
        ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 1 ⟹ dest5 kk ≠ AE_Left
            ⟹ mt_pos c5 kk = 0
    and step6_sub: (c5, c6) ∈ mttm_step (ae_delta_ss6_ss7 M)
    and step6: (c5, c6) ∈ mttm_step (alphabet_enlarge_delta M)
    and step7: (c6, c7) ∈ mttm_step (alphabet_enlarge_delta M)
    and c7_state: mt_state c7 = (q6, ofs6, buf6, dest6, SS8)
    and buf6_eq_buf5: buf6 = buf5
    and dest6_eq_dest5: dest6 = dest5
    and c5_pos_for_pos1_dest_left:
        ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 1 ⟹ dest5 kk = AE_Left
            ⟹ mt_pos c5 kk = 2
    and c7_pos_for_pos1_dest_left:
        ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 1 ⟹ dest5 kk = AE_Left
            ⟹ mt_pos c7 kk = 0
    and step8_sub: (c7, c8) ∈ mttm_step (ae_delta_ss8_ss1 M)
    and step8: (c7, c8) ∈ mttm_step (alphabet_enlarge_delta M)
    and c6_pos_for_pos1:
        ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 1 ⟹ mt_pos c6 kk = 1
    and c7_pos_for_pos1:
        ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 1
            ⟹ mt_pos c7 kk = (if dest5 kk = AE_Left then 0 else 2)
  shows ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 1
        ⟹ mt_tape c8 kk 2 = snd (snd (buf5 kk))
proof −
  fix kk
  assume kklt: kk < k_tm M
  assume pos_kk: mt_pos c' kk = 1
    obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
      using prod.exhaust by metis
    have buf6_kk: buf6 kk = (ll, hh, rr)
      using buf5_kk buf6_eq_buf5 by simp
    have h_ne_LE: hh ≠ LE_block (le_tm M)
    proof −
      have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
        using buf5_not_le_per_tape[rule_format, OF kklt] pos_kk by blast
      thus ?thesis using buf5_kk by simp
    qed
    have two_ne_c4: (2 :: nat) ≠ mt_pos c4 kk
      using c4_pos kklt pos_kk by simp
    have c4_at_2_ne_LE: mt_tape c4 kk 2 ≠ LE_block (le_tm M)
    proof −
      have mt_tape c4 kk 2 = mt_tape c' kk 2 using tape_c4_eq_c' by simp
      moreover have mt_tape c' kk (mt_pos c' kk + 1) ≠ LE_block (le_tm M)
        using right_not_le_c' by blast
      ultimately have mt_tape c4 kk 2 = mt_tape c' kk 2
                    ∧ mt_tape c' kk 2 ≠ LE_block (le_tm M)
        using pos_kk by (simp add: numeral_2_eq_2)
```

```
    thus ?thesis by simp
  qed
  have c5_at_2: mt_tape c5 kk 2 = mt_tape c4 kk 2
    using mttm_step_tape_off_head[OF step5 two_ne_c4] .
  show mt_tape c8 kk 2 = snd (snd (buf5 kk))
  proof (cases dest5 kk)
    case AE_Left
    — SS6 writes r at c5_pos = 2; SS7/SS8 off cell 2.
    have c5_pos_kk: mt_pos c5 kk = 2
      using c5_pos_for_pos1_dest_left[OF kklt pos_kk AE_Left] .
    obtain qq tts nn qq' aa6 dr6 where
        c5_eq6: c5 = Config_M qq tts nn
      and c6_eq: c6 = Config_M qq'
                      (λk. (tts k)(nn k := aa6 k))
                      (λk. go_dir (dr6 k) (nn k))
      and tr_in: (qq, λk. tts k (nn k), qq', aa6, dr6)
                 ∈ ae_delta_ss6_ss7 M
      using step6_sub by (auto elim: mttm_step.cases)
    obtain q ofs buf dest where
      qq_eq: qq = (q, ofs, buf, dest, SS6)
      and aa6_eq:
        aa6 = (λk. if k < k_tm M then fst (ae_ss6_action (le_tm M)
                    (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
      using tr_in by (auto simp: ae_delta_ss6_ss7_def)
    have qq_state: qq = (q5, ofs5, buf5, dest5, SS6)
      using c5_state c5_eq6 by simp
    have buf_eq: buf = buf5 using qq_eq qq_state by simp
    have dest_eq: dest = dest5 using qq_eq qq_state by simp
    have nn_kk: nn kk = mt_pos c5 kk using c5_eq6 by simp
    have nn_kk_2: nn kk = 2 using nn_kk c5_pos_kk by simp
    have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
    proof −
      have tts kk (nn kk) = mt_tape c5 kk (mt_pos c5 kk)
        using c5_eq6 nn_kk by simp
      also have ... = mt_tape c5 kk 2 using c5_pos_kk by simp
      also have ... = mt_tape c4 kk 2 using c5_at_2 .
      finally have tts kk (nn kk) = mt_tape c4 kk 2 .
      thus ?thesis using c4_at_2_ne_LE by simp
    qed
    have aa6_kk: aa6 kk = rr
     using aa6_eq buf_eq dest_eq AE_Left buf5_kk read_ne_LE h_ne_LE kklt
by simp
    have c6_at_2: mt_tape c6 kk 2 = rr
    proof −
      have mt_tape c6 kk 2 = aa6 kk using c6_eq nn_kk_2 by simp
      thus ?thesis using aa6_kk by simp
    qed
    have two_ne_c6_pos: (2 :: nat) ≠ mt_pos c6 kk
      using c6_pos_for_pos1[OF kklt pos_kk] by simp
```

```
  have c7_at_2: mt_tape c7 kk 2 = mt_tape c6 kk 2
    using mttm_step_tape_off_head[OF step7 two_ne_c6_pos] .
  have c7_pos_kk: mt_pos c7 kk = 0
    using c7_pos_for_pos1_dest_left[OF kklt pos_kk AE_Left] .
  have two_ne_c7_pos: (2 :: nat) ≠ mt_pos c7 kk
    using c7_pos_kk by simp
  have c8_at_2: mt_tape c8 kk 2 = mt_tape c7 kk 2
    using mttm_step_tape_off_head[OF step8 two_ne_c7_pos] .
  have mt_tape c8 kk 2 = mt_tape c7 kk 2 using c8_at_2 .
  also have ... = mt_tape c6 kk 2 using c7_at_2 .
  also have ... = rr using c6_at_2 .
  also have ... = snd (snd (buf5 kk)) using buf5_kk by simp
  finally show ?thesis .
next
  case AE_Home
  — SS6/SS7 off cell 2; SS8 writes r at c7_pos = 2.
  have dest_ne_left: dest5 kk ≠ AE_Left using AE_Home by simp
  have c5_pos_kk: mt_pos c5 kk = 0
    using c5_pos_for_pos1[OF kklt pos_kk dest_ne_left] .
  have two_ne_c5_pos: (2 :: nat) ≠ mt_pos c5 kk
    using c5_pos_kk by simp
  have c6_at_2_eq_c4: mt_tape c6 kk 2 = mt_tape c4 kk 2
  proof −
    have mt_tape c6 kk 2 = mt_tape c5 kk 2
      using mttm_step_tape_off_head[OF step6 two_ne_c5_pos] .
    also have ... = mt_tape c4 kk 2 using c5_at_2 .
    finally show ?thesis .
  qed
  have two_ne_c6_pos: (2 :: nat) ≠ mt_pos c6 kk
    using c6_pos_for_pos1[OF kklt pos_kk] by simp
  have c7_at_2_eq_c4: mt_tape c7 kk 2 = mt_tape c4 kk 2
  proof −
    have mt_tape c7 kk 2 = mt_tape c6 kk 2
      using mttm_step_tape_off_head[OF step7 two_ne_c6_pos] .
    also have ... = mt_tape c4 kk 2 using c6_at_2_eq_c4 .
    finally show ?thesis .
  qed
  have c7_pos_kk: mt_pos c7 kk = 2
    using c7_pos_for_pos1[OF kklt pos_kk] AE_Home by simp
  obtain qq tts nn qq' aa8 dr8 where
      c7_eq8: c7 = Config_M qq tts nn
    and c8_eq: c8 = Config_M qq'
                    (λk. (tts k)(nn k := aa8 k))
                    (λk. go_dir (dr8 k) (nn k))
    and tr_in: (qq, λk. tts k (nn k), qq', aa8, dr8)
               ∈ ae_delta_ss8_ss1 M
    using step8_sub by (auto elim: mttm_step.cases)
  obtain q ofs buf dest where
      qq_eq: qq = (q, ofs, buf, dest, SS8)
```

```
      and aa8_eq:
          aa8 = (λk. if k < k_tm M then fst (ae_ss8_action (le_tm M)
                              (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
        using tr_in by (auto simp: ae_delta_ss8_ss1_def)
      have qq_state: qq = (q6, ofs6, buf6, dest6, SS8)
        using c7_state c7_eq8 by simp
      have buf_eq: buf = buf6 using qq_eq qq_state by simp
      have dest_eq: dest = dest6 using qq_eq qq_state by simp
      have nn_kk: nn kk = mt_pos c7 kk using c7_eq8 by simp
      have nn_kk_2: nn kk = 2 using nn_kk c7_pos_kk by simp
      have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
      proof −
        have tts kk (nn kk) = mt_tape c7 kk (mt_pos c7 kk)
          using c7_eq8 nn_kk by simp
        also have ... = mt_tape c7 kk 2 using c7_pos_kk by simp
        also have ... = mt_tape c4 kk 2 using c7_at_2_eq_c4 .
        finally have tts kk (nn kk) = mt_tape c4 kk 2 .
        thus ?thesis using c4_at_2_ne_LE by simp
      qed
      have dest6_AE_Home: dest6 kk = AE_Home
        using AE_Home dest6_eq_dest5 by simp
      have aa8_kk: aa8 kk = rr
            using aa8_eq buf_eq dest_eq dest6_AE_Home buf6_kk h_ne_LE
read_ne_LE kklt
        by simp
      have mt_tape c8 kk 2 = aa8 kk using c8_eq nn_kk_2 by simp
      also have ... = rr using aa8_kk .
      also have ... = snd (snd (buf5 kk)) using buf5_kk by simp
      finally show ?thesis .
    next
      case AE_Right
      — SS6/SS7 off cell 2; SS8 writes r at c7_pos = 2.
      have dest_ne_left: dest5 kk ≠ AE_Left using AE_Right by simp
      have c5_pos_kk: mt_pos c5 kk = 0
        using c5_pos_for_pos1[OF kklt pos_kk dest_ne_left] .
      have two_ne_c5_pos: (2 :: nat) ≠ mt_pos c5 kk
        using c5_pos_kk by simp
      have c6_at_2_eq_c4: mt_tape c6 kk 2 = mt_tape c4 kk 2
      proof −
        have mt_tape c6 kk 2 = mt_tape c5 kk 2
          using mttm_step_tape_off_head[OF step6 two_ne_c5_pos] .
        also have ... = mt_tape c4 kk 2 using c5_at_2 .
        finally show ?thesis .
      qed
      have two_ne_c6_pos: (2 :: nat) ≠ mt_pos c6 kk
        using c6_pos_for_pos1[OF kklt pos_kk] by simp
      have c7_at_2_eq_c4: mt_tape c7 kk 2 = mt_tape c4 kk 2
      proof −
        have mt_tape c7 kk 2 = mt_tape c6 kk 2
```

```
      using mttm_step_tape_off_head[OF step7 two_ne_c6_pos] .
    also have ... = mt_tape c4 kk 2 using c6_at_2_eq_c4 .
    finally show ?thesis .
  qed
  have c7_pos_kk: mt_pos c7 kk = 2
    using c7_pos_for_pos1[OF kklt pos_kk] AE_Right by simp
  obtain qq tts nn qq' aa8 dr8 where
      c7_eq8: c7 = Config_M qq tts nn
    and c8_eq: c8 = Config_M qq'
                    (λk. (tts k)(nn k := aa8 k))
                    (λk. go_dir (dr8 k) (nn k))
    and tr_in: (qq, λk. tts k (nn k), qq', aa8, dr8)
               ∈ ae_delta_ss8_ss1 M
    using step8_sub by (auto elim: mttm_step.cases)
  obtain q ofs buf dest where
      qq_eq: qq = (q, ofs, buf, dest, SS8)
    and aa8_eq:
      aa8 = (λk. if k < k_tm M then fst (ae_ss8_action (le_tm M)
                  (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
    using tr_in by (auto simp: ae_delta_ss8_ss1_def)
  have qq_state: qq = (q6, ofs6, buf6, dest6, SS8)
    using c7_state c7_eq8 by simp
  have buf_eq: buf = buf6 using qq_eq qq_state by simp
  have dest_eq: dest = dest6 using qq_eq qq_state by simp
  have nn_kk: nn kk = mt_pos c7 kk using c7_eq8 by simp
  have nn_kk_2: nn kk = 2 using nn_kk c7_pos_kk by simp
  have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
  proof −
    have tts kk (nn kk) = mt_tape c7 kk (mt_pos c7 kk)
      using c7_eq8 nn_kk by simp
    also have ... = mt_tape c7 kk 2 using c7_pos_kk by simp
    also have ... = mt_tape c4 kk 2 using c7_at_2_eq_c4 .
    finally have tts kk (nn kk) = mt_tape c4 kk 2 .
    thus ?thesis using c4_at_2_ne_LE by simp
  qed
  have dest6_AE_Right: dest6 kk = AE_Right
    using AE_Right dest6_eq_dest5 by simp
  have aa8_kk: aa8 kk = rr
        using aa8_eq buf_eq dest_eq dest6_AE_Right buf6_kk h_ne_LE
read_ne_LE kklt
    by simp
  have mt_tape c8 kk 2 = aa8 kk using c8_eq nn_kk_2 by simp
  also have ... = rr using aa8_kk .
  also have ... = snd (snd (buf5 kk)) using buf5_kk by simp
  finally show ?thesis .
 qed
qed
```

Reconstruction leaf of *ae_simulates_forward_stage_general*: the in-window *c8* value at block *pos + 1* for a steady tape (*mt_pos c′ kk* ≥ *2*). Uni-

formly *c8@(pos+1) = rr* (the right buffer slot). Three-way *dest5* case-split: for *AE_Left* SS6 writes *r* at *c5_pos = pos + 1*; for *AE_Home/AE_Right* SS8 writes *r* at *c7_pos = pos + 1*; the other substeps are off cell *pos + 1*.

lemma *ae_fwd_c8_at_pos_plus_1_ge2*:
fixes *M* :: ('*q*, '*a*) *mttm*
and *c' c4 c5 c6 c7 c8* :: ('*c* :: *enum* ⇒ '*a*,
'*q* × ('*a*, '*c*) *ae_stage*) *mt_config*
and *q5 q6* :: '*q*
and *ofs5 ofs6* :: *nat* ⇒ '*c*
and *buf5 buf6* :: *nat* ⇒ ('*c* ⇒ '*a*) × ('*c* ⇒ '*a*) × ('*c* ⇒ '*a*)
and *dest5 dest6* :: *nat* ⇒ *ae_dest*
assumes *step5*: (*c4*, *c5*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *c5_state*: *mt_state c5* = (*q5*, *ofs5*, *buf5*, *dest5*, *SS6*)
and *tape_c4_eq_c'*: *mt_tape c4* = *mt_tape c'*
and *buf5_not_le_per_tape*:
∀ *k*<*k_tm M*. (*mt_pos c' k* = *0*
⟶ *fst* (*buf5 k*) ≠ *LE_block* (*le_tm M*)
∧ *snd* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*))
∧ (*mt_pos c' k* = *1*
⟶ *fst* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*)
∧ *snd* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*))
∧ (*mt_pos c' k* ≥ *2*
⟶ *fst* (*buf5 k*) ≠ *LE_block* (*le_tm M*)
∧ *fst* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*)
∧ *snd* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*))
and *right_not_le_c'*:
∀ *k*. *mt_tape c' k* (*mt_pos c' k* + *1*) ≠ *LE_block* (*le_tm M*)
and *c4_pos*: ⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c4 kk* = *mt_pos c' kk*
and *step6_sub*: (*c5*, *c6*) ∈ *mttm_step* (*ae_delta_ss6_ss7 M*)
and *step6*: (*c5*, *c6*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *step7*: (*c6*, *c7*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *c7_state*: *mt_state c7* = (*q6*, *ofs6*, *buf6*, *dest6*, *SS8*)
and *buf6_eq_buf5*: *buf6* = *buf5*
and *dest6_eq_dest5*: *dest6* = *dest5*
and *step8_sub*: (*c7*, *c8*) ∈ *mttm_step* (*ae_delta_ss8_ss1 M*)
and *step8*: (*c7*, *c8*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *c5_pos_for_pos_ge2*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* ≥ *2* ⟹ *dest5 kk* ≠ *AE_Left*
⟹ *mt_pos c5 kk* = *mt_pos c' kk* − *1*
and *c5_pos_for_pos_ge2_dest_left*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* ≥ *2* ⟹ *dest5 kk* = *AE_Left*
⟹ *mt_pos c5 kk* = *mt_pos c' kk* + *1*
and *c6_pos_for_pos_ge2*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* ≥ *2* ⟹ *mt_pos c6 kk* = *mt_pos c' kk*
and *c7_pos_for_pos_ge2*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* ≥ *2*
⟹ *mt_pos c7 kk* = (*if dest5 kk* = *AE_Left*
then mt_pos c' kk − *1*

```
            else mt_pos c' kk + 1)
  shows ⋀kk. kk < k_tm M ⟹ mt_pos c' kk ≥ 2
        ⟹ mt_tape c8 kk (mt_pos c' kk + 1) = snd (snd (buf5 kk))
proof −
  fix kk
  assume kklt: kk < k_tm M
  assume hge2: mt_pos c' kk ≥ 2
    obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
      using prod.exhaust by metis
    have buf6_kk: buf6 kk = (ll, hh, rr)
      using buf5_kk buf6_eq_buf5 by simp
    have h_ne_LE: hh ≠ LE_block (le_tm M)
    proof −
      have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
        using buf5_not_le_per_tape[rule_format, OF kklt] hge2 by blast
      thus ?thesis using buf5_kk by simp
    qed
    have c4_kk_eq: mt_pos c4 kk = mt_pos c' kk using c4_pos kklt by simp
    have pp1_ne_c4: (mt_pos c' kk + 1) ≠ mt_pos c4 kk
      using c4_kk_eq by linarith
    have c4_at_pp1_ne_LE:
       mt_tape c4 kk (mt_pos c' kk + 1) ≠ LE_block (le_tm M)
    proof −
     have mt_tape c4 kk (mt_pos c' kk + 1) = mt_tape c' kk (mt_pos c' kk + 1)
       using tape_c4_eq_c' by simp
      moreover have mt_tape c' kk (mt_pos c' kk + 1) ≠ LE_block (le_tm M)
       using right_not_le_c' by blast
      ultimately show ?thesis by simp
    qed
    have c5_at_pp1_eq_c4:
       mt_tape c5 kk (mt_pos c' kk + 1) = mt_tape c4 kk (mt_pos c' kk + 1)
      using mttm_step_tape_off_head[OF step5 pp1_ne_c4] .
    show mt_tape c8 kk (mt_pos c' kk + 1) = snd (snd (buf5 kk))
    proof (cases dest5 kk)
      case AE_Left
      — SS6 writes r at c5_pos = pos + 1; SS7/SS8 off pos+1.
      have c5_pos_kk: mt_pos c5 kk = mt_pos c' kk + 1
        using c5_pos_for_pos_ge2_dest_left[OF kklt hge2 AE_Left] .
      obtain qq tts nn qq' aa6 dr6 where
         c5_eq6: c5 = Config_M qq tts nn
        and c6_eq: c6 = Config_M qq'
                        (λk. (tts k)(nn k := aa6 k))
                        (λk. go_dir (dr6 k) (nn k))
        and tr_in: (qq, λk. tts k (nn k), qq', aa6, dr6)
                   ∈ ae_delta_ss6_ss7 M
        using step6_sub by (auto elim: mttm_step.cases)
      obtain q ofs buf dest where
        qq_eq: qq = (q, ofs, buf, dest, SS6)
        and aa6_eq:
```

```
        aa6 = (λk. if k < k_tm M then fst (ae_ss6_action (le_tm M)
                              (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
    using tr_in by (auto simp: ae_delta_ss6_ss7_def)
  have qq_state: qq = (q5, ofs5, buf5, dest5, SS6)
    using c5_state c5_eq6 by simp
  have buf_eq: buf = buf5 using qq_eq qq_state by simp
  have dest_eq: dest = dest5 using qq_eq qq_state by simp
  have nn_kk: nn kk = mt_pos c5 kk using c5_eq6 by simp
   have nn_kk_val: nn kk = mt_pos c' kk + 1 using nn_kk c5_pos_kk by
simp
  have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
  proof −
    have tts kk (nn kk) = mt_tape c5 kk (mt_pos c5 kk)
      using c5_eq6 nn_kk by simp
     also have ... = mt_tape c5 kk (mt_pos c' kk + 1) using c5_pos_kk by
simp
   also have ... = mt_tape c4 kk (mt_pos c' kk + 1) using c5_at_pp1_eq_c4
.
    finally have tts kk (nn kk) = mt_tape c4 kk (mt_pos c' kk + 1) .
    thus ?thesis using c4_at_pp1_ne_LE by simp
  qed
  have aa6_kk: aa6 kk = rr
   using aa6_eq buf_eq dest_eq AE_Left buf5_kk read_ne_LE h_ne_LE kklt
by simp
  have c6_at_pp1: mt_tape c6 kk (mt_pos c' kk + 1) = rr
  proof −
    have mt_tape c6 kk (mt_pos c' kk + 1) = aa6 kk
      using c6_eq nn_kk_val by simp
    thus ?thesis using aa6_kk by simp
  qed
  have c6_pos_kk: mt_pos c6 kk = mt_pos c' kk
    using c6_pos_for_pos_ge2[OF kklt hge2] .
  have pp1_ne_c6: (mt_pos c' kk + 1) ≠ mt_pos c6 kk
    using c6_pos_kk by linarith
  have c7_pos_kk: mt_pos c7 kk = mt_pos c' kk − 1
    using c7_pos_for_pos_ge2[OF kklt hge2] AE_Left by simp
  have pp1_ne_c7: (mt_pos c' kk + 1) ≠ mt_pos c7 kk
    using c7_pos_kk hge2 by linarith
   have mt_tape c8 kk (mt_pos c' kk + 1) = mt_tape c7 kk (mt_pos c' kk +
1)
    using mttm_step_tape_off_head[OF step8 pp1_ne_c7] .
  also have ... = mt_tape c6 kk (mt_pos c' kk + 1)
    using mttm_step_tape_off_head[OF step7 pp1_ne_c6] .
  also have ... = rr using c6_at_pp1 .
  also have ... = snd (snd (buf5 kk)) using buf5_kk by simp
  finally show ?thesis .
next
  case AE_Home
  — SS6/SS7 off pos+1; SS8 writes r at c7_pos = pos + 1.
```

have *dest_ne_left*: *dest5 kk* ≠ *AE_Left* **using** *AE_Home* **by** *simp*
have *c5_pos_kk*: *mt_pos c5 kk* = *mt_pos c′ kk* − *1*
using *c5_pos_for_pos_ge2*[*OF kklt hge2 dest_ne_left*] .
have *pp1_ne_c5*: (*mt_pos c′ kk* + *1*) ≠ *mt_pos c5 kk*
using *c5_pos_kk hge2* **by** *linarith*
have *c6_pos_kk*: *mt_pos c6 kk* = *mt_pos c′ kk*
using *c6_pos_for_pos_ge2*[*OF kklt hge2*] .
have *pp1_ne_c6*: (*mt_pos c′ kk* + *1*) ≠ *mt_pos c6 kk*
using *c6_pos_kk* **by** *linarith*
have *c7_at_pp1_eq_c4*:
mt_tape c7 kk (*mt_pos c′ kk* + *1*) = *mt_tape c4 kk* (*mt_pos c′ kk* + *1*)
proof −
have *mt_tape c7 kk* (*mt_pos c′ kk* + *1*) = *mt_tape c6 kk* (*mt_pos c′ kk* + *1*)
using *mttm_step_tape_off_head*[*OF step7 pp1_ne_c6*] .
also have ... = *mt_tape c5 kk* (*mt_pos c′ kk* + *1*)
using *mttm_step_tape_off_head*[*OF step6 pp1_ne_c5*] .
also have ... = *mt_tape c4 kk* (*mt_pos c′ kk* + *1*) **using** *c5_at_pp1_eq_c4* .
finally show *?thesis* .
qed
have *c7_pos_kk*: *mt_pos c7 kk* = *mt_pos c′ kk* + *1*
using *c7_pos_for_pos_ge2*[*OF kklt hge2*] *AE_Home* **by** *simp*
obtain *qq tts nn qq′ aa8 dr8* **where**
c7_eq8: *c7* = *Config*$_M$ *qq tts nn*
and *c8_eq*: *c8* = *Config*$_M$ *qq′*
(λ*k*. (*tts k*)(*nn k* := *aa8 k*))
(λ*k*. *go_dir* (*dr8 k*) (*nn k*))
and *tr_in*: (*qq*, λ*k*. *tts k* (*nn k*), *qq′*, *aa8*, *dr8*)
∈ *ae_delta_ss8_ss1 M*
using *step8_sub* **by** (*auto elim*: *mttm_step.cases*)
obtain *q ofs buf dest* **where**
qq_eq: *qq* = (*q*, *ofs*, *buf*, *dest*, *SS8*)
and *aa8_eq*:
aa8 = (λ*k*. *if k* < *k_tm M then fst* (*ae_ss8_action* (*le_tm M*)
(*tts k* (*nn k*)) (*buf k*) (*dest k*)) *else bl_block* (*bl_tm M*))
using *tr_in* **by** (*auto simp*: *ae_delta_ss8_ss1_def*)
have *qq_state*: *qq* = (*q6*, *ofs6*, *buf6*, *dest6*, *SS8*)
using *c7_state c7_eq8* **by** *simp*
have *buf_eq*: *buf* = *buf6* **using** *qq_eq qq_state* **by** *simp*
have *dest_eq*: *dest* = *dest6* **using** *qq_eq qq_state* **by** *simp*
have *nn_kk*: *nn kk* = *mt_pos c7 kk* **using** *c7_eq8* **by** *simp*
have *nn_kk_val*: *nn kk* = *mt_pos c′ kk* + *1* **using** *nn_kk c7_pos_kk* **by** *simp*
have *read_ne_LE*: *tts kk* (*nn kk*) ≠ *LE_block* (*le_tm M*)
proof −
have *tts kk* (*nn kk*) = *mt_tape c7 kk* (*mt_pos c7 kk*)
using *c7_eq8 nn_kk* **by** *simp*
also have ... = *mt_tape c7 kk* (*mt_pos c′ kk* + *1*) **using** *c7_pos_kk* **by**

```
simp
      also have ... = mt_tape c4 kk (mt_pos c' kk + 1) using c7_at_pp1_eq_c4
.
       finally have tts kk (nn kk) = mt_tape c4 kk (mt_pos c' kk + 1) .
       thus ?thesis using c4_at_pp1_ne_LE by simp
      qed
      have dest6_AE_Home: dest6 kk = AE_Home
       using AE_Home dest6_eq_dest5 by simp
      have aa8_kk: aa8 kk = rr
            using aa8_eq buf_eq dest_eq dest6_AE_Home buf6_kk h_ne_LE
read_ne_LE kklt
       by simp
      have mt_tape c8 kk (mt_pos c' kk + 1) = aa8 kk
       using c8_eq nn_kk_val by simp
      also have ... = rr using aa8_kk .
      also have ... = snd (snd (buf5 kk)) using buf5_kk by simp
      finally show ?thesis .
    next
      case AE_Right
      — SS6/SS7 off pos+1; SS8 writes r at c7_pos = pos + 1.
      have dest_ne_left: dest5 kk ≠ AE_Left using AE_Right by simp
      have c5_pos_kk: mt_pos c5 kk = mt_pos c' kk − 1
       using c5_pos_for_pos_ge2[OF kklt hge2 dest_ne_left] .
      have pp1_ne_c5: (mt_pos c' kk + 1) ≠ mt_pos c5 kk
       using c5_pos_kk hge2 by linarith
      have c6_pos_kk: mt_pos c6 kk = mt_pos c' kk
       using c6_pos_for_pos_ge2[OF kklt hge2] .
      have pp1_ne_c6: (mt_pos c' kk + 1) ≠ mt_pos c6 kk
       using c6_pos_kk by linarith
      have c7_at_pp1_eq_c4:
         mt_tape c7 kk (mt_pos c' kk + 1) = mt_tape c4 kk (mt_pos c' kk + 1)
      proof −
       have mt_tape c7 kk (mt_pos c' kk + 1) = mt_tape c6 kk (mt_pos c' kk +
1)
         using mttm_step_tape_off_head[OF step7 pp1_ne_c6] .
       also have ... = mt_tape c5 kk (mt_pos c' kk + 1)
         using mttm_step_tape_off_head[OF step6 pp1_ne_c5] .
      also have ... = mt_tape c4 kk (mt_pos c' kk + 1) using c5_at_pp1_eq_c4
.
       finally show ?thesis .
      qed
      have c7_pos_kk: mt_pos c7 kk = mt_pos c' kk + 1
       using c7_pos_for_pos_ge2[OF kklt hge2] AE_Right by simp
      obtain qq tts nn qq' aa8 dr8 where
         c7_eq8: c7 = Config_M qq tts nn
       and c8_eq: c8 = Config_M qq'
                         (λk. (tts k)(nn k := aa8 k))
                         (λk. go_dir (dr8 k) (nn k))
       and tr_in: (qq, λk. tts k (nn k), qq', aa8, dr8)
```

```
                ∈ ae_delta_ss8_ss1 M
      using step8_sub by (auto elim: mttm_step.cases)
    obtain q ofs buf dest where
      qq_eq: qq = (q, ofs, buf, dest, SS8)
      and aa8_eq:
        aa8 = (λk. if k < k_tm M then fst (ae_ss8_action (le_tm M)
                    (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
      using tr_in by (auto simp: ae_delta_ss8_ss1_def)
    have qq_state: qq = (q6, ofs6, buf6, dest6, SS8)
      using c7_state c7_eq8 by simp
    have buf_eq: buf = buf6 using qq_eq qq_state by simp
    have dest_eq: dest = dest6 using qq_eq qq_state by simp
    have nn_kk: nn kk = mt_pos c7 kk using c7_eq8 by simp
     have nn_kk_val: nn kk = mt_pos c' kk + 1 using nn_kk c7_pos_kk by
simp
    have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
    proof −
      have tts kk (nn kk) = mt_tape c7 kk (mt_pos c7 kk)
        using c7_eq8 nn_kk by simp
       also have ... = mt_tape c7 kk (mt_pos c' kk + 1) using c7_pos_kk by
simp
     also have ... = mt_tape c4 kk (mt_pos c' kk + 1) using c7_at_pp1_eq_c4
.
      finally have tts kk (nn kk) = mt_tape c4 kk (mt_pos c' kk + 1) .
      thus ?thesis using c4_at_pp1_ne_LE by simp
    qed
    have dest6_AE_Right: dest6 kk = AE_Right
      using AE_Right dest6_eq_dest5 by simp
    have aa8_kk: aa8 kk = rr
          using aa8_eq buf_eq dest_eq dest6_AE_Right buf6_kk h_ne_LE
read_ne_LE kklt
      by simp
    have mt_tape c8 kk (mt_pos c' kk + 1) = aa8 kk
      using c8_eq nn_kk_val by simp
    also have ... = rr using aa8_kk .
    also have ... = snd (snd (buf5 kk)) using buf5_kk by simp
    finally show ?thesis .
  qed
 qed
```

Reconstruction leaf of *ae_simulates_forward_stage_general*: the *c8* head position for a steady tape (*mt_pos c′ kk ≥ 2*), as a *dest5*-case displacement of *mt_pos c′ kk* (*AE_Left -> pos − 1*, *AE_Home -> pos*, *AE_Right -> pos + 1*). SS8's action reads the steady-window head cell and steps per the recorded *dest*.

```
lemma ae_fwd_c8_pos_for_pos_ge2:
 fixes M :: ('q, 'a) mttm
   and c' c4 c5 c6 c7 c8 :: ('c :: enum ⇒ 'a,
            'q × ('a, 'c) ae_stage) mt_config
```

and *q5 q6* :: *'q*
and *ofs5 ofs6* :: *nat* ⇒ *'c*
and *buf5 buf6* :: *nat* ⇒ (*'c* ⇒ *'a*) × (*'c* ⇒ *'a*) × (*'c* ⇒ *'a*)
and *dest5 dest6* :: *nat* ⇒ *ae_dest*
assumes *step5*: (*c4*, *c5*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *tape_c4_eq_c'*: *mt_tape c4* = *mt_tape c'*
and *buf5_not_le_per_tape*:
∀ *k*<*k_tm M*. (*mt_pos c' k* = *0*
⟶ *fst* (*buf5 k*) ≠ *LE_block* (*le_tm M*)
∧ *snd* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*))
∧ (*mt_pos c' k* = *1*
⟶ *fst* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*)
∧ *snd* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*))
∧ (*mt_pos c' k* ≥ *2*
⟶ *fst* (*buf5 k*) ≠ *LE_block* (*le_tm M*)
∧ *fst* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*)
∧ *snd* (*snd* (*buf5 k*)) ≠ *LE_block* (*le_tm M*))
and *right_not_le_c'*:
∀ *k*. *mt_tape c' k* (*mt_pos c' k* + *1*) ≠ *LE_block* (*le_tm M*)
and *c4_pos*: ⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c4 kk* = *mt_pos c' kk*
and *step6*: (*c5*, *c6*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *step7*: (*c6*, *c7*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *c7_state*: *mt_state c7* = (*q6*, *ofs6*, *buf6*, *dest6*, *SS8*)
and *buf6_eq_buf5*: *buf6* = *buf5*
and *dest6_eq_dest5*: *dest6* = *dest5*
and *step8_sub*: (*c7*, *c8*) ∈ *mttm_step* (*ae_delta_ss8_ss1 M*)
and *c5_pos_for_pos_ge2*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* ≥ *2* ⟹ *dest5 kk* ≠ *AE_Left*
⟹ *mt_pos c5 kk* = *mt_pos c' kk* − *1*
and *c5_pos_for_pos_ge2_dest_left*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* ≥ *2* ⟹ *dest5 kk* = *AE_Left*
⟹ *mt_pos c5 kk* = *mt_pos c' kk* + *1*
and *left_not_le_c'_steady*:
∀ *k*. *mt_pos c' k* ≥ *2*
⟶ *mt_tape c' k* (*mt_pos c' k* − *1*) ≠ *LE_block* (*le_tm M*)
and *c6_pos_for_pos_ge2*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* ≥ *2* ⟹ *mt_pos c6 kk* = *mt_pos c' kk*
and *c7_pos_for_pos_ge2*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* ≥ *2*
⟹ *mt_pos c7 kk* = (*if dest5 kk* = *AE_Left*
then mt_pos c' kk − *1*
else mt_pos c' kk + *1*)
shows ⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* ≥ *2*
⟹ *mt_pos c8 kk* = (*case dest5 kk of*
AE_Left ⇒ *mt_pos c' kk* − *1*
| *AE_Home* ⇒ *mt_pos c' kk*
| *AE_Right* ⇒ *mt_pos c' kk* + *1*)
proof −

fix *kk*
assume *kklt*: *kk* < *k_tm M*
assume *hge2*: *mt_pos c' kk* ≥ *2*
obtain *qq tts nn qq' aa8 dr8* **where**
c7_eq8: *c7* = *Config*$_M$ *qq tts nn*
and *c8_eq*: *c8* = *Config*$_M$ *qq'*
(λ*k*. (*tts k*)(*nn k* := *aa8 k*))
(λ*k*. *go_dir* (*dr8 k*) (*nn k*))
and *tr_in*: (*qq*, λ*k*. *tts k* (*nn k*), *qq'*, *aa8*, *dr8*)
∈ *ae_delta_ss8_ss1 M*
using *step8_sub* **by** (*auto elim*: *mttm_step.cases*)
obtain *q ofs buf dest* **where**
qq_eq: *qq* = (*q*, *ofs*, *buf*, *dest*, *SS8*)
and *dr8_eq*:
dr8 = (λ*k*. *if k* < *k_tm M then snd* (*ae_ss8_action* (*le_tm M*)
(*tts k* (*nn k*)) (*buf k*) (*dest k*)) *else dir.N*)
using *tr_in* **by** (*auto simp*: *ae_delta_ss8_ss1_def*)
have *qq_state*: *qq* = (*q6*, *ofs6*, *buf6*, *dest6*, *SS8*)
using *c7_state c7_eq8* **by** *simp*
have *buf_eq*: *buf* = *buf6* **using** *qq_eq qq_state* **by** *simp*
have *dest_eq*: *dest* = *dest6* **using** *qq_eq qq_state* **by** *simp*
have *nn_kk*: *nn kk* = *mt_pos c7 kk* **using** *c7_eq8* **by** *simp*
obtain *ll hh rr* **where** *buf5_kk*: *buf5 kk* = (*ll*, *hh*, *rr*)
using *prod.exhaust* **by** *metis*
have *buf6_kk*: *buf6 kk* = (*ll*, *hh*, *rr*)
using *buf5_kk buf6_eq_buf5* **by** *simp*
have *h_ne_LE*: *hh* ≠ *LE_block* (*le_tm M*)
proof −
have *fst* (*snd* (*buf5 kk*)) ≠ *LE_block* (*le_tm M*)
using *buf5_not_le_per_tape*[*rule_format*, *OF kklt*] *hge2* **by** *blast*
thus *?thesis* **using** *buf5_kk* **by** *simp*
qed
have *c4_kk*: *mt_pos c4 kk* = *mt_pos c' kk* **using** *c4_pos kklt* **by** *simp*
show *mt_pos c8 kk* = (*case dest5 kk of*
AE_Left ⇒ *mt_pos c' kk* − *1*
| *AE_Home* ⇒ *mt_pos c' kk*
| *AE_Right* ⇒ *mt_pos c' kk* + *1*)
proof (*cases dest5 kk*)
case *AE_Left*
have *c7_kk*: *mt_pos c7 kk* = *mt_pos c' kk* − *1*
using *c7_pos_for_pos_ge2*[*OF kklt hge2*] *AE_Left* **by** *simp*
have *nn_kk_val*: *nn kk* = *mt_pos c' kk* − *1* **using** *nn_kk c7_kk* **by** *simp*
have *c5_pos_kk*: *mt_pos c5 kk* = *mt_pos c' kk* + *1*
using *c5_pos_for_pos_ge2_dest_left*[*OF kklt hge2 AE_Left*] **.**
have *pm1_ne_c6*: (*mt_pos c' kk* − *1*) ≠ *mt_pos c6 kk*
using *c6_pos_for_pos_ge2*[*OF kklt hge2*] *hge2* **by** *linarith*
have *pm1_ne_c5*: (*mt_pos c' kk* − *1*) ≠ *mt_pos c5 kk*
using *c5_pos_kk hge2* **by** *linarith*
have *pm1_ne_c4*: (*mt_pos c' kk* − *1*) ≠ *mt_pos c4 kk*

```
      using c4_kk hge2 by linarith
    have c7_at_addr:
       mt_tape c7 kk (mt_pos c' kk − 1) = mt_tape c' kk (mt_pos c' kk − 1)
    proof −
     have mt_tape c7 kk (mt_pos c' kk − 1) = mt_tape c6 kk (mt_pos c' kk −
1)
       using mttm_step_tape_off_head[OF step7 pm1_ne_c6] .
     also have ... = mt_tape c5 kk (mt_pos c' kk − 1)
       using mttm_step_tape_off_head[OF step6 pm1_ne_c5] .
     also have ... = mt_tape c4 kk (mt_pos c' kk − 1)
       using mttm_step_tape_off_head[OF step5 pm1_ne_c4] .
     also have ... = mt_tape c' kk (mt_pos c' kk − 1)
       using tape_c4_eq_c' by simp
     finally show ?thesis .
    qed
    have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
    proof −
     have tts kk (nn kk) = mt_tape c7 kk (mt_pos c7 kk)
       using c7_eq8 nn_kk by simp
     also have ... = mt_tape c7 kk (mt_pos c' kk − 1)
       using c7_kk by simp
     also have ... = mt_tape c' kk (mt_pos c' kk − 1) using c7_at_addr .
     finally have read_eq:
        tts kk (nn kk) = mt_tape c' kk (mt_pos c' kk − 1) .
     have mt_tape c' kk (mt_pos c' kk − 1) ≠ LE_block (le_tm M)
       using left_not_le_c'_steady hge2 by blast
     thus ?thesis using read_eq by simp
    qed
    have dest6_AE_Left: dest6 kk = AE_Left
     using AE_Left dest6_eq_dest5 by simp
    have dr8_kk: dr8 kk = dir.N
    using dr8_eq buf_eq dest_eq dest6_AE_Left buf6_kk h_ne_LE read_ne_LE
kklt
      by simp
    have mt_pos c8 kk = go_dir (dr8 kk) (nn kk)
     using c8_eq by simp
    also have ... = nn kk using dr8_kk by simp
    also have ... = mt_pos c' kk − 1 using nn_kk_val .
    finally show ?thesis using AE_Left by simp
   next
    case AE_Home
    have c7_kk: mt_pos c7 kk = mt_pos c' kk + 1
     using c7_pos_for_pos_ge2[OF kklt hge2] AE_Home by simp
    have nn_kk_val: nn kk = mt_pos c' kk + 1 using nn_kk c7_kk by simp
    have dest_ne_left: dest5 kk ≠ AE_Left using AE_Home by simp
    have c5_pos_kk: mt_pos c5 kk = mt_pos c' kk − 1
     using c5_pos_for_pos_ge2[OF kklt hge2 dest_ne_left] .
    have pp1_ne_c6: (mt_pos c' kk + 1) ≠ mt_pos c6 kk
     using c6_pos_for_pos_ge2[OF kklt hge2] by simp
```

```
    have pp1_ne_c5: (mt_pos c' kk + 1) ≠ mt_pos c5 kk
      using c5_pos_kk hge2 by linarith
    have pp1_ne_c4: (mt_pos c' kk + 1) ≠ mt_pos c4 kk
      using c4_kk by simp
    have c7_at_addr:
       mt_tape c7 kk (mt_pos c' kk + 1) = mt_tape c' kk (mt_pos c' kk + 1)
    proof −
     have mt_tape c7 kk (mt_pos c' kk + 1) = mt_tape c6 kk (mt_pos c' kk +
1)
       using mttm_step_tape_off_head[OF step7 pp1_ne_c6] .
      also have ... = mt_tape c5 kk (mt_pos c' kk + 1)
       using mttm_step_tape_off_head[OF step6 pp1_ne_c5] .
      also have ... = mt_tape c4 kk (mt_pos c' kk + 1)
       using mttm_step_tape_off_head[OF step5 pp1_ne_c4] .
      also have ... = mt_tape c' kk (mt_pos c' kk + 1)
       using tape_c4_eq_c' by simp
      finally show ?thesis .
    qed
    have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
    proof −
      have tts kk (nn kk) = mt_tape c7 kk (mt_pos c7 kk)
       using c7_eq8 nn_kk by simp
      also have ... = mt_tape c7 kk (mt_pos c' kk + 1)
       using c7_kk by simp
      also have ... = mt_tape c' kk (mt_pos c' kk + 1) using c7_at_addr .
      finally have read_eq:
         tts kk (nn kk) = mt_tape c' kk (mt_pos c' kk + 1) .
      have mt_tape c' kk (mt_pos c' kk + 1) ≠ LE_block (le_tm M)
       using right_not_le_c' by blast
      thus ?thesis using read_eq by simp
    qed
    have dest6_AE_Home: dest6 kk = AE_Home
      using AE_Home dest6_eq_dest5 by simp
    have dr8_kk: dr8 kk = dir.L
          using dr8_eq buf_eq dest_eq dest6_AE_Home buf6_kk h_ne_LE
read_ne_LE kklt
      by simp
    have mt_pos c8 kk = go_dir (dr8 kk) (nn kk)
      using c8_eq by simp
    also have ... = (nn kk) − 1 using dr8_kk by simp
    also have ... = (mt_pos c' kk + 1) − 1 using nn_kk_val by simp
    also have ... = mt_pos c' kk by simp
    finally show ?thesis using AE_Home by simp
  next
    case AE_Right
    have c7_kk: mt_pos c7 kk = mt_pos c' kk + 1
      using c7_pos_for_pos_ge2[OF kklt hge2] AE_Right by simp
    have nn_kk_val: nn kk = mt_pos c' kk + 1 using nn_kk c7_kk by simp
    have dest_ne_left: dest5 kk ≠ AE_Left using AE_Right by simp
```

```
    have c5_pos_kk: mt_pos c5 kk = mt_pos c' kk − 1
      using c5_pos_for_pos_ge2[OF kklt hge2 dest_ne_left] .
    have pp1_ne_c6: (mt_pos c' kk + 1) ≠ mt_pos c6 kk
      using c6_pos_for_pos_ge2[OF kklt hge2] by simp
    have pp1_ne_c5: (mt_pos c' kk + 1) ≠ mt_pos c5 kk
      using c5_pos_kk hge2 by linarith
    have pp1_ne_c4: (mt_pos c' kk + 1) ≠ mt_pos c4 kk
      using c4_kk by simp
    have c7_at_addr:
       mt_tape c7 kk (mt_pos c' kk + 1) = mt_tape c' kk (mt_pos c' kk + 1)
    proof −
     have mt_tape c7 kk (mt_pos c' kk + 1) = mt_tape c6 kk (mt_pos c' kk +
1)
       using mttm_step_tape_off_head[OF step7 pp1_ne_c6] .
     also have ... = mt_tape c5 kk (mt_pos c' kk + 1)
       using mttm_step_tape_off_head[OF step6 pp1_ne_c5] .
     also have ... = mt_tape c4 kk (mt_pos c' kk + 1)
       using mttm_step_tape_off_head[OF step5 pp1_ne_c4] .
     also have ... = mt_tape c' kk (mt_pos c' kk + 1)
       using tape_c4_eq_c' by simp
     finally show ?thesis .
    qed
    have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
    proof −
     have tts kk (nn kk) = mt_tape c7 kk (mt_pos c7 kk)
       using c7_eq8 nn_kk by simp
     also have ... = mt_tape c7 kk (mt_pos c' kk + 1)
       using c7_kk by simp
     also have ... = mt_tape c' kk (mt_pos c' kk + 1) using c7_at_addr .
     finally have read_eq:
        tts kk (nn kk) = mt_tape c' kk (mt_pos c' kk + 1) .
     have mt_tape c' kk (mt_pos c' kk + 1) ≠ LE_block (le_tm M)
       using right_not_le_c' by blast
     thus ?thesis using read_eq by simp
    qed
    have dest6_AE_Right: dest6 kk = AE_Right
      using AE_Right dest6_eq_dest5 by simp
    have dr8_kk: dr8 kk = dir.N
          using dr8_eq buf_eq dest_eq dest6_AE_Right buf6_kk h_ne_LE
read_ne_LE kklt
      by simp
    have mt_pos c8 kk = go_dir (dr8 kk) (nn kk)
      using c8_eq by simp
    also have ... = nn kk using dr8_kk by simp
    also have ... = mt_pos c' kk + 1 using nn_kk_val .
    finally show ?thesis using AE_Right by simp
   qed
  qed
```

Reconstruction leaf of *ae_simulates_forward_stage_general*: the

position-decode invariant at *c8*. When *c8* is at *SS1* (the non-halt super-step boundary), each simulated head *mt_pos cM_k k* is recovered from the M'-side head *mt_pos c8 k* and the recorded offset *off k* via *ae_decode_pos*. Per-tape regime case-split on *mt_pos c′ kk* feeding the three *c8_pos_for_∗* head trajectories; the *idx = SS1* antecedent rules out the halt branch (which would land at *init_stage* = VFwd).

```
lemma ae_fwd_pos_decode_c8:
  fixes M :: ('q, 'a) mttm
    and c' c8 :: ('c :: enum ⇒ 'a,
              'q × ('a, 'c) ae_stage) mt_config
    and cM_k :: ('a, 'q) mt_config
    and q6 :: 'q
    and ofs5 ofs6 :: nat ⇒ 'c
    and buf5 buf6 :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
    and dest5 :: nat ⇒ ae_dest
  assumes ofs6_eq_ofs5: ofs6 = ofs5
      and c8_state: mt_state c8 = (q6, if q6 ∈ {t_tm M, r_tm M}
                then init_stage (le_tm M)
                else (ofs6, buf6, init_dest, SS1))
      and cM_k_window_general:
          ∀ kk<k_tm M. ae_window_invariant_general
                (mt_tape cM_k kk) (mt_pos cM_k kk)
                (dest5 kk, ofs5 kk) (buf5 kk)
                (mt_pos c' kk) (le_tm M)
      and c8_pos_for_pos0:
          ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 0
                ⟹ mt_pos c8 kk = (if dest5 kk = AE_Right then 1 else 0)
      and c8_pos_for_pos_ge2:
          ⋀kk. kk < k_tm M ⟹ mt_pos c' kk ≥ 2
                ⟹ mt_pos c8 kk = (case dest5 kk of
                AE_Left  ⇒ mt_pos c' kk − 1
                | AE_Home  ⇒ mt_pos c' kk
                | AE_Right ⇒ mt_pos c' kk + 1)
      and c8_pos_for_pos1:
          ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 1
                ⟹ mt_pos c8 kk = (case dest5 kk of
                AE_Left  ⇒ 0
                | AE_Home  ⇒ 1
                | AE_Right ⇒ 2)
  shows (case mt_state c8 of (_, _, _, _, idx) ⇒ idx = SS1)
      ⟶ (∀ k<k_tm M. mt_pos cM_k k
                = ae_decode_pos (mt_pos c8 k)
                   (case mt_state c8 of (_, off, _, _, _) ⇒ off k))
proof −
  let ?c = card (UNIV :: 'c set)
  show ?thesis
  proof
    assume idx_ss1: case mt_state c8 of (_, _, _, _, idx) ⇒ idx = SS1
```

```
have q6_nhalt: q6 ∉ {t_tm M, r_tm M}
proof (rule ccontr)
  assume ¬ q6 ∉ {t_tm M, r_tm M}
  hence q6_in: q6 ∈ {t_tm M, r_tm M} by simp
  hence c8_eq_halt: mt_state c8 = (q6, init_stage (le_tm M))
    using c8_state by simp
  thus False using idx_ss1 by (simp add: init_stage_def)
qed
have c8_nh: mt_state c8 = (q6, ofs6, buf6, init_dest, SS1)
  using c8_state q6_nhalt by simp
show ∀ k<k_tm M. mt_pos cM_k k
        = ae_decode_pos (mt_pos c8 k)
            (case mt_state c8 of (_, off, _, _, _) ⇒ off k)
proof (intro allI impI)
  fix kk assume kklt: kk < k_tm M
  have ofs_at_c8:
     (case mt_state c8 of (_, off, _, _, _) ⇒ off kk) = ofs6 kk
    using c8_nh by simp
  have ofs5_eq_kk: ofs5 kk = ofs6 kk using ofs6_eq_ofs5 by simp
  have wi: ae_window_invariant_general
            (mt_tape cM_k kk) (mt_pos cM_k kk)
            (dest5 kk, ofs5 kk) (buf5 kk)
            (mt_pos c' kk) (le_tm M)
    using cM_k_window_general[rule_format, OF kklt] by simp
  consider (le0) mt_pos c' kk = 0
        | (le1) mt_pos c' kk = 1
        | (steady) mt_pos c' kk ≥ 2
    by linarith
  thus mt_pos cM_k kk
        = ae_decode_pos (mt_pos c8 kk)
            (case mt_state c8 of (_, off, _, _, _) ⇒ off kk)
  proof cases
    case le0
    have win_le0: ae_window_invariant_le0
            (mt_tape cM_k kk) (mt_pos cM_k kk)
            (dest5 kk, ofs5 kk) (buf5 kk) (le_tm M)
      using wi le0 unfolding ae_window_invariant_general_def by simp
    show ?thesis
    proof (cases dest5 kk)
      case AE_Left
      have False
        using win_le0 AE_Left
        unfolding ae_window_invariant_le0_def by simp
      thus ?thesis ..
    next
      case AE_Home
      have c8_kk: mt_pos c8 kk = 0
        using c8_pos_for_pos0[OF kklt le0] AE_Home by simp
      have mp_cM_k: mt_pos cM_k kk = 0
```

```
    using win_le0 AE_Home
    unfolding ae_window_invariant_le0_def by simp
  have decode: ae_decode_pos 0 (ofs6 kk) = 0
    unfolding ae_decode_pos_def by simp
  show ?thesis using c8_kk mp_cM_k decode ofs_at_c8 by simp
 next
  case AE_Right
  have c8_kk: mt_pos c8 kk = 1
    using c8_pos_for_pos0[OF kklt le0] AE_Right by simp
  have mp_cM_k: mt_pos cM_k kk = Suc (c_idx (ofs5 kk))
    using win_le0 AE_Right
    unfolding ae_window_invariant_le0_def by simp
  have decode: ae_decode_pos 1 (ofs6 kk) = Suc (c_idx (ofs6 kk))
    unfolding ae_decode_pos_def by simp
  show ?thesis
    using c8_kk mp_cM_k decode ofs_at_c8 ofs5_eq_kk by simp
 qed
next
 case le1
 have win_le1: ae_window_invariant_le1
         (mt_tape cM_k kk) (mt_pos cM_k kk)
         (dest5 kk, ofs5 kk) (buf5 kk) (le_tm M)
  using wi le1 unfolding ae_window_invariant_general_def by simp
 show ?thesis
 proof (cases dest5 kk)
  case AE_Left
  have c8_kk: mt_pos c8 kk = 0
    using c8_pos_for_pos1[OF kklt le1] AE_Left by simp
  have mp_cM_k: mt_pos cM_k kk = 0
    using win_le1 AE_Left
    unfolding ae_window_invariant_le1_def by simp
  have decode: ae_decode_pos 0 (ofs6 kk) = 0
    unfolding ae_decode_pos_def by simp
  show ?thesis using c8_kk mp_cM_k decode ofs_at_c8 by simp
 next
  case AE_Home
  have c8_kk: mt_pos c8 kk = 1
    using c8_pos_for_pos1[OF kklt le1] AE_Home by simp
  have mp_cM_k: mt_pos cM_k kk = Suc (c_idx (ofs5 kk))
    using win_le1 AE_Home
    unfolding ae_window_invariant_le1_def by simp
  have decode: ae_decode_pos 1 (ofs6 kk) = Suc (c_idx (ofs6 kk))
    unfolding ae_decode_pos_def by simp
  show ?thesis
    using c8_kk mp_cM_k decode ofs_at_c8 ofs5_eq_kk by simp
 next
  case AE_Right
  have c8_kk: mt_pos c8 kk = 2
    using c8_pos_for_pos1[OF kklt le1] AE_Right by simp
```

```
    have mp_cM_k:
       mt_pos cM_k kk = Suc (?c + c_idx (ofs5 kk))
      using win_le1 AE_Right
      unfolding ae_window_invariant_le1_def by simp
    have decode:
       ae_decode_pos 2 (ofs6 kk) = Suc (?c + c_idx (ofs6 kk))
      unfolding ae_decode_pos_def by simp
    show ?thesis
      using c8_kk mp_cM_k decode ofs_at_c8 ofs5_eq_kk by simp
  qed
next
  case steady
  have win_steady: ae_window_invariant
          (mt_tape cM_k kk) (mt_pos cM_k kk)
          (dest5 kk, ofs5 kk) (buf5 kk)
          ((mt_pos c' kk − 2) ∗ ?c + 1)
    using wi steady unfolding ae_window_invariant_general_def by simp
  have mp_cM_k:
     mt_pos cM_k kk
       = (mt_pos c' kk − 2) ∗ ?c + 1 + bp_linear (dest5 kk, ofs5 kk)
    using win_steady unfolding ae_window_invariant_def by simp
  show ?thesis
  proof (cases dest5 kk)
    case AE_Left
    have c8_kk: mt_pos c8 kk = mt_pos c' kk − 1
      using c8_pos_for_pos_ge2[OF kklt steady] AE_Left by simp
    have bp_lin: bp_linear (dest5 kk, ofs5 kk) = c_idx (ofs5 kk)
      using AE_Left unfolding bp_linear_def by simp
    have decode:
       ae_decode_pos (mt_pos c' kk − 1) (ofs6 kk)
         = (mt_pos c' kk − 2) ∗ ?c + c_idx (ofs6 kk) + 1
    proof −
      have ae_decode_pos (mt_pos c' kk − 1) (ofs6 kk)
            = (mt_pos c' kk − 1 − 1) ∗ ?c + c_idx (ofs6 kk) + 1
        using steady unfolding ae_decode_pos_def by simp
      also have ... = (mt_pos c' kk − 2) ∗ ?c + c_idx (ofs6 kk) + 1
        using steady by (simp add: numeral_2_eq_2)
      finally show ?thesis .
    qed
    show ?thesis
      using c8_kk mp_cM_k bp_lin decode ofs_at_c8 ofs5_eq_kk by simp
  next
    case AE_Home
    have c8_kk: mt_pos c8 kk = mt_pos c' kk
      using c8_pos_for_pos_ge2[OF kklt steady] AE_Home by simp
    have bp_lin: bp_linear (dest5 kk, ofs5 kk) = ?c + c_idx (ofs5 kk)
      using AE_Home unfolding bp_linear_def by simp
    have decode:
       ae_decode_pos (mt_pos c' kk) (ofs6 kk)
```

```
        = (mt_pos c' kk − 1) * ?c + c_idx (ofs6 kk) + 1
      using steady unfolding ae_decode_pos_def by simp
    have addr_eq:
      (mt_pos c' kk − 2) * ?c + 1 + (?c + c_idx (ofs5 kk))
        = (mt_pos c' kk − 1) * ?c + c_idx (ofs5 kk) + 1
    proof −
      have c_split: (mt_pos c' kk − 1) * ?c
                  = (mt_pos c' kk − 2) * ?c + ?c
        using steady by (simp add: algebra_simps diff_mult_distrib)
      thus ?thesis by linarith
    qed
    show ?thesis
      using c8_kk mp_cM_k bp_lin decode addr_eq ofs_at_c8 ofs5_eq_kk
      by simp
  next
    case AE_Right
    have c8_kk: mt_pos c8 kk = mt_pos c' kk + 1
      using c8_pos_for_pos_ge2[OF kklt steady] AE_Right by simp
    have bp_lin:
      bp_linear (dest5 kk, ofs5 kk) = 2 * ?c + c_idx (ofs5 kk)
      using AE_Right unfolding bp_linear_def by simp
    have decode:
      ae_decode_pos (mt_pos c' kk + 1) (ofs6 kk)
        = mt_pos c' kk * ?c + c_idx (ofs6 kk) + 1
      unfolding ae_decode_pos_def by simp
    have addr_eq:
      (mt_pos c' kk − 2) * ?c + 1 + (2 * ?c + c_idx (ofs5 kk))
        = mt_pos c' kk * ?c + c_idx (ofs5 kk) + 1
    proof −
      have c_split: mt_pos c' kk * ?c
                  = (mt_pos c' kk − 2) * ?c + 2 * ?c
        using steady by (simp add: algebra_simps diff_mult_distrib)
      thus ?thesis by linarith
    qed
    show ?thesis
      using c8_kk mp_cM_k bp_lin decode addr_eq ofs_at_c8 ofs5_eq_kk
      by simp
  qed
 qed
qed
qed
qed
```

Reconstruction leaf of *ae_simulates_forward_stage_general*: the *c8* head position for an le1 tape (*mt_pos c' kk = 1*), as a *dest5*-case value (*AE_Left* -> 0, *AE_Home* -> 1, *AE_Right* -> 2). For *AE_Left* SS8 reads the LE-marked cell 0 at *c7_pos = 0* (via *tape_c7_zero_le_pos1_dest_left*) and stays; otherwise SS8 steps from *c7_pos* per the recorded *dest*.

lemma *ae_fwd_c8_pos_for_pos1*:

fixes $M :: ('q, 'a)\ mttm$
and $c'\ c4\ c5\ c6\ c7\ c8 :: ('c :: enum \Rightarrow 'a, 'q \times ('a, 'c)\ ae_stage)\ mt_config$
and $q5\ q6 :: 'q$
and $ofs5\ ofs6 :: nat \Rightarrow 'c$
and $buf5\ buf6 :: nat \Rightarrow ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$
and $dest5\ dest6 :: nat \Rightarrow ae_dest$
assumes $step5: (c4, c5) \in mttm_step\ (alphabet_enlarge_delta\ M)$
and $tape_c4_eq_c': mt_tape\ c4 = mt_tape\ c'$
and $buf5_not_le_per_tape$:
$\forall k{<}k_tm\ M.\ (mt_pos\ c'\ k = 0 \longrightarrow fst\ (buf5\ k) \neq LE_block\ (le_tm\ M) \wedge snd\ (snd\ (buf5\ k)) \neq LE_block\ (le_tm\ M)) \wedge (mt_pos\ c'\ k = 1 \longrightarrow fst\ (snd\ (buf5\ k)) \neq LE_block\ (le_tm\ M) \wedge snd\ (snd\ (buf5\ k)) \neq LE_block\ (le_tm\ M)) \wedge (mt_pos\ c'\ k \geq 2 \longrightarrow fst\ (buf5\ k) \neq LE_block\ (le_tm\ M) \wedge fst\ (snd\ (buf5\ k)) \neq LE_block\ (le_tm\ M) \wedge snd\ (snd\ (buf5\ k)) \neq LE_block\ (le_tm\ M))$
and $right_not_le_c'$:
$\forall k.\ mt_tape\ c'\ k\ (mt_pos\ c'\ k + 1) \neq LE_block\ (le_tm\ M)$
and $c4_pos: \bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c4\ kk = mt_pos\ c'\ kk$
and $c5_pos_for_pos1$:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 1 \Longrightarrow dest5\ kk \neq AE_Left \Longrightarrow mt_pos\ c5\ kk = 0$
and $step6: (c5, c6) \in mttm_step\ (alphabet_enlarge_delta\ M)$
and $step7: (c6, c7) \in mttm_step\ (alphabet_enlarge_delta\ M)$
and $c7_state: mt_state\ c7 = (q6, ofs6, buf6, dest6, SS8)$
and $buf6_eq_buf5: buf6 = buf5$
and $dest6_eq_dest5: dest6 = dest5$
and $c7_pos_for_pos1_dest_left$:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 1 \Longrightarrow dest5\ kk = AE_Left \Longrightarrow mt_pos\ c7\ kk = 0$
and $tape_c7_zero_le_pos1_dest_left$:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 1 \Longrightarrow dest5\ kk = AE_Left \Longrightarrow mt_tape\ c7\ kk\ 0 = LE_block\ (le_tm\ M)$
and $step8_sub: (c7, c8) \in mttm_step\ (ae_delta_ss8_ss1\ M)$
and $c6_pos_for_pos1$:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 1 \Longrightarrow mt_pos\ c6\ kk = 1$
and $c7_pos_for_pos1$:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 1 \Longrightarrow mt_pos\ c7\ kk = (if\ dest5\ kk = AE_Left\ then\ 0\ else\ 2)$
shows $\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 1 \Longrightarrow mt_pos\ c8\ kk = (case\ dest5\ kk\ of\ AE_Left \Rightarrow 0 \mid AE_Home \Rightarrow 1 \mid AE_Right \Rightarrow 2)$
proof −

fix *kk*
assume *kklt*: *kk* < *k_tm M*
assume *pos_kk*: *mt_pos c' kk* = *1*
obtain *qq tts nn qq' aa8 dr8* **where**
c7_eq8: *c7* = *Config*$_M$ *qq tts nn*
and *c8_eq*: *c8* = *Config*$_M$ *qq'*
(λ*k*. (*tts k*)(*nn k* := *aa8 k*))
(λ*k*. *go_dir* (*dr8 k*) (*nn k*))
and *tr_in*: (*qq*, λ*k*. *tts k* (*nn k*), *qq'*, *aa8*, *dr8*)
∈ *ae_delta_ss8_ss1 M*
using *step8_sub* **by** (*auto elim*: *mttm_step.cases*)
obtain *q ofs buf dest* **where**
qq_eq: *qq* = (*q*, *ofs*, *buf*, *dest*, *SS8*)
and *dr8_eq*:
dr8 = (λ*k*. *if k* < *k_tm M then snd* (*ae_ss8_action* (*le_tm M*)
(*tts k* (*nn k*)) (*buf k*) (*dest k*)) *else dir.N*)
using *tr_in* **by** (*auto simp*: *ae_delta_ss8_ss1_def*)
have *qq_state*: *qq* = (*q6*, *ofs6*, *buf6*, *dest6*, *SS8*)
using *c7_state c7_eq8* **by** *simp*
have *buf_eq*: *buf* = *buf6* **using** *qq_eq qq_state* **by** *simp*
have *dest_eq*: *dest* = *dest6* **using** *qq_eq qq_state* **by** *simp*
have *nn_kk*: *nn kk* = *mt_pos c7 kk* **using** *c7_eq8* **by** *simp*
obtain *ll hh rr* **where** *buf5_kk*: *buf5 kk* = (*ll*, *hh*, *rr*)
using *prod.exhaust* **by** *metis*
have *buf6_kk*: *buf6 kk* = (*ll*, *hh*, *rr*)
using *buf5_kk buf6_eq_buf5* **by** *simp*
have *h_ne_LE*: *hh* ≠ *LE_block* (*le_tm M*)
proof −
have *fst* (*snd* (*buf5 kk*)) ≠ *LE_block* (*le_tm M*)
using *buf5_not_le_per_tape*[*rule_format*, *OF kklt*] *pos_kk* **by** *blast*
thus *?thesis* **using** *buf5_kk* **by** *simp*
qed
show *mt_pos c8 kk* = (*case dest5 kk of*
AE_Left ⇒ *0*
| *AE_Home* ⇒ *1*
| *AE_Right* ⇒ *2*)
proof (*cases dest5 kk*)
case *AE_Left*
have *c7_kk*: *mt_pos c7 kk* = *0*
using *c7_pos_for_pos1_dest_left*[*OF kklt pos_kk AE_Left*] **.**
have *nn_kk_val*: *nn kk* = *0* **using** *nn_kk c7_kk* **by** *simp*
have *read_LE*: *tts kk* (*nn kk*) = *LE_block* (*le_tm M*)
proof −
have *tts kk* (*nn kk*) = *mt_tape c7 kk* (*mt_pos c7 kk*)
using *c7_eq8 nn_kk* **by** *simp*
also have ... = *mt_tape c7 kk 0* **using** *c7_kk* **by** *simp*
also have ... = *LE_block* (*le_tm M*)
using *tape_c7_zero_le_pos1_dest_left*[*OF kklt pos_kk AE_Left*] **.**
finally show *?thesis* **.**

```
  qed
  have dr8_kk: dr8 kk = dir.N using dr8_eq read_LE kklt by simp
  have mt_pos c8 kk = go_dir (dr8 kk) (nn kk)
    using c8_eq by simp
  also have ... = nn kk using dr8_kk by simp
  also have ... = 0 using nn_kk_val .
  finally show ?thesis using AE_Left by simp
next
  case AE_Home
  have c7_kk: mt_pos c7 kk = 2
    using c7_pos_for_pos1[OF kklt pos_kk] AE_Home by simp
  have nn_kk_val: nn kk = 2 using nn_kk c7_kk by simp
  have dest_ne_left: dest5 kk ≠ AE_Left using AE_Home by simp
  have c5_pos_kk: mt_pos c5 kk = 0
    using c5_pos_for_pos1[OF kklt pos_kk dest_ne_left] .
  have two_ne_c6: (2 :: nat) ≠ mt_pos c6 kk
    using c6_pos_for_pos1[OF kklt pos_kk] by simp
  have two_ne_c5: (2 :: nat) ≠ mt_pos c5 kk
    using c5_pos_kk by simp
  have two_ne_c4: (2 :: nat) ≠ mt_pos c4 kk
    using c4_pos kklt pos_kk by simp
  have c7_at_2: mt_tape c7 kk 2 = mt_tape c' kk 2
  proof −
    have mt_tape c7 kk 2 = mt_tape c6 kk 2
      using mttm_step_tape_off_head[OF step7 two_ne_c6] .
    also have ... = mt_tape c5 kk 2
      using mttm_step_tape_off_head[OF step6 two_ne_c5] .
    also have ... = mt_tape c4 kk 2
      using mttm_step_tape_off_head[OF step5 two_ne_c4] .
    also have ... = mt_tape c' kk 2 using tape_c4_eq_c' by simp
    finally show ?thesis .
  qed
  have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
  proof −
    have tts kk (nn kk) = mt_tape c7 kk (mt_pos c7 kk)
      using c7_eq8 nn_kk by simp
    also have ... = mt_tape c7 kk 2 using c7_kk by simp
    also have ... = mt_tape c' kk 2 using c7_at_2 .
    finally have read_eq: tts kk (nn kk) = mt_tape c' kk 2 .
    have mt_tape c' kk (mt_pos c' kk + 1) ≠ LE_block (le_tm M)
      using right_not_le_c' by blast
    hence mt_tape c' kk 2 ≠ LE_block (le_tm M)
      using pos_kk by (simp add: numeral_2_eq_2)
    thus ?thesis using read_eq by simp
  qed
  have dest6_AE_Home: dest6 kk = AE_Home
    using AE_Home dest6_eq_dest5 by simp
  have dr8_kk: dr8 kk = dir.L
        using dr8_eq buf_eq dest_eq dest6_AE_Home buf6_kk h_ne_LE
```

```
read_ne_LE kklt
      by simp
    have mt_pos c8 kk = go_dir (dr8 kk) (nn kk)
      using c8_eq by simp
    also have ... = (nn kk) − 1 using dr8_kk by simp
    also have ... = 1 using nn_kk_val by simp
    finally show ?thesis using AE_Home by simp
  next
    case AE_Right
    have c7_kk: mt_pos c7 kk = 2
      using c7_pos_for_pos1[OF kklt pos_kk] AE_Right by simp
    have nn_kk_val: nn kk = 2 using nn_kk c7_kk by simp
    have dest_ne_left: dest5 kk ≠ AE_Left using AE_Right by simp
    have c5_pos_kk: mt_pos c5 kk = 0
      using c5_pos_for_pos1[OF kklt pos_kk dest_ne_left] .
    have two_ne_c6: (2 :: nat) ≠ mt_pos c6 kk
      using c6_pos_for_pos1[OF kklt pos_kk] by simp
    have two_ne_c5: (2 :: nat) ≠ mt_pos c5 kk
      using c5_pos_kk by simp
    have two_ne_c4: (2 :: nat) ≠ mt_pos c4 kk
      using c4_pos kklt pos_kk by simp
    have c7_at_2: mt_tape c7 kk 2 = mt_tape c' kk 2
    proof −
      have mt_tape c7 kk 2 = mt_tape c6 kk 2
        using mttm_step_tape_off_head[OF step7 two_ne_c6] .
      also have ... = mt_tape c5 kk 2
        using mttm_step_tape_off_head[OF step6 two_ne_c5] .
      also have ... = mt_tape c4 kk 2
        using mttm_step_tape_off_head[OF step5 two_ne_c4] .
      also have ... = mt_tape c' kk 2 using tape_c4_eq_c' by simp
      finally show ?thesis .
    qed
    have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
    proof −
      have tts kk (nn kk) = mt_tape c7 kk (mt_pos c7 kk)
        using c7_eq8 nn_kk by simp
      also have ... = mt_tape c7 kk 2 using c7_kk by simp
      also have ... = mt_tape c' kk 2 using c7_at_2 .
      finally have read_eq: tts kk (nn kk) = mt_tape c' kk 2 .
      have mt_tape c' kk (mt_pos c' kk + 1) ≠ LE_block (le_tm M)
        using right_not_le_c' by blast
      hence mt_tape c' kk 2 ≠ LE_block (le_tm M)
        using pos_kk by (simp add: numeral_2_eq_2)
      thus ?thesis using read_eq by simp
    qed
    have dest6_AE_Right: dest6 kk = AE_Right
      using AE_Right dest6_eq_dest5 by simp
    have dr8_kk: dr8 kk = dir.N
          using dr8_eq buf_eq dest_eq dest6_AE_Right buf6_kk h_ne_LE
```

```
    read_ne_LE kklt
      by simp
    have mt_pos c8 kk = go_dir (dr8 kk) (nn kk)
      using c8_eq by simp
    also have ... = nn kk using dr8_kk by simp
    also have ... = 2 using nn_kk_val .
    finally show ?thesis using AE_Right by simp
  qed
 qed
```

Reconstruction leaf of *ae_simulates_forward_stage_general*: the in-window *c8* value at block 1 for an le0 tape ($mt_pos\ c'\ kk = 0$). Uniformly $c8@1 = rr$ (the right buffer slot $snd\ (snd\ (buf5\ kk))$); the SS6/SS7/SS8 substeps write at cells $c5_pos = 0$ / $c6_pos = 1$ / $c7_pos$, with cell 1 carrying the buffered right value.

```
lemma ae_fwd_c8_at_one_for_pos0:
 fixes M :: ('q, 'a) mttm
   and c' c4 c5 c6 c7 c8 :: ('c :: enum ⇒ 'a,
            'q × ('a, 'c) ae_stage) mt_config
   and q5 q6 :: 'q
   and ofs5 ofs6 :: nat ⇒ 'c
   and buf5 buf6 :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
   and dest5 dest6 :: nat ⇒ ae_dest
 assumes step5: (c4, c5) ∈ mttm_step (alphabet_enlarge_delta M)
     and tape_c4_eq_c': mt_tape c4 = mt_tape c'
     and buf5_not_le_per_tape:
         ∀k<k_tm M. (mt_pos c' k = 0
               ⟶ fst (buf5 k) ≠ LE_block (le_tm M)
               ∧ snd (snd (buf5 k)) ≠ LE_block (le_tm M))
             ∧ (mt_pos c' k = 1
               ⟶ fst (snd (buf5 k)) ≠ LE_block (le_tm M)
               ∧ snd (snd (buf5 k)) ≠ LE_block (le_tm M))
             ∧ (mt_pos c' k ≥ 2
               ⟶ fst (buf5 k) ≠ LE_block (le_tm M)
               ∧ fst (snd (buf5 k)) ≠ LE_block (le_tm M)
               ∧ snd (snd (buf5 k)) ≠ LE_block (le_tm M))
     and right_not_le_c':
         ∀k. mt_tape c' k (mt_pos c' k + 1) ≠ LE_block (le_tm M)
     and c4_pos: ⋀kk. kk < k_tm M ⟹ mt_pos c4 kk = mt_pos c' kk
     and step6: (c5, c6) ∈ mttm_step (alphabet_enlarge_delta M)
     and step7_sub: (c6, c7) ∈ mttm_step (ae_delta_ss7_ss8 M)
     and c6_state: mt_state c6 = (q6, ofs6, buf6, dest6, SS7)
     and c7_state: mt_state c7 = (q6, ofs6, buf6, dest6, SS8)
     and buf6_eq_buf5: buf6 = buf5
     and dest6_eq_dest5: dest6 = dest5
     and step8_sub: (c7, c8) ∈ mttm_step (ae_delta_ss8_ss1 M)
     and step8: (c7, c8) ∈ mttm_step (alphabet_enlarge_delta M)
     and c5_pos_for_pos0:
         ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 0 ⟹ mt_pos c5 kk = 0
```

and *buf5_h_le_for_pos0*:
$\forall k{<}k_tm\ M.\ mt_pos\ c'\ k = 0 \longrightarrow fst\ (snd\ (buf5\ k)) = LE_block\ (le_tm\ M)$
and *c6_pos_for_pos0*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 0 \Longrightarrow mt_pos\ c6\ kk = 1$
and *c7_pos_for_pos0*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 0$
$\Longrightarrow mt_pos\ c7\ kk = (if\ dest5\ kk = AE_Right\ then\ 1\ else\ 0)$
shows $\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 0$
$\Longrightarrow mt_tape\ c8\ kk\ 1 = snd\ (snd\ (buf5\ kk))$
proof −
fix *kk*
assume *kklt*: $kk < k_tm\ M$
assume *pos_kk*: $mt_pos\ c'\ kk = 0$
obtain *qq tts nn qq' aa7 dr7* **where**
c6_eq7: $c6 = Config_M\ qq\ tts\ nn$
and *c7_eq*: $c7 = Config_M\ qq'$
$(\lambda k.\ (tts\ k)(nn\ k := aa7\ k))$
$(\lambda k.\ go_dir\ (dr7\ k)\ (nn\ k))$
and *tr7_in*: $(qq, \lambda k.\ tts\ k\ (nn\ k), qq', aa7, dr7)$
$\in ae_delta_ss7_ss8\ M$
using *step7_sub* **by** (*auto elim*: *mttm_step.cases*)
obtain *q7 ofs7 buf7 dest7* **where**
qq_eq7: $qq = (q7, ofs7, buf7, dest7, SS7)$
and *aa7_eq*:
$aa7 = (\lambda k.\ if\ k < k_tm\ M\ then\ fst\ (ae_ss7_action\ (le_tm\ M)$
$(tts\ k\ (nn\ k))\ (buf7\ k)\ (dest7\ k))\ else\ bl_block\ (bl_tm\ M))$
using *tr7_in* **by** (*auto simp*: *ae_delta_ss7_ss8_def*)
have *qq_state*: $qq = (q6, ofs6, buf6, dest6, SS7)$
using *c6_state c6_eq7* **by** *simp*
have *buf7_eq*: $buf7 = buf6$ **using** *qq_eq7 qq_state* **by** *simp*
have *dest7_eq*: $dest7 = dest6$ **using** *qq_eq7 qq_state* **by** *simp*
have *nn7_kk*: $nn\ kk = mt_pos\ c6\ kk$ **using** *c6_eq7* **by** *simp*
have *c6_kk*: $mt_pos\ c6\ kk = 1$ **using** *c6_pos_for_pos0*[*OF kklt pos_kk*] **.**
have *nn7_kk_1*: $nn\ kk = 1$ **using** *nn7_kk c6_kk* **by** *simp*
have *one_ne_c5_pos*: $(1 :: nat) \neq mt_pos\ c5\ kk$
using *c5_pos_for_pos0*[*OF kklt pos_kk*] **by** *simp*
have *one_ne_c4_pos*: $(1 :: nat) \neq mt_pos\ c4\ kk$
using *c4_pos kklt pos_kk* **by** *simp*
have *c6_at_1*: $mt_tape\ c6\ kk\ 1 = mt_tape\ c'\ kk\ 1$
proof −
have $mt_tape\ c6\ kk\ 1 = mt_tape\ c5\ kk\ 1$
using *mttm_step_tape_off_head*[*OF step6 one_ne_c5_pos*] **.**
also have $\ldots = mt_tape\ c4\ kk\ 1$
using *mttm_step_tape_off_head*[*OF step5 one_ne_c4_pos*] **.**
also have $\ldots = mt_tape\ c'\ kk\ 1$ **using** *tape_c4_eq_c'* **by** *simp*
finally show *?thesis* **.**
qed
have *read7_ne_LE*: $tts\ kk\ (nn\ kk) \neq LE_block\ (le_tm\ M)$

proof −
have *tts kk* (*nn kk*) = *mt_tape c6 kk* (*mt_pos c6 kk*)
using *c6_eq7 nn7_kk* **by** *simp*
also have ... = *mt_tape c6 kk 1* **using** *c6_kk* **by** *simp*
also have ... = *mt_tape c′ kk 1* **using** *c6_at_1* **.**
finally have *read_eq*: *tts kk* (*nn kk*) = *mt_tape c′ kk 1* **.**
have *mt_tape c′ kk* (*mt_pos c′ kk* + *1*) ≠ *LE_block* (*le_tm M*)
using *right_not_le_c′* **by** *blast*
hence *mt_tape c′ kk 1* ≠ *LE_block* (*le_tm M*) **using** *pos_kk* **by** *simp*
thus *?thesis* **using** *read_eq* **by** *simp*
qed
obtain *ll hh rr* **where** *buf5_kk*: *buf5 kk* = (*ll*, *hh*, *rr*)
using *prod.exhaust* **by** *metis*
have *buf6_kk*: *buf6 kk* = (*ll*, *hh*, *rr*)
using *buf5_kk buf6_eq_buf5* **by** *simp*
have *hh_le*: *hh* = *LE_block* (*le_tm M*)
proof −
have *fst* (*snd* (*buf5 kk*)) = *LE_block* (*le_tm M*)
using *buf5_h_le_for_pos0*[*rule_format*, *OF kklt*] *pos_kk* **by** *blast*
thus *?thesis* **using** *buf5_kk* **by** *simp*
qed
have *c7_at_1_eq_rr*: *mt_tape c7 kk 1* = *rr*
proof −
— SS7 writes *aa7* at *nn* = *1*; with h=LE the action's second branch fires, returning (*r*, _) regardless of *dest*, so *aa7 kk* = *r* = *rr*.
have *aa7_kk*: *aa7 kk* = *rr*
using *aa7_eq buf7_eq buf6_kk hh_le read7_ne_LE kklt* **by** *simp*
have *mt_tape c7 kk 1* = *aa7 kk* **using** *c7_eq nn7_kk_1* **by** *simp*
thus *?thesis* **using** *aa7_kk* **by** *simp*
qed
show *mt_tape c8 kk 1* = *snd* (*snd* (*buf5 kk*))
proof (*cases dest5 kk* = *AE_Right*)
case *True*
— SS8 writes at *c7_pos* = *1*; reads *c7@1* = *rr* ≠ *LE*, *h* = *LE*, dest=Right gives (*r*, *N*); *c8@1* = *rr*.
have *c7_pos_kk*: *mt_pos c7 kk* = *1*
using *c7_pos_for_pos0*[*OF kklt pos_kk*] *True* **by** *simp*
obtain *qq8 tts8 nn8 qq8′ aa8 dr8* **where**
c7_eq8: *c7* = *Config*$_M$ *qq8 tts8 nn8*
and *c8_eq*: *c8* = *Config*$_M$ *qq8′*
(*λk*. (*tts8 k*)(*nn8 k* := *aa8 k*))
(*λk*. *go_dir* (*dr8 k*) (*nn8 k*))
and *tr8_in*: (*qq8*, *λk*. *tts8 k* (*nn8 k*), *qq8′*, *aa8*, *dr8*)
∈ *ae_delta_ss8_ss1 M*
using *step8_sub* **by** (*auto elim*: *mttm_step.cases*)
obtain *q8 ofs8 buf8 dest8* **where**
qq8_eq: *qq8* = (*q8*, *ofs8*, *buf8*, *dest8*, *SS8*)
and *aa8_eq*:
aa8 = (*λk*. *if k* < *k_tm M then fst* (*ae_ss8_action* (*le_tm M*)

```
                (tts8 k (nn8 k)) (buf8 k) (dest8 k)) else bl_block (bl_tm
M))
      using tr8_in by (auto simp: ae_delta_ss8_ss1_def)
    have qq8_state: qq8 = (q6, ofs6, buf6, dest6, SS8)
      using c7_state c7_eq8 by simp
    have buf8_eq: buf8 = buf6 using qq8_eq qq8_state by simp
    have dest8_eq: dest8 = dest6 using qq8_eq qq8_state by simp
    have nn8_kk: nn8 kk = mt_pos c7 kk using c7_eq8 by simp
    have nn8_kk_1: nn8 kk = 1 using nn8_kk c7_pos_kk by simp
    have rr_ne_LE: rr ≠ LE_block (le_tm M)
    proof −
      have snd (snd (buf5 kk)) ≠ LE_block (le_tm M)
        using buf5_not_le_per_tape[rule_format, OF kklt] pos_kk by blast
      thus ?thesis using buf5_kk by simp
    qed
    have read8_ne_LE: tts8 kk (nn8 kk) ≠ LE_block (le_tm M)
    proof −
      have tts8 kk (nn8 kk) = mt_tape c7 kk (mt_pos c7 kk)
        using c7_eq8 nn8_kk by simp
      also have ... = mt_tape c7 kk 1 using c7_pos_kk by simp
      also have ... = rr using c7_at_1_eq_rr .
      finally have tts8 kk (nn8 kk) = rr .
      thus ?thesis using rr_ne_LE by simp
    qed
    have dest6_eq: dest6 kk = AE_Right using True dest6_eq_dest5 by simp
    have aa8_kk: aa8 kk = rr
      using aa8_eq buf8_eq dest8_eq dest6_eq buf6_kk hh_le read8_ne_LE kklt
      by simp
    have mt_tape c8 kk 1 = aa8 kk using c8_eq nn8_kk_1 by simp
    also have ... = rr using aa8_kk .
    also have ... = snd (snd (buf5 kk)) using buf5_kk by simp
    finally show ?thesis .
  next
    case False
    — SS8 writes at c7_pos = 0 ≠ 1, off cell 1. c8@1 = c7@1 = rr.
    have c7_pos_kk: mt_pos c7 kk = 0
      using c7_pos_for_pos0[OF kklt pos_kk] False by simp
    have one_ne_c7_pos: (1 :: nat) ≠ mt_pos c7 kk
      using c7_pos_kk by simp
    have mt_tape c8 kk 1 = mt_tape c7 kk 1
      using mttm_step_tape_off_head[OF step8 one_ne_c7_pos] .
    also have ... = rr using c7_at_1_eq_rr .
    also have ... = snd (snd (buf5 kk)) using buf5_kk by simp
    finally show ?thesis .
  qed
qed
```

Reconstruction leaf of *ae_simulates_forward_stage_general*: the *c6* head position for a steady tape (*mt_pos c′ kk* ≥ *2*) is unchanged from *mt_pos c′ kk*. SS6's action writes at *c5_pos* (*pos* ∓ *1* per *dest5*) and steps

back, so *c6* lands on the home block *pos*; the non-LE window facts rule out the LE-marker short-circuit.

```
lemma ae_fwd_c6_pos_for_pos_ge2:
  fixes M :: ('q, 'a) mttm
    and c' c4 c5 c6 :: ('c :: enum ⇒ 'a,
              'q × ('a, 'c) ae_stage) mt_config
    and q5 :: 'q
    and ofs5 :: nat ⇒ 'c
    and buf5 :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
    and dest5 :: nat ⇒ ae_dest
  assumes step5: (c4, c5) ∈ mttm_step (alphabet_enlarge_delta M)
      and c5_state: mt_state c5 = (q5, ofs5, buf5, dest5, SS6)
      and tape_c4_eq_c': mt_tape c4 = mt_tape c'
      and buf5_not_le_per_tape:
          ∀ k<k_tm M. (mt_pos c' k = 0
                ⟶ fst (buf5 k) ≠ LE_block (le_tm M)
                ∧ snd (snd (buf5 k)) ≠ LE_block (le_tm M))
              ∧ (mt_pos c' k = 1
                ⟶ fst (snd (buf5 k)) ≠ LE_block (le_tm M)
                ∧ snd (snd (buf5 k)) ≠ LE_block (le_tm M))
              ∧ (mt_pos c' k ≥ 2
                ⟶ fst (buf5 k) ≠ LE_block (le_tm M)
                ∧ fst (snd (buf5 k)) ≠ LE_block (le_tm M)
                ∧ snd (snd (buf5 k)) ≠ LE_block (le_tm M))
      and right_not_le_c':
          ∀ k. mt_tape c' k (mt_pos c' k + 1) ≠ LE_block (le_tm M)
      and c4_pos: ⋀kk. kk < k_tm M ⟹ mt_pos c4 kk = mt_pos c' kk
      and step6_sub: (c5, c6) ∈ mttm_step (ae_delta_ss6_ss7 M)
      and c5_pos_for_pos_ge2:
          ⋀kk. kk < k_tm M ⟹ mt_pos c' kk ≥ 2 ⟹ dest5 kk ≠ AE_Left
              ⟹ mt_pos c5 kk = mt_pos c' kk − 1
      and c5_pos_for_pos_ge2_dest_left:
          ⋀kk. kk < k_tm M ⟹ mt_pos c' kk ≥ 2 ⟹ dest5 kk = AE_Left
              ⟹ mt_pos c5 kk = mt_pos c' kk + 1
      and left_not_le_c'_steady:
          ∀ k. mt_pos c' k ≥ 2
              ⟶ mt_tape c' k (mt_pos c' k − 1) ≠ LE_block (le_tm M)
  shows ⋀kk. kk < k_tm M ⟹ mt_pos c' kk ≥ 2 ⟹ mt_pos c6 kk = mt_pos
c' kk
proof −
  fix kk
  assume kklt: kk < k_tm M
  assume hge2: mt_pos c' kk ≥ 2
    obtain qq tts nn qq' aa6 dr6 where
      c5_eq6: c5 = Config_M qq tts nn
    and c6_eq: c6 = Config_M qq'
                    (λk. (tts k)(nn k := aa6 k))
                    (λk. go_dir (dr6 k) (nn k))
    and tr_in: (qq, λk. tts k (nn k), qq', aa6, dr6)
```

```
            ∈ ae_delta_ss6_ss7 M
  using step6_sub by (auto elim: mttm_step.cases)
obtain q ofs buf dest where
    qq_eq: qq = (q, ofs, buf, dest, SS6)
  and dr6_eq:
      dr6 = (λk. if k < k_tm M then snd (ae_ss6_action (le_tm M)
                        (tts k (nn k)) (buf k)) (dest k)) else dir.N)
  using tr_in by (auto simp: ae_delta_ss6_ss7_def)
have qq_state: qq = (q5, ofs5, buf5, dest5, SS6)
  using c5_state c5_eq6 by simp
have buf_eq: buf = buf5 using qq_eq qq_state by simp
have dest_eq: dest = dest5 using qq_eq qq_state by simp
have nn_kk: nn kk = mt_pos c5 kk using c5_eq6 by simp
obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
  using prod.exhaust by metis
have h_ne_LE: hh ≠ LE_block (le_tm M)
proof −
  have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
    using buf5_not_le_per_tape[rule_format, OF kklt] hge2 by blast
  thus ?thesis using buf5_kk by simp
qed
have c4_kk: mt_pos c4 kk = mt_pos c' kk using c4_pos kklt by simp
show mt_pos c6 kk = mt_pos c' kk
proof (cases dest5 kk = AE_Left)
  case True
  have c5_kk: mt_pos c5 kk = mt_pos c' kk + 1
    using c5_pos_for_pos_ge2_dest_left[OF kklt hge2 True] .
  have nn_kk_val: nn kk = mt_pos c' kk + 1 using nn_kk c5_kk by simp
  have addr_ne_c4_pos: (mt_pos c' kk + 1) ≠ mt_pos c4 kk
    using c4_kk by simp
  have c5_at_addr:
      mt_tape c5 kk (mt_pos c' kk + 1) = mt_tape c' kk (mt_pos c' kk + 1)
  proof −
    have mt_tape c5 kk (mt_pos c' kk + 1)
          = mt_tape c4 kk (mt_pos c' kk + 1)
      using mttm_step_tape_off_head[OF step5 addr_ne_c4_pos] .
    also have ... = mt_tape c' kk (mt_pos c' kk + 1)
      using tape_c4_eq_c' by simp
    finally show ?thesis .
  qed
  have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
  proof −
    have tts kk (nn kk) = mt_tape c5 kk (mt_pos c5 kk)
      using c5_eq6 nn_kk by simp
    also have ... = mt_tape c5 kk (mt_pos c' kk + 1)
      using c5_kk by simp
    also have ... = mt_tape c' kk (mt_pos c' kk + 1)
      using c5_at_addr .
    finally have read_eq:
```

```
      tts kk (nn kk) = mt_tape c' kk (mt_pos c' kk + 1) .
    have mt_tape c' kk (mt_pos c' kk + 1) ≠ LE_block (le_tm M)
      using right_not_le_c' by blast
    thus ?thesis using read_eq by simp
  qed
  have dr6_kk:
      dr6 kk = (if dest5 kk = AE_Left then dir.L else dir.R)
    using dr6_eq buf_eq dest_eq buf5_kk read_ne_LE h_ne_LE kklt by simp
  have dr6_kk_L: dr6 kk = dir.L using dr6_kk True by simp
  have mt_pos c6 kk = go_dir (dr6 kk) (nn kk)
    using c6_eq by simp
  also have ... = (nn kk) − 1 using dr6_kk_L by simp
  also have ... = (mt_pos c' kk + 1) − 1 using nn_kk_val by simp
  also have ... = mt_pos c' kk by simp
  finally show ?thesis .
next
  case False
  have c5_kk: mt_pos c5 kk = mt_pos c' kk − 1
    using c5_pos_for_pos_ge2[OF kklt hge2 False] .
  have nn_kk_val: nn kk = mt_pos c' kk − 1 using nn_kk c5_kk by simp
  have addr_ne_c4_pos: (mt_pos c' kk − 1) ≠ mt_pos c4 kk
    using c4_kk hge2 by linarith
  have c5_at_addr:
      mt_tape c5 kk (mt_pos c' kk − 1) = mt_tape c' kk (mt_pos c' kk − 1)
  proof −
    have mt_tape c5 kk (mt_pos c' kk − 1)
          = mt_tape c4 kk (mt_pos c' kk − 1)
      using mttm_step_tape_off_head[OF step5 addr_ne_c4_pos] .
    also have ... = mt_tape c' kk (mt_pos c' kk − 1)
      using tape_c4_eq_c' by simp
    finally show ?thesis .
  qed
  have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
  proof −
    have tts kk (nn kk) = mt_tape c5 kk (mt_pos c5 kk)
      using c5_eq6 nn_kk by simp
    also have ... = mt_tape c5 kk (mt_pos c' kk − 1)
      using c5_kk by simp
    also have ... = mt_tape c' kk (mt_pos c' kk − 1)
      using c5_at_addr .
    finally have read_eq:
        tts kk (nn kk) = mt_tape c' kk (mt_pos c' kk − 1) .
    have mt_tape c' kk (mt_pos c' kk − 1) ≠ LE_block (le_tm M)
      using left_not_le_c'_steady hge2 by blast
    thus ?thesis using read_eq by simp
  qed
  have dr6_kk:
      dr6 kk = (if dest5 kk = AE_Left then dir.L else dir.R)
    using dr6_eq buf_eq dest_eq buf5_kk read_ne_LE h_ne_LE kklt by simp
```

```
    have dr6_kk_R: dr6 kk = dir.R using dr6_kk False by simp
    have mt_pos c6 kk = go_dir (dr6 kk) (nn kk)
      using c6_eq by simp
    also have ... = Suc (nn kk) using dr6_kk_R by simp
    also have ... = Suc (mt_pos c' kk − 1) using nn_kk_val by simp
    also have ... = mt_pos c' kk using hge2 by simp
    finally show ?thesis .
  qed
 qed

end
theory AlphabetEnlargement_ForwardStep
 imports AlphabetEnlargement_ForwardCells
begin
```

2.25 Forward-stage SS5–SS8 super-step

The middle link of the forward-stage chain *ForwardCells* → *ForwardStep* → *ForwardStage*: the SS5→SS8 substep chain of *ae_simulates_forward_stage_general*. From the SS5-entry config *c4* (its invariant *ae_inv_ss5*, the γ-block and buffer side-bands, and the per-tape regime data) it runs the four substeps *ae_delta_ss5_ss6*, *ae_delta_ss6_ss7*, *ae_delta_ss7_ss8*, *ae_delta_ss8_ss1*, exposing the intermediate configs *c5*, *c6*, *c7*, *c8*, their shared SS-stage state tuple (*q5*, *ofs5*, *buf5*, *dest5*) = (*q6*, *ofs6*, *buf6*, *dest6*), and the per-tape position and tape-content facts the reconstruction leaves (*AlphabetEnlargement_ForwardCells*) consume. Carved out of *ae_simulates_forward_stage_general* as its widest single seam (17 assumptions, 29 conclusions); the call site re-binds the named facts verbatim.

```
lemma ae_forward_stage_step_chain:
 fixes M :: ('q, 'a) mttm
   and cM :: ('a, 'q) mt_config
   and c' c1 c2 c3 c4 :: ('c :: enum ⇒ 'a,
            'q × ('a, 'c) ae_stage) mt_config
 assumes vM:        valid_mttm M
    and le_anchor:
    ∀k<k_tm M. mt_tape c' k 0 = LE_block (le_tm M)
    and no_le_per_tape:
    ∀k. (mt_pos c' k ≥ 2
    ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
    ⟶ mt_tape cM k
    ((mt_pos c' k − 2) * card (UNIV :: 'c set) + 1 + i)
    ≠ le_tm M))
    ∧ (mt_pos c' k = 1
    ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
    ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
    ∧ (mt_pos c' k = 0
    ⟶ (∀i. i < card (UNIV :: 'c set)
```

$\longrightarrow$ *mt_tape cM k* (*Suc i*) $\neq$ *le_tm M*))
and *tape_corr*: $\forall$ *k*<*k_tm M*. *ae_tape_correspondence* (*le_tm M*)
(*mt_tape cM k*) (*mt_tape c′ k*)
and *step1_sub*: (*c′*, *c1*) $\in$ *mttm_step* (*ae_delta_ss1_ss2 M*)
and *step2_sub*: (*c1*, *c2*) $\in$ *mttm_step* (*ae_delta_ss2_ss3 M*)
and *step3_sub*: (*c2*, *c3*) $\in$ *mttm_step* (*ae_delta_ss3_ss4 M*)
and *c_ge_1*: *1* $\leq$ *card* (*UNIV* :: *′c set*)
and *c_eq_len*: *card* (*UNIV* :: *′c set*) = *length* (*enum_class.enum* :: *′c list*)
and *home_classification*:
$\forall$ *k*<*k_tm M*. (*mt_pos c′ k* = *0*
$\longrightarrow$ *mt_tape c′ k* (*mt_pos c′ k*) = *LE_block* (*le_tm M*))
$\wedge$ (*mt_pos c′ k* $\geq$ *1*
$\longrightarrow$ *mt_tape c′ k* (*mt_pos c′ k*) $\neq$ *LE_block* (*le_tm M*))
and *c3_pos*: $\bigwedge$*k*. *k* < *k_tm M* $\Longrightarrow$ *mt_pos c3 k* = *mt_pos c′ k* + *1*
and *bl_neq_le*:
(*bl_block* (*bl_tm M*) :: *′c* $\Rightarrow$ *′a*) $\neq$ *LE_block* (*le_tm M*)
and *step4_sub*: (*c3*, *c4*) $\in$ *mttm_step* (*ae_delta_ss4_ss5 M*)
and *c4_buf_not_le_per_tape*:
case mt_state c4 of (_, _, *b*, _, _) $\Rightarrow$
$\forall$ *kk*<*k_tm M*. (*mt_pos c′ kk* = *0*
$\longrightarrow$ *fst* (*b kk*) $\neq$ *LE_block* (*le_tm M*)
$\wedge$ *snd* (*snd* (*b kk*)) $\neq$ *LE_block* (*le_tm M*))
$\wedge$ (*mt_pos c′ kk* = *1*
$\longrightarrow$ *fst* (*snd* (*b kk*)) $\neq$ *LE_block* (*le_tm M*)
$\wedge$ *snd* (*snd* (*b kk*)) $\neq$ *LE_block* (*le_tm M*))
$\wedge$ (*mt_pos c′ kk* $\geq$ *2*
$\longrightarrow$ *fst* (*b kk*) $\neq$ *LE_block* (*le_tm M*)
$\wedge$ *fst* (*snd* (*b kk*)) $\neq$ *LE_block* (*le_tm M*)
$\wedge$ *snd* (*snd* (*b kk*)) $\neq$ *LE_block* (*le_tm M*))
and *inv_ss5*: *ae_inv_ss5 M c4*
and *gamma_c4*: *ae_tape_in_gamma_block M c4*
and *buf_gamma_c4*: *ae_buffer_in_gamma_block M c4*
obtains *c5 q5 ofs5 buf5 dest5 c6 c7 q6 ofs6 buf6 dest6 c8*
where (*c4*, *c5*) $\in$ *mttm_step* (*ae_delta_ss5_ss6 M*)
and (*c4*, *c5*) $\in$ *mttm_step* (*alphabet_enlarge_delta M*)
and *mt_state c4* = (*q5*, *ofs5*, *buf5*, *dest5*, *SS5*)
and *mt_state c5* = (*q5*, *ofs5*, *buf5*, *dest5*, *SS6*)
and *mt_tape c4* = *mt_tape c′*
and
$\forall$ *k*<*k_tm M*. (*mt_pos c′ k* = *0*
$\longrightarrow$ *fst* (*buf5 k*) $\neq$ *LE_block* (*le_tm M*)
$\wedge$ *snd* (*snd* (*buf5 k*)) $\neq$ *LE_block* (*le_tm M*))
$\wedge$ (*mt_pos c′ k* = *1*
$\longrightarrow$ *fst* (*snd* (*buf5 k*)) $\neq$ *LE_block* (*le_tm M*)
$\wedge$ *snd* (*snd* (*buf5 k*)) $\neq$ *LE_block* (*le_tm M*))
$\wedge$ (*mt_pos c′ k* $\geq$ *2*
$\longrightarrow$ *fst* (*buf5 k*) $\neq$ *LE_block* (*le_tm M*)
$\wedge$ *fst* (*snd* (*buf5 k*)) $\neq$ *LE_block* (*le_tm M*)
$\wedge$ *snd* (*snd* (*buf5 k*)) $\neq$ *LE_block* (*le_tm M*))

and
$\forall k.\ mt_tape\ c'\ k\ (mt_pos\ c'\ k + 1) \neq LE_block\ (le_tm\ M)$
and $\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c4\ kk = mt_pos\ c'\ kk$
and $\forall k{<}k_tm\ M.\ mt_tape\ c5\ k\ 0 = LE_block\ (le_tm\ M)$
and
$\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c'\ kk = 1 \longrightarrow dest5\ kk \neq AE_Left$
$\longrightarrow mt_pos\ c5\ kk = 0$
and $(c5,\ c6) \in mttm_step\ (ae_delta_ss6_ss7\ M)$
and $(c5,\ c6) \in mttm_step\ (alphabet_enlarge_delta\ M)$
and $(c6,\ c7) \in mttm_step\ (ae_delta_ss7_ss8\ M)$
and $(c6,\ c7) \in mttm_step\ (alphabet_enlarge_delta\ M)$
and $ae_tape_in_gamma_block\ M\ c7$
and $ae_buffer_in_gamma_block\ M\ c7$
and $mt_state\ c6 = (q6,\ ofs6,\ buf6,\ dest6,\ SS7)$
and $mt_state\ c7 = (q6,\ ofs6,\ buf6,\ dest6,\ SS8)$
and $buf6 = buf5$
and $q6 = q5$
and $ofs6 = ofs5$
and $dest6 = dest5$
and
$\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c'\ kk = 1 \longrightarrow dest5\ kk = AE_Left$
$\longrightarrow mt_pos\ c5\ kk = 2$
and
$\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c'\ kk = 1$
$\longrightarrow mt_tape\ c5\ kk\ 1 = fst\ (snd\ (buf5\ kk))$
and
$\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c'\ kk = 1 \longrightarrow dest5\ kk = AE_Left$
$\longrightarrow mt_pos\ c6\ kk = 1$
and
$\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c'\ kk = 1 \longrightarrow dest5\ kk = AE_Left$
$\longrightarrow mt_pos\ c7\ kk = 0$
and
$\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c'\ kk = 1 \longrightarrow dest5\ kk = AE_Left$
$\longrightarrow mt_tape\ c7\ kk\ 0 = LE_block\ (le_tm\ M)$
and $(c7,\ c8) \in mttm_step\ (ae_delta_ss8_ss1\ M)$
and $(c7,\ c8) \in mttm_step\ (alphabet_enlarge_delta\ M)$
proof −
obtain *c5* **where**
$step5_sub: (c4,\ c5) \in mttm_step\ (ae_delta_ss5_ss6\ M)$
and $step5: (c4,\ c5) \in mttm_step\ (alphabet_enlarge_delta\ M)$
using $ae_step_ss5_ss6_exists[OF\ inv_ss5\ gamma_c4\ buf_gamma_c4]$ **by** *blast*
have $inv_ss6: ae_inv_ss6\ M\ c5$
using $ae_step_ss5_ss6_invariant[OF\ vM\ inv_ss5\ step5_sub]$ **.**
have $gamma_c5: ae_tape_in_gamma_block\ M\ c5$
using $ae_step_alphabet_enlarge_gamma_preserve[OF\ gamma_c4\ step5]$ **.**
have $buf_gamma_c5: ae_buffer_in_gamma_block\ M\ c5$
using $ae_step_alphabet_enlarge_buffer_gamma_preserve[OF\ \ vM\ buf_gamma_c4$

gamma_c4 step5] .

— State shapes at *c4* (idx SS5) and *c5* (idx SS6). Identical derivation across regimes; the substep preserves (*q*, *ofs*, *buf*, *dest*) and flips only the stage index.
obtain *q5 ofs5 buf5 dest5* **where**
c4_state: *mt_state c4* = (*q5*, *ofs5*, *buf5*, *dest5*, *SS5*)
using *inv_ss5* **unfolding** *ae_inv_ss5_def*
by (*cases mt_state c4*) *auto*
have *c5_state*: *mt_state c5* = (*q5*, *ofs5*, *buf5*, *dest5*, *SS6*)
using *step5_sub c4_state*
by (*auto simp*: *ae_delta_ss5_ss6_def elim*: *mttm_step.cases*)

— Sub-step 5b: M'-tape preservation through SS1→SS2 → ... →SS5. Each sub-delta has $a' = a$ (idempotent on M'-tape), so the cumulative tape is unchanged through *c1*...*c4*. Uniform across regimes.
have *nw_12*: $\bigwedge q\ a\ q'\ a'\ d.$
$(q, a, q', a', d) \in ae_delta_ss1_ss2\ M \Longrightarrow a' = a$
by (*auto simp*: *ae_delta_ss1_ss2_def*)
have *nw_23*: $\bigwedge q\ a\ q'\ a'\ d.$
$(q, a, q', a', d) \in ae_delta_ss2_ss3\ M \Longrightarrow a' = a$
by (*auto simp*: *ae_delta_ss2_ss3_def*)
have *nw_34*: $\bigwedge q\ a\ q'\ a'\ d.$
$(q, a, q', a', d) \in ae_delta_ss3_ss4\ M \Longrightarrow a' = a$
by (*auto simp*: *ae_delta_ss3_ss4_def*)
have *nw_45*: $\bigwedge q\ a\ q'\ a'\ d.$
$(q, a, q', a', d) \in ae_delta_ss4_ss5\ M \Longrightarrow a' = a$
by (*auto simp*: *ae_delta_ss4_ss5_def*)
have *tape_c1_eq*: *mt_tape c1* = *mt_tape c'*
using *mttm_step_no_write_tape*[*OF step1_sub nw_12*] .
have *tape_c2_eq*: *mt_tape c2* = *mt_tape c1*
using *mttm_step_no_write_tape*[*OF step2_sub nw_23*] .
have *tape_c3_eq*: *mt_tape c3* = *mt_tape c2*
using *mttm_step_no_write_tape*[*OF step3_sub nw_34*] .
have *tape_c4_eq*: *mt_tape c4* = *mt_tape c3*
using *mttm_step_no_write_tape*[*OF step4_sub nw_45*] .
have *tape_c4_eq_c'*: *mt_tape c4* = *mt_tape c'*
using *tape_c1_eq tape_c2_eq tape_c3_eq tape_c4_eq* **by** *simp*

— Buf5 per-tape regime-aware non-LE shape, transferred from *c4_buf_not_le_per_tape* via *c4_state*. Pos=0 tapes have left+right non-LE; pos=1 tapes have home+right non-LE; pos≥2 tapes have all three non-LE. This is the regime-aware generalisation of *buf5_not_le_lr* (le0) and *buf5_not_le_hr* (le1) and the steady-state full *buf5_not_le*.
have *buf5_not_le_per_tape*:
$\forall k{<}k_tm\ M.\ (mt_pos\ c'\ k = 0$
$\longrightarrow fst\ (buf5\ k) \neq LE_block\ (le_tm\ M)$
$\wedge\ snd\ (snd\ (buf5\ k)) \neq LE_block\ (le_tm\ M))$
$\wedge\ (mt_pos\ c'\ k = 1$
$\longrightarrow fst\ (snd\ (buf5\ k)) \neq LE_block\ (le_tm\ M)$

```
        ∧ snd (snd (buf5 k)) ≠ LE_block (le_tm M))
    ∧ (mt_pos c' k ≥ 2
        ⟶ fst (buf5 k) ≠ LE_block (le_tm M)
          ∧ fst (snd (buf5 k)) ≠ LE_block (le_tm M)
          ∧ snd (snd (buf5 k)) ≠ LE_block (le_tm M))
  using c4_buf_not_le_per_tape c4_state by simp
```

— Sub-step 5c: *right_not_le_c'*. The unified "right-of-home not LE" fact at c': for every tape, the block at *mt_pos c' k + 1* is not *LE_block*. Powers the SS4→SS5 head-trajectory derivation (*c3* reads at *mt_pos c3 = mt_pos c' + 1*, direction *L* when read ≠ *LE*). Per-tape case-split on regime: pos=0 uses *no_le* i=0 (cell 1); pos=1 uses *no_le* i=c (cell c+1); pos≥2 uses *no_le* i=2c (cell pos·c+1). All three cell indices coincide with the correspondence-derived address *mt_pos c' k · c + 1*.

```
  have right_not_le_c':
    ∀k. mt_tape c' k (mt_pos c' k + 1) ≠ LE_block (le_tm M)
  proof (intro allI)
    fix k :: nat
    show mt_tape c' k (mt_pos c' k + 1) ≠ LE_block (le_tm M)
    proof (cases k < k_tm M)
      case True
    let ?x = (enum_class.enum :: 'c list) ! 0
    let ?c = card (UNIV :: 'c set)
    have zero_lt_len: 0 < length (enum_class.enum :: 'c list)
      using c_ge_1 c_eq_len by linarith
    have x_idx0: c_idx ?x = 0
      using c_idx_enum_nth[OF zero_lt_len] .
    have tc_k: ae_tape_correspondence (le_tm M) (mt_tape cM k) (mt_tape c'
k)
      using tape_corr[rule_format, OF True] .
    have addr_via_tc:
       mt_tape cM k (mt_pos c' k * ?c + 1)
         = mt_tape c' k (mt_pos c' k + 1) ?x
    proof −
      from tc_k have unf:
         ∀s≥1. ∀i. mt_tape cM k ((s − 1) * ?c + c_idx i + 1)
                   = mt_tape c' k s i
        unfolding ae_tape_correspondence_def by simp
      have s_ge1: mt_pos c' k + 1 ≥ 1 by simp
      from unf s_ge1 have eq:
         mt_tape cM k ((mt_pos c' k + 1 − 1) * ?c + c_idx ?x + 1)
           = mt_tape c' k (mt_pos c' k + 1) ?x
        by blast
      thus ?thesis using x_idx0 by simp
    qed
    consider (le0) mt_pos c' k = 0
        | (le1) mt_pos c' k = 1
        | (steady) mt_pos c' k ≥ 2
      by linarith
    hence mt_tape c' k (mt_pos c' k + 1) ?x ≠ le_tm M
```

```
  proof cases
    case le0
    have idx_lt: (0 :: nat) < ?c using c_ge_1 by simp
    have nle: mt_tape cM k (Suc 0) ≠ le_tm M
      using no_le_per_tape le0 idx_lt by blast
    thus ?thesis using addr_via_tc le0 by simp
  next
    case le1
    have idx_lt: ?c < 2 * ?c using c_ge_1 by simp
    have nle: mt_tape cM k (Suc ?c) ≠ le_tm M
      using no_le_per_tape le1 idx_lt by blast
    hence mt_tape cM k (1 * ?c + 1) ≠ le_tm M by simp
    thus ?thesis using addr_via_tc le1 by simp
  next
    case steady
    have idx_lt: 2 * ?c < 3 * ?c using c_ge_1 by simp
    have addr_eq:
       (mt_pos c' k - 2) * ?c + 1 + 2 * ?c = mt_pos c' k * ?c + 1
      using steady by (simp add: algebra_simps diff_mult_distrib)
    have nle: mt_tape cM k ((mt_pos c' k - 2) * ?c + 1 + 2 * ?c) ≠ le_tm M
      using no_le_per_tape steady idx_lt by blast
    thus ?thesis using addr_via_tc addr_eq by simp
  qed
  thus mt_tape c' k (mt_pos c' k + 1) ≠ LE_block (le_tm M)
    unfolding LE_block_def by auto
  next
    case False
    hence kge: k_tm M ≤ k by simp
    have mt_tape c' k (mt_pos c' k + 1) = bl_block (bl_tm M)
      using gamma_c4 kge
      unfolding ae_tape_in_gamma_block_def tape_c4_eq_c'[symmetric]
      by simp
    thus ?thesis using bl_neq_le by simp
  qed
qed

— Sub-step 5d: SS4→SS5 head trajectory. At c3's head position mt_pos
c3 k = mt_pos c' k + 1, the M'-tape reads mt_tape c' k (mt_pos c' k +
1) (via tape_c3_eq_c') which is not LE_block by right_not_le_c'. Hence
ae_delta_ss4_ss5's direction is L, landing c4's head at mt_pos c' k. Uniform
across regimes: pos=0 lands at 0, pos=1 lands at 1, pos≥2 lands at pos.
  have tape_c3_eq_c': mt_tape c3 = mt_tape c'
    using tape_c1_eq tape_c2_eq tape_c3_eq by simp
  have c4_pos: ⋀kk. kk < k_tm M ⟹ mt_pos c4 kk = mt_pos c' kk
  proof −
    fix kk assume kklt: kk < k_tm M
    obtain qq tts nn qq' aa4 dr4 where
       c3_eq4: c3 = Config_M qq tts nn
      and c4_eq: c4 = Config_M qq'
```

```
                    (λk. (tts k)(nn k := aa4 k))
                    (λk. go_dir (dr4 k) (nn k))
    and tr_in: (qq, λk. tts k (nn k), qq', aa4, dr4)
              ∈ ae_delta_ss4_ss5 M
    using step4_sub by (auto elim: mttm_step.cases)
  have dr4_eq:
     dr4 = (λk. if k < k_tm M
                then (if tts k (nn k) = LE_block (le_tm M)
                      then dir.N else dir.L)
                else dir.N)
    using tr_in by (auto simp: ae_delta_ss4_ss5_def)
  have nn_kk: nn kk = mt_pos c3 kk using c3_eq4 by simp
  have tts_kk: tts kk (nn kk) = mt_tape c3 kk (mt_pos c3 kk)
    using c3_eq4 nn_kk by simp
  have c3_kk: mt_pos c3 kk = mt_pos c' kk + 1
    using c3_pos kklt by simp
  have read_at_c3:
     tts kk (nn kk) = mt_tape c' kk (mt_pos c' kk + 1)
    using tts_kk c3_kk tape_c3_eq_c' by simp
  have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
    using read_at_c3 right_not_le_c' by simp
  have dr4_L: dr4 kk = dir.L
    using dr4_eq read_ne_LE kklt by simp
  have mt_pos c4 kk = go_dir (dr4 kk) (nn kk) using c4_eq by simp
  also have ... = (nn kk) − 1 using dr4_L by simp
  also have ... = (mt_pos c3 kk) − 1 using nn_kk by simp
  also have ... = (mt_pos c' kk + 1) − 1 using c3_kk by simp
  also have ... = mt_pos c' kk by simp
  finally show mt_pos c4 kk = mt_pos c' kk .
qed
```

— Sub-step 5e: *tape_c5_zero_le*. The M'-tape cell 0 stays *LE_block* across SS5. For pos≥1 tapes *c4_pos* ≠ *0*, so cell 0 is off-head and preserved from *c4* (which equals *c'* by *tape_c4_eq_c'*, hence *LE_block* by *le_anchor*). For pos=0 tapes *c4_pos* = *0*, so the SS5 action fires at cell 0; but the LE-guard fires (read = *LE_block* via *le_anchor*) and idempotently writes *LE_block* back. Either way cell 0 stays.

```
  have tape_c5_zero_le: ∀k<k_tm M. mt_tape c5 k 0 = LE_block (le_tm M)
  proof (intro allI impI)
    fix k assume klt: k < k_tm M
    consider (le0) mt_pos c' k = 0 | (rest) mt_pos c' k ≥ 1 by linarith
    thus mt_tape c5 k 0 = LE_block (le_tm M)
    proof cases
      case rest
      have c4_kk_ne0: mt_pos c4 k ≠ 0
        using c4_pos[OF klt] rest by simp
      have zero_ne_pos: (0 :: nat) ≠ mt_pos c4 k using c4_kk_ne0 by simp
      have c5_zero_eq_c4: mt_tape c5 k 0 = mt_tape c4 k 0
        using mttm_step_tape_off_head[OF step5 zero_ne_pos] .
```

```
    have mt_tape c4 k 0 = mt_tape c' k 0
      using tape_c4_eq_c' by simp
    also have ... = LE_block (le_tm M)
      using le_anchor[rule_format, OF klt] by simp
    finally show ?thesis using c5_zero_eq_c4 by simp
  next
    case le0
    have c4_pos_kk: mt_pos c4 k = 0
      using c4_pos[OF klt] le0 by simp
    obtain qq tts nn qq' aa5 dr5 where
        c4_eq5: c4 = Config_M qq tts nn
      and c5_eq: c5 = Config_M qq'
                      (λk. (tts k)(nn k := aa5 k))
                      (λk. go_dir (dr5 k) (nn k))
      and tr_in: (qq, λk. tts k (nn k), qq', aa5, dr5)
                 ∈ ae_delta_ss5_ss6 M
      using step5_sub by (auto elim: mttm_step.cases)
    obtain q ofs buf dest where
        qq_eq: qq = (q, ofs, buf, dest, SS5)
      and aa5_eq: aa5 = (λk. if k < k_tm M
                             then fst (ae_ss5_action (le_tm M)
                                        (tts k (nn k)) (buf k) (dest k))
                             else bl_block (bl_tm M))
      using tr_in by (auto simp: ae_delta_ss5_ss6_def)
    have nn_kk: nn k = mt_pos c4 k using c4_eq5 by simp
    have nn_kk_0: nn k = 0 using nn_kk c4_pos_kk by simp
    have tts_kk: tts k (nn k) = mt_tape c4 k (mt_pos c4 k)
      using c4_eq5 nn_kk by simp
    have read_at: tts k (nn k) = mt_tape c' k 0
      using tts_kk c4_pos_kk tape_c4_eq_c' by simp
    have read_LE: tts k (nn k) = LE_block (le_tm M)
      using read_at le_anchor[rule_format, OF klt] by simp
    obtain l h r where buf_k: buf k = (l, h, r)
      by (cases buf k) auto
    have aa5_LE: aa5 k = LE_block (le_tm M)
      using aa5_eq read_LE buf_k klt by simp
    have step_apply: mt_tape c5 k 0 = ((tts k)(nn k := aa5 k)) 0
      using c5_eq by simp
    have hit: ((tts k)(nn k := aa5 k)) 0 = aa5 k
      using nn_kk_0 by simp
    show ?thesis using step_apply hit aa5_LE by simp
  qed
qed
```

— Sub-step 5f: head trajectory at SS5 for pos=1 tapes. When the regime is pos=1 and *dest5 k* ≠ *AE_Left*, the SS5 action moves L, landing *c5*'s head at block 0 (the LE position). Read at *c4_pos = 1* is *mt_tape c' k 1* ≠ *LE* (via *home_classification* pos≥1 branch); buf5 home ≠ *LE* (via *buf5_not_le_per_tape* pos=1 branch). Neither LE-guard in *ae_ss5_action* fires; direction is L when *dest*

```
≠ AE_Left.
  have c5_pos_for_pos1:
     ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 1 ⟹ dest5 kk ≠ AE_Left
          ⟹ mt_pos c5 kk = 0
  proof −
    fix kk
    assume kklt: kk < k_tm M
    assume pos_kk: mt_pos c' kk = 1
    assume dest_ne: dest5 kk ≠ AE_Left
    obtain qq tts nn qq' aa5 dr5 where
       c4_eq5: c4 = Config_M qq tts nn
      and c5_eq: c5 = Config_M qq'
                      (λk. (tts k)(nn k := aa5 k))
                      (λk. go_dir (dr5 k) (nn k))
      and tr_in: (qq, λk. tts k (nn k), qq', aa5, dr5)
                 ∈ ae_delta_ss5_ss6 M
      using step5_sub by (auto elim: mttm_step.cases)
    obtain q ofs buf dest where
       qq_eq: qq = (q, ofs, buf, dest, SS5)
      and dr5_eq:
         dr5 = (λk. if k < k_tm M
                    then snd (ae_ss5_action (le_tm M)
                                 (tts k (nn k)) (buf k) (dest k))
                    else dir.N)
      using tr_in by (auto simp: ae_delta_ss5_ss6_def)
    have qq_state: qq = (q5, ofs5, buf5, dest5, SS5)
      using c4_state c4_eq5 by simp
    have buf_eq: buf = buf5 using qq_eq qq_state by simp
    have dest_eq: dest = dest5 using qq_eq qq_state by simp
    have nn_kk: nn kk = mt_pos c4 kk using c4_eq5 by simp
    have c4_kk: mt_pos c4 kk = 1 using c4_pos kklt pos_kk by simp
    have tts_kk: tts kk (nn kk) = mt_tape c4 kk (mt_pos c4 kk)
      using c4_eq5 nn_kk by simp
    have read_at_home:
       tts kk (nn kk) = mt_tape c' kk 1
      using tts_kk c4_kk tape_c4_eq_c' by simp
    have home_ne_LE: mt_tape c' kk 1 ≠ LE_block (le_tm M)
    proof −
      have pos_ge1: (1 :: nat) ≤ mt_pos c' kk using pos_kk by simp
      have mt_tape c' kk (mt_pos c' kk) ≠ LE_block (le_tm M)
        using home_classification[rule_format, OF kklt] pos_ge1 by blast
      thus ?thesis using pos_kk by simp
    qed
    have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
      using read_at_home home_ne_LE by simp
    obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
      using prod.exhaust by metis
    have h_ne_LE: hh ≠ LE_block (le_tm M)
    proof −
```

```
    have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
      using buf5_not_le_per_tape[rule_format, OF kklt] pos_kk by blast
    thus ?thesis using buf5_kk by simp
  qed
  have dr5_kk:
     dr5 kk = (if dest5 kk = AE_Left then dir.R else dir.L)
    using dr5_eq buf_eq dest_eq buf5_kk read_ne_LE h_ne_LE kklt by simp
  have dr5_kk_L: dr5 kk = dir.L
    using dr5_kk dest_ne by simp
  have mt_pos c5 kk = go_dir (dr5 kk) (nn kk)
    using c5_eq by simp
  also have ... = (nn kk) − 1 using dr5_kk_L by simp
  also have ... = mt_pos c4 kk − 1 using nn_kk by simp
  also have ... = 1 − 1 using c4_kk by simp
  also have ... = 0 by simp
  finally show mt_pos c5 kk = 0 .
qed
```

— Sub-step 5g: *pos_link_c5*. Discharges *ae_position_link M c5* from regime-aware buffer facts + head trajectory. Two conjuncts at idx=SS6: right-slot non-LE (uniform, follows from each regime branch of *buf5_not_le_per_tape*) and the SS6 conditional (*dest ≠ AE_Left + fst (buf k) = LE_block* implies head reads *LE_block*). The SS6 conditional case-splits per-tape: pos=0/≥2 tapes have *fst (buf5 k) ≠ LE_block* so the antecedent is false (vacuous); pos=1 tapes use *c5_pos_for_pos1 + tape_c5_zero_le* for the substantive case.

```
  have pos_link_c5: ae_position_link M c5
  proof −
    have r_not_le: ∀k<k_tm M. snd (snd (buf5 k)) ≠ LE_block (le_tm M)
    proof (intro allI impI)
      fix k
      assume klt: k < k_tm M
      consider (le0) mt_pos c' k = 0
             | (le1) mt_pos c' k = 1
             | (steady) mt_pos c' k ≥ 2
        by linarith
      thus snd (snd (buf5 k)) ≠ LE_block (le_tm M)
      proof cases
        case le0   thus ?thesis using buf5_not_le_per_tape[rule_format, OF klt]
by blast
      next
        case le1   thus ?thesis using buf5_not_le_per_tape[rule_format, OF klt]
by blast
      next
       case steady thus ?thesis using buf5_not_le_per_tape[rule_format, OF klt]
by blast
      qed
    qed
    have ss6_cond:
       ∀k<k_tm M. dest5 k ≠ AE_Left
```

```
        ⟶ fst (buf5 k) = LE_block (le_tm M)
        ⟶ mt_tape c5 k (mt_pos c5 k) = LE_block (le_tm M)
  proof (intro allI impI)
    fix k
    assume klt: k < k_tm M
    assume dest_ne: dest5 k ≠ AE_Left
    assume l_eq_le: fst (buf5 k) = LE_block (le_tm M)
    consider (le0) mt_pos c' k = 0
        | (le1) mt_pos c' k = 1
        | (steady) mt_pos c' k ≥ 2
      by linarith
    thus mt_tape c5 k (mt_pos c5 k) = LE_block (le_tm M)
    proof cases
      case le0
      have fst (buf5 k) ≠ LE_block (le_tm M)
        using buf5_not_le_per_tape[rule_format, OF klt] le0 by blast
      thus ?thesis using l_eq_le by simp
    next
      case le1
      have c5_pos_zero: mt_pos c5 k = 0
        using c5_pos_for_pos1[OF klt le1 dest_ne] .
      have mt_tape c5 k 0 = LE_block (le_tm M)
        using tape_c5_zero_le[rule_format, OF klt] by simp
      thus ?thesis using c5_pos_zero by simp
    next
      case steady
      have fst (buf5 k) ≠ LE_block (le_tm M)
        using buf5_not_le_per_tape[rule_format, OF klt] steady by blast
      thus ?thesis using l_eq_le by simp
    qed
  qed
  show ?thesis
    unfolding ae_position_link_def
    using r_not_le ss6_cond c5_state by simp
qed
```

— Sub-step 5h: SS6→SS7 invocation. Discharge *ae_le_compat_ss6* from *pos_link_c5*; invoke *ae_step_ss6_ss7_exists*. Propagate *pos_link* from *c5* (idx SS6) to *c6* via the arm-agnostic preservation lemma (the three aux conditionals are vacuous: idx-pre is SS6, not SS4/5/7). Identical to the corresponding le0/le1 / steady blocks.

```
  have le_compat_ss6_c5: ae_le_compat_ss6 M c5
    using ae_position_link_discharges_ss6[OF inv_ss6 pos_link_c5] .
  obtain c6 where
      step6_sub: (c5, c6) ∈ mttm_step (ae_delta_ss6_ss7 M)
    and step6: (c5, c6) ∈ mttm_step (alphabet_enlarge_delta M)
    using ae_step_ss6_ss7_exists[OF inv_ss6 gamma_c5 buf_gamma_c5
                                    le_compat_ss6_c5] by blast
  have inv_ss7: ae_inv_ss7 M c6
```

using *ae_step_ss6_ss7_invariant*[*OF vM inv_ss6 step6_sub*] .
have *gamma_c6*: *ae_tape_in_gamma_block M c6*
using *ae_step_alphabet_enlarge_gamma_preserve*[*OF gamma_c5 step6*] .
have *buf_gamma_c6*: *ae_buffer_in_gamma_block M c6*
using *ae_step_alphabet_enlarge_buffer_gamma_preserve*[*OF vM buf_gamma_c5 gamma_c5 step6*] .
have *c5_idx_ss6*: *snd* (*snd* (*snd* (*snd* (*mt_state c5*)))) = *SS6*
using *c5_state* **by** *simp*
have *aux_45_at_c5*: (*c5*, *c6*) ∈ *mttm_step* (*ae_delta_ss4_ss5 M*) ⟹ *ae_position_link M c6*
using *ae_pos_link_aux_void_ss4_ss5*[**where** *M* = *M* **and** *c_pre* = *c5* **and** *c_post* = *c6*] *c5_idx_ss6* **by** *simp*
have *aux_56_at_c5*: (*c5*, *c6*) ∈ *mttm_step* (*ae_delta_ss5_ss6 M*) ⟹ *ae_position_link M c6*
using *ae_pos_link_aux_void_ss5_ss6*[**where** *M* = *M* **and** *c_pre* = *c5* **and** *c_post* = *c6*] *c5_idx_ss6* **by** *simp*
have *aux_78_at_c5*: (*c5*, *c6*) ∈ *mttm_step* (*ae_delta_ss7_ss8 M*) ⟹ *ae_position_link M c6*
using *ae_pos_link_aux_void_ss7_ss8*[**where** *M* = *M* **and** *c_pre* = *c5* **and** *c_post* = *c6*] *c5_idx_ss6* **by** *simp*
have *pos_link_c6*: *ae_position_link M c6*
using *ae_step_alphabet_enlarge_position_link_preserve*[*OF pos_link_c5 step6 aux_45_at_c5 aux_56_at_c5 aux_78_at_c5*] .

— Sub-step 5i: SS7→SS8 invocation. Discharge *ae_le_compat_ss7* from *pos_link_c6*; invoke *ae_step_ss7_ss8_exists*. Buffer content is preserved across SS5→SS6→SS7 (the substeps change only the tape and stage index, not the buffer component), so the *buf5* per-tape non-LE shape transfers to *buf6* = *buf5* and *c7*'s state inherits the same buffer triple.
have *le_compat_ss7_c6*: *ae_le_compat_ss7 M c6*
using *ae_position_link_discharges_ss7*[*OF inv_ss7 pos_link_c6*] .
obtain *c7* **where**
step7_sub: (*c6*, *c7*) ∈ *mttm_step* (*ae_delta_ss7_ss8 M*)
and *step7*: (*c6*, *c7*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
using *ae_step_ss7_ss8_exists*[*OF inv_ss7 gamma_c6 buf_gamma_c6 le_compat_ss7_c6*] **by** *blast*
have *inv_ss8*: *ae_inv_ss8 M c7*
using *ae_step_ss7_ss8_invariant*[*OF vM inv_ss7 step7_sub*] .
have *gamma_c7*: *ae_tape_in_gamma_block M c7*
using *ae_step_alphabet_enlarge_gamma_preserve*[*OF gamma_c6 step7*] .
have *buf_gamma_c7*: *ae_buffer_in_gamma_block M c7*
using *ae_step_alphabet_enlarge_buffer_gamma_preserve*[*OF vM buf_gamma_c6 gamma_c6 step7*] .

obtain *q6 ofs6 buf6 dest6* **where**
c5_state_eq: *mt_state c5* = (*q6*, *ofs6*, *buf6*, *dest6*, *SS6*)
using *inv_ss6* **unfolding** *ae_inv_ss6_def*
by (*cases mt_state c5*) *auto*
have *c6_state*: *mt_state c6* = (*q6*, *ofs6*, *buf6*, *dest6*, *SS7*)
using *step6_sub c5_state_eq*
by (*auto simp*: *ae_delta_ss6_ss7_def elim*: *mttm_step.cases*)
have *c7_state*: *mt_state c7* = (*q6*, *ofs6*, *buf6*, *dest6*, *SS8*)
using *step7_sub c6_state*
by (*auto simp*: *ae_delta_ss7_ss8_def elim*: *mttm_step.cases*)
have *buf6_eq_buf5*: *buf6* = *buf5*
using *c5_state_eq c5_state* **by** *simp*
have *q6_eq_q5*: *q6* = *q5*
using *c5_state_eq c5_state* **by** *simp*
have *ofs6_eq_ofs5*: *ofs6* = *ofs5*
using *c5_state_eq c5_state* **by** *simp*
have *dest6_eq_dest5*: *dest6* = *dest5*
using *c5_state_eq c5_state* **by** *simp*

— Sub-step 5j: head trajectory at SS5 for pos=1 tapes under *dest* = *AE_Left*. Mirror of *c5_pos_for_pos1* with R-direction (when *dest* = *AE_Left*, *ae_ss5_action*'s else-branch returns (*h*, *R*)); head lands at block 2. Plus *tape_c5_at_one_pos1*: the SS5 write at *c4_pos* = *1* records the home buffer slot *fst* (*snd* (*buf5 k*)) at *c5*'s cell 1.

have *c5_pos_for_pos1_dest_left*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c′ kk* = *1* ⟹ *dest5 kk* = *AE_Left*
⟹ *mt_pos c5 kk* = *2*
proof −
fix *kk*
assume *kklt*: *kk* < *k_tm M*
assume *pos_kk*: *mt_pos c′ kk* = *1*
assume *dest_eq*: *dest5 kk* = *AE_Left*
obtain *qq tts nn qq′ aa5 dr5* **where**
c4_eq5: *c4* = *Config*$_M$ *qq tts nn*
and *c5_eq*: *c5* = *Config*$_M$ *qq′*
(λ*k*. (*tts k*)(*nn k* := *aa5 k*))
(λ*k*. *go_dir* (*dr5 k*) (*nn k*))
and *tr_in*: (*qq*, λ*k*. *tts k* (*nn k*), *qq′*, *aa5*, *dr5*)
∈ *ae_delta_ss5_ss6 M*
using *step5_sub* **by** (*auto elim*: *mttm_step.cases*)
obtain *q ofs buf dest* **where**
qq_eq: *qq* = (*q*, *ofs*, *buf*, *dest*, *SS5*)
and *dr5_eq*:
dr5 = (λ*k*. *if k* < *k_tm M*
then snd (*ae_ss5_action* (*le_tm M*)
(*tts k* (*nn k*)) (*buf k*) (*dest k*))
else dir.N)
using *tr_in* **by** (*auto simp*: *ae_delta_ss5_ss6_def*)
have *qq_state*: *qq* = (*q5*, *ofs5*, *buf5*, *dest5*, *SS5*)

```
    using c4_state c4_eq5 by simp
  have buf_eq: buf = buf5 using qq_eq qq_state by simp
  have dest_eq2: dest = dest5 using qq_eq qq_state by simp
  have nn_kk: nn kk = mt_pos c4 kk using c4_eq5 by simp
  have c4_kk: mt_pos c4 kk = 1 using c4_pos kklt pos_kk by simp
  have tts_kk: tts kk (nn kk) = mt_tape c4 kk (mt_pos c4 kk)
    using c4_eq5 nn_kk by simp
  have read_at_home: tts kk (nn kk) = mt_tape c' kk 1
    using tts_kk c4_kk tape_c4_eq_c' by simp
  have home_ne_LE: mt_tape c' kk 1 ≠ LE_block (le_tm M)
  proof −
    have pos_ge1: (1 :: nat) ≤ mt_pos c' kk using pos_kk by simp
    have mt_tape c' kk (mt_pos c' kk) ≠ LE_block (le_tm M)
      using home_classification[rule_format, OF kklt] pos_ge1 by blast
    thus ?thesis using pos_kk by simp
  qed
  have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
    using read_at_home home_ne_LE by simp
  obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
    using prod.exhaust by metis
  have h_ne_LE: hh ≠ LE_block (le_tm M)
  proof −
    have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
      using buf5_not_le_per_tape[rule_format, OF kklt] pos_kk by blast
    thus ?thesis using buf5_kk by simp
  qed
  have dr5_kk:
      dr5 kk = (if dest5 kk = AE_Left then dir.R else dir.L)
    using dr5_eq buf_eq dest_eq2 buf5_kk read_ne_LE h_ne_LE kklt by simp
  have dr5_kk_R: dr5 kk = dir.R
    using dr5_kk dest_eq by simp
  have mt_pos c5 kk = go_dir (dr5 kk) (nn kk)
    using c5_eq by simp
  also have ... = Suc (nn kk) using dr5_kk_R by simp
  also have ... = Suc (mt_pos c4 kk) using nn_kk by simp
  also have ... = Suc 1 using c4_kk by simp
  also have ... = 2 by simp
  finally show mt_pos c5 kk = 2 .
qed

have tape_c5_at_one_pos1:
    ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 1
         ⟹ mt_tape c5 kk 1 = fst (snd (buf5 kk))
proof −
  fix kk
  assume kklt: kk < k_tm M
  assume pos_kk: mt_pos c' kk = 1
  obtain qq tts nn qq' aa5 dr5 where
      c4_eq5: c4 = Config_M qq tts nn
```

```
  and c5_eq: c5 = Config_M qq'
                  (λk. (tts k)(nn k := aa5 k))
                  (λk. go_dir (dr5 k) (nn k))
  and tr_in: (qq, λk. tts k (nn k), qq', aa5, dr5)
             ∈ ae_delta_ss5_ss6 M
  using step5_sub by (auto elim: mttm_step.cases)
obtain q ofs buf dest where
   qq_eq: qq = (q, ofs, buf, dest, SS5)
  and aa5_eq:
     aa5 = (λk. if k < k_tm M then fst (ae_ss5_action (le_tm M)
                    (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
  using tr_in by (auto simp: ae_delta_ss5_ss6_def)
have qq_state: qq = (q5, ofs5, buf5, dest5, SS5)
  using c4_state c4_eq5 by simp
have buf_eq: buf = buf5 using qq_eq qq_state by simp
have dest_eq: dest = dest5 using qq_eq qq_state by simp
have nn_kk: nn kk = mt_pos c4 kk using c4_eq5 by simp
have c4_kk: mt_pos c4 kk = 1 using c4_pos kklt pos_kk by simp
have tts_kk: tts kk (nn kk) = mt_tape c4 kk (mt_pos c4 kk)
  using c4_eq5 nn_kk by simp
have read_at_home: tts kk (nn kk) = mt_tape c' kk 1
  using tts_kk c4_kk tape_c4_eq_c' by simp
have home_ne_LE: mt_tape c' kk 1 ≠ LE_block (le_tm M)
proof −
  have pos_ge1: (1 :: nat) ≤ mt_pos c' kk using pos_kk by simp
  have mt_tape c' kk (mt_pos c' kk) ≠ LE_block (le_tm M)
    using home_classification[rule_format, OF kklt] pos_ge1 by blast
  thus ?thesis using pos_kk by simp
qed
have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
  using read_at_home home_ne_LE by simp
obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
  using prod.exhaust by metis
have h_ne_LE: hh ≠ LE_block (le_tm M)
proof −
  have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
    using buf5_not_le_per_tape[rule_format, OF kklt] pos_kk by blast
  thus ?thesis using buf5_kk by simp
qed
have aa5_kk: aa5 kk = fst (snd (buf5 kk))
  using aa5_eq buf_eq dest_eq buf5_kk read_ne_LE h_ne_LE kklt by simp
have c5_tape_kk: mt_tape c5 kk = (tts kk)(nn kk := aa5 kk)
  using c5_eq by simp
have step_apply: mt_tape c5 kk 1 = ((tts kk)(nn kk := aa5 kk)) 1
  using c5_tape_kk by simp
have hit: ((tts kk)(nn kk := aa5 kk)) 1 = aa5 kk
  using nn_kk c4_kk by simp
show mt_tape c5 kk 1 = fst (snd (buf5 kk))
  using step_apply hit aa5_kk by simp
```

qed

— Sub-step 5k: c6 trajectory for pos=1 + dest=*AE_Left*. *c5_pos = 2*; SS6 reads *mt_tape c5 k 2* which is off-head from SS5's write at *c4_pos = 1*, so = *mt_tape c4 k 2 = mt_tape c′ k 2*. *right_not_le_c′* instantiated at pos=1 gives ≠ *LE_block*. Buf5 home ≠ *LE* (*buf5_not_le_per_tape*). *ae_ss6_action*'s else-branch fires with *ds = AE_Left*: writes *r*, direction *L*; head lands at *c6_pos = c5_pos − 1 = 1*. Plus *tape_c6_zero_le_pos1_dest_left*: cell 0 is off-head from SS6's write at *c5_pos = 2*, hence = *LE* via *tape_c5_zero_le*.

have *c6_pos_for_pos1_dest_left*:
  $\bigwedge$*kk*. *kk < k_tm M* $\Longrightarrow$ *mt_pos c′ kk = 1* $\Longrightarrow$ *dest5 kk = AE_Left*
    $\Longrightarrow$ *mt_pos c6 kk = 1*
proof −
 fix *kk*
 assume *kklt*: *kk < k_tm M*
 assume *pos_kk*: *mt_pos c′ kk = 1*
 assume *dest_eq*: *dest5 kk = AE_Left*
 obtain *qq tts nn qq′ aa6 dr6* **where**
   c5_eq6: *c5 = Config*$_M$ *qq tts nn*
  and *c6_eq*: *c6 = Config*$_M$ *qq′*
                (λ*k*. (*tts k*)(*nn k* := *aa6 k*))
                (λ*k*. *go_dir* (*dr6 k*) (*nn k*))
  and *tr_in*: (*qq*, λ*k*. *tts k* (*nn k*), *qq′*, *aa6*, *dr6*)
          ∈ *ae_delta_ss6_ss7 M*
  using *step6_sub* **by** (*auto elim*: *mttm_step.cases*)
 obtain *q ofs buf dest* **where**
   qq_eq: *qq* = (*q*, *ofs*, *buf*, *dest*, *SS6*)
  and *dr6_eq*:
    dr6 = (λ*k*. *if k < k_tm M then snd* (*ae_ss6_action* (*le_tm M*)
                (*tts k* (*nn k*)) (*buf k*) (*dest k*)) *else dir.N*)
  using *tr_in* **by** (*auto simp*: *ae_delta_ss6_ss7_def*)
 have *qq_state*: *qq* = (*q5*, *ofs5*, *buf5*, *dest5*, *SS6*)
  using *c5_state c5_eq6* **by** *simp*
 have *buf_eq*: *buf = buf5* **using** *qq_eq qq_state* **by** *simp*
 have *dest_eq2*: *dest = dest5* **using** *qq_eq qq_state* **by** *simp*
 have *nn_kk*: *nn kk = mt_pos c5 kk* **using** *c5_eq6* **by** *simp*
 have *c5_kk*: *mt_pos c5 kk = 2*
  using *c5_pos_for_pos1_dest_left*[*OF kklt pos_kk dest_eq*] **.**
 have *tts_kk*: *tts kk* (*nn kk*) = *mt_tape c5 kk* (*mt_pos c5 kk*)
  using *c5_eq6 nn_kk* **by** *simp*
 have *c5_tape_at_2*: *mt_tape c5 kk 2 = mt_tape c4 kk 2*
 proof −
  have *two_ne_pos_c4*: (*2* :: *nat*) ≠ *mt_pos c4 kk*
   using *c4_pos kklt pos_kk* **by** *simp*
  show *?thesis*
   using *mttm_step_tape_off_head*[*OF step5 two_ne_pos_c4*] **.**
 qed
 have *c4_tape_at_2*: *mt_tape c4 kk 2 = mt_tape c′ kk 2*
  using *tape_c4_eq_c′* **by** *simp*

```
  have read_at_right:
     tts kk (nn kk) = mt_tape c' kk 2
   using tts_kk c5_kk c5_tape_at_2 c4_tape_at_2 by simp
  have right_ne_LE_pos1: mt_tape c' kk 2 ≠ LE_block (le_tm M)
  proof −
   have mt_tape c' kk (mt_pos c' kk + 1) ≠ LE_block (le_tm M)
     using right_not_le_c' by blast
   thus ?thesis using pos_kk by (simp add: numeral_2_eq_2)
  qed
  have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
   using read_at_right right_ne_LE_pos1 by simp
  obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
   using prod.exhaust by metis
  have h_ne_LE: hh ≠ LE_block (le_tm M)
  proof −
   have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
     using buf5_not_le_per_tape[rule_format, OF kklt] pos_kk by blast
   thus ?thesis using buf5_kk by simp
  qed
  have dr6_kk:
     dr6 kk = (if dest5 kk = AE_Left then dir.L else dir.R)
   using dr6_eq buf_eq dest_eq2 buf5_kk read_ne_LE h_ne_LE kklt by simp
  have dr6_kk_L: dr6 kk = dir.L
   using dr6_kk dest_eq by simp
  have mt_pos c6 kk = go_dir (dr6 kk) (nn kk)
   using c6_eq by simp
  also have ... = (nn kk) − 1 using dr6_kk_L by simp
  also have ... = (mt_pos c5 kk) − 1 using nn_kk by simp
  also have ... = 2 − 1 using c5_kk by simp
  also have ... = 1 by simp
  finally show mt_pos c6 kk = 1 .
qed

have tape_c6_zero_le_pos1_dest_left:
   ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 1 ⟹ dest5 kk = AE_Left
        ⟹ mt_tape c6 kk 0 = LE_block (le_tm M)
proof −
  fix kk
  assume kklt: kk < k_tm M
  assume pos_kk: mt_pos c' kk = 1
  assume dest_eq: dest5 kk = AE_Left
  have zero_ne_pos_c5: (0 :: nat) ≠ mt_pos c5 kk
   using c5_pos_for_pos1_dest_left[OF kklt pos_kk dest_eq] by simp
  have c6_zero_eq_c5:
     mt_tape c6 kk 0 = mt_tape c5 kk 0
   using mttm_step_tape_off_head[OF step6 zero_ne_pos_c5] .
  have c5_zero_le: mt_tape c5 kk 0 = LE_block (le_tm M)
   using tape_c5_zero_le[rule_format, OF kklt] by simp
  show mt_tape c6 kk 0 = LE_block (le_tm M)
```

using *c6_zero_eq_c5 c5_zero_le* **by** *simp*
qed

— Sub-step 5l: c7 trajectory for pos=1 + dest=*AE_Left*. *c6_pos = 1*; SS7 reads *mt_tape c6 k 1 = mt_tape c5 k 1* (off-head from SS6's write at *c5_pos = 2*), and *mt_tape c5 k 1 = fst (snd (buf5 k))* (via *tape_c5_at_one_pos1*) = home buffer slot, ≠ *LE*. *ae_ss7_action*'s else-branch fires with *ds = AE_Left*: writes *a*, direction *L*; *c7_pos = c6_pos − 1 = 0*. Plus *tape_c7_zero_le*: cell 0 off-head from SS7's write at *c6_pos = 1*, so = *LE* via *tape_c6_zero_le_pos1_dest_left*.
have *c7_pos_for_pos1_dest_left*:
⋀*kk*. *kk < k_tm M* ⟹ *mt_pos c' kk = 1* ⟹ *dest5 kk = AE_Left*
⟹ *mt_pos c7 kk = 0*
proof −
fix *kk*
assume *kklt*: *kk < k_tm M*
assume *pos_kk*: *mt_pos c' kk = 1*
assume *dest_eq*: *dest5 kk = AE_Left*
obtain *qq tts nn qq' aa7 dr7* **where**
c6_eq7: *c6 = Config*$_M$ *qq tts nn*
and *c7_eq*: *c7 = Config*$_M$ *qq'*
(λ*k*. (*tts k*)(*nn k* := *aa7 k*))
(λ*k*. *go_dir* (*dr7 k*) (*nn k*))
and *tr_in*: (*qq*, λ*k*. *tts k* (*nn k*), *qq'*, *aa7*, *dr7*)
∈ *ae_delta_ss7_ss8 M*
using *step7_sub* **by** (*auto elim*: *mttm_step.cases*)
obtain *q ofs buf dest* **where**
qq_eq: *qq* = (*q*, *ofs*, *buf*, *dest*, *SS7*)
and *dr7_eq*:
dr7 = (λ*k*. *if k < k_tm M then snd* (*ae_ss7_action* (*le_tm M*)
(*tts k* (*nn k*)) (*buf k*) (*dest k*)) *else dir.N*)
using *tr_in* **by** (*auto simp*: *ae_delta_ss7_ss8_def*)
have *qq_state*: *qq* = (*q5*, *ofs5*, *buf5*, *dest5*, *SS7*)
using *c6_state c6_eq7 q6_eq_q5 ofs6_eq_ofs5*
buf6_eq_buf5 dest6_eq_dest5 **by** *simp*
have *buf_eq*: *buf = buf5* **using** *qq_eq qq_state* **by** *simp*
have *dest_eq2*: *dest = dest5* **using** *qq_eq qq_state* **by** *simp*
have *nn_kk*: *nn kk = mt_pos c6 kk* **using** *c6_eq7* **by** *simp*
have *c6_kk*: *mt_pos c6 kk = 1*
using *c6_pos_for_pos1_dest_left*[*OF kklt pos_kk dest_eq*] **.**
have *tts_kk*: *tts kk* (*nn kk*) = *mt_tape c6 kk* (*mt_pos c6 kk*)
using *c6_eq7 nn_kk* **by** *simp*
have *one_ne_pos_c5*: (*1* :: *nat*) ≠ *mt_pos c5 kk*
using *c5_pos_for_pos1_dest_left*[*OF kklt pos_kk dest_eq*] **by** *simp*
have *c6_at_one*: *mt_tape c6 kk 1 = mt_tape c5 kk 1*
using *mttm_step_tape_off_head*[*OF step6 one_ne_pos_c5*] **.**
have *c6_at_one_eq_h*:
mt_tape c6 kk 1 = fst (*snd* (*buf5 kk*))
using *c6_at_one tape_c5_at_one_pos1*[*OF kklt pos_kk*] **by** *simp*
obtain *ll hh rr* **where** *buf5_kk*: *buf5 kk* = (*ll*, *hh*, *rr*)

```
    using prod.exhaust by metis
  have h_ne_LE: hh ≠ LE_block (le_tm M)
  proof −
    have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
      using buf5_not_le_per_tape[rule_format, OF kklt] pos_kk by blast
    thus ?thesis using buf5_kk by simp
  qed
  have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
    using tts_kk c6_kk c6_at_one_eq_h buf5_kk h_ne_LE by simp
  have dr7_kk:
      dr7 kk = (if dest5 kk = AE_Left then dir.L else dir.R)
    using dr7_eq buf_eq dest_eq2 buf5_kk read_ne_LE h_ne_LE kklt by simp
  have dr7_kk_L: dr7 kk = dir.L
    using dr7_kk dest_eq by simp
  have mt_pos c7 kk = go_dir (dr7 kk) (nn kk)
    using c7_eq by simp
  also have ... = (nn kk) − 1 using dr7_kk_L by simp
  also have ... = (mt_pos c6 kk) − 1 using nn_kk by simp
  also have ... = 1 − 1 using c6_kk by simp
  also have ... = 0 by simp
  finally show mt_pos c7 kk = 0 .
qed

have tape_c7_zero_le_pos1_dest_left:
    ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 1 ⟹ dest5 kk = AE_Left
        ⟹ mt_tape c7 kk 0 = LE_block (le_tm M)
proof −
  fix kk
  assume kklt: kk < k_tm M
  assume pos_kk: mt_pos c' kk = 1
  assume dest_eq: dest5 kk = AE_Left
  have zero_ne_pos_c6: (0 :: nat) ≠ mt_pos c6 kk
    using c6_pos_for_pos1_dest_left[OF kklt pos_kk dest_eq] by simp
  have c7_zero_eq_c6:
      mt_tape c7 kk 0 = mt_tape c6 kk 0
    using mttm_step_tape_off_head[OF step7 zero_ne_pos_c6] .
  show mt_tape c7 kk 0 = LE_block (le_tm M)
      using c7_zero_eq_c6 tape_c6_zero_le_pos1_dest_left[OF kklt pos_kk
dest_eq]
    by simp
qed
```

— Sub-step 5m: *pos_link_c7* + SS8→SS1 invocation. Mirror of *pos_link_c5*: right-slot uniform non-LE from buf5 (= buf6) regime case-split; SS8 conditional substantive only for pos=1 + dest=*AE_Left*, vacuous for pos=0/≥2. Then *le_compat_ss8* from *pos_link_c7*, invoke *ae_step_ss8_ss1_exists*.

```
have pos_link_c7: ae_position_link M c7
proof −
  have r_not_le: ∀k<k_tm M. snd (snd (buf6 k)) ≠ LE_block (le_tm M)
```

proof (*intro allI impI*)
fix *k*
assume *klt*: $k < k_tm\ M$
consider (*le0*) $mt_pos\ c'\ k = 0$
| (*le1*) $mt_pos\ c'\ k = 1$
| (*steady*) $mt_pos\ c'\ k \geq 2$
by *linarith*
thus $snd\ (snd\ (buf6\ k)) \neq LE_block\ (le_tm\ M)$
proof *cases*
case *le0* **thus** *?thesis* **using** *buf5_not_le_per_tape*[*rule_format*, *OF klt*] *buf6_eq_buf5* **by** *blast*
next
case *le1* **thus** *?thesis* **using** *buf5_not_le_per_tape*[*rule_format*, *OF klt*] *buf6_eq_buf5* **by** *blast*
next
case *steady* **thus** *?thesis* **using** *buf5_not_le_per_tape*[*rule_format*, *OF klt*] *buf6_eq_buf5* **by** *blast*
qed
qed
have *ss8_cond*:
$\forall\, k{<}k_tm\ M.\ dest6\ k = AE_Left$
$\longrightarrow fst\ (buf6\ k) = LE_block\ (le_tm\ M)$
$\longrightarrow mt_tape\ c7\ k\ (mt_pos\ c7\ k) = LE_block\ (le_tm\ M)$
proof (*intro allI impI*)
fix *k*
assume *klt*: $k < k_tm\ M$
assume *dest_eq*: $dest6\ k = AE_Left$
assume *l_eq_le*: $fst\ (buf6\ k) = LE_block\ (le_tm\ M)$
have *dest5_eq*: $dest5\ k = AE_Left$ **using** *dest_eq dest6_eq_dest5* **by** *simp*
have *l_eq_le_buf5*: $fst\ (buf5\ k) = LE_block\ (le_tm\ M)$
using *l_eq_le buf6_eq_buf5* **by** *simp*
consider (*le0*) $mt_pos\ c'\ k = 0$
| (*le1*) $mt_pos\ c'\ k = 1$
| (*steady*) $mt_pos\ c'\ k \geq 2$
by *linarith*
thus $mt_tape\ c7\ k\ (mt_pos\ c7\ k) = LE_block\ (le_tm\ M)$
proof *cases*
case *le0*
have $fst\ (buf5\ k) \neq LE_block\ (le_tm\ M)$
using *buf5_not_le_per_tape*[*rule_format*, *OF klt*] *le0* **by** *blast*
thus *?thesis* **using** *l_eq_le_buf5* **by** *simp*
next
case *le1*
have *c7_pos_k*: $mt_pos\ c7\ k = 0$
using *c7_pos_for_pos1_dest_left*[*OF klt le1 dest5_eq*] **.**
have $mt_tape\ c7\ k\ 0 = LE_block\ (le_tm\ M)$
using *tape_c7_zero_le_pos1_dest_left*[*OF klt le1 dest5_eq*] **.**
thus *?thesis* **using** *c7_pos_k* **by** *simp*
next

```
      case steady
      have fst (buf5 k) ≠ LE_block (le_tm M)
        using buf5_not_le_per_tape[rule_format, OF klt] steady by blast
      thus ?thesis using l_eq_le_buf5 by simp
    qed
  qed
  show ?thesis
    unfolding ae_position_link_def
    using r_not_le ss8_cond c7_state by simp
qed

have le_compat_ss8_c7: ae_le_compat_ss8 M c7
  using ae_position_link_discharges_ss8[OF inv_ss8 pos_link_c7] .
obtain c8 where
    step8_sub: (c7, c8) ∈ mttm_step (ae_delta_ss8_ss1 M)
  and step8: (c7, c8) ∈ mttm_step (alphabet_enlarge_delta M)
  using ae_step_ss8_ss1_exists[OF vM inv_ss8 gamma_c7 buf_gamma_c7
                                  le_compat_ss8_c7] by blast
```

— Re-express the per-tape $\bigwedge kk$ facts as object-level $\forall\, kk$ before feeding *that*: the nested meta-implication under the obtains witness binders defeats elim-resolution at the call site. The call site recovers the $\bigwedge kk$ form via [*rule_format*].

```
  have c4_pos_all: ∀ kk. kk < k_tm M ⟶ mt_pos c4 kk = mt_pos c' kk
    using c4_pos by blast
  have c5_pos_for_pos1_all:
      ∀ kk. kk < k_tm M ⟶ mt_pos c' kk = 1 ⟶ dest5 kk ≠ AE_Left
        ⟶ mt_pos c5 kk = 0
    using c5_pos_for_pos1 by blast
  have c5_pos_for_pos1_dest_left_all:
      ∀ kk. kk < k_tm M ⟶ mt_pos c' kk = 1 ⟶ dest5 kk = AE_Left
        ⟶ mt_pos c5 kk = 2
    using c5_pos_for_pos1_dest_left by blast
  have tape_c5_at_one_pos1_all:
      ∀ kk. kk < k_tm M ⟶ mt_pos c' kk = 1
        ⟶ mt_tape c5 kk 1 = fst (snd (buf5 kk))
    using tape_c5_at_one_pos1 by blast
  have c6_pos_for_pos1_dest_left_all:
      ∀ kk. kk < k_tm M ⟶ mt_pos c' kk = 1 ⟶ dest5 kk = AE_Left
        ⟶ mt_pos c6 kk = 1
    using c6_pos_for_pos1_dest_left by blast
  have c7_pos_for_pos1_dest_left_all:
      ∀ kk. kk < k_tm M ⟶ mt_pos c' kk = 1 ⟶ dest5 kk = AE_Left
        ⟶ mt_pos c7 kk = 0
    using c7_pos_for_pos1_dest_left by blast
  have tape_c7_zero_le_pos1_dest_left_all:
      ∀ kk. kk < k_tm M ⟶ mt_pos c' kk = 1 ⟶ dest5 kk = AE_Left
        ⟶ mt_tape c7 kk 0 = LE_block (le_tm M)
    using tape_c7_zero_le_pos1_dest_left by blast
  show thesis by (rule that[OF step5_sub step5 c4_state c5_state tape_c4_eq_c'
      buf5_not_le_per_tape right_not_le_c' c4_pos_all tape_c5_zero_le
```

c5_pos_for_pos1_all step6_sub step6 step7_sub step7 gamma_c7 buf_gamma_c7
c6_state c7_state buf6_eq_buf5 q6_eq_q5 ofs6_eq_ofs5 dest6_eq_dest5
c5_pos_for_pos1_dest_left_all tape_c5_at_one_pos1_all
c6_pos_for_pos1_dest_left_all c7_pos_for_pos1_dest_left_all
tape_c7_zero_le_pos1_dest_left_all step8_sub step8])
qed

end
theory *AlphabetEnlargement_ForwardStage*
imports *AlphabetEnlargement_ForwardStep*
begin

2.26 Per-tape unified forward stage

The primary forward-stage statement, feeding *ae_simulation_phase_step_count* (theory *AlphabetEnlargement_Acceptance*). Tip of the forward-stage chain *ForwardCells* → *ForwardStep* → *ForwardStage*: it applies the SS5→SS8 super-step (*ae_forward_stage_step_chain*) and the per-tape reconstruction leaves (theory *AlphabetEnlargement_ForwardCells*) to advance the simulation by one M-step.

Each tape k picks its regime (steady / le1 / le0) independently from its own *mt_pos* c' k; mixed configurations across tapes are handled natively, with no uniform-regime case-split at the call site. Per-tape dispatch is mandatory: the regimes can differ from tape to tape.

The *no_le_per_tape* hypothesis bundles a per-tape LE-free window whose start and width depend on the tape's block position: *3c*-cell window starting at the left neighbour for steady tapes, *2c*-cell window starting at position 1 for le1 tapes, c-cell window starting at position 1 for le0 tapes. Discharged at the call site via substrate's *valid_reach_LE_only_pos0_mttm* for any reach-from-init configuration.

The displacement disjunct's leftward branch is conditioned on *mt_pos* c' $kk > 0$ le0 tapes cannot move left from block 0.

lemma *ae_simulates_forward_stage_general*:
fixes M :: ('q, 'a) *mttm*
and *cM cM_k* :: ('a, 'q) *mt_config*
and c' :: ('c :: *enum* ⇒ 'a,
'q × ('a, 'c) *ae_stage*) *mt_config*
and k :: *nat*
assumes *vM*: *valid_mttm M*
and *lu*: *le_unique M*
and *sim*: *ae_simulates M cM c'*
and *qM_in_Q*: *mt_state cM* ∈ *Q_tm M*
and *q_neq_t*: *mt_state cM* ≠ *t_tm M*
and *q_neq_r*: *mt_state cM* ≠ *r_tm M*
and *buf_gamma*: *ae_buffer_in_gamma_block M c'*

and *k_le*: $k \le card\ (UNIV :: 'c\ set)$
and *trace*: $(cM,\ cM_k) \in mttm_step\ (delta_tm\ M)\ ^^\ k$
and *end_or_halt*:
$k = card\ (UNIV :: 'c\ set)$
$\lor\ mt_state\ cM_k \in \{t_tm\ M,\ r_tm\ M\}$
and *le_anchor*:
$\forall\, k{<}k_tm\ M.\ mt_tape\ c'\ k\ 0 = LE_block\ (le_tm\ M)$
and *le_neq_bl*:
$le_tm\ M \neq bl_tm\ M$
and *no_le_per_tape*:
$\forall\, k.\ (mt_pos\ c'\ k \ge 2$
$\longrightarrow\ (\forall\, i.\ i < 3 * card\ (UNIV :: 'c\ set)$
$\longrightarrow\ mt_tape\ cM\ k$
$((mt_pos\ c'\ k - 2) * card\ (UNIV :: 'c\ set) + 1 + i)$
$\neq le_tm\ M))$
$\land\ (mt_pos\ c'\ k = 1$
$\longrightarrow\ (\forall\, i.\ i < 2 * card\ (UNIV :: 'c\ set)$
$\longrightarrow\ mt_tape\ cM\ k\ (Suc\ i) \neq le_tm\ M))$
$\land\ (mt_pos\ c'\ k = 0$
$\longrightarrow\ (\forall\, i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow\ mt_tape\ cM\ k\ (Suc\ i) \neq le_tm\ M))$
obtains *c'' c1 c2 c3 c4 c5 c6 c7 ofs_f buf_f dest_f* **where**
$(c',\ c'') \in mttm_step\ (alphabet_enlarge_delta\ M)\ ^^\ 8$
and *ae_simulates M cM_k c''*
and *ae_buffer_in_gamma_block M c''*
and $\forall\, kk{<}k_tm\ M.\ mt_tape\ c''\ kk\ 0 = LE_block\ (le_tm\ M)$
— SS5 exposure for the det-free reverse arm (*nae_sim_proto*): the full substep chain, the SS5 state shape + window correspondence in (*dest*, *ofs*) bp form, and the buffer γ-block carrying the frozen inactive tails.
and $(c',\ c1) \in mttm_step\ (ae_delta_ss1_ss2\ M)$
and $(c1,\ c2) \in mttm_step\ (ae_delta_ss2_ss3\ M)$
and $(c2,\ c3) \in mttm_step\ (ae_delta_ss3_ss4\ M)$
and $(c3,\ c4) \in mttm_step\ (ae_delta_ss4_ss5\ M)$
and $(c4,\ c5) \in mttm_step\ (ae_delta_ss5_ss6\ M)$
and $(c5,\ c6) \in mttm_step\ (ae_delta_ss6_ss7\ M)$
and $(c6,\ c7) \in mttm_step\ (ae_delta_ss7_ss8\ M)$
and $(c7,\ c'') \in mttm_step\ (ae_delta_ss8_ss1\ M)$
and $mt_state\ c4 = (mt_state\ cM_k,\ ofs_f,\ buf_f,\ dest_f,\ SS5)$
and $\forall\, kk{<}k_tm\ M.\ ae_window_invariant_general$
$(mt_tape\ cM_k\ kk)\ (mt_pos\ cM_k\ kk)$
$(dest_f\ kk,\ ofs_f\ kk)\ (buf_f\ kk)\ (mt_pos\ c'\ kk)\ (le_tm\ M)$
and *ae_buffer_in_gamma_block M c4*
proof −
— Unpack *ae_simulates* at SS1 entry via the shared regime-agnostic helper. No regime-specific hypothesis enters here the unpack is uniform across steady / le1 / le0.
obtain *qM' ofs buf dest* **where**
c'_state: $mt_state\ c' = (qM',\ ofs,\ buf,\ dest,\ SS1)$
and *qM_eq*: $mt_state\ cM = qM'$

and *qM′_in_Q*: *qM′ ∈ Q_tm M*
and *qM′_neq_t*: *qM′ ≠ t_tm M*
and *qM′_neq_r*: *qM′ ≠ r_tm M*
and *tape_corr*: *∀ k<k_tm M. ae_tape_correspondence (le_tm M)*
(mt_tape cM k) (mt_tape c′ k)
and *pos_corr*: *∀ k<k_tm M. mt_pos cM k = ae_decode_pos (mt_pos c′ k)*
(ofs k)
and *gamma_c′*: *ae_tape_in_gamma_block M c′*
and *inv_ss1*: *ae_inv_ss1 M c′*
and *pos_link_c′*: *ae_position_link M c′*
by (*rule ae_forward_stage_unpack_sim*[*OF sim qM_in_Q q_neq_t q_neq_r*])

— Buffer-load substeps SS1→SS2→SS3 →SS4 via the shared regime-agnostic load-chain helper. Yields the SS4-entry config *c3* together with its SS4 invariant, gamma and buffer-gamma side-bands, state preservation, and position-link side-band. Each per-tape branch will refine the buffer-content characterisation separately in the regime-aware substeps that follow.
obtain *c1 c2 c3* **where**
step1_sub: (*c′*, *c1*) ∈ *mttm_step* (*ae_delta_ss1_ss2 M*)
and *step1*: (*c′*, *c1*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *step2_sub*: (*c1*, *c2*) ∈ *mttm_step* (*ae_delta_ss2_ss3 M*)
and *step2*: (*c1*, *c2*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *step3_sub*: (*c2*, *c3*) ∈ *mttm_step* (*ae_delta_ss3_ss4 M*)
and *step3*: (*c2*, *c3*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *inv_ss4*: *ae_inv_ss4 M c3*
and *gamma_c3*: *ae_tape_in_gamma_block M c3*
and *buf_gamma_c3*: *ae_buffer_in_gamma_block M c3*
and *c3_fst_eq*: *fst* (*mt_state c3*) = *qM′*
and *pos_link_c3*: *ae_position_link M c3*
by (*rule ae_forward_stage_load_chain*[*OF vM c′_state inv_ss1 gamma_c′*
buf_gamma pos_link_c′
qM′_neq_t qM′_neq_r])
have *c3_fst_neq_t*: *fst* (*mt_state c3*) ≠ *t_tm M*
using *c3_fst_eq qM′_neq_t* **by** *simp*
have *c3_fst_neq_r*: *fst* (*mt_state c3*) ≠ *r_tm M*
using *c3_fst_eq qM′_neq_r* **by** *simp*

— Per-tape home-cell classification: each tape *k* either has *mt_pos c′ k = 0* with home = *LE_block* (forward arm via *le_anchor*) or *mt_pos c′ k ≥ 1* with home ≠ *LE_block* (backward arm via *ae_tape_correspondence* at the home block with witness *c_idx x = 0*, projecting *no_le_per_tape* to the regime branch).
have *c_ge_1*: *1 ≤ card* (*UNIV* :: *′c set*)
using *c_idx_lt_card*[*of SOME x* :: *′c. True*] **by** *simp*
have *c_eq_len*: *card* (*UNIV* :: *′c set*) = *length* (*enum_class.enum* :: *′c list*)
using *enum_class.UNIV_enum enum_class.enum_distinct*
by (*metis distinct_card length_remdups_card_conv set_remdups*)

have *home_classification*:
∀ k<k_tm M. (*mt_pos c′ k = 0*

```
      ⟶ mt_tape c' k (mt_pos c' k) = LE_block (le_tm M))
  ∧ (mt_pos c' k ≥ 1
      ⟶ mt_tape c' k (mt_pos c' k) ≠ LE_block (le_tm M))
 by (rule ae_home_classification[OF le_anchor no_le_per_tape tape_corr])
```

— SS4-entry characterisation via the per-tape unified buffer-chars helper. Buffer at SS4 is per-tape: tapes at *pos = 0* get (*bl_block*, *LE_block*, *snd* (*snd* (*buf k*))) (the LE-arm shape from *ae_ss1_to_ss4_buffer_chars_le0*); tapes at *pos ≥ 1* get (*mt_tape c' k* (*pos − 1*), *mt_tape c' k pos*, *snd* (*snd* (*buf k*))) (the steady-arm shape from *ae_ss1_to_ss4_buffer_chars*). Tape and position conclusions are uniform across regimes.

```
 have c3_state:
   mt_state c3 = (qM', ofs,
                  (λk. if k < k_tm M
                       then (if mt_pos c' k = 0
                             then (bl_block (bl_tm M),
                                   LE_block (le_tm M),
                                   snd (snd (buf k)))
                             else (mt_tape c' k (mt_pos c' k − 1),
                                   mt_tape c' k (mt_pos c' k),
                                   snd (snd (buf k))))
                       else init_buffer (le_tm M) k),
                  dest, SS4)
  using ae_ss1_to_ss4_buffer_chars_general(1)[OF c'_state step1_sub
                                              step2_sub step3_sub
                                              home_classification] .
 have c3_tape: mt_tape c3 = mt_tape c'
  using ae_ss1_to_ss4_buffer_chars_general(2)[OF c'_state step1_sub
                                              step2_sub step3_sub
                                              home_classification] .
 have c3_pos: ⋀k. k < k_tm M ⟹ mt_pos c3 k = mt_pos c' k + 1
  using ae_ss1_to_ss4_buffer_chars_general(3)[OF c'_state step1_sub
                                              step2_sub step3_sub
                                              home_classification] .
```

— Per-tape SS4 window invariant via the unified from-correspondence helper. The buffer triple is the regime-aware shape that *ae_delta_ss4_ss5* reads from *c3*: left and home from the SS4-entry per-tape buffer, right freshly read at *mt_pos c' k + 1*. The conclusion is the per-tape *ae_window_invariant_general*, indexed by *mt_pos c' k* (the frozen regime selector).

```
 have window_at_ss4:
   ∀k<k_tm M. ae_window_invariant_general
         (mt_tape cM k) (mt_pos cM k)
         (AE_Home, ofs k)
         (if mt_pos c' k = 0
            then (bl_block (bl_tm M),
                  LE_block (le_tm M),
                  mt_tape c' k 1)
          else if mt_pos c' k = 1
```

```
      then (mt_tape c' k 0,
            mt_tape c' k 1,
            mt_tape c' k 2)
      else (mt_tape c' k (mt_pos c' k − 1),
            mt_tape c' k (mt_pos c' k),
            mt_tape c' k (mt_pos c' k + 1)))
     (mt_pos c' k)
     (le_tm M)
using ae_ss4_window_from_correspondence_general[OF sim c'_state
                                                 le_anchor
                                                 no_le_per_tape] .
```

— Re-express *cM* via $Config_M$ to match the trace lemma's toolkit hypothesis shape (arm-uniform with le0 / le1 / steady). Surfaces *ts_cM* = *mt_tape cM* and *n_cM* = *mt_pos cM* for direct substitution into the trace, no-LE and window hypotheses.

```
obtain ts_cM n_cM where cM_eq:
  cM = Config_M qM' ts_cM n_cM
  using qM_eq by (cases cM) auto
have ts_cM_eq: ts_cM = mt_tape cM using cM_eq by simp
have n_cM_eq: n_cM = mt_pos cM using cM_eq by simp
```

— Trace toolkit: rewrite the precondition's (*cM*, *cM_k*) ∈ *mttm_step*⁀⁀*k* shape into the trace lemma's (*Config*$_M$ (*fst* (*mt_state c3*)) *ts_cM n_cM*, *cM_k*) ∈ ... form, via *cM_eq* and *c3_fst_eq*.

```
have trace_toolkit:
  (Config_M (fst (mt_state c3)) ts_cM n_cM, cM_k)
    ∈ mttm_step (delta_tm M) ⁀⁀ k
  using trace cM_eq c3_fst_eq by simp
```

— No-LE toolkit: rephrase *no_le_per_tape* in terms of *ts_cM* (= *mt_tape cM*) for the trace lemma's *tsM*-parameter. Same shape modulo the rewrite of *mt_tape cM* to *ts_cM*.

```
have no_le_toolkit_general:
  ∀ kk. (mt_pos c' kk = 0
         ⟶ (∀ i. i < card (UNIV :: 'c set)
                 ⟶ ts_cM kk (Suc i) ≠ le_tm M))
      ∧ (mt_pos c' kk = 1
         ⟶ (∀ i. i < 2 * card (UNIV :: 'c set)
                 ⟶ ts_cM kk (Suc i) ≠ le_tm M))
      ∧ (mt_pos c' kk ≥ 2
         ⟶ (∀ i. i < 3 * card (UNIV :: 'c set)
                 ⟶ ts_cM kk
                     ((mt_pos c' kk − 2) * card (UNIV :: 'c set)
                       + 1 + i)
                   ≠ le_tm M))
  using no_le_per_tape ts_cM_eq by simp
```

— Left-slot guard for pos=0 tapes at *c3*: the SS4-entry left slot for tapes at

block 0 is *bl_block (bl_tm M)* (per *c3_state*'s if-branch), distinct from *LE_block (le_tm M)* via *le_neq_bl*. Le1 and steady tapes carry no obligation here (the trace lemma's hypothesis is *pos k = 0 ⟶ ...*).

```
have bl_neq_le:
  (bl_block (bl_tm M) :: 'c ⇒ 'a) ≠ LE_block (le_tm M)
proof
  assume eq: (bl_block (bl_tm M) :: 'c ⇒ 'a) = LE_block (le_tm M)
  have bl_block (bl_tm M) (SOME x :: 'c. True)
       = LE_block (le_tm M) (SOME x :: 'c. True)
    using eq by simp
  hence bl_tm M = le_tm M
    unfolding bl_block_def LE_block_def by simp
  with le_neq_bl show False by simp
qed
have left_not_le_c3_toolkit:
  case mt_state c3 of (_, _, b, _, _) ⇒
      ∀ kk<k_tm M. mt_pos c' kk = 0
          ⟶ fst (b kk) ≠ LE_block (le_tm M)
  using c3_state bl_neq_le by simp
```

— Window-invariant toolkit: convert *window_at_ss4* from its *c'*-tape if-then-else form into the *case mt_state c3* form expected by the trace lemma. Per-tape regime case-split on *mt_pos c' kk*; in each branch, *c3_state*'s if-then-else and the *window_at_ss4* nested-if reduce to the same buffer triple.

```
have window_toolkit_general:
  ∀ kk<k_tm M. ae_window_invariant_general (ts_cM kk) (n_cM kk)
      (case mt_state c3 of (_, ofs', _, _, _)
         ⇒ (AE_Home, ofs' kk))
      (case mt_state c3 of (_, _, buf', _, _)
         ⇒ (fst (buf' kk), fst (snd (buf' kk)),
            mt_tape c3 kk (mt_pos c3 kk)))
      (mt_pos c' kk) (le_tm M)
proof (intro allI impI)
  fix kk
  assume kklt: kk < k_tm M
  have c3_buf_triple:
    (case mt_state c3 of (_, _, buf', _, _) ⇒
        (fst (buf' kk), fst (snd (buf' kk)),
         mt_tape c3 kk (mt_pos c3 kk)))
      = (if mt_pos c' kk = 0
         then (bl_block (bl_tm M), LE_block (le_tm M),
               mt_tape c' kk (mt_pos c' kk + 1))
        else (mt_tape c' kk (mt_pos c' kk − 1),
              mt_tape c' kk (mt_pos c' kk),
              mt_tape c' kk (mt_pos c' kk + 1)))
    using c3_state c3_tape c3_pos kklt by simp
  have c3_ofs_pair:
    (case mt_state c3 of (_, ofs', _, _, _) ⇒ (AE_Home, ofs' kk))
      = (AE_Home, ofs kk)
```

```
    using c3_state by simp
  have win_at_ss4_kk:
      ae_window_invariant_general (mt_tape cM kk) (mt_pos cM kk)
          (AE_Home, ofs kk)
          (if mt_pos c' kk = 0
            then (bl_block (bl_tm M), LE_block (le_tm M),
                  mt_tape c' kk 1)
           else if mt_pos c' kk = 1
            then (mt_tape c' kk 0,
                  mt_tape c' kk 1,
                  mt_tape c' kk 2)
           else (mt_tape c' kk (mt_pos c' kk − 1),
                 mt_tape c' kk (mt_pos c' kk),
                 mt_tape c' kk (mt_pos c' kk + 1)))
          (mt_pos c' kk) (le_tm M)
    using window_at_ss4 kklt by blast
  consider (le0) mt_pos c' kk = 0
         | (le1) mt_pos c' kk = 1
         | (steady) mt_pos c' kk ≥ 2
    by linarith
  thus ae_window_invariant_general (ts_cM kk) (n_cM kk)
         (case mt_state c3 of (_, ofs', _, _, _)
            ⇒ (AE_Home, ofs' kk))
         (case mt_state c3 of (_, _, buf', _, _)
            ⇒ (fst (buf' kk), fst (snd (buf' kk)),
               mt_tape c3 kk (mt_pos c3 kk)))
         (mt_pos c' kk) (le_tm M)
  proof cases
    case le0
    show ?thesis
      using win_at_ss4_kk c3_buf_triple c3_ofs_pair le0
            ts_cM_eq n_cM_eq
      by simp
  next
    case le1
    show ?thesis
      using win_at_ss4_kk c3_buf_triple c3_ofs_pair le1
            ts_cM_eq n_cM_eq
      by (simp add: numeral_2_eq_2)
  next
    case steady
    have pos_ne_0: mt_pos c' kk ≠ 0 using steady by simp
    have pos_ne_1: mt_pos c' kk ≠ 1 using steady by simp
    show ?thesis
      using win_at_ss4_kk c3_buf_triple c3_ofs_pair pos_ne_0 pos_ne_1
            ts_cM_eq n_cM_eq
      by simp
  qed
qed
```

— Invoke the trace-driven SS4→SS5 helper for the per-tape unified case. The trace lemma's *pos* parameter is inferred from *window_toolkit_general*'s *mt_pos c′ kk* regime selector, giving *pos* := *mt_pos c′*.

obtain *c4* **where**
step4_sub: (*c3*, *c4*) ∈ *mttm_step* (*ae_delta_ss4_ss5 M*)
and *step4*: (*c3*, *c4*) ∈ *mttm_step* (*alphabet_enlarge_delta M*)
and *c4_q_out*: *case mt_state c4 of* (*q_out*, _, _, _, _)
⇒ *q_out* = *mt_state cM_k*
and *c4_buf_not_le_per_tape*:
case mt_state c4 of (_, _, *b*, _, _) ⇒
∀ *kk*<*k_tm M*. (*mt_pos c′ kk* = *0*
⟶ *fst* (*b kk*) ≠ *LE_block* (*le_tm M*)
∧ *snd* (*snd* (*b kk*)) ≠ *LE_block* (*le_tm M*))
∧ (*mt_pos c′ kk* = *1*
⟶ *fst* (*snd* (*b kk*)) ≠ *LE_block* (*le_tm M*)
∧ *snd* (*snd* (*b kk*)) ≠ *LE_block* (*le_tm M*))
∧ (*mt_pos c′ kk* ≥ *2*
⟶ *fst* (*b kk*) ≠ *LE_block* (*le_tm M*)
∧ *fst* (*snd* (*b kk*)) ≠ *LE_block* (*le_tm M*)
∧ *snd* (*snd* (*b kk*)) ≠ *LE_block* (*le_tm M*))
and *c4_window_general*:
case mt_state c4 of (_, *ofs′*, *buf′*, *dest′*, _) ⇒
∀ *kk*<*k_tm M*. *ae_window_invariant_general*
(*mt_tape cM_k kk*) (*mt_pos cM_k kk*)
(*dest′ kk*, *ofs′ kk*) (*buf′ kk*)
(*mt_pos c′ kk*) (*le_tm M*)
using *ae_step_ss4_ss5_exists_general_trace*[*OF vM lu inv_ss4 gamma_c3*
buf_gamma_c3
c3_fst_neq_t c3_fst_neq_r
window_toolkit_general
no_le_toolkit_general
trace_toolkit k_le
end_or_halt
left_not_le_c3_toolkit]
by *blast*

— Sub-step 5a: SS5→SS6 invocation. Arm-uniform with the three per-regime variants: *inv_ss5*, gamma + buf-gamma at *c4* propagate from *c3* via the arm-agnostic preservation lemmas. *ae_step_ss5_ss6_exists* takes only these three (no pos-link required at SS5), so *c5* is obtained directly.

have *inv_ss5*: *ae_inv_ss5 M c4*
using *ae_step_ss4_ss5_invariant*[*OF vM inv_ss4 step4_sub*] .
have *gamma_c4*: *ae_tape_in_gamma_block M c4*
using *ae_step_alphabet_enlarge_gamma_preserve*[*OF gamma_c3 step4*] .
have *buf_gamma_c4*: *ae_buffer_in_gamma_block M c4*
using *ae_step_alphabet_enlarge_buffer_gamma_preserve*[*OF vM buf_gamma_c3*
gamma_c3 step4] .

— The SS5→SS8 super-step: extracted to *ae_forward_stage_step_chain*. Rebinds the 12 intermediate witnesses and 29 named facts the per-tape reconstruction below consumes.

obtain *c5 q5 ofs5 buf5 dest5 c6 c7 q6 ofs6 buf6 dest6 c8*
where *step5_sub*: $(c4, c5) \in mttm_step\ (ae_delta_ss5_ss6\ M)$
and *step5*: $(c4, c5) \in mttm_step\ (alphabet_enlarge_delta\ M)$
and *c4_state*: $mt_state\ c4 = (q5, ofs5, buf5, dest5, SS5)$
and *c5_state*: $mt_state\ c5 = (q5, ofs5, buf5, dest5, SS6)$
and *tape_c4_eq_c'*: $mt_tape\ c4 = mt_tape\ c'$
and *buf5_not_le_per_tape*:
$\forall k{<}k_tm\ M.\ (mt_pos\ c'\ k = 0$
$\longrightarrow fst\ (buf5\ k) \neq LE_block\ (le_tm\ M)$
$\wedge\ snd\ (snd\ (buf5\ k)) \neq LE_block\ (le_tm\ M))$
$\wedge\ (mt_pos\ c'\ k = 1$
$\longrightarrow fst\ (snd\ (buf5\ k)) \neq LE_block\ (le_tm\ M)$
$\wedge\ snd\ (snd\ (buf5\ k)) \neq LE_block\ (le_tm\ M))$
$\wedge\ (mt_pos\ c'\ k \geq 2$
$\longrightarrow fst\ (buf5\ k) \neq LE_block\ (le_tm\ M)$
$\wedge\ fst\ (snd\ (buf5\ k)) \neq LE_block\ (le_tm\ M)$
$\wedge\ snd\ (snd\ (buf5\ k)) \neq LE_block\ (le_tm\ M))$
and *right_not_le_c'*:
$\forall k.\ mt_tape\ c'\ k\ (mt_pos\ c'\ k + 1) \neq LE_block\ (le_tm\ M)$
and *c4_pos_all*: $\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c4\ kk = mt_pos\ c'\ kk$
and *tape_c5_zero_le*: $\forall k{<}k_tm\ M.\ mt_tape\ c5\ k\ 0 = LE_block\ (le_tm\ M)$
and *c5_pos_for_pos1_all*:
$\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c'\ kk = 1 \longrightarrow dest5\ kk \neq AE_Left$
$\longrightarrow mt_pos\ c5\ kk = 0$
and *step6_sub*: $(c5, c6) \in mttm_step\ (ae_delta_ss6_ss7\ M)$
and *step6*: $(c5, c6) \in mttm_step\ (alphabet_enlarge_delta\ M)$
and *step7_sub*: $(c6, c7) \in mttm_step\ (ae_delta_ss7_ss8\ M)$
and *step7*: $(c6, c7) \in mttm_step\ (alphabet_enlarge_delta\ M)$
and *gamma_c7*: $ae_tape_in_gamma_block\ M\ c7$
and *buf_gamma_c7*: $ae_buffer_in_gamma_block\ M\ c7$
and *c6_state*: $mt_state\ c6 = (q6, ofs6, buf6, dest6, SS7)$
and *c7_state*: $mt_state\ c7 = (q6, ofs6, buf6, dest6, SS8)$
and *buf6_eq_buf5*: $buf6 = buf5$
and *q6_eq_q5*: $q6 = q5$
and *ofs6_eq_ofs5*: $ofs6 = ofs5$
and *dest6_eq_dest5*: $dest6 = dest5$
and *c5_pos_for_pos1_dest_left_all*:
$\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c'\ kk = 1 \longrightarrow dest5\ kk = AE_Left$
$\longrightarrow mt_pos\ c5\ kk = 2$
and *tape_c5_at_one_pos1_all*:
$\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c'\ kk = 1$
$\longrightarrow mt_tape\ c5\ kk\ 1 = fst\ (snd\ (buf5\ kk))$
and *c6_pos_for_pos1_dest_left_all*:
$\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c'\ kk = 1 \longrightarrow dest5\ kk = AE_Left$
$\longrightarrow mt_pos\ c6\ kk = 1$
and *c7_pos_for_pos1_dest_left_all*:

$\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c'\ kk = 1 \longrightarrow dest5\ kk = AE_Left$
$\longrightarrow mt_pos\ c7\ kk = 0$
and *tape_c7_zero_le_pos1_dest_left_all*:
$\forall kk.\ kk < k_tm\ M \longrightarrow mt_pos\ c'\ kk = 1 \longrightarrow dest5\ kk = AE_Left$
$\longrightarrow mt_tape\ c7\ kk\ 0 = LE_block\ (le_tm\ M)$
and *step8_sub*: $(c7, c8) \in mttm_step\ (ae_delta_ss8_ss1\ M)$
and *step8*: $(c7, c8) \in mttm_step\ (alphabet_enlarge_delta\ M)$
by (*rule ae_forward_stage_step_chain*[*OF vM le_anchor no_le_per_tape tape_corr step1_sub step2_sub step3_sub c_ge_1 c_eq_len home_classification c3_pos bl_neq_le step4_sub c4_buf_not_le_per_tape inv_ss5 gamma_c4 buf_gamma_c4*])

— Recover the $\bigwedge kk$ form of the per-tape facts (the chain helper exposes them as $\forall kk$ to survive elim-resolution) so the per-tape reconstruction below uses them unchanged.

note *c4_pos* = *c4_pos_all*[*rule_format*]
note *c5_pos_for_pos1* = *c5_pos_for_pos1_all*[*rule_format*]
note *c5_pos_for_pos1_dest_left* = *c5_pos_for_pos1_dest_left_all*[*rule_format*]
note *tape_c5_at_one_pos1* = *tape_c5_at_one_pos1_all*[*rule_format*]
note *c6_pos_for_pos1_dest_left* = *c6_pos_for_pos1_dest_left_all*[*rule_format*]
note *c7_pos_for_pos1_dest_left* = *c7_pos_for_pos1_dest_left_all*[*rule_format*]
note *tape_c7_zero_le_pos1_dest_left* = *tape_c7_zero_le_pos1_dest_left_all*[*rule_format*]

— Sub-step 6a: chain composition + c8 state shape + arm-uniform reconstruction facts (*sim_i*, *sim_ii*, gamma, buf-gamma). The chain composition uses *relpow_Suc_I* Suc-by-Suc; the c8 state shape is the halt-aware *?stage8* let-term from *ae_delta_ss8_ss1*.

let *?rel* = *mttm_step* (*alphabet_enlarge_delta M*)
have *chain1*: $(c', c1) \in ?rel\ ^{\frown\frown}\ Suc\ 0$
using *step1* **by** *simp*
have *chain2*: $(c', c2) \in ?rel\ ^{\frown\frown}\ Suc\ (Suc\ 0)$
using *chain1 step2* **by** (*rule relpow_Suc_I*)
have *chain3*: $(c', c3) \in ?rel\ ^{\frown\frown}\ Suc\ (Suc\ (Suc\ 0))$
using *chain2 step3* **by** (*rule relpow_Suc_I*)
have *chain4*: $(c', c4) \in ?rel\ ^{\frown\frown}\ Suc\ (Suc\ (Suc\ (Suc\ 0)))$
using *chain3 step4* **by** (*rule relpow_Suc_I*)
have *chain5*: $(c', c5) \in ?rel\ ^{\frown\frown}\ Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))$
using *chain4 step5* **by** (*rule relpow_Suc_I*)
have *chain6*: $(c', c6) \in ?rel\ ^{\frown\frown}\ Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0)))))$
using *chain5 step6* **by** (*rule relpow_Suc_I*)
have *chain7*:
$(c', c7) \in ?rel\ ^{\frown\frown}\ Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))))$
using *chain6 step7* **by** (*rule relpow_Suc_I*)
have *chain8_suc*:
$(c', c8) \in ?rel\ ^{\frown\frown}\ Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0)))))))$
using *chain7 step8* **by** (*rule relpow_Suc_I*)
have *chain8*: $(c', c8) \in ?rel\ ^{\frown\frown}\ 8$
using *chain8_suc* **by** (*simp add*: *numeral_eq_Suc*)

let *?stage8* = *if q6* $\in \{t_tm\ M, r_tm\ M\}$

then init_stage (le_tm M)
else (ofs6, buf6, init_dest, SS1)
have *c8_state*: *mt_state c8 = (q6, ?stage8)*
using *step8_sub c7_state*
by (*auto simp*: *ae_delta_ss8_ss1_def elim*: *mttm_step.cases*)

— *sim_i*: state-tuple disjunction at *c8*. The SS8→SS1 rule's *?stage′* case-splits on $q \in \{t_tm\ M,\ r_tm\ M\}$: halt routes to *init_stage le*, non-halt to (*ofs*, *buf*, *init_dest*, *SS1*). The disjunction is XOR-style, with each branch matching one of the *ae_simulates*-defining disjuncts directly.
have *sim_i_c8*:
case mt_state c8 of (qq8, ofs8, buf8, dest8, idx8) ⇒
(idx8 = SS1 ∧ qq8 ∉ {t_tm M, r_tm M})
∨ (qq8 ∈ {t_tm M, r_tm M}
∧ (ofs8, buf8, dest8, idx8) = init_stage (le_tm M))
proof (*cases q6 ∈ {t_tm M, r_tm M}*)
case *True*
have *shape*: *?stage8 = init_stage (le_tm M)* **using** *True* **by** *simp*
have *mt_state c8 = (q6, init_stage (le_tm M))*
using *c8_state shape* **by** *simp*
thus *?thesis* **using** *True* **by** (*simp add*: *init_stage_def*)
next
case *False*
have *shape*: *?stage8 = (ofs6, buf6, init_dest, SS1)* **using** *False* **by** *simp*
have *mt_state c8 = (q6, ofs6, buf6, init_dest, SS1)*
using *c8_state shape* **by** *simp*
thus *?thesis* **using** *False* **by** *simp*
qed

— *sim_ii*: *mt_state cM_k = q6*. q5 = q6 (no q changes through SS5/SS6/SS7); q5 = c4's M-state projection = *mt_state cM_k* (via *c4_q_out* from the trace lemma).
have *q5_eq_q6*: *q5 = q6* **using** *q6_eq_q5* **by** *simp*
have *q5_eq_cM_k*: *q5 = mt_state cM_k*
using *c4_q_out c4_state* **by** *simp*
have *q6_eq_cM_k*: *q6 = mt_state cM_k*
using *q5_eq_q6 q5_eq_cM_k* **by** *simp*
have *sim_ii_c8*:
case mt_state c8 of (qq8, _, _, _, _) ⇒ mt_state cM_k = qq8
using *c8_state q6_eq_cM_k* **by** *simp*

— gamma + buf-gamma at *c8*: chained preservation.
have *gamma_c8*: *ae_tape_in_gamma_block M c8*
using *ae_step_alphabet_enlarge_gamma_preserve*[*OF gamma_c7 step8*] **.**
have *buf_gamma_c8*: *ae_buffer_in_gamma_block M c8*
using *ae_step_alphabet_enlarge_buffer_gamma_preserve*[*OF vM buf_gamma_c7 gamma_c7 step8*] **.**

— Sub-step 6b1: per-tape regime-aware *cM_k*-side window invariant. Projection of *c4_window_general* through *c4_state* = (*q5*, *ofs5*, *buf5*, *dest5*, *SS5*): each tape *kk* has the hybrid *ae_window_invariant_general* indexed by its regime selector *mt_pos c′ kk*. This single fact unpacks into le0/le1/steady sub-invariants per tape via the definition's case-split on *pos* = *0*, *pos* = *1*, *pos* ≥ *2*.

```
have cM_k_window_general:
  ∀ kk<k_tm M. ae_window_invariant_general
        (mt_tape cM_k kk) (mt_pos cM_k kk)
        (dest5 kk, ofs5 kk) (buf5 kk)
        (mt_pos c′ kk) (le_tm M)
  using c4_window_general c4_state by simp
```

— Sub-step 6b1: *cM_k*'s tape at position 0 is *LE_block* (*le_tm M*), equivalently *le_tm M* as a bare tape value the s=0 conjunct of *ae_tape_correspondence*. Per-tape regime case-split: pos=0 and pos=1 extract the *tM 0* = *le* clause directly from the regime-specific window invariant (*ae_window_invariant_le0_def* and *_le1_def*); pos≥2 has no such clause in its window invariant (which only constrains [*p_start*, *p_start* + *3c*)) so the M-side entry correspondence at s=0 (*mt_tape cM kk 0* = *le*) chains with *mttm_relpow_tape_off_window*'s zero preservation (*mt_pos cM kk* ≥ *c* + *1* > *k* for pos≥2 tapes).

```
let ?c = card (UNIV :: 'c set)
have c_ge1: (1 :: nat) ≤ ?c
  using c_idx_lt_card[of SOME x :: 'c. True] by linarith
have cM_k_zero: ⋀kk. kk < k_tm M ⟹ mt_tape cM_k kk 0 = le_tm M
proof −
  fix kk
  assume kklt: kk < k_tm M
  show mt_tape cM_k kk 0 = le_tm M
  proof (cases mt_pos c′ kk = 0)
    case True
    have win_kk:
      ae_window_invariant_le0
          (mt_tape cM_k kk) (mt_pos cM_k kk)
          (dest5 kk, ofs5 kk) (buf5 kk) (le_tm M)
      using cM_k_window_general[rule_format, OF kklt] True
      unfolding ae_window_invariant_general_def by blast
    thus ?thesis unfolding ae_window_invariant_le0_def by simp
  next
    case ne0: False
    show ?thesis
    proof (cases mt_pos c′ kk = 1)
      case True
      have win_kk:
        ae_window_invariant_le1
            (mt_tape cM_k kk) (mt_pos cM_k kk)
            (dest5 kk, ofs5 kk) (buf5 kk) (le_tm M)
        using cM_k_window_general[rule_format, OF kklt] True
        unfolding ae_window_invariant_general_def by blast
      thus ?thesis unfolding ae_window_invariant_le1_def by simp
```

```
    next
      case ne1: False
      have hge2: mt_pos c' kk ≥ 2 using ne0 ne1 by linarith
      have decode:
          mt_pos cM kk = (mt_pos c' kk − 1) ∗ ?c + c_idx (ofs kk) + 1
        using pos_corr kklt hge2 by (simp add: ae_decode_pos_def)
      have step_mult: (mt_pos c' kk − 1) ∗ ?c ≥ ?c
      proof −
        have (mt_pos c' kk − 1) ≥ 1 using hge2 by linarith
        hence (mt_pos c' kk − 1) ∗ ?c ≥ 1 ∗ ?c
          by (rule mult_le_mono1)
        thus ?thesis by simp
      qed
      have pos_ge_c1: mt_pos cM kk ≥ ?c + 1
        using decode step_mult by linarith
      have k_lt: k < mt_pos cM kk using k_le pos_ge_c1 by linarith
      have far_disj:
          (0 :: nat) > mt_pos cM kk + k ∨ (0 :: nat) + k < mt_pos cM kk
        using k_lt by simp
      have entry_zero: mt_tape cM kk 0 = le_tm M
                       using tape_corr[rule_format, OF kklt] unfolding
ae_tape_correspondence_def by simp
      have mt_tape cM_k kk 0 = mt_tape cM kk 0
        using mttm_relpow_tape_off_window[OF trace far_disj] .
      thus ?thesis using entry_zero by simp
    qed
  qed
qed
```

— Sub-step 6b2: M-side off-window preservation per tape regime. Each regime gives a different M-side window characterisation; *mttm_relpow_tape_off_window* applied with the entry M-head position (*pos_corr* + *ae_decode_pos*) and displacement bound $\leq k \leq c$ closes each branch.

```
have m_tape_off_window_le0:
    ⋀kk p. kk < k_tm M ⟹ mt_pos c' kk = 0
          ⟹ p > ?c
          ⟹ mt_tape cM_k kk p = mt_tape cM kk p
proof −
  fix kk and p :: nat
  assume kklt: kk < k_tm M
  assume pos_eq: mt_pos c' kk = 0
  assume p_gt: p > ?c
  have m_pos_zero: mt_pos cM kk = 0
    using pos_corr kklt pos_eq by (simp add: ae_decode_pos_def)
  have far: p > mt_pos cM kk + k using p_gt m_pos_zero k_le by linarith
  have far_disj: p > mt_pos cM kk + k ∨ p + k < mt_pos cM kk
    using far by simp
  show mt_tape cM_k kk p = mt_tape cM kk p
    using mttm_relpow_tape_off_window[OF trace far_disj] .
```

qed
have *m_tape_off_window_le1*:
  $\bigwedge kk\ p.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 1$
    $\Longrightarrow p > 2 * ?c$
    $\Longrightarrow mt_tape\ cM_k\ kk\ p = mt_tape\ cM\ kk\ p$
proof −
 fix *kk* **and** *p* :: *nat*
 assume *kklt*: $kk < k_tm\ M$
 assume *pos_eq*: $mt_pos\ c'\ kk = 1$
 assume *p_gt*: $p > 2 * ?c$
 have *m_pos_le_c*: $mt_pos\ cM\ kk \leq ?c$
 proof −
  have *decode*: $mt_pos\ cM\ kk = Suc\ (c_idx\ (ofs\ kk))$
   using *pos_corr kklt pos_eq* **by** (*simp add*: *ae_decode_pos_def*)
  have *idx_lt*: $c_idx\ (ofs\ kk) < ?c$ **by** (*rule c_idx_lt_card*)
  show *?thesis* **using** *decode idx_lt* **by** *linarith*
 qed
 have *far*: $p > mt_pos\ cM\ kk + k$
  using *p_gt m_pos_le_c k_le* **by** *linarith*
 have *far_disj*: $p > mt_pos\ cM\ kk + k \vee p + k < mt_pos\ cM\ kk$
  using *far* **by** *simp*
 show $mt_tape\ cM_k\ kk\ p = mt_tape\ cM\ kk\ p$
  using *mttm_relpow_tape_off_window*[*OF trace far_disj*] **.**
qed
have *m_tape_off_window_steady*:
  $\bigwedge kk\ p.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk \geq 2$
    $\Longrightarrow p < (mt_pos\ c'\ kk - 2) * ?c + 1$
      $\vee\ p \geq (mt_pos\ c'\ kk - 2) * ?c + 1 + 3 * ?c$
    $\Longrightarrow mt_tape\ cM_k\ kk\ p = mt_tape\ cM\ kk\ p$
proof −
 fix *kk* **and** *p* :: *nat*
 assume *kklt*: $kk < k_tm\ M$
 assume *hge2*: $mt_pos\ c'\ kk \geq 2$
 assume *p_off*: $p < (mt_pos\ c'\ kk - 2) * ?c + 1$
      $\vee\ p \geq (mt_pos\ c'\ kk - 2) * ?c + 1 + 3 * ?c$
 have *decode*:
   $mt_pos\ cM\ kk = (mt_pos\ c'\ kk - 1) * ?c + c_idx\ (ofs\ kk) + 1$
  using *pos_corr kklt hge2* **by** (*simp add*: *ae_decode_pos_def*)
 have *ws_alt*:
   $(mt_pos\ c'\ kk - 1) * ?c = (mt_pos\ c'\ kk - 2) * ?c + ?c$
 proof −
  have *suc_eq*: $Suc\ (mt_pos\ c'\ kk - 2) = mt_pos\ c'\ kk - 1$
   using *hge2* **by** *simp*
  have $Suc\ (mt_pos\ c'\ kk - 2) * ?c = (mt_pos\ c'\ kk - 2) * ?c + ?c$
   by *simp*
  with *suc_eq* **show** *?thesis* **by** *simp*
 qed
 have *m_pos_eq*:
   $mt_pos\ cM\ kk = (mt_pos\ c'\ kk - 2) * ?c + 1 + ?c + c_idx\ (ofs\ kk)$

using *decode ws_alt* **by** *simp*
have *idx_lt*: *c_idx* (*ofs kk*) < *?c* **by** (*rule c_idx_lt_card*)
have *far_disj*: *p* > *mt_pos cM kk* + *k* ∨ *p* + *k* < *mt_pos cM kk*
using *p_off k_le m_pos_eq idx_lt* **by** *linarith*
show *mt_tape cM_k kk p* = *mt_tape cM kk p*
using *mttm_relpow_tape_off_window*[*OF trace far_disj*] **.**
qed

— Sub-step 6b3: head trajectory *c5_pos* for pos=0 tapes. For pos=0, *c4_pos kk* = *0* by *c4_pos*, so SS5 reads *mt_tape c4 kk 0* = *mt_tape c′ kk 0* = *LE_block* by *le_anchor*. *ae_ss5_action*'s first branch (*a* = *LE_block*) fires: write *a* (no-op), direction N. Head stays at 0.
have *c5_pos_for_pos0*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c′ kk* = *0* ⟹ *mt_pos c5 kk* = *0*
proof −
fix *kk*
assume *kklt*: *kk* < *k_tm M*
assume *pos_kk*: *mt_pos c′ kk* = *0*
obtain *qq tts nn qq′ aa5 dr5* **where**
c4_eq5: *c4* = *Config*$_M$ *qq tts nn*
and *c5_eq*: *c5* = *Config*$_M$ *qq′*
(λ*k*. (*tts k*)(*nn k* := *aa5 k*))
(λ*k*. *go_dir* (*dr5 k*) (*nn k*))
and *tr_in*: (*qq*, λ*k*. *tts k* (*nn k*), *qq′*, *aa5*, *dr5*)
∈ *ae_delta_ss5_ss6 M*
using *step5_sub* **by** (*auto elim*: *mttm_step.cases*)
obtain *q ofs buf dest* **where**
qq_eq: *qq* = (*q*, *ofs*, *buf*, *dest*, *SS5*)
and *dr5_eq*:
dr5 = (λ*k*. *if k* < *k_tm M*
then snd (*ae_ss5_action* (*le_tm M*)
(*tts k* (*nn k*)) (*buf k*) (*dest k*))
else dir.*N*)
using *tr_in* **by** (*auto simp*: *ae_delta_ss5_ss6_def*)
have *qq_state*: *qq* = (*q5*, *ofs5*, *buf5*, *dest5*, *SS5*)
using *c4_state c4_eq5* **by** *simp*
have *buf_eq*: *buf* = *buf5* **using** *qq_eq qq_state* **by** *simp*
have *nn_kk*: *nn kk* = *mt_pos c4 kk* **using** *c4_eq5* **by** *simp*
have *c4_kk*: *mt_pos c4 kk* = *0* **using** *c4_pos kklt pos_kk* **by** *simp*
have *nn_kk_0*: *nn kk* = *0* **using** *nn_kk c4_kk* **by** *simp*
have *read_LE*: *tts kk* (*nn kk*) = *LE_block* (*le_tm M*)
proof −
have *tts kk* (*nn kk*) = *mt_tape c4 kk* (*mt_pos c4 kk*)
using *c4_eq5 nn_kk* **by** *simp*
also have … = *mt_tape c′ kk 0*
using *c4_kk tape_c4_eq_c′* **by** *simp*
also have … = *LE_block* (*le_tm M*) **using** *le_anchor*[*rule_format*, *OF kklt*] **by** *simp*
finally show *?thesis* **.**

```
  qed
  obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
    using prod.exhaust by metis
  have dr5_kk: dr5 kk = dir.N
    using dr5_eq buf_eq buf5_kk read_LE kklt by simp
  have mt_pos c5 kk = go_dir (dr5 kk) (nn kk)
    using c5_eq by simp
  also have ... = nn kk using dr5_kk by simp
  also have ... = 0 using nn_kk_0 .
  finally show mt_pos c5 kk = 0 .
qed
```

— Sub-step 6b3: head trajectory *c5_pos* for pos≥2 tapes. Two sub-cases on *dest5 kk*. For $dest \neq AE_Left$: SS5's else branch fires with direction L, head lands at $c4_pos - 1 = mt_pos\ c'\ kk - 1$. For $dest = AE_Left$: SS5's else branch fires with direction R, head lands at $c4_pos + 1 = mt_pos\ c'\ kk + 1$. Mirrors the pos=1 derivations *c5_pos_for_pos1* and *_dest_left*, but with $mt_pos\ c'\ kk \geq 2$ so the result is $mt_pos\ c'\ kk \mp 1$ rather than landing at the *LE_block* sentinel.

```
  have c5_pos_for_pos_ge2:
     ⋀kk. kk < k_tm M ⟹ mt_pos c' kk ≥ 2 ⟹ dest5 kk ≠ AE_Left
          ⟹ mt_pos c5 kk = mt_pos c' kk − 1
  proof −
    fix kk
    assume kklt: kk < k_tm M
    assume hge2: mt_pos c' kk ≥ 2
    assume dest_ne: dest5 kk ≠ AE_Left
    obtain qq tts nn qq' aa5 dr5 where
       c4_eq5: c4 = Config_M qq tts nn
      and c5_eq: c5 = Config_M qq'
                      (λk. (tts k)(nn k := aa5 k))
                      (λk. go_dir (dr5 k) (nn k))
      and tr_in: (qq, λk. tts k (nn k), qq', aa5, dr5)
                 ∈ ae_delta_ss5_ss6 M
      using step5_sub by (auto elim: mttm_step.cases)
    obtain q ofs buf dest where
       qq_eq: qq = (q, ofs, buf, dest, SS5)
      and dr5_eq:
         dr5 = (λk. if k < k_tm M
                    then snd (ae_ss5_action (le_tm M)
                                (tts k (nn k)) (buf k) (dest k))
                    else dir.N)
      using tr_in by (auto simp: ae_delta_ss5_ss6_def)
    have qq_state: qq = (q5, ofs5, buf5, dest5, SS5)
      using c4_state c4_eq5 by simp
    have buf_eq: buf = buf5 using qq_eq qq_state by simp
    have dest_eq: dest = dest5 using qq_eq qq_state by simp
    have nn_kk: nn kk = mt_pos c4 kk using c4_eq5 by simp
    have c4_kk: mt_pos c4 kk = mt_pos c' kk
      using c4_pos kklt by simp
```

```
  have pos_ge1: (1 :: nat) ≤ mt_pos c' kk using hge2 by linarith
  have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
  proof −
    have tts kk (nn kk) = mt_tape c4 kk (mt_pos c4 kk)
      using c4_eq5 nn_kk by simp
    also have ... = mt_tape c' kk (mt_pos c' kk)
      using c4_kk tape_c4_eq_c' by simp
    finally have read_eq: tts kk (nn kk) = mt_tape c' kk (mt_pos c' kk) .
    have mt_tape c' kk (mt_pos c' kk) ≠ LE_block (le_tm M)
      using home_classification[rule_format, OF kklt] pos_ge1 by blast
    thus ?thesis using read_eq by simp
  qed
  obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
    using prod.exhaust by metis
  have h_ne_LE: hh ≠ LE_block (le_tm M)
  proof −
    have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
      using buf5_not_le_per_tape[rule_format, OF kklt] hge2 by blast
    thus ?thesis using buf5_kk by simp
  qed
  have dr5_kk:
      dr5 kk = (if dest5 kk = AE_Left then dir.R else dir.L)
    using dr5_eq buf_eq dest_eq buf5_kk read_ne_LE h_ne_LE kklt by simp
  have dr5_kk_L: dr5 kk = dir.L using dr5_kk dest_ne by simp
  have mt_pos c5 kk = go_dir (dr5 kk) (nn kk)
    using c5_eq by simp
  also have ... = (nn kk) − 1 using dr5_kk_L by simp
  also have ... = mt_pos c' kk − 1 using nn_kk c4_kk by simp
  finally show mt_pos c5 kk = mt_pos c' kk − 1 .
qed

have c5_pos_for_pos_ge2_dest_left:
    ⋀kk. kk < k_tm M ⟹ mt_pos c' kk ≥ 2 ⟹ dest5 kk = AE_Left
        ⟹ mt_pos c5 kk = mt_pos c' kk + 1
proof −
  fix kk
  assume kklt: kk < k_tm M
  assume hge2: mt_pos c' kk ≥ 2
  assume dest_eq: dest5 kk = AE_Left
  obtain qq tts nn qq' aa5 dr5 where
      c4_eq5: c4 = Config_M qq tts nn
    and c5_eq: c5 = Config_M qq'
                    (λk. (tts k)(nn k := aa5 k))
                    (λk. go_dir (dr5 k) (nn k))
    and tr_in: (qq, λk. tts k (nn k), qq', aa5, dr5)
              ∈ ae_delta_ss5_ss6 M
    using step5_sub by (auto elim: mttm_step.cases)
  obtain q ofs buf dest where
      qq_eq: qq = (q, ofs, buf, dest, SS5)
```

```
    and dr5_eq:
      dr5 = (λk. if k < k_tm M
                 then snd (ae_ss5_action (le_tm M)
                                         (tts k (nn k)) (buf k) (dest k))
                 else dir.N)
    using tr_in by (auto simp: ae_delta_ss5_ss6_def)
  have qq_state: qq = (q5, ofs5, buf5, dest5, SS5)
    using c4_state c4_eq5 by simp
  have buf_eq: buf = buf5 using qq_eq qq_state by simp
  have dest_eq': dest = dest5 using qq_eq qq_state by simp
  have nn_kk: nn kk = mt_pos c4 kk using c4_eq5 by simp
  have c4_kk: mt_pos c4 kk = mt_pos c' kk using c4_pos kklt by simp
  have pos_ge1: (1 :: nat) ≤ mt_pos c' kk using hge2 by linarith
  have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
  proof −
    have tts kk (nn kk) = mt_tape c4 kk (mt_pos c4 kk)
      using c4_eq5 nn_kk by simp
    also have ... = mt_tape c' kk (mt_pos c' kk)
      using c4_kk tape_c4_eq_c' by simp
    finally have read_eq: tts kk (nn kk) = mt_tape c' kk (mt_pos c' kk) .
    have mt_tape c' kk (mt_pos c' kk) ≠ LE_block (le_tm M)
      using home_classification[rule_format, OF kklt] pos_ge1 by blast
    thus ?thesis using read_eq by simp
  qed
  obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
    using prod.exhaust by metis
  have h_ne_LE: hh ≠ LE_block (le_tm M)
  proof −
    have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
      using buf5_not_le_per_tape[rule_format, OF kklt] hge2 by blast
    thus ?thesis using buf5_kk by simp
  qed
  have dr5_kk:
      dr5 kk = (if dest5 kk = AE_Left then dir.R else dir.L)
    using dr5_eq buf_eq dest_eq' buf5_kk read_ne_LE h_ne_LE kklt by simp
  have dr5_kk_R: dr5 kk = dir.R using dr5_kk dest_eq by simp
  have mt_pos c5 kk = go_dir (dr5 kk) (nn kk)
    using c5_eq by simp
  also have ... = Suc (nn kk) using dr5_kk_R by simp
  also have ... = mt_pos c' kk + 1 using nn_kk c4_kk by simp
  finally show mt_pos c5 kk = mt_pos c' kk + 1 .
qed
```

— Sub-step 6b3 helper: *left_not_le_c'_steady*. Mirror of *right_not_le_c'* for the left side at pos≥2: block *mt_pos c' k − 1* mapped via the tape correspondence at sub-index 0 (which gives cell address $(mt_pos\ c'\ k - 2) \cdot c + 1$), then no-LE via *no_le_per_tape*'s steady branch at *i = 0*. Powers the SS5→SS6 head-trajectory for pos≥2 tapes with *dest ≠ AE_Left*: c5 reads the cell at *c5_pos = pos − 1*, which is c'-side block pos-1.

```
  have left_not_le_c'_steady:
     ∀k. mt_pos c' k ≥ 2
           ⟶ mt_tape c' k (mt_pos c' k − 1) ≠ LE_block (le_tm M)
  proof (intro allI impI)
    fix k :: nat
    assume hge2: mt_pos c' k ≥ 2
    show mt_tape c' k (mt_pos c' k − 1) ≠ LE_block (le_tm M)
    proof (cases k < k_tm M)
      case True
    let ?x = (enum_class.enum :: 'c list) ! 0
    let ?c = card (UNIV :: 'c set)
    have zero_lt_len: 0 < length (enum_class.enum :: 'c list)
      using c_ge_1 c_eq_len by linarith
    have x_idx0: c_idx ?x = 0
      using c_idx_enum_nth[OF zero_lt_len] .
    have tc_k: ae_tape_correspondence (le_tm M) (mt_tape cM k) (mt_tape c'
k)
      using tape_corr[rule_format, OF True] by simp
    have s_ge1: mt_pos c' k − 1 ≥ 1 using hge2 by linarith
    have addr_via_tc:
       mt_tape cM k ((mt_pos c' k − 2) * ?c + 1)
         = mt_tape c' k (mt_pos c' k − 1) ?x
    proof −
      from tc_k have unf:
         ∀s≥1. ∀i. mt_tape cM k ((s − 1) * ?c + c_idx i + 1)
                   = mt_tape c' k s i
        unfolding ae_tape_correspondence_def by simp
      from unf s_ge1 have eq:
         mt_tape cM k ((mt_pos c' k − 1 − 1) * ?c + c_idx ?x + 1)
           = mt_tape c' k (mt_pos c' k − 1) ?x
        by blast
      have idx_arith:
         mt_pos c' k − 1 − 1 = mt_pos c' k − 2
        using hge2 by simp
      thus ?thesis using eq x_idx0 idx_arith by simp
    qed
    have idx_lt: (0 :: nat) < 3 * ?c using c_ge_1 by simp
    have nle: mt_tape cM k ((mt_pos c' k − 2) * ?c + 1 + 0) ≠ le_tm M
      using no_le_per_tape hge2 idx_lt by blast
    hence nle': mt_tape cM k ((mt_pos c' k − 2) * ?c + 1) ≠ le_tm M
      by simp
    have mt_tape c' k (mt_pos c' k − 1) ?x ≠ le_tm M
      using nle' addr_via_tc by simp
    thus mt_tape c' k (mt_pos c' k − 1) ≠ LE_block (le_tm M)
      unfolding LE_block_def by auto
    next
      case False
      hence kge: k_tm M ≤ k by simp
      have mt_tape c' k (mt_pos c' k − 1) = bl_block (bl_tm M)
```

```
      using gamma_c4 kge
      unfolding ae_tape_in_gamma_block_def tape_c4_eq_c'[symmetric]
      by simp
    thus ?thesis using bl_neq_le by simp
  qed
qed
```

— Sub-step 6b3 helpers: positive buffer-slot characterisations projected from *c4_window_general*'s regime-specific clauses. For pos=0 tapes, *ae_window_invariant_le0* contributes *fst* (*snd blocks*) = *LE_block* (the head buffer slot is LE). For pos=1 tapes, *ae_window_invariant_le1* contributes *fst blocks* = *LE_block* (the left buffer slot is LE). These complement *buf5_not_le_per_tape* (negative facts) with positive identifications, powering the SS7 LE-guard branches that fire for pos=0 ($h = LE$) and the SS8 *AE_Left* dispatch for pos=1 ($l = LE$).

```
  have buf5_h_le_for_pos0:
    ∀k<k_tm M. mt_pos c' k = 0 ⟶ fst (snd (buf5 k)) = LE_block (le_tm M)
  proof (intro allI impI)
    fix k :: nat
    assume klt: k < k_tm M
    assume pos_k: mt_pos c' k = 0
    have wi: ae_window_invariant_general
               (mt_tape cM_k k) (mt_pos cM_k k)
               (dest5 k, ofs5 k) (buf5 k)
               (mt_pos c' k) (le_tm M)
      using cM_k_window_general[rule_format, OF klt] by simp
    hence ae_window_invariant_le0
            (mt_tape cM_k k) (mt_pos cM_k k)
            (dest5 k, ofs5 k) (buf5 k) (le_tm M)
      using pos_k unfolding ae_window_invariant_general_def by simp
    thus fst (snd (buf5 k)) = LE_block (le_tm M)
      unfolding ae_window_invariant_le0_def by simp
  qed

  have buf5_l_le_for_pos1:
    ∀k<k_tm M. mt_pos c' k = 1 ⟶ fst (buf5 k) = LE_block (le_tm M)
  proof (intro allI impI)
    fix k :: nat
    assume klt: k < k_tm M
    assume pos_k: mt_pos c' k = 1
    have wi: ae_window_invariant_general
               (mt_tape cM_k k) (mt_pos cM_k k)
               (dest5 k, ofs5 k) (buf5 k)
               (mt_pos c' k) (le_tm M)
      using cM_k_window_general[rule_format, OF klt] by simp
    hence ae_window_invariant_le1
            (mt_tape cM_k k) (mt_pos cM_k k)
            (dest5 k, ofs5 k) (buf5 k) (le_tm M)
      using pos_k unfolding ae_window_invariant_general_def by simp
    thus fst (buf5 k) = LE_block (le_tm M)
```

unfolding *ae_window_invariant_le1_def* **by** *simp*
qed

— Sub-step 6b3: head trajectory *c6_pos* for pos=0 tapes. *c5_pos = 0* (*c5_pos_for_pos0*); SS6 reads *mt_tape c5 kk 0 = LE_block* via *tape_c5_zero_le*. *ae_ss6_action*'s first branch fires (*a = LE_block*), writing a (no-op), direction R. Head lands at *c5_pos + 1 = 1*.
have *c6_pos_for_pos0*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c′ kk = 0* ⟹ *mt_pos c6 kk = 1*
proof −
fix *kk*
assume *kklt*: *kk* < *k_tm M*
assume *pos_kk*: *mt_pos c′ kk = 0*
obtain *qq tts nn qq′ aa6 dr6* **where**
c5_eq6: *c5* = *Config*$_M$ *qq tts nn*
and *c6_eq*: *c6* = *Config*$_M$ *qq′*
(λ*k*. (*tts k*)(*nn k* := *aa6 k*))
(λ*k*. *go_dir* (*dr6 k*) (*nn k*))
and *tr_in*: (*qq*, λ*k*. *tts k* (*nn k*), *qq′*, *aa6*, *dr6*)
∈ *ae_delta_ss6_ss7 M*
using *step6_sub* **by** (*auto elim*: *mttm_step.cases*)
obtain *q ofs buf dest* **where**
qq_eq: *qq* = (*q*, *ofs*, *buf*, *dest*, *SS6*)
and *dr6_eq*:
dr6 = (λ*k*. *if k* < *k_tm M then snd* (*ae_ss6_action* (*le_tm M*)
(*tts k* (*nn k*)) (*buf k*) (*dest k*)) *else dir.N*)
using *tr_in* **by** (*auto simp*: *ae_delta_ss6_ss7_def*)
have *qq_state*: *qq* = (*q5*, *ofs5*, *buf5*, *dest5*, *SS6*)
using *c5_state c5_eq6* **by** *simp*
have *buf_eq*: *buf* = *buf5* **using** *qq_eq qq_state* **by** *simp*
have *nn_kk*: *nn kk* = *mt_pos c5 kk* **using** *c5_eq6* **by** *simp*
have *c5_kk*: *mt_pos c5 kk* = *0* **using** *c5_pos_for_pos0*[*OF kklt pos_kk*] **.**
have *nn_kk_0*: *nn kk* = *0* **using** *nn_kk c5_kk* **by** *simp*
have *read_LE*: *tts kk* (*nn kk*) = *LE_block* (*le_tm M*)
proof −
have *tts kk* (*nn kk*) = *mt_tape c5 kk* (*mt_pos c5 kk*)
using *c5_eq6 nn_kk* **by** *simp*
also have ... = *mt_tape c5 kk 0* **using** *c5_kk* **by** *simp*
also have ... = *LE_block* (*le_tm M*) **using** *tape_c5_zero_le*[*rule_format*, *OF kklt*] **by** *simp*
finally show *?thesis* **.**
qed
obtain *ll hh rr* **where** *buf5_kk*: *buf5 kk* = (*ll*, *hh*, *rr*)
using *prod.exhaust* **by** *metis*
have *dr6_kk*: *dr6 kk* = *dir.R*
using *dr6_eq buf_eq buf5_kk read_LE kklt* **by** *simp*
have *mt_pos c6 kk* = *go_dir* (*dr6 kk*) (*nn kk*)
using *c6_eq* **by** *simp*
also have ... = *Suc* (*nn kk*) **using** *dr6_kk* **by** *simp*

also have ... = *1* **using** *nn_kk_0* **by** *simp*
finally show *mt_pos c6 kk = 1* **.**
qed

— Sub-step 6b3: head trajectory *c6_pos* for pos≥2 tapes. Both dest branches converge to *c6_pos = pos* (SS6 writes the home block and returns the head to home). Case-split on *dest5 kk = AE_Left*. For *dest = AE_Left*: *c5_pos = pos + 1*, SS6 reads at *pos + 1* non-LE (*right_not_le_c'*), *h* non-LE (*buf5_not_le_per_tape* pos≥2), action's else branch direction L, head lands at *c5_pos − 1 = pos*. For *dest ≠ AE_Left*: *c5_pos = pos − 1*, SS6 reads at *pos − 1* non-LE (*left_not_le_c'_steady*), *h* non-LE, action's else branch direction R, head lands at *c5_pos + 1 = pos*. Extracted to *ae_fwd_c6_pos_for_pos_ge2*.

have *c6_pos_for_pos_ge2*:
⋀*kk*. *kk < k_tm M* ⟹ *mt_pos c' kk* ≥ *2* ⟹ *mt_pos c6 kk = mt_pos c' kk*
by (*rule ae_fwd_c6_pos_for_pos_ge2*[*OF step5 c5_state tape_c4_eq_c' buf5_not_le_per_tape right_not_le_c' c4_pos step6_sub c5_pos_for_pos_ge2 c5_pos_for_pos_ge2_dest_left left_not_le_c'_steady*])

— Sub-step 6b4: head trajectory *c7_pos* for pos=0 tapes. *c6_pos = 1* (*c6_pos_for_pos0*); SS7 reads *mt_tape c6 kk 1* which off-traces through SS6 (*c5_pos = 0*), SS5 (*c4_pos = 0*) to *mt_tape c' kk 1*, non-LE via *right_not_le_c'*. Buffer's head slot *h = LE_block* (*buf5_h_le_for_pos0*), so *ae_ss7_action*'s second branch fires: writes *r*, direction *N* if *dest = AE_Right* else *L*. Head lands at *1* (*dest = AE_Right*) or *0* (otherwise).

have *c7_pos_for_pos0*:
⋀*kk*. *kk < k_tm M* ⟹ *mt_pos c' kk = 0*
⟹ *mt_pos c7 kk* = (*if dest5 kk = AE_Right then 1 else 0*)
proof −
fix *kk*
assume *kklt*: *kk < k_tm M*
assume *pos_kk*: *mt_pos c' kk = 0*
obtain *qq tts nn qq' aa7 dr7* **where**
c6_eq7: *c6* = $Config_M$ *qq tts nn*
and *c7_eq*: *c7* = $Config_M$ *qq'*
(λ*k*. (*tts k*)(*nn k* := *aa7 k*))
(λ*k*. *go_dir* (*dr7 k*) (*nn k*))
and *tr_in*: (*qq*, λ*k*. *tts k* (*nn k*), *qq'*, *aa7*, *dr7*)
∈ *ae_delta_ss7_ss8 M*
using *step7_sub* **by** (*auto elim*: *mttm_step.cases*)
obtain *q ofs buf dest* **where**
qq_eq: *qq* = (*q*, *ofs*, *buf*, *dest*, *SS7*)
and *dr7_eq*:
dr7 = (λ*k*. *if k < k_tm M then snd* (*ae_ss7_action* (*le_tm M*)
(*tts k* (*nn k*)) (*buf k*) (*dest k*)) *else dir.N*)
using *tr_in* **by** (*auto simp*: *ae_delta_ss7_ss8_def*)
have *qq_state*: *qq* = (*q6*, *ofs6*, *buf6*, *dest6*, *SS7*)
using *c6_state c6_eq7* **by** *simp*
have *buf_eq*: *buf = buf6* **using** *qq_eq qq_state* **by** *simp*
have *dest_eq*: *dest = dest6* **using** *qq_eq qq_state* **by** *simp*

have *nn_kk*: *nn kk* = *mt_pos c6 kk* **using** *c6_eq7* **by** *simp*
have *c6_kk*: *mt_pos c6 kk* = *1* **using** *c6_pos_for_pos0*[*OF kklt pos_kk*] **.**
have *nn_kk_1*: *nn kk* = *1* **using** *nn_kk c6_kk* **by** *simp*
have *one_ne_c5_pos*: (*1* :: *nat*) ≠ *mt_pos c5 kk*
using *c5_pos_for_pos0*[*OF kklt pos_kk*] **by** *simp*
have *one_ne_c4_pos*: (*1* :: *nat*) ≠ *mt_pos c4 kk*
using *c4_pos kklt pos_kk* **by** *simp*
have *c6_at_1*: *mt_tape c6 kk 1* = *mt_tape c′ kk 1*
proof −
have *mt_tape c6 kk 1* = *mt_tape c5 kk 1*
using *mttm_step_tape_off_head*[*OF step6 one_ne_c5_pos*] **.**
also have ... = *mt_tape c4 kk 1*
using *mttm_step_tape_off_head*[*OF step5 one_ne_c4_pos*] **.**
also have ... = *mt_tape c′ kk 1*
using *tape_c4_eq_c′* **by** *simp*
finally show *?thesis* **.**
qed
have *read_ne_LE*: *tts kk* (*nn kk*) ≠ *LE_block* (*le_tm M*)
proof −
have *tts kk* (*nn kk*) = *mt_tape c6 kk* (*mt_pos c6 kk*)
using *c6_eq7 nn_kk* **by** *simp*
also have ... = *mt_tape c6 kk 1* **using** *c6_kk* **by** *simp*
also have ... = *mt_tape c′ kk 1* **using** *c6_at_1* **.**
finally have *read_eq*: *tts kk* (*nn kk*) = *mt_tape c′ kk 1* **.**
have *mt_tape c′ kk* (*mt_pos c′ kk* + *1*) ≠ *LE_block* (*le_tm M*)
using *right_not_le_c′* **by** *blast*
hence *mt_tape c′ kk 1* ≠ *LE_block* (*le_tm M*) **using** *pos_kk* **by** *simp*
thus *?thesis* **using** *read_eq* **by** *simp*
qed
obtain *ll hh rr* **where** *buf5_kk*: *buf5 kk* = (*ll*, *hh*, *rr*)
using *prod.exhaust* **by** *metis*
have *hh_le*: *hh* = *LE_block* (*le_tm M*)
proof −
have *fst* (*snd* (*buf5 kk*)) = *LE_block* (*le_tm M*)
using *buf5_h_le_for_pos0*[*rule_format*, *OF kklt*] *pos_kk* **by** *blast*
thus *?thesis* **using** *buf5_kk* **by** *simp*
qed
have *buf6_kk*: *buf6 kk* = (*ll*, *hh*, *rr*)
using *buf5_kk buf6_eq_buf5* **by** *simp*
have *dr7_kk*:
dr7 kk = (*if dest6 kk* = *AE_Right then dir.N else dir.L*)
using *dr7_eq buf_eq dest_eq buf6_kk hh_le read_ne_LE kklt* **by** *simp*
show *mt_pos c7 kk* = (*if dest5 kk* = *AE_Right then 1 else 0*)
proof (*cases dest5 kk* = *AE_Right*)
case *True*
have *dest6_eq*: *dest6 kk* = *AE_Right*
using *True dest6_eq_dest5* **by** *simp*
have *dr7_kk_N*: *dr7 kk* = *dir.N* **using** *dr7_kk dest6_eq* **by** *simp*
have *mt_pos c7 kk* = *go_dir* (*dr7 kk*) (*nn kk*)

```
    using c7_eq by simp
   also have ... = nn kk using dr7_kk_N by simp
   also have ... = 1 using nn_kk_1 .
   finally show ?thesis using True by simp
  next
   case False
   have dest6_ne: dest6 kk ≠ AE_Right
    using False dest6_eq_dest5 by simp
   have dr7_kk_L: dr7 kk = dir.L using dr7_kk dest6_ne by simp
   have mt_pos c7 kk = go_dir (dr7 kk) (nn kk)
    using c7_eq by simp
   also have ... = (nn kk) − 1 using dr7_kk_L by simp
   also have ... = 0 using nn_kk_1 by simp
   finally show ?thesis using False by simp
  qed
 qed
```

— Sub-step 6b4 helper: SS5's write at *c4_pos = pos* for pos≥2 tapes records the home buffer slot *fst (snd (buf5 k))*. Action *ae_ss5_action*'s else branch fires (read non-LE via *home_classification*, head buf slot non-LE via *buf5_not_le_per_tape* pos≥2), writing *h*. Powers the SS7 read characterisation at *c6_pos = pos* (off-write from SS6 which writes at *c5_pos = pos ∓ 1*).

```
 have tape_c5_at_pos_for_pos_ge2:
    ⋀kk. kk < k_tm M ⟹ mt_pos c' kk ≥ 2
         ⟹ mt_tape c5 kk (mt_pos c' kk) = fst (snd (buf5 kk))
 proof −
  fix kk
  assume kklt: kk < k_tm M
  assume hge2: mt_pos c' kk ≥ 2
  obtain qq tts nn qq' aa5 dr5 where
     c4_eq5: c4 = Config_M qq tts nn
   and c5_eq: c5 = Config_M qq'
                   (λk. (tts k)(nn k := aa5 k))
                   (λk. go_dir (dr5 k) (nn k))
   and tr_in: (qq, λk. tts k (nn k), qq', aa5, dr5)
              ∈ ae_delta_ss5_ss6 M
   using step5_sub by (auto elim: mttm_step.cases)
  obtain q ofs buf dest where
     qq_eq: qq = (q, ofs, buf, dest, SS5)
   and aa5_eq:
      aa5 = (λk. if k < k_tm M then fst (ae_ss5_action (le_tm M)
                        (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
   using tr_in by (auto simp: ae_delta_ss5_ss6_def)
  have qq_state: qq = (q5, ofs5, buf5, dest5, SS5)
   using c4_state c4_eq5 by simp
  have buf_eq: buf = buf5 using qq_eq qq_state by simp
  have nn_kk: nn kk = mt_pos c4 kk using c4_eq5 by simp
  have c4_kk: mt_pos c4 kk = mt_pos c' kk using c4_pos kklt by simp
  have pos_ge1: (1 :: nat) ≤ mt_pos c' kk using hge2 by linarith
```

```
  have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
  proof −
    have tts kk (nn kk) = mt_tape c4 kk (mt_pos c4 kk)
      using c4_eq5 nn_kk by simp
    also have ... = mt_tape c' kk (mt_pos c' kk)
      using c4_kk tape_c4_eq_c' by simp
    finally have read_eq:
       tts kk (nn kk) = mt_tape c' kk (mt_pos c' kk) .
    have mt_tape c' kk (mt_pos c' kk) ≠ LE_block (le_tm M)
      using home_classification[rule_format, OF kklt] pos_ge1 by blast
    thus ?thesis using read_eq by simp
  qed
  obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
    using prod.exhaust by metis
  have h_ne_LE: hh ≠ LE_block (le_tm M)
  proof −
    have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
      using buf5_not_le_per_tape[rule_format, OF kklt] hge2 by blast
    thus ?thesis using buf5_kk by simp
  qed
  have aa5_kk: aa5 kk = hh
    using aa5_eq buf_eq buf5_kk read_ne_LE h_ne_LE kklt by simp
  have mt_tape c5 kk (mt_pos c' kk) = aa5 kk
    using c5_eq nn_kk c4_kk by simp
  also have ... = hh using aa5_kk .
  also have ... = fst (snd (buf5 kk)) using buf5_kk by simp
  finally show mt_tape c5 kk (mt_pos c' kk) = fst (snd (buf5 kk)) .
qed
```

— Sub-step 6b4: head trajectory *c7_pos* for pos≥2 tapes. *c6_pos = pos* (*c6_pos_for_pos_ge2*); SS7 reads *mt_tape c6 kk pos = mt_tape c5 kk pos* (off-write from SS6 at *c5_pos = pos ∓ 1*) *= fst* (*snd* (*buf5 kk*)) (*tape_c5_at_pos_for_pos_ge2*), non-LE via *buf5_not_le_per_tape*. *h = fst* (*snd* (*buf6 kk*)) non-LE. *ae_ss7_action*'s else branch fires: dir L if *dest = AE_Left* else R. Head lands at *pos − 1* (*dest = AE_Left*) or *pos + 1* (otherwise).

```
  have c7_pos_for_pos_ge2:
     ⋀kk. kk < k_tm M ⟹ mt_pos c' kk ≥ 2
          ⟹ mt_pos c7 kk = (if dest5 kk = AE_Left
                            then mt_pos c' kk − 1
                            else mt_pos c' kk + 1)
  proof −
    fix kk
    assume kklt: kk < k_tm M
    assume hge2: mt_pos c' kk ≥ 2
    obtain qq tts nn qq' aa7 dr7 where
       c6_eq7: c6 = Config_M qq tts nn
      and c7_eq: c7 = Config_M qq'
                      (λk. (tts k)(nn k := aa7 k))
                      (λk. go_dir (dr7 k) (nn k))
```

```
  and tr_in: (qq, λk. tts k (nn k), qq', aa7, dr7)
               ∈ ae_delta_ss7_ss8 M
  using step7_sub by (auto elim: mttm_step.cases)
obtain q ofs buf dest where
   qq_eq: qq = (q, ofs, buf, dest, SS7)
  and dr7_eq:
     dr7 = (λk. if k < k_tm M then snd (ae_ss7_action (le_tm M)
                    (tts k (nn k)) (buf k) (dest k)) else dir.N)
  using tr_in by (auto simp: ae_delta_ss7_ss8_def)
have qq_state: qq = (q6, ofs6, buf6, dest6, SS7)
  using c6_state c6_eq7 by simp
have buf_eq: buf = buf6 using qq_eq qq_state by simp
have dest_eq: dest = dest6 using qq_eq qq_state by simp
have nn_kk: nn kk = mt_pos c6 kk using c6_eq7 by simp
have c6_kk: mt_pos c6 kk = mt_pos c' kk
  using c6_pos_for_pos_ge2[OF kklt hge2] .
have nn_kk_val: nn kk = mt_pos c' kk using nn_kk c6_kk by simp
obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
  using prod.exhaust by metis
have buf6_kk: buf6 kk = (ll, hh, rr)
  using buf5_kk buf6_eq_buf5 by simp
have h_ne_LE: hh ≠ LE_block (le_tm M)
proof −
  have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
    using buf5_not_le_per_tape[rule_format, OF kklt] hge2 by blast
  thus ?thesis using buf5_kk by simp
qed
have pos_ne_c5_pos: mt_pos c' kk ≠ mt_pos c5 kk
proof (cases dest5 kk = AE_Left)
  case True
  have mt_pos c5 kk = mt_pos c' kk + 1
    using c5_pos_for_pos_ge2_dest_left[OF kklt hge2 True] .
  thus ?thesis by linarith
next
  case False
  have mt_pos c5 kk = mt_pos c' kk − 1
    using c5_pos_for_pos_ge2[OF kklt hge2 False] .
  thus ?thesis using hge2 by linarith
qed
 have c6_at_pos: mt_tape c6 kk (mt_pos c' kk) = mt_tape c5 kk (mt_pos c'
kk)
  using mttm_step_tape_off_head[OF step6 pos_ne_c5_pos] .
have c5_at_pos:
   mt_tape c5 kk (mt_pos c' kk) = fst (snd (buf5 kk))
  using tape_c5_at_pos_for_pos_ge2[OF kklt hge2] .
have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
proof −
  have tts kk (nn kk) = mt_tape c6 kk (mt_pos c6 kk)
    using c6_eq7 nn_kk by simp
```

```
    also have ... = mt_tape c6 kk (mt_pos c' kk) using c6_kk by simp
    also have ... = mt_tape c5 kk (mt_pos c' kk) using c6_at_pos .
    also have ... = fst (snd (buf5 kk)) using c5_at_pos .
    also have ... = hh using buf5_kk by simp
    finally have read_eq: tts kk (nn kk) = hh .
    show ?thesis using read_eq h_ne_LE by simp
  qed
  have dr7_kk:
     dr7 kk = (if dest6 kk = AE_Left then dir.L else dir.R)
    using dr7_eq buf_eq dest_eq buf6_kk h_ne_LE read_ne_LE kklt by simp
  show mt_pos c7 kk = (if dest5 kk = AE_Left
                       then mt_pos c' kk − 1
                       else mt_pos c' kk + 1)
  proof (cases dest5 kk = AE_Left)
    case True
    have dest6_eq: dest6 kk = AE_Left using True dest6_eq_dest5 by simp
    have dr7_kk_L: dr7 kk = dir.L using dr7_kk dest6_eq by simp
    have mt_pos c7 kk = go_dir (dr7 kk) (nn kk)
      using c7_eq by simp
    also have ... = (nn kk) − 1 using dr7_kk_L by simp
    also have ... = mt_pos c' kk − 1 using nn_kk_val by simp
    finally show ?thesis using True by simp
  next
    case False
    have dest6_ne: dest6 kk ≠ AE_Left using False dest6_eq_dest5 by simp
    have dr7_kk_R: dr7 kk = dir.R using dr7_kk dest6_ne by simp
    have mt_pos c7 kk = go_dir (dr7 kk) (nn kk)
      using c7_eq by simp
    also have ... = Suc (nn kk) using dr7_kk_R by simp
    also have ... = mt_pos c' kk + 1 using nn_kk_val by simp
    finally show ?thesis using False by simp
  qed
qed
```

— Sub-step 6b4: head trajectory *c6_pos* for pos=1 tapes, uniform across *dest*. *dest* = *AE_Left* routes via *c6_pos_for_pos1_dest_left* (*c5_pos* = *2*, dir L); *dest* ≠ *AE_Left* routes via *c5_pos_for_pos1* (*c5_pos* = *0*): SS6 reads LE at 0 (*tape_c5_zero_le*), action's first branch (a = LE) fires with direction R, head lands at *c5_pos* + *1* = *1*.

```
  have c6_pos_for_pos1:
     ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 1 ⟹ mt_pos c6 kk = 1
  proof −
    fix kk
    assume kklt: kk < k_tm M
    assume pos_kk: mt_pos c' kk = 1
    show mt_pos c6 kk = 1
    proof (cases dest5 kk = AE_Left)
      case True
      show ?thesis using c6_pos_for_pos1_dest_left[OF kklt pos_kk True] .
```

```
    next
      case False
      obtain qq tts nn qq' aa6 dr6 where
          c5_eq6: c5 = Config_M qq tts nn
        and c6_eq: c6 = Config_M qq'
                        (λk. (tts k)(nn k := aa6 k))
                        (λk. go_dir (dr6 k) (nn k))
        and tr_in: (qq, λk. tts k (nn k), qq', aa6, dr6)
                   ∈ ae_delta_ss6_ss7 M
        using step6_sub by (auto elim: mttm_step.cases)
      obtain q ofs buf dest where
          qq_eq: qq = (q, ofs, buf, dest, SS6)
        and dr6_eq:
            dr6 = (λk. if k < k_tm M then snd (ae_ss6_action (le_tm M)
                                (tts k (nn k)) (buf k) (dest k)) else dir.N)
        using tr_in by (auto simp: ae_delta_ss6_ss7_def)
      have qq_state: qq = (q5, ofs5, buf5, dest5, SS6)
        using c5_state c5_eq6 by simp
      have buf_eq: buf = buf5 using qq_eq qq_state by simp
      have nn_kk: nn kk = mt_pos c5 kk using c5_eq6 by simp
      have c5_kk: mt_pos c5 kk = 0
        using c5_pos_for_pos1[OF kklt pos_kk False] .
      have nn_kk_0: nn kk = 0 using nn_kk c5_kk by simp
      have read_LE: tts kk (nn kk) = LE_block (le_tm M)
      proof −
        have tts kk (nn kk) = mt_tape c5 kk (mt_pos c5 kk)
          using c5_eq6 nn_kk by simp
        also have ... = mt_tape c5 kk 0 using c5_kk by simp
       also have ... = LE_block (le_tm M) using tape_c5_zero_le[rule_format,
OF kklt] by simp
        finally show ?thesis .
      qed
      obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
        using prod.exhaust by metis
      have dr6_kk: dr6 kk = dir.R
        using dr6_eq buf_eq buf5_kk read_LE kklt by simp
      have mt_pos c6 kk = go_dir (dr6 kk) (nn kk)
        using c6_eq by simp
      also have ... = Suc (nn kk) using dr6_kk by simp
      also have ... = 1 using nn_kk_0 by simp
      finally show ?thesis .
    qed
  qed
```

— Sub-step 6b4: head trajectory *c7_pos* for pos=1 tapes. *c6_pos = 1* uniformly. For *dest = AE_Left*: defers to the sub-step 5 lemma *c7_pos_for_pos1_dest_left* (*c7_pos = 0*). For *dest ≠ AE_Left*: SS7 reads *mt_tape c6 kk 1 = mt_tape c5 kk 1* (off-write from SS6 at *c5_pos = 0*) = *fst* (*snd* (*buf5 kk*)) = *hh* (*tape_c5_at_one_pos1*), non-LE (*buf5_not_le_per_tape* pos=1). *h* non-LE;

ae_ss7_action's else branch with *dest* ≠ *AE_Left* gives dir R, head lands at *2*.
have *c7_pos_for_pos1*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c' kk* = *1*
⟹ *mt_pos c7 kk* = (*if dest5 kk* = *AE_Left then 0 else 2*)
proof −
fix *kk*
assume *kklt*: *kk* < *k_tm M*
assume *pos_kk*: *mt_pos c' kk* = *1*
show *mt_pos c7 kk* = (*if dest5 kk* = *AE_Left then 0 else 2*)
proof (*cases dest5 kk* = *AE_Left*)
case *True*
have *mt_pos c7 kk* = *0*
using *c7_pos_for_pos1_dest_left*[*OF kklt pos_kk True*] .
thus *?thesis* **using** *True* **by** *simp*
next
case *False*
obtain *qq tts nn qq' aa7 dr7* **where**
c6_eq7: *c6* = *Config*$_M$ *qq tts nn*
and *c7_eq*: *c7* = *Config*$_M$ *qq'*
(λ*k*. (*tts k*)(*nn k* := *aa7 k*))
(λ*k*. *go_dir* (*dr7 k*) (*nn k*))
and *tr_in*: (*qq*, λ*k*. *tts k* (*nn k*), *qq'*, *aa7*, *dr7*)
∈ *ae_delta_ss7_ss8 M*
using *step7_sub* **by** (*auto elim*: *mttm_step.cases*)
obtain *q ofs buf dest* **where**
qq_eq: *qq* = (*q*, *ofs*, *buf*, *dest*, *SS7*)
and *dr7_eq*:
dr7 = (λ*k*. *if k* < *k_tm M then snd* (*ae_ss7_action* (*le_tm M*)
(*tts k* (*nn k*)) (*buf k*) (*dest k*)) *else dir.N*)
using *tr_in* **by** (*auto simp*: *ae_delta_ss7_ss8_def*)
have *qq_state*: *qq* = (*q6*, *ofs6*, *buf6*, *dest6*, *SS7*)
using *c6_state c6_eq7* **by** *simp*
have *buf_eq*: *buf* = *buf6* **using** *qq_eq qq_state* **by** *simp*
have *dest_eq*: *dest* = *dest6* **using** *qq_eq qq_state* **by** *simp*
have *nn_kk*: *nn kk* = *mt_pos c6 kk* **using** *c6_eq7* **by** *simp*
have *c6_kk*: *mt_pos c6 kk* = *1* **using** *c6_pos_for_pos1*[*OF kklt pos_kk*] .
have *nn_kk_1*: *nn kk* = *1* **using** *nn_kk c6_kk* **by** *simp*
have *one_ne_c5_pos*: (*1* :: *nat*) ≠ *mt_pos c5 kk*
using *c5_pos_for_pos1*[*OF kklt pos_kk False*] **by** *simp*
have *c6_at_1*: *mt_tape c6 kk 1* = *mt_tape c5 kk 1*
using *mttm_step_tape_off_head*[*OF step6 one_ne_c5_pos*] .
obtain *ll hh rr* **where** *buf5_kk*: *buf5 kk* = (*ll*, *hh*, *rr*)
using *prod.exhaust* **by** *metis*
have *buf6_kk*: *buf6 kk* = (*ll*, *hh*, *rr*)
using *buf5_kk buf6_eq_buf5* **by** *simp*
have *c5_at_1*: *mt_tape c5 kk 1* = *hh*
proof −
have *mt_tape c5 kk 1* = *fst* (*snd* (*buf5 kk*))
using *tape_c5_at_one_pos1*[*OF kklt pos_kk*] .

```
    thus ?thesis using buf5_kk by simp
   qed
   have h_ne_LE: hh ≠ LE_block (le_tm M)
   proof −
    have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
     using buf5_not_le_per_tape[rule_format, OF kklt] pos_kk by blast
    thus ?thesis using buf5_kk by simp
   qed
   have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
   proof −
    have tts kk (nn kk) = mt_tape c6 kk (mt_pos c6 kk)
     using c6_eq7 nn_kk by simp
    also have ... = mt_tape c6 kk 1 using c6_kk by simp
    also have ... = mt_tape c5 kk 1 using c6_at_1 .
    also have ... = hh using c5_at_1 .
    finally have tts kk (nn kk) = hh .
    thus ?thesis using h_ne_LE by simp
   qed
   have dr7_kk:
      dr7 kk = (if dest6 kk = AE_Left then dir.L else dir.R)
    using dr7_eq buf_eq dest_eq buf6_kk h_ne_LE read_ne_LE kklt by simp
   have dest6_ne: dest6 kk ≠ AE_Left using False dest6_eq_dest5 by simp
   have dr7_kk_R: dr7 kk = dir.R using dr7_kk dest6_ne by simp
   have mt_pos c7 kk = go_dir (dr7 kk) (nn kk)
    using c7_eq by simp
   also have ... = Suc (nn kk) using dr7_kk_R by simp
   also have ... = 2 using nn_kk_1 by simp
   finally show ?thesis using False by simp
  qed
 qed
```

— Sub-step 6b5 helper: *c6* tape at cell 0 for pos=0 tapes. SS6 writes at *c5_pos = 0*, reading *c5@0 = LE_block* (*tape_c5_zero_le*); action's first branch *a = LE* fires, writing *a = LE_block* back. So *c6@0 = LE_block*.

```
 have tape_c6_zero_le_for_pos0:
    ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 0 ⟹ mt_tape c6 kk 0 = LE_block
(le_tm M)
 proof −
  fix kk
  assume kklt: kk < k_tm M
  assume pos_kk: mt_pos c' kk = 0
  obtain qq tts nn qq' aa6 dr6 where
     c5_eq6: c5 = Config_M qq tts nn
   and c6_eq: c6 = Config_M qq'
                   (λk. (tts k)(nn k := aa6 k))
                   (λk. go_dir (dr6 k) (nn k))
   and tr_in: (qq, λk. tts k (nn k), qq', aa6, dr6)
              ∈ ae_delta_ss6_ss7 M
   using step6_sub by (auto elim: mttm_step.cases)
```

```
    obtain q ofs buf dest where
        qq_eq: qq = (q, ofs, buf, dest, SS6)
      and aa6_eq:
          aa6 = (λk. if k < k_tm M then fst (ae_ss6_action (le_tm M)
                                (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
      using tr_in by (auto simp: ae_delta_ss6_ss7_def)
    have qq_state: qq = (q5, ofs5, buf5, dest5, SS6)
      using c5_state c5_eq6 by simp
    have buf_eq: buf = buf5 using qq_eq qq_state by simp
    have nn_kk: nn kk = mt_pos c5 kk using c5_eq6 by simp
    have c5_kk: mt_pos c5 kk = 0 using c5_pos_for_pos0[OF kklt pos_kk] .
    have nn_kk_0: nn kk = 0 using nn_kk c5_kk by simp
    have read_LE: tts kk (nn kk) = LE_block (le_tm M)
    proof −
      have tts kk (nn kk) = mt_tape c5 kk (mt_pos c5 kk)
        using c5_eq6 nn_kk by simp
      also have ... = mt_tape c5 kk 0 using c5_kk by simp
      also have ... = LE_block (le_tm M) using tape_c5_zero_le[rule_format,
OF kklt] by simp
      finally show ?thesis .
    qed
    obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
      using prod.exhaust by metis
    have aa6_kk: aa6 kk = LE_block (le_tm M)
      using aa6_eq buf_eq buf5_kk read_LE kklt by simp
    have mt_tape c6 kk 0 = aa6 kk
      using c6_eq nn_kk_0 by simp
    also have ... = LE_block (le_tm M) using aa6_kk .
    finally show mt_tape c6 kk 0 = LE_block (le_tm M) .
  qed
```

— Sub-step 6b5 helper: $c7$ tape at cell 1 for pos=0 tapes under $dest = AE_Right$. SS7 writes at $c6_pos = 1$, reading $c6@1$ non-LE and with home-slot $h = LE$; action's second branch fires (h = LE), and with $dest = AE_Right$ writes $r = snd\ (snd\ (buf5\ kk))$.

```
  have tape_c7_at_one_for_pos0_right:
      ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 0 ⟹ dest5 kk = AE_Right
          ⟹ mt_tape c7 kk 1 = snd (snd (buf5 kk))
  proof −
    fix kk
    assume kklt: kk < k_tm M
    assume pos_kk: mt_pos c' kk = 0
    assume dest_eq: dest5 kk = AE_Right
    obtain qq tts nn qq' aa7 dr7 where
        c6_eq7: c6 = Config_M qq tts nn
      and c7_eq: c7 = Config_M qq'
                      (λk. (tts k)(nn k := aa7 k))
                      (λk. go_dir (dr7 k) (nn k))
      and tr_in: (qq, λk. tts k (nn k), qq', aa7, dr7)
```

```
            ∈ ae_delta_ss7_ss8 M
  using step7_sub by (auto elim: mttm_step.cases)
obtain q ofs buf dest where
    qq_eq: qq = (q, ofs, buf, dest, SS7)
  and aa7_eq:
      aa7 = (λk. if k < k_tm M then fst (ae_ss7_action (le_tm M)
                        (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
  using tr_in by (auto simp: ae_delta_ss7_ss8_def)
have qq_state: qq = (q6, ofs6, buf6, dest6, SS7)
  using c6_state c6_eq7 by simp
have buf_eq: buf = buf6 using qq_eq qq_state by simp
have dest_eq2: dest = dest6 using qq_eq qq_state by simp
have nn_kk: nn kk = mt_pos c6 kk using c6_eq7 by simp
have c6_kk: mt_pos c6 kk = 1 using c6_pos_for_pos0[OF kklt pos_kk] .
have nn_kk_1: nn kk = 1 using nn_kk c6_kk by simp
have one_ne_c5_pos: (1 :: nat) ≠ mt_pos c5 kk
  using c5_pos_for_pos0[OF kklt pos_kk] by simp
have one_ne_c4_pos: (1 :: nat) ≠ mt_pos c4 kk
  using c4_pos kklt pos_kk by simp
have c6_at_1: mt_tape c6 kk 1 = mt_tape c' kk 1
proof −
  have mt_tape c6 kk 1 = mt_tape c5 kk 1
    using mttm_step_tape_off_head[OF step6 one_ne_c5_pos] .
  also have ... = mt_tape c4 kk 1
    using mttm_step_tape_off_head[OF step5 one_ne_c4_pos] .
  also have ... = mt_tape c' kk 1 using tape_c4_eq_c' by simp
  finally show ?thesis .
qed
have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
proof −
  have tts kk (nn kk) = mt_tape c6 kk (mt_pos c6 kk)
    using c6_eq7 nn_kk by simp
  also have ... = mt_tape c6 kk 1 using c6_kk by simp
  also have ... = mt_tape c' kk 1 using c6_at_1 .
  finally have read_eq: tts kk (nn kk) = mt_tape c' kk 1 .
  have mt_tape c' kk (mt_pos c' kk + 1) ≠ LE_block (le_tm M)
    using right_not_le_c' by blast
  hence mt_tape c' kk 1 ≠ LE_block (le_tm M) using pos_kk by simp
  thus ?thesis using read_eq by simp
qed
obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
  using prod.exhaust by metis
have buf6_kk: buf6 kk = (ll, hh, rr)
  using buf5_kk buf6_eq_buf5 by simp
have hh_le: hh = LE_block (le_tm M)
proof −
  have fst (snd (buf5 kk)) = LE_block (le_tm M)
    using buf5_h_le_for_pos0[rule_format, OF kklt] pos_kk by blast
  thus ?thesis using buf5_kk by simp
```

```
    qed
    have dest6_eq: dest6 kk = AE_Right using dest_eq dest6_eq_dest5 by simp
    have aa7_kk: aa7 kk = rr
      using aa7_eq buf_eq dest_eq2 dest6_eq buf6_kk hh_le read_ne_LE kklt by
simp
    have mt_tape c7 kk 1 = aa7 kk using c7_eq nn_kk_1 by simp
    also have ... = rr using aa7_kk .
    also have ... = snd (snd (buf5 kk)) using buf5_kk by simp
    finally show mt_tape c7 kk 1 = snd (snd (buf5 kk)) .
  qed

  — Sub-step 6b4: head trajectory c8_pos for pos=0 tapes. Internal case-split on
dest5 kk = AE_Right. For dest = AE_Right: c7_pos = 1, read r non-LE, h = LE,
action's h-LE branch with dest = AE_Right gives (r, N); c8_pos = 1. For dest
≠ AE_Right: c7_pos = 0, read LE (c6@0 = LE via tape_c6_zero_le_for_pos0,
off-write at SS7); action's first branch a = LE gives (a, N); c8_pos = 0.
  have c8_pos_for_pos0:
      ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 0
          ⟹ mt_pos c8 kk = (if dest5 kk = AE_Right then 1 else 0)
  proof −
    fix kk
    assume kklt: kk < k_tm M
    assume pos_kk: mt_pos c' kk = 0
    obtain qq tts nn qq' aa8 dr8 where
      c7_eq8: c7 = Config_M qq tts nn
      and c8_eq: c8 = Config_M qq'
                      (λk. (tts k)(nn k := aa8 k))
                      (λk. go_dir (dr8 k) (nn k))
      and tr_in: (qq, λk. tts k (nn k), qq', aa8, dr8)
                 ∈ ae_delta_ss8_ss1 M
      using step8_sub by (auto elim: mttm_step.cases)
    obtain q ofs buf dest where
      qq_eq: qq = (q, ofs, buf, dest, SS8)
      and dr8_eq:
        dr8 = (λk. if k < k_tm M then snd (ae_ss8_action (le_tm M)
                        (tts k (nn k)) (buf k) (dest k)) else dir.N)
      using tr_in by (auto simp: ae_delta_ss8_ss1_def)
    have qq_state: qq = (q6, ofs6, buf6, dest6, SS8)
      using c7_state c7_eq8 by simp
    have buf_eq: buf = buf6 using qq_eq qq_state by simp
    have dest_eq: dest = dest6 using qq_eq qq_state by simp
    have nn_kk: nn kk = mt_pos c7 kk using c7_eq8 by simp
    obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
      using prod.exhaust by metis
    have buf6_kk: buf6 kk = (ll, hh, rr)
      using buf5_kk buf6_eq_buf5 by simp
    have hh_le: hh = LE_block (le_tm M)
    proof −
      have fst (snd (buf5 kk)) = LE_block (le_tm M)
```

```
      using buf5_h_le_for_pos0[rule_format, OF kklt] pos_kk by blast
    thus ?thesis using buf5_kk by simp
  qed
  show mt_pos c8 kk = (if dest5 kk = AE_Right then 1 else 0)
  proof (cases dest5 kk = AE_Right)
    case True
    have dest6_eq: dest6 kk = AE_Right using True dest6_eq_dest5 by simp
     have c7_kk: mt_pos c7 kk = 1 using c7_pos_for_pos0[OF kklt pos_kk]
True by simp
    have nn_kk_1: nn kk = 1 using nn_kk c7_kk by simp
    have rr_ne_LE: rr ≠ LE_block (le_tm M)
    proof −
      have snd (snd (buf5 kk)) ≠ LE_block (le_tm M)
        using buf5_not_le_per_tape[rule_format, OF kklt] pos_kk by blast
      thus ?thesis using buf5_kk by simp
    qed
    have c7_at_1: mt_tape c7 kk 1 = rr
    proof −
      have mt_tape c7 kk 1 = snd (snd (buf5 kk))
        using tape_c7_at_one_for_pos0_right[OF kklt pos_kk True] .
      thus ?thesis using buf5_kk by simp
    qed
    have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
    proof −
      have tts kk (nn kk) = mt_tape c7 kk (mt_pos c7 kk)
        using c7_eq8 nn_kk by simp
      also have ... = mt_tape c7 kk 1 using c7_kk by simp
      also have ... = rr using c7_at_1 .
      finally have tts kk (nn kk) = rr .
      thus ?thesis using rr_ne_LE by simp
    qed
    have dr8_kk: dr8 kk = dir.N
     using dr8_eq buf_eq dest_eq dest6_eq buf6_kk hh_le read_ne_LE kklt by
simp
    have mt_pos c8 kk = go_dir (dr8 kk) (nn kk)
      using c8_eq by simp
    also have ... = nn kk using dr8_kk by simp
    also have ... = 1 using nn_kk_1 .
    finally show ?thesis using True by simp
  next
    case False
    have c7_kk: mt_pos c7 kk = 0
      using c7_pos_for_pos0[OF kklt pos_kk] False by simp
    have nn_kk_0: nn kk = 0 using nn_kk c7_kk by simp
    have zero_ne_c6_pos: (0 :: nat) ≠ mt_pos c6 kk
      using c6_pos_for_pos0[OF kklt pos_kk] by simp
    have c7_at_0: mt_tape c7 kk 0 = mt_tape c6 kk 0
      using mttm_step_tape_off_head[OF step7 zero_ne_c6_pos] .
    have read_LE: tts kk (nn kk) = LE_block (le_tm M)
```

```
    proof −
      have tts kk (nn kk) = mt_tape c7 kk (mt_pos c7 kk)
        using c7_eq8 nn_kk by simp
      also have ... = mt_tape c7 kk 0 using c7_kk by simp
      also have ... = mt_tape c6 kk 0 using c7_at_0 .
      also have ... = LE_block (le_tm M)
        using tape_c6_zero_le_for_pos0[OF kklt pos_kk] .
      finally show ?thesis .
    qed
    have dr8_kk: dr8 kk = dir.N
      using dr8_eq read_LE kklt by simp
    have mt_pos c8 kk = go_dir (dr8 kk) (nn kk)
      using c8_eq by simp
    also have ... = nn kk using dr8_kk by simp
    also have ... = 0 using nn_kk_0 .
    finally show ?thesis using False by simp
  qed
qed
```

— Sub-step 6b4: head trajectory *c8_pos* for pos≥2 tapes. Three-way case-split on *dest5 kk*. For *AE_Left*: *c7_pos = pos − 1*, SS8 reads *mt_tape c′ kk* (*pos − 1*) non-LE (via *left_not_le_c′_steady*, chained off SS5/SS6/SS7 writes); action's else branch *AE_Left* gives (*l*, *N*); *c8_pos = pos − 1*. For *AE_Home*: *c7_pos = pos + 1*, reads *mt_tape c′ kk* (*pos + 1*) non-LE (via *right_not_le_c′*); else branch *AE_Home* gives (*r*, *L*); *c8_pos = pos*. For *AE_Right*: same read as *AE_Home*; else branch *AE_Right* gives (*r*, *N*); *c8_pos = pos + 1*. Extracted to *ae_fwd_c8_pos_for_pos_ge2*.

```
  have c8_pos_for_pos_ge2:
    ⋀kk. kk < k_tm M ⟹ mt_pos c′ kk ≥ 2
        ⟹ mt_pos c8 kk = (case dest5 kk of
                            AE_Left  ⇒ mt_pos c′ kk − 1
                          | AE_Home  ⇒ mt_pos c′ kk
                          | AE_Right ⇒ mt_pos c′ kk + 1)
    by (rule ae_fwd_c8_pos_for_pos_ge2[OF step5 tape_c4_eq_c′
        buf5_not_le_per_tape right_not_le_c′ c4_pos step6 step7 c7_state
        buf6_eq_buf5 dest6_eq_dest5 step8_sub c5_pos_for_pos_ge2
                        c5_pos_for_pos_ge2_dest_left  left_not_le_c′_steady
c6_pos_for_pos_ge2
        c7_pos_for_pos_ge2])
```

— Sub-step 6b4: head trajectory *c8_pos* for pos=1 tapes. Three-way case-split on *dest5 kk*. For *AE_Left*: defers to sub-step 5's *c7_pos_for_pos1_dest_left* (*c7_pos = 0*) and *tape_c7_zero_le_pos1_dest_left* (*c7@0 = LE*); SS8's a=LE first branch gives (*a*, *N*); *c8_pos = 0*. For *AE_Home*/*AE_Right*: *c7_pos = 2*, SS8 reads *mt_tape c′ kk 2* non-LE (via *right_not_le_c′* at pos=1); *h* non-LE (*buf5_not_le_per_tape* pos=1); else branch dispatches *AE_Home* to dir L (*c8_pos = 1*), *AE_Right* to dir N (*c8_pos = 2*). Extracted to *ae_fwd_c8_pos_for_pos1*.

```
  have c8_pos_for_pos1:
    ⋀kk. kk < k_tm M ⟹ mt_pos c′ kk = 1
        ⟹ mt_pos c8 kk = (case dest5 kk of
```

```
                    AE_Left ⇒ 0
                  | AE_Home ⇒ 1
                  | AE_Right ⇒ 2)
          by (rule ae_fwd_c8_pos_for_pos1[OF step5 tape_c4_eq_c' buf5_not_le_per_tape
              right_not_le_c' c4_pos c5_pos_for_pos1 step6 step7 c7_state
              buf6_eq_buf5 dest6_eq_dest5 c7_pos_for_pos1_dest_left
              tape_c7_zero_le_pos1_dest_left step8_sub c6_pos_for_pos1
              c7_pos_for_pos1])
```

— Sub-step 6d: *tape_c6_zero_le*. Cell 0 of *c6* is *LE_block* for every tape. Case-split on whether SS6's write site *mt_pos c5 kk* equals 0. If yes (pos=0 always, or pos=1 with *dest* ≠ *AE_Left*): SS6 reads *c5@0* = *LE* (*tape_c5_zero_le*) and writes LE back via the first branch. If no (pos=1 with *dest* = *AE_Left* has *c5_pos* = *2*; pos≥2 has *c5_pos* ∈ {*pos−1*, *pos+1*}): SS6 writes off cell 0, so *c6@0* = *c5@0* = *LE*.

```
have tape_c6_zero_le: ∀ kk<k_tm M. mt_tape c6 kk 0 = LE_block (le_tm M)
proof (intro allI impI)
  fix kk
  assume kklt: kk < k_tm M
  obtain qq tts nn qq' aa6 dr6 where
      c5_eq6: c5 = Config_M qq tts nn
    and c6_eq: c6 = Config_M qq'
                    (λk. (tts k)(nn k := aa6 k))
                    (λk. go_dir (dr6 k) (nn k))
    and tr_in: (qq, λk. tts k (nn k), qq', aa6, dr6)
               ∈ ae_delta_ss6_ss7 M
    using step6_sub by (auto elim: mttm_step.cases)
  obtain q ofs buf dest where
      qq_eq: qq = (q, ofs, buf, dest, SS6)
    and aa6_eq:
      aa6 = (λk. if k < k_tm M then fst (ae_ss6_action (le_tm M)
                  (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
    using tr_in by (auto simp: ae_delta_ss6_ss7_def)
  have qq_state: qq = (q5, ofs5, buf5, dest5, SS6)
    using c5_state c5_eq6 by simp
  have buf_eq: buf = buf5 using qq_eq qq_state by simp
  have nn_kk: nn kk = mt_pos c5 kk using c5_eq6 by simp
  show mt_tape c6 kk 0 = LE_block (le_tm M)
  proof (cases mt_pos c5 kk = 0)
    case True
    have nn_kk_0: nn kk = 0 using nn_kk True by simp
    have read_LE: tts kk (nn kk) = LE_block (le_tm M)
    proof −
      have tts kk (nn kk) = mt_tape c5 kk (mt_pos c5 kk)
        using c5_eq6 nn_kk by simp
      also have ... = mt_tape c5 kk 0 using True by simp
     also have ... = LE_block (le_tm M) using tape_c5_zero_le[rule_format,
OF kklt] by simp
      finally show ?thesis .
```

```
    qed
    obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
      using prod.exhaust by metis
    have aa6_kk: aa6 kk = LE_block (le_tm M)
      using aa6_eq buf_eq buf5_kk read_LE kklt by simp
    have mt_tape c6 kk 0 = aa6 kk using c6_eq nn_kk_0 by simp
    also have ... = LE_block (le_tm M) using aa6_kk .
    finally show ?thesis .
  next
    case False
    have zero_ne_c5: (0 :: nat) ≠ mt_pos c5 kk using False by simp
    have mt_tape c6 kk 0 = mt_tape c5 kk 0
      using mttm_step_tape_off_head[OF step6 zero_ne_c5] .
    also have ... = LE_block (le_tm M) using tape_c5_zero_le[rule_format,
OF kklt] by simp
    finally show ?thesis .
  qed
qed
```

— Sub-step 6d: *tape_c7_zero_le*. Cell 0 of *c7* is *LE_block*. SS7's write site *mt_pos c6 kk* is never 0: pos=0 has *c6_pos = 1*; pos=1 has *c6_pos = 1*; pos≥2 has *c6_pos = pos ≥ 2*. Off-write, *c7@0 = c6@0 = LE*.

```
have tape_c7_zero_le: ∀ kk<k_tm M. mt_tape c7 kk 0 = LE_block (le_tm M)
proof (intro allI impI)
  fix kk
  assume kklt: kk < k_tm M
  have zero_ne_c6: (0 :: nat) ≠ mt_pos c6 kk
  proof −
    consider (le0) mt_pos c' kk = 0
          | (le1) mt_pos c' kk = 1
          | (steady) mt_pos c' kk ≥ 2
      by linarith
    thus ?thesis
    proof cases
      case le0   thus ?thesis using c6_pos_for_pos0[OF kklt le0] by simp
    next
      case le1   thus ?thesis using c6_pos_for_pos1[OF kklt le1] by simp
    next
      case steady thus ?thesis using c6_pos_for_pos_ge2[OF kklt steady] by
linarith
    qed
  qed
  have mt_tape c7 kk 0 = mt_tape c6 kk 0
    using mttm_step_tape_off_head[OF step7 zero_ne_c6] .
  also have ... = LE_block (le_tm M)
    using tape_c6_zero_le[rule_format, OF kklt] by simp
  finally show mt_tape c7 kk 0 = LE_block (le_tm M) .
qed
```

— Sub-step 6d: *tape_c8_zero_le*. Cell 0 of *c8* is *LE_block*. Case-split on whether SS8's write site *mt_pos c7 kk* equals 0. If yes (pos=0 with *dest* ≠ *AE_Right*, or pos=1 with *dest* = *AE_Left*): SS8 reads *c7@0* = *LE* (*tape_c7_zero_le*) and writes LE back via the first branch. If no: SS8 writes off 0, so *c8@0* = *c7@0* = *LE*.

```
have tape_c8_zero_le: ∀ kk<k_tm M. mt_tape c8 kk 0 = LE_block (le_tm M)
proof (intro allI impI)
  fix kk
  assume kklt: kk < k_tm M
  obtain qq tts nn qq' aa8 dr8 where
      c7_eq8: c7 = Config_M qq tts nn
    and c8_eq: c8 = Config_M qq'
                    (λk. (tts k)(nn k := aa8 k))
                    (λk. go_dir (dr8 k) (nn k))
    and tr_in: (qq, λk. tts k (nn k), qq', aa8, dr8)
               ∈ ae_delta_ss8_ss1 M
    using step8_sub by (auto elim: mttm_step.cases)
  obtain q ofs buf dest where
      qq_eq: qq = (q, ofs, buf, dest, SS8)
    and aa8_eq:
        aa8 = (λk. if k < k_tm M then fst (ae_ss8_action (le_tm M)
                        (tts k (nn k)) (buf k) (dest k)) else bl_block (bl_tm M))
    using tr_in by (auto simp: ae_delta_ss8_ss1_def)
  have nn_kk: nn kk = mt_pos c7 kk using c7_eq8 by simp
  show mt_tape c8 kk 0 = LE_block (le_tm M)
  proof (cases mt_pos c7 kk = 0)
    case True
    have nn_kk_0: nn kk = 0 using nn_kk True by simp
    have read_LE: tts kk (nn kk) = LE_block (le_tm M)
    proof −
      have tts kk (nn kk) = mt_tape c7 kk (mt_pos c7 kk)
        using c7_eq8 nn_kk by simp
      also have ... = mt_tape c7 kk 0 using True by simp
      also have ... = LE_block (le_tm M)
        using tape_c7_zero_le[rule_format, OF kklt] by simp
      finally show ?thesis .
    qed
    obtain ll hh rr where buf_kk: buf kk = (ll, hh, rr)
      using prod.exhaust by metis
    have aa8_kk: aa8 kk = LE_block (le_tm M)
      using aa8_eq buf_kk read_LE kklt by simp
    have mt_tape c8 kk 0 = aa8 kk using c8_eq nn_kk_0 by simp
    also have ... = LE_block (le_tm M) using aa8_kk .
    finally show ?thesis .
  next
    case False
    have zero_ne_c7: (0 :: nat) ≠ mt_pos c7 kk using False by simp
    have mt_tape c8 kk 0 = mt_tape c7 kk 0
      using mttm_step_tape_off_head[OF step8 zero_ne_c7] .
    also have ... = LE_block (le_tm M)
```

using *tape_c7_zero_le*[*rule_format*, *OF kklt*] **by** *simp*
finally show *?thesis* **.**
qed
qed

— Sub-step 6b6: M'-side off-window preservation. Regime-agnostic form: cells outside all four SS5…SS8 write sites are preserved from c'. Each substep writes at one per-tape position; chaining the four *mttm_step_tape_off_head* facts plus *tape_c4_eq_c'* gives the result. Per-regime instantiation provides the s-not-in-window precondition.
have *c8_tape_off_window*:
$\bigwedge kk\ s.\ s \neq mt_pos\ c4\ kk \Longrightarrow s \neq mt_pos\ c5\ kk$
$\Longrightarrow s \neq mt_pos\ c6\ kk \Longrightarrow s \neq mt_pos\ c7\ kk$
$\Longrightarrow mt_tape\ c8\ kk\ s = mt_tape\ c'\ kk\ s$
proof −
fix *kk*
fix *s* :: *nat*
assume *s_ne_c4*: $s \neq mt_pos\ c4\ kk$
assume *s_ne_c5*: $s \neq mt_pos\ c5\ kk$
assume *s_ne_c6*: $s \neq mt_pos\ c6\ kk$
assume *s_ne_c7*: $s \neq mt_pos\ c7\ kk$
have *c5_eq_c4*: $mt_tape\ c5\ kk\ s = mt_tape\ c4\ kk\ s$
using *mttm_step_tape_off_head*[*OF step5 s_ne_c4*] **.**
have *c6_eq_c5*: $mt_tape\ c6\ kk\ s = mt_tape\ c5\ kk\ s$
using *mttm_step_tape_off_head*[*OF step6 s_ne_c5*] **.**
have *c7_eq_c6*: $mt_tape\ c7\ kk\ s = mt_tape\ c6\ kk\ s$
using *mttm_step_tape_off_head*[*OF step7 s_ne_c6*] **.**
have *c8_eq_c7*: $mt_tape\ c8\ kk\ s = mt_tape\ c7\ kk\ s$
using *mttm_step_tape_off_head*[*OF step8 s_ne_c7*] **.**
show $mt_tape\ c8\ kk\ s = mt_tape\ c'\ kk\ s$
using *c8_eq_c7 c7_eq_c6 c6_eq_c5 c5_eq_c4 tape_c4_eq_c'* **by** *simp*
qed

— Sub-step 6b5: pos=0 in-window c8 value at block 1. Uniformly $c8@1 = rr$ (= the buf5 right slot). Case-split on *dest5 kk = AE_Right*. For *dest = AE_Right*: SS7 wrote r at $c6_pos = 1$; SS8 wrote at $c7_pos = 1$ with action h=LE+dest=Right giving (r, N); $c8@1 = r$. For *dest* ≠ *AE_Right*: SS7 wrote r at $c6_pos = 1$ (action h=LE+dest≠Right gives (r, L)); SS8 wrote at $c7_pos = 0 \neq 1$, off-write; $c8@1 = c7@1 = r$. Extracted to *ae_fwd_c8_at_one_for_pos0*.
have *c8_tape_at_one_for_pos0*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 0$
$\Longrightarrow mt_tape\ c8\ kk\ 1 = snd\ (snd\ (buf5\ kk))$
by (*rule ae_fwd_c8_at_one_for_pos0*[*OF step5 tape_c4_eq_c' buf5_not_le_per_tape right_not_le_c' c4_pos step6 step7_sub c6_state c7_state buf6_eq_buf5 dest6_eq_dest5 step8_sub step8 c5_pos_for_pos0 buf5_h_le_for_pos0 c6_pos_for_pos0 c7_pos_for_pos0*])
— Sub-step 6b5 helper: *c7*'s block 1 for pos=1 tapes. Uniformly $c7@1 = hh$. SS7 writes *aa7* at $c6_pos = 1$; action reads $a = c6@1 = c5@1 = hh$ (via off SS6 write at $c5_pos \in \{0, 2\}$, then *tape_c5_at_one_pos1*), h non-LE; else branch writes *a*

= *hh* regardless of *dest*.
have *c7_tape_at_one_for_pos1*:
$\bigwedge kk.\ kk < k_tm\ M \Longrightarrow mt_pos\ c'\ kk = 1$
$\Longrightarrow mt_tape\ c7\ kk\ 1 = fst\ (snd\ (buf5\ kk))$
proof −
fix *kk*
assume *kklt*: $kk < k_tm\ M$
assume *pos_kk*: $mt_pos\ c'\ kk = 1$
obtain *qq tts nn qq' aa7 dr7* **where**
c6_eq7: $c6 = Config_M\ qq\ tts\ nn$
and *c7_eq*: $c7 = Config_M\ qq'$
$(\lambda k.\ (tts\ k)(nn\ k := aa7\ k))$
$(\lambda k.\ go_dir\ (dr7\ k)\ (nn\ k))$
and *tr_in*: $(qq, \lambda k.\ tts\ k\ (nn\ k), qq', aa7, dr7)$
$\in ae_delta_ss7_ss8\ M$
using *step7_sub* **by** (*auto elim*: *mttm_step.cases*)
obtain *q7 ofs7 buf7 dest7* **where**
qq_eq: $qq = (q7, ofs7, buf7, dest7, SS7)$
and *aa7_eq*:
$aa7 = (\lambda k.\ if\ k < k_tm\ M\ then\ fst\ (ae_ss7_action\ (le_tm\ M)$
$(tts\ k\ (nn\ k))\ (buf7\ k)\ (dest7\ k))\ else\ bl_block\ (bl_tm\ M))$
using *tr_in* **by** (*auto simp*: *ae_delta_ss7_ss8_def*)
have *qq_state*: $qq = (q6, ofs6, buf6, dest6, SS7)$
using *c6_state c6_eq7* **by** *simp*
have *buf7_eq*: $buf7 = buf6$ **using** *qq_eq qq_state* **by** *simp*
have *dest7_eq*: $dest7 = dest6$ **using** *qq_eq qq_state* **by** *simp*
have *nn_kk*: $nn\ kk = mt_pos\ c6\ kk$ **using** *c6_eq7* **by** *simp*
have *c6_kk*: $mt_pos\ c6\ kk = 1$ **using** *c6_pos_for_pos1*[*OF kklt pos_kk*] .
have *nn_kk_1*: $nn\ kk = 1$ **using** *nn_kk c6_kk* **by** *simp*
have *one_ne_c5_pos*: $(1 :: nat) \neq mt_pos\ c5\ kk$
proof (*cases dest5 kk = AE_Left*)
case *True*
have $mt_pos\ c5\ kk = 2$
using *c5_pos_for_pos1_dest_left*[*OF kklt pos_kk True*] .
thus *?thesis* **by** *simp*
next
case *False*
have $mt_pos\ c5\ kk = 0$
using *c5_pos_for_pos1*[*OF kklt pos_kk False*] .
thus *?thesis* **by** *simp*
qed
have *c6_at_1_eq_hh*:
$mt_tape\ c6\ kk\ 1 = fst\ (snd\ (buf5\ kk))$
proof −
have $mt_tape\ c6\ kk\ 1 = mt_tape\ c5\ kk\ 1$
using *mttm_step_tape_off_head*[*OF step6 one_ne_c5_pos*] .
also have $\ldots = fst\ (snd\ (buf5\ kk))$
using *tape_c5_at_one_pos1*[*OF kklt pos_kk*] .
finally show *?thesis* .

```
    qed
    obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
      using prod.exhaust by metis
    have buf6_kk: buf6 kk = (ll, hh, rr)
      using buf5_kk buf6_eq_buf5 by simp
    have h_ne_LE: hh ≠ LE_block (le_tm M)
    proof −
      have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
        using buf5_not_le_per_tape[rule_format, OF kklt] pos_kk by blast
      thus ?thesis using buf5_kk by simp
    qed
    have read_ne_LE: tts kk (nn kk) ≠ LE_block (le_tm M)
    proof −
      have tts kk (nn kk) = mt_tape c6 kk (mt_pos c6 kk)
        using c6_eq7 nn_kk by simp
      also have ... = mt_tape c6 kk 1 using c6_kk by simp
      also have ... = fst (snd (buf5 kk)) using c6_at_1_eq_hh .
      also have ... = hh using buf5_kk by simp
      finally have tts kk (nn kk) = hh .
      thus ?thesis using h_ne_LE by simp
    qed
    — Action's else branch fires, returning (a, _) regardless of dest.
    have aa7_kk: aa7 kk = tts kk (nn kk)
      using aa7_eq buf7_eq buf6_kk h_ne_LE read_ne_LE kklt by simp
    have tts_eq_hh: tts kk (nn kk) = hh
    proof −
      have tts kk (nn kk) = mt_tape c6 kk 1
        using c6_eq7 nn_kk_1 by simp
      also have ... = fst (snd (buf5 kk)) using c6_at_1_eq_hh .
      also have ... = hh using buf5_kk by simp
      finally show ?thesis .
    qed
    have mt_tape c7 kk 1 = aa7 kk using c7_eq nn_kk_1 by simp
    also have ... = tts kk (nn kk) using aa7_kk .
    also have ... = hh using tts_eq_hh .
    also have ... = fst (snd (buf5 kk)) using buf5_kk by simp
    finally show mt_tape c7 kk 1 = fst (snd (buf5 kk)) .
  qed

  — Sub-step 6b5: pos=1 in-window c8 value at block 1. c7_pos ∈ {0, 2}, never
1; so SS8 writes off cell 1. c8@1 = c7@1 = hh.
  have c8_tape_at_one_for_pos1:
      ⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 1
          ⟹ mt_tape c8 kk 1 = fst (snd (buf5 kk))
  proof −
    fix kk
    assume kklt: kk < k_tm M
    assume pos_kk: mt_pos c' kk = 1
    have c7_pos_ne_1: mt_pos c7 kk ≠ 1
```

proof (*cases dest5 kk = AE_Left*)
case *True*
have *mt_pos c7 kk = 0*
using *c7_pos_for_pos1_dest_left*[*OF kklt pos_kk True*] **.**
thus *?thesis* **by** *simp*
next
case *False*
have *mt_pos c7 kk = 2*
using *c7_pos_for_pos1*[*OF kklt pos_kk*] *False* **by** *simp*
thus *?thesis* **by** *simp*
qed
have *ne*: (*1* :: *nat*) ≠ *mt_pos c7 kk* **using** *c7_pos_ne_1* **by** *simp*
have *mt_tape c8 kk 1 = mt_tape c7 kk 1*
using *mttm_step_tape_off_head*[*OF step8 ne*] **.**
also have ... = *fst* (*snd* (*buf5 kk*))
using *c7_tape_at_one_for_pos1*[*OF kklt pos_kk*] **.**
finally show *mt_tape c8 kk 1 = fst* (*snd* (*buf5 kk*)) **.**
qed

— Sub-step 6b5: pos=1 in-window c8 value at block 2. Three-way case-split on *dest5 kk*; uniformly *c8@2 = rr*. For *AE_Left*: SS6 writes *r* at *c5_pos = 2* (action's else branch with *ds = AE_Left* gives (*r*, *L*)); SS7, SS8 off cell 2. For *AE_Home*/*AE_Right*: SS6, SS7 off cell 2; SS8 writes *r* at *c7_pos = 2* (action's else branch with *ds = AE_Home* gives (*r*, *L*), *ds = AE_Right* gives (*r*, *N*)). Extracted to *ae_fwd_c8_at_two_for_pos1*.
have *c8_tape_at_two_for_pos1*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c′ kk = 1*
⟹ *mt_tape c8 kk 2 = snd* (*snd* (*buf5 kk*))
by (*rule ae_fwd_c8_at_two_for_pos1*[*OF step5 c5_state tape_c4_eq_c′*
buf5_not_le_per_tape right_not_le_c′ c4_pos c5_pos_for_pos1 step6_sub
step6 step7 c7_state buf6_eq_buf5 dest6_eq_dest5
c5_pos_for_pos1_dest_left c7_pos_for_pos1_dest_left step8_sub step8
c6_pos_for_pos1 c7_pos_for_pos1])
— Sub-step 6b5 helper: *c7*'s block pos for pos≥2. Uniformly *c7@pos = hh*. SS7 writes *aa7* at *c6_pos = pos*; action reads *a = c6@pos = c5@pos = hh* (via off SS6 write at *c5_pos = pos* ∓ *1*, then *tape_c5_at_pos_for_pos_ge2*); else branch writes *a = hh* regardless of *dest*.
have *c7_tape_at_pos_for_pos_ge2*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c′ kk* ≥ *2*
⟹ *mt_tape c7 kk* (*mt_pos c′ kk*) = *fst* (*snd* (*buf5 kk*))
proof −
fix *kk*
assume *kklt*: *kk* < *k_tm M*
assume *hge2*: *mt_pos c′ kk* ≥ *2*
obtain *qq tts nn qq′ aa7 dr7* **where**
c6_eq7: *c6* = *Config*$_M$ *qq tts nn*
and *c7_eq*: *c7* = *Config*$_M$ *qq′*
(λ*k*. (*tts k*)(*nn k* := *aa7 k*))
(λ*k*. *go_dir* (*dr7 k*) (*nn k*))

```
  and tr_in: (qq, λk. tts k (nn k), qq', aa7, dr7)
            ∈ ae_delta_ss7_ss8 M
  using step7_sub by (auto elim: mttm_step.cases)
obtain q ofs buf dest where
    qq_eq: qq = (q, ofs, buf, dest, SS7)
  and aa7_eq:
      aa7 = (λk. if k < k_tm M then fst (ae_ss7_action (le_tm M)
                        (tts k (nn k)) (buf k)) (dest k)) else bl_block (bl_tm M))
  using tr_in by (auto simp: ae_delta_ss7_ss8_def)
have qq_state: qq = (q6, ofs6, buf6, dest6, SS7)
  using c6_state c6_eq7 by simp
have buf_eq: buf = buf6 using qq_eq qq_state by simp
have nn_kk: nn kk = mt_pos c6 kk using c6_eq7 by simp
have c6_kk: mt_pos c6 kk = mt_pos c' kk
  using c6_pos_for_pos_ge2[OF kklt hge2] .
have nn_kk_pos: nn kk = mt_pos c' kk using nn_kk c6_kk by simp
have pos_ne_c5: mt_pos c' kk ≠ mt_pos c5 kk
proof (cases dest5 kk = AE_Left)
  case True
  have mt_pos c5 kk = mt_pos c' kk + 1
    using c5_pos_for_pos_ge2_dest_left[OF kklt hge2 True] .
  thus ?thesis by linarith
next
  case False
  have mt_pos c5 kk = mt_pos c' kk − 1
    using c5_pos_for_pos_ge2[OF kklt hge2 False] .
  thus ?thesis using hge2 by linarith
qed
have c6_at_pos:
    mt_tape c6 kk (mt_pos c' kk) = fst (snd (buf5 kk))
proof −
  have mt_tape c6 kk (mt_pos c' kk) = mt_tape c5 kk (mt_pos c' kk)
    using mttm_step_tape_off_head[OF step6 pos_ne_c5] .
  also have ... = fst (snd (buf5 kk))
    using tape_c5_at_pos_for_pos_ge2[OF kklt hge2] .
  finally show ?thesis .
qed
obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
  using prod.exhaust by metis
have buf6_kk: buf6 kk = (ll, hh, rr)
  using buf5_kk buf6_eq_buf5 by simp
have h_ne_LE: hh ≠ LE_block (le_tm M)
proof −
  have fst (snd (buf5 kk)) ≠ LE_block (le_tm M)
    using buf5_not_le_per_tape[rule_format, OF kklt] hge2 by blast
  thus ?thesis using buf5_kk by simp
qed
have read_eq_hh: tts kk (nn kk) = hh
proof −
```

have *tts kk* (*nn kk*) = *mt_tape c6 kk* (*mt_pos c6 kk*)
using *c6_eq7 nn_kk* **by** *simp*
also have ... = *mt_tape c6 kk* (*mt_pos c′ kk*) **using** *c6_kk* **by** *simp*
also have ... = *fst* (*snd* (*buf5 kk*)) **using** *c6_at_pos* **.**
also have ... = *hh* **using** *buf5_kk* **by** *simp*
finally show *?thesis* **.**
qed
have *read_ne_LE*: *tts kk* (*nn kk*) ≠ *LE_block* (*le_tm M*)
using *read_eq_hh h_ne_LE* **by** *simp*
have *aa7_kk*: *aa7 kk* = *tts kk* (*nn kk*)
using *aa7_eq buf_eq buf6_kk h_ne_LE read_ne_LE kklt* **by** *simp*
have *mt_tape c7 kk* (*mt_pos c′ kk*) = *aa7 kk*
using *c7_eq nn_kk_pos* **by** *simp*
also have ... = *tts kk* (*nn kk*) **using** *aa7_kk* **.**
also have ... = *hh* **using** *read_eq_hh* **.**
also have ... = *fst* (*snd* (*buf5 kk*)) **using** *buf5_kk* **by** *simp*
finally show *mt_tape c7 kk* (*mt_pos c′ kk*) = *fst* (*snd* (*buf5 kk*)) **.**
qed

— Sub-step 6b5: pos≥2 in-window c8 value at block pos. Uniformly *c8@pos* = *hh*. SS8 writes at *c7_pos* = *pos* ∓ *1*, never pos for pos≥2; *c8@pos* = *c7@pos* = *hh* via off-write preservation and *c7_tape_at_pos_for_pos_ge2*.
have *c8_tape_at_pos_for_pos_ge2*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c′ kk* ≥ *2*
⟹ *mt_tape c8 kk* (*mt_pos c′ kk*) = *fst* (*snd* (*buf5 kk*))
proof −
fix *kk*
assume *kklt*: *kk* < *k_tm M*
assume *hge2*: *mt_pos c′ kk* ≥ *2*
have *pos_ne_c7*: *mt_pos c′ kk* ≠ *mt_pos c7 kk*
proof (*cases dest5 kk* = *AE_Left*)
case *True*
have *mt_pos c7 kk* = *mt_pos c′ kk* − *1*
using *c7_pos_for_pos_ge2*[*OF kklt hge2*] *True* **by** *simp*
thus *?thesis* **using** *hge2* **by** *linarith*
next
case *False*
have *mt_pos c7 kk* = *mt_pos c′ kk* + *1*
using *c7_pos_for_pos_ge2*[*OF kklt hge2*] *False* **by** *simp*
thus *?thesis* **by** *linarith*
qed
have *mt_tape c8 kk* (*mt_pos c′ kk*) = *mt_tape c7 kk* (*mt_pos c′ kk*)
using *mttm_step_tape_off_head*[*OF step8 pos_ne_c7*] **.**
also have ... = *fst* (*snd* (*buf5 kk*))
using *c7_tape_at_pos_for_pos_ge2*[*OF kklt hge2*] **.**
finally show *mt_tape c8 kk* (*mt_pos c′ kk*) = *fst* (*snd* (*buf5 kk*)) **.**
qed

— Sub-step 6b5: pos≥2 in-window c8 value at block pos-1. Uniformly *c8@*(*pos*−*1*)

= *ll*. Three-way dest case-split. For *AE_Left*: SS5/SS6/SS7 off pos-1; SS8 writes *l* at *c7_pos* = *pos* − *1* (action's else branch with *ds* = *AE_Left* gives (*l*, *N*)). For *AE_Home*/*AE_Right*: SS6 writes *l* at *c5_pos* = *pos* − *1* (action's else branch with *ds* ≠ *AE_Left* gives (*l*, *R*)); SS7, SS8 off pos-1. Extracted to *ae_fwd_c8_at_pos_minus_1_ge2*.

have *c8_tape_at_pos_minus_1_for_pos_ge2*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c′ kk* ≥ *2*
⟹ *mt_tape c8 kk* (*mt_pos c′ kk* − *1*) = *fst* (*buf5 kk*)
by (*rule ae_fwd_c8_at_pos_minus_1_ge2*[*OF step5 c5_state tape_c4_eq_c′ buf5_not_le_per_tape c4_pos step6_sub step6 step7 c7_state buf6_eq_buf5 dest6_eq_dest5 step8_sub step8 c5_pos_for_pos_ge2 c5_pos_for_pos_ge2_dest_left left_not_le_c′_steady c6_pos_for_pos_ge2 c7_pos_for_pos_ge2*])

— Sub-step 6b5: pos≥2 in-window c8 value at block pos+1. Uniformly *c8*@(*pos*+*1*) = *rr*. Three-way dest case-split. For *AE_Left*: SS6 writes *r* at *c5_pos* = *pos* + *1* (else branch with *ds* = *AE_Left* gives (*r*, *L*)); SS7, SS8 off pos+1. For *AE_Home*/*AE_Right*: SS6/SS7 off pos+1; SS8 writes *r* at *c7_pos* = *pos* + *1* (else branch with *ds* = *AE_Home* gives (*r*, *L*), *ds* = *AE_Right* gives (*r*, *N*)). Extracted to *ae_fwd_c8_at_pos_plus_1_ge2*.

have *c8_tape_at_pos_plus_1_for_pos_ge2*:
⋀*kk*. *kk* < *k_tm M* ⟹ *mt_pos c′ kk* ≥ *2*
⟹ *mt_tape c8 kk* (*mt_pos c′ kk* + *1*) = *snd* (*snd* (*buf5 kk*))
by (*rule ae_fwd_c8_at_pos_plus_1_ge2*[*OF step5 c5_state tape_c4_eq_c′ buf5_not_le_per_tape right_not_le_c′ c4_pos step6_sub step6 step7 c7_state buf6_eq_buf5 dest6_eq_dest5 step8_sub step8 c5_pos_for_pos_ge2 c5_pos_for_pos_ge2_dest_left c6_pos_for_pos_ge2 c7_pos_for_pos_ge2*])
let *?c* = *card* (*UNIV* :: *'c set*)

— Sub-step 6b5: M-side buf-lin transfers per regime, per in-window block. Projects the window invariant's linearisation clause for each (*regime*, *s*) pair. Six lemmas: pos=0/s=1; pos=1/s=1,2; pos≥2/s=pos-1,pos,pos+1.

have *tape_cM_k_at_one_for_pos0*:
⋀*kk i*. *kk* < *k_tm M* ⟹ *mt_pos c′ kk* = *0*
⟹ *mt_tape cM_k kk* (*Suc* (*c_idx* (*i* :: *'c*)))
= (*snd* (*snd* (*buf5 kk*))) *i*
proof −
fix *kk*
fix *i* :: *'c*
assume *kklt*: *kk* < *k_tm M*
assume *pos_kk*: *mt_pos c′ kk* = *0*
have *c_idx_lt*: *c_idx i* < *?c* **by** (*rule c_idx_lt_card*)
have *wi*: *ae_window_invariant_general*
(*mt_tape cM_k kk*) (*mt_pos cM_k kk*)
(*dest5 kk*, *ofs5 kk*) (*buf5 kk*)
(*mt_pos c′ kk*) (*le_tm M*)
using *cM_k_window_general*[*rule_format*, *OF kklt*] **by** *simp*
hence *win_le0*: *ae_window_invariant_le0*
(*mt_tape cM_k kk*) (*mt_pos cM_k kk*)

```
        (dest5 kk, ofs5 kk) (buf5 kk) (le_tm M)
    using pos_kk unfolding ae_window_invariant_general_def by simp
  have buf_eq:
      mt_tape cM_k kk (Suc (c_idx i))
        = buf_lin_at (buf5 kk) (2 * ?c + c_idx i)
    using win_le0 c_idx_lt unfolding ae_window_invariant_le0_def by blast
  obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
    using prod.exhaust by metis
  have not_lt_c: ¬ 2 * ?c + c_idx i < ?c by simp
  have not_lt_2c: ¬ 2 * ?c + c_idx i < 2 * ?c by simp
  have buf_lin_eq:
      buf_lin_at (buf5 kk) (2 * ?c + c_idx i)
        = rr ((enum_class.enum :: 'c list) ! ((2 * ?c + c_idx i) - 2 * ?c))
    using buf5_kk not_lt_c not_lt_2c
    unfolding buf_lin_at_def Let_def by simp
  have sub_simp: (2 * ?c + c_idx i) - 2 * ?c = c_idx i by simp
  have enum_at_idx:
      (enum_class.enum :: 'c list) ! c_idx i = i
    by (rule c_idx_in_range(2))
  show mt_tape cM_k kk (Suc (c_idx i)) = (snd (snd (buf5 kk))) i
    using buf_eq buf_lin_eq sub_simp enum_at_idx buf5_kk by simp
qed

have tape_cM_k_at_one_for_pos1:
    ⋀kk i. kk < k_tm M ⟹ mt_pos c' kk = 1
          ⟹ mt_tape cM_k kk (Suc (c_idx (i :: 'c)))
               = (fst (snd (buf5 kk))) i
proof −
  fix kk
  fix i :: 'c
  assume kklt: kk < k_tm M
  assume pos_kk: mt_pos c' kk = 1
  have c_idx_lt: c_idx i < ?c by (rule c_idx_lt_card)
  have c_idx_lt_2c: c_idx i < 2 * ?c using c_idx_lt by linarith
  have wi: ae_window_invariant_general
             (mt_tape cM_k kk) (mt_pos cM_k kk)
             (dest5 kk, ofs5 kk) (buf5 kk)
             (mt_pos c' kk) (le_tm M)
    using cM_k_window_general[rule_format, OF kklt] by simp
  hence win_le1: ae_window_invariant_le1
           (mt_tape cM_k kk) (mt_pos cM_k kk)
           (dest5 kk, ofs5 kk) (buf5 kk) (le_tm M)
    using pos_kk unfolding ae_window_invariant_general_def by simp
  have buf_eq:
      mt_tape cM_k kk (Suc (c_idx i))
        = buf_lin_at (buf5 kk) (?c + c_idx i)
    using win_le1 c_idx_lt_2c unfolding ae_window_invariant_le1_def by
blast
  obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
```

```
    using prod.exhaust by metis
  have lt_2c: ?c + c_idx i < 2 * ?c using c_idx_lt by linarith
  have not_lt_c: ¬ ?c + c_idx i < ?c by simp
  have buf_lin_eq:
     buf_lin_at (buf5 kk) (?c + c_idx i)
       = hh ((enum_class.enum :: 'c list) ! ((?c + c_idx i) - ?c))
    using buf5_kk not_lt_c lt_2c
    unfolding buf_lin_at_def Let_def by simp
  have sub_simp: (?c + c_idx i) - ?c = c_idx i by simp
  have enum_at_idx:
     (enum_class.enum :: 'c list) ! c_idx i = i
    by (rule c_idx_in_range(2))
  show mt_tape cM_k kk (Suc (c_idx i)) = (fst (snd (buf5 kk))) i
    using buf_eq buf_lin_eq sub_simp enum_at_idx buf5_kk by simp
qed

have tape_cM_k_at_two_for_pos1:
   ⋀kk i. kk < k_tm M ⟹ mt_pos c' kk = 1
        ⟹ mt_tape cM_k kk (Suc (?c + c_idx (i :: 'c)))
             = (snd (snd (buf5 kk))) i
proof −
  fix kk
  fix i :: 'c
  assume kklt: kk < k_tm M
  assume pos_kk: mt_pos c' kk = 1
  have c_idx_lt: c_idx i < ?c by (rule c_idx_lt_card)
  have addr_lt_2c: ?c + c_idx i < 2 * ?c using c_idx_lt by linarith
  have wi: ae_window_invariant_general
            (mt_tape cM_k kk) (mt_pos cM_k kk)
            (dest5 kk, ofs5 kk) (buf5 kk)
            (mt_pos c' kk) (le_tm M)
    using cM_k_window_general[rule_format, OF kklt] by simp
  hence win_le1: ae_window_invariant_le1
          (mt_tape cM_k kk) (mt_pos cM_k kk)
          (dest5 kk, ofs5 kk) (buf5 kk) (le_tm M)
    using pos_kk unfolding ae_window_invariant_general_def by simp
  have buf_eq:
     mt_tape cM_k kk (Suc (?c + c_idx i))
       = buf_lin_at (buf5 kk) (?c + (?c + c_idx i))
    using win_le1 addr_lt_2c unfolding ae_window_invariant_le1_def by
blast
  obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
    using prod.exhaust by metis
  have addr_eq: ?c + (?c + c_idx i) = 2 * ?c + c_idx i by simp
  have not_lt_c: ¬ 2 * ?c + c_idx i < ?c by simp
  have not_lt_2c: ¬ 2 * ?c + c_idx i < 2 * ?c by simp
  have buf_lin_eq:
     buf_lin_at (buf5 kk) (2 * ?c + c_idx i)
       = rr ((enum_class.enum :: 'c list) ! ((2 * ?c + c_idx i) - 2 * ?c))
```

```
    using buf5_kk not_lt_c not_lt_2c
    unfolding buf_lin_at_def Let_def by simp
  have sub_simp: (2 * ?c + c_idx i) - 2 * ?c = c_idx i by simp
  have enum_at_idx:
      (enum_class.enum :: 'c list) ! c_idx i = i
    by (rule c_idx_in_range(2))
  show mt_tape cM_k kk (Suc (?c + c_idx i)) = (snd (snd (buf5 kk))) i
    using buf_eq addr_eq buf_lin_eq sub_simp enum_at_idx buf5_kk by simp
qed

have tape_cM_k_at_pos_minus_1_for_pos_ge2:
    ⋀kk i. kk < k_tm M ⟹ mt_pos c' kk ≥ 2
          ⟹ mt_tape cM_k kk ((mt_pos c' kk - 2) * ?c
                             + c_idx (i :: 'c) + 1)
              = (fst (buf5 kk)) i
proof −
  fix kk
  fix i :: 'c
  assume kklt: kk < k_tm M
  assume hge2: mt_pos c' kk ≥ 2
  have c_idx_lt: c_idx i < ?c by (rule c_idx_lt_card)
  have c_idx_lt_3c: c_idx i < 3 * ?c using c_idx_lt by linarith
  have wi: ae_window_invariant_general
            (mt_tape cM_k kk) (mt_pos cM_k kk)
            (dest5 kk, ofs5 kk) (buf5 kk)
            (mt_pos c' kk) (le_tm M)
    using cM_k_window_general[rule_format, OF kklt] by simp
  hence win_steady:
      ae_window_invariant (mt_tape cM_k kk) (mt_pos cM_k kk)
        (dest5 kk, ofs5 kk) (buf5 kk)
        ((mt_pos c' kk - 2) * ?c + 1)
    using hge2 unfolding ae_window_invariant_general_def by simp
  have buf_eq:
      mt_tape cM_k kk (((mt_pos c' kk - 2) * ?c + 1) + c_idx i)
        = buf_lin_at (buf5 kk) (c_idx i)
    using win_steady c_idx_lt_3c
    unfolding ae_window_invariant_def by blast
  have addr_simp:
      ((mt_pos c' kk - 2) * ?c + 1) + c_idx i
        = (mt_pos c' kk - 2) * ?c + c_idx i + 1 by simp
  obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
    using prod.exhaust by metis
  have buf_lin_eq:
      buf_lin_at (buf5 kk) (c_idx i)
        = ll ((enum_class.enum :: 'c list) ! c_idx i)
    using buf5_kk c_idx_lt unfolding buf_lin_at_def Let_def by simp
  have enum_at_idx:
      (enum_class.enum :: 'c list) ! c_idx i = i
    by (rule c_idx_in_range(2))
```

```
  show mt_tape cM_k kk ((mt_pos c' kk − 2) * ?c + c_idx i + 1)
        = (fst (buf5 kk)) i
    using buf_eq addr_simp buf_lin_eq enum_at_idx buf5_kk by simp
qed

have tape_cM_k_at_pos_for_pos_ge2:
    ⋀kk i. kk < k_tm M ⟹ mt_pos c' kk ≥ 2
          ⟹ mt_tape cM_k kk ((mt_pos c' kk − 1) * ?c
                                + c_idx (i :: 'c) + 1)
              = (fst (snd (buf5 kk))) i
proof −
  fix kk
  fix i :: 'c
  assume kklt: kk < k_tm M
  assume hge2: mt_pos c' kk ≥ 2
  have c_idx_lt: c_idx i < ?c by (rule c_idx_lt_card)
  have addr_lt_3c: ?c + c_idx i < 3 * ?c using c_idx_lt by linarith
  have wi: ae_window_invariant_general
             (mt_tape cM_k kk) (mt_pos cM_k kk)
             (dest5 kk, ofs5 kk) (buf5 kk)
             (mt_pos c' kk) (le_tm M)
    using cM_k_window_general[rule_format, OF kklt] by simp
  hence win_steady:
      ae_window_invariant (mt_tape cM_k kk) (mt_pos cM_k kk)
        (dest5 kk, ofs5 kk) (buf5 kk)
        ((mt_pos c' kk − 2) * ?c + 1)
    using hge2 unfolding ae_window_invariant_general_def by simp
  have buf_eq:
      mt_tape cM_k kk (((mt_pos c' kk − 2) * ?c + 1) + (?c + c_idx i))
        = buf_lin_at (buf5 kk) (?c + c_idx i)
    using win_steady addr_lt_3c
    unfolding ae_window_invariant_def by blast
  have addr_simp:
      ((mt_pos c' kk − 2) * ?c + 1) + (?c + c_idx i)
        = (mt_pos c' kk − 1) * ?c + c_idx i + 1
    using hge2 by (simp add: algebra_simps diff_mult_distrib)
  have buf_eq_rewritten:
      mt_tape cM_k kk ((mt_pos c' kk − 1) * ?c + c_idx i + 1)
        = buf_lin_at (buf5 kk) (?c + c_idx i)
    using buf_eq[unfolded addr_simp] .
  obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
    using prod.exhaust by metis
  have not_lt_c: ¬ ?c + c_idx i < ?c by simp
  have lt_2c: ?c + c_idx i < 2 * ?c using c_idx_lt by linarith
  have buf_lin_eq:
      buf_lin_at (buf5 kk) (?c + c_idx i)
        = hh ((enum_class.enum :: 'c list) ! ((?c + c_idx i) − ?c))
    using buf5_kk not_lt_c lt_2c
    unfolding buf_lin_at_def Let_def by simp
```

```
  have sub_simp: (?c + c_idx i) - ?c = c_idx i by simp
  have enum_at_idx:
     (enum_class.enum :: 'c list) ! c_idx i = i
    by (rule c_idx_in_range(2))
  show mt_tape cM_k kk ((mt_pos c' kk - 1) * ?c + c_idx i + 1)
        = (fst (snd (buf5 kk))) i
   using buf_eq_rewritten buf_lin_eq sub_simp enum_at_idx buf5_kk by simp
qed

have tape_cM_k_at_pos_plus_1_for_pos_ge2:
    ⋀kk i. kk < k_tm M ⟹ mt_pos c' kk ≥ 2
          ⟹ mt_tape cM_k kk (mt_pos c' kk * ?c
                                + c_idx (i :: 'c) + 1)
              = (snd (snd (buf5 kk))) i
proof −
  fix kk
  fix i :: 'c
  assume kklt: kk < k_tm M
  assume hge2: mt_pos c' kk ≥ 2
  have c_idx_lt: c_idx i < ?c by (rule c_idx_lt_card)
  have addr_lt_3c: 2 * ?c + c_idx i < 3 * ?c using c_idx_lt by linarith
  have wi: ae_window_invariant_general
             (mt_tape cM_k kk) (mt_pos cM_k kk)
             (dest5 kk, ofs5 kk) (buf5 kk)
             (mt_pos c' kk) (le_tm M)
    using cM_k_window_general[rule_format, OF kklt] by simp
  hence win_steady:
     ae_window_invariant (mt_tape cM_k kk) (mt_pos cM_k kk)
        (dest5 kk, ofs5 kk) (buf5 kk)
        ((mt_pos c' kk - 2) * ?c + 1)
    using hge2 unfolding ae_window_invariant_general_def by simp
  have buf_eq:
     mt_tape cM_k kk (((mt_pos c' kk - 2) * ?c + 1) + (2 * ?c + c_idx i))
        = buf_lin_at (buf5 kk) (2 * ?c + c_idx i)
    using win_steady addr_lt_3c
    unfolding ae_window_invariant_def by blast
  have addr_simp:
     ((mt_pos c' kk - 2) * ?c + 1) + (2 * ?c + c_idx i)
       = mt_pos c' kk * ?c + c_idx i + 1
    using hge2 by (simp add: algebra_simps diff_mult_distrib)
  have buf_eq_rewritten:
     mt_tape cM_k kk (mt_pos c' kk * ?c + c_idx i + 1)
        = buf_lin_at (buf5 kk) (2 * ?c + c_idx i)
    using buf_eq[unfolded addr_simp] .
  obtain ll hh rr where buf5_kk: buf5 kk = (ll, hh, rr)
    using prod.exhaust by metis
  have not_lt_c: ¬ 2 * ?c + c_idx i < ?c by simp
  have not_lt_2c: ¬ 2 * ?c + c_idx i < 2 * ?c by simp
  have buf_lin_eq:
```

buf_lin_at (*buf5 kk*) (*2 * ?c + c_idx i*)
= *rr* ((*enum_class.enum* :: *'c list*) ! ((*2 * ?c + c_idx i*) − *2 * ?c*))
using *buf5_kk not_lt_c not_lt_2c*
unfolding *buf_lin_at_def Let_def* **by** *simp*
have *sub_simp*: (*2 * ?c + c_idx i*) − *2 * ?c* = *c_idx i* **by** *simp*
have *enum_at_idx*:
(*enum_class.enum* :: *'c list*) ! *c_idx i* = *i*
by (*rule c_idx_in_range*(*2*))
show *mt_tape cM_k kk* (*mt_pos c' kk * ?c + c_idx i + 1*)
= (*snd* (*snd* (*buf5 kk*))) *i*
using *buf_eq_rewritten buf_lin_eq sub_simp enum_at_idx buf5_kk* **by** *simp*
qed

— Sub-step 6b-assemble: per-tape regime-aware *ae_tape_correspondence* between *cM_k* and *c8*. Per-tape branches on *mt_pos c' kk*; within each regime, per-block branches on *s* match an in-window value (buf-lin transfer + c8 in-window value coincide on buf5 slot) or an off-window value (*cM_k* = *cM* = *c'* M-side via *m_tape_off_window* + *tape_corr*; *c8* = *c'* M'-side via *c8_tape_off_window*). Extracted to *ae_fwd_tape_corr_c8*.
have *tape_corr_c8*:
∀ *kk*<*k_tm M*. *ae_tape_correspondence* (*le_tm M*)
(*mt_tape cM_k kk*) (*mt_tape c8 kk*)
by (*rule ae_fwd_tape_corr_c8*[*OF tape_corr c_ge_1 c4_pos c5_pos_for_pos1*
c5_pos_for_pos1_dest_left c7_pos_for_pos1_dest_left cM_k_zero
m_tape_off_window_le0 m_tape_off_window_le1 m_tape_off_window_steady
c5_pos_for_pos0 c5_pos_for_pos_ge2 c5_pos_for_pos_ge2_dest_left
c6_pos_for_pos0 c6_pos_for_pos_ge2 c7_pos_for_pos0 c7_pos_for_pos_ge2
c6_pos_for_pos1 c7_pos_for_pos1 c8_tape_off_window
c8_tape_at_one_for_pos0 c8_tape_at_one_for_pos1 c8_tape_at_two_for_pos1
c8_tape_at_pos_for_pos_ge2 c8_tape_at_pos_minus_1_for_pos_ge2
c8_tape_at_pos_plus_1_for_pos_ge2 tape_cM_k_at_one_for_pos0
tape_cM_k_at_one_for_pos1 tape_cM_k_at_two_for_pos1
tape_cM_k_at_pos_minus_1_for_pos_ge2 tape_cM_k_at_pos_for_pos_ge2
tape_cM_k_at_pos_plus_1_for_pos_ge2])
— Sub-step 6c: position decode at *c8*. Per-tape regime case-split on *mt_pos c' kk*, then per-dest case-split. The *idx* = *SS1* antecedent excludes the halt branch (which lands at *init_stage* = VFwd). Extracted to *ae_fwd_pos_decode_c8*.
have *pos_decode_c8*:
(*case mt_state c8 of* (_, _, _, _, *idx*) ⇒ *idx* = *SS1*)
⟶ (∀ *k*<*k_tm M*. *mt_pos cM_k k*
= *ae_decode_pos* (*mt_pos c8 k*)
(*case mt_state c8 of* (_, *off*, _, _, _) ⇒ *off k*))
by (*rule ae_fwd_pos_decode_c8*[*OF ofs6_eq_ofs5 c8_state cM_k_window_general*

c8_pos_for_pos0 c8_pos_for_pos_ge2 c8_pos_for_pos1])

— Sub-step 6f: final assembly via *rule that*. The four obligations of the *obtains*: chain composition (*chain8*), *ae_simulates M cM_k c8* (the four per-conjunct facts *conj_i*, *conj_ii*, *tape_corr_c8*, *pos_decode_c8*, plus gamma and buf-gamma side-bands), *ae_buffer_in_gamma_block M c8*, and *c8* cell 0 is *LE_block*.

— SS5 exposure (det-free reverse arm): the SS5 state in (*q_out*, *ofs*, *buf*, *dest*) form (*q_out = mt_state cM_k* via *c4_q_out*) and the window correspondence in (*dest*, *ofs*) bp form (from *c4_window_general* under *c4_state*).

have *q5_eq*: *q5 = mt_state cM_k*
using *c4_q_out c4_state* **by** *simp*
have *c4f_state_exp*: *mt_state c4 = (mt_state cM_k, ofs5, buf5, dest5, SS5)*
using *c4_state q5_eq* **by** *simp*
have *fwd_window_exp*:
∀ *kk*<*k_tm M*. *ae_window_invariant_general*
(*mt_tape cM_k kk*) (*mt_pos cM_k kk*)
(*dest5 kk*, *ofs5 kk*) (*buf5 kk*) (*mt_pos c′ kk*) (*le_tm M*)
using *c4_window_general c4_state* **by** *simp*

obtain *qq8 ofs8 buf8 dest8 idx8* **where**
c8_full: *mt_state c8 = (qq8, ofs8, buf8, dest8, idx8)*
by (*cases mt_state c8*) *auto*
have *conj_i*:
(*idx8 = SS1 ∧ qq8 ∉ {t_tm M, r_tm M}*)
∨ (*qq8 ∈ {t_tm M, r_tm M}*
∧ (*ofs8, buf8, dest8, idx8*) = *init_stage* (*le_tm M*))
using *sim_i_c8 c8_full* **by** *simp*
have *conj_ii*: *mt_state cM_k = qq8*
using *sim_ii_c8 c8_full* **by** *simp*
have *conj_iv*:
idx8 = SS1
⟶ (∀ *k*<*k_tm M*. *mt_pos cM_k k = ae_decode_pos* (*mt_pos c8 k*) (*ofs8 k*))
using *pos_decode_c8 c8_full* **by** *simp*
have *sim_c8*: *ae_simulates M cM_k c8*
unfolding *ae_simulates_def*
using *conj_i conj_ii conj_iv tape_corr_c8 gamma_c8 c8_full*
by *simp*

show *?thesis*
by (*rule that*[*OF chain8 sim_c8 buf_gamma_c8 tape_c8_zero_le*
step1_sub step2_sub step3_sub step4_sub
step5_sub step6_sub step7_sub step8_sub
c4f_state_exp fwd_window_exp buf_gamma_c4])
qed

end
theory *AlphabetEnlargement_Acceptance*
imports *AlphabetEnlargement_ForwardStage*
begin

2.27 Acceptance correspondence and step-count engine

2.27.1 Acceptance correspondence and initial setup

Acceptance correspondence: M accepts iff M''s state equals the canonical M'-accept config (*t_tm M*, *init_stage le_M*). Direct from *ae_simulates_def*'s XOR-style halt-arm disjunct (which pins down (*off*, *buf*, *dest*, *idx*) = *init_stage le_M* exactly when $qM' \in \{t_tm\ M,\ r_tm\ M\}$) plus the q-correspondence conjunct.

```
lemma ae_simulates_accept_iff:
  fixes M :: ('q, 'a) mttm
    and cM :: ('a, 'q) mt_config
    and c' :: ('c :: enum ⇒ 'a,
                'q × ('a, 'c) ae_stage) mt_config
  assumes vM:  valid_mttm M
      and sim: ae_simulates M cM c'
    shows (mt_state cM = t_tm M)
          ⟷ (mt_state c' = (t_tm M, init_stage (le_tm M)))
proof −
  obtain qM' ofs buf dest idx where
      state_comp: mt_state c' = (qM', ofs, buf, dest, idx)
    by (cases mt_state c')
  have sim_body:
      ((idx = SS1 ∧ qM' ∉ {t_tm M, r_tm M})
         ∨ (qM' ∈ {t_tm M, r_tm M}
              ∧ (ofs, buf, dest, idx) = init_stage (le_tm M)))
       ∧ mt_state cM = qM'
       ∧ (∀k<k_tm M. ae_tape_correspondence (le_tm M)
                (mt_tape cM k) (mt_tape c' k))
       ∧ (idx = SS1
            ⟶ (∀k<k_tm M. mt_pos cM k = ae_decode_pos (mt_pos c' k) (ofs
k)))
       ∧ ae_tape_in_gamma_block M c'
    using sim state_comp unfolding ae_simulates_def by simp
  have disj:
      (idx = SS1 ∧ qM' ∉ {t_tm M, r_tm M})
        ∨ (qM' ∈ {t_tm M, r_tm M}
             ∧ (ofs, buf, dest, idx) = init_stage (le_tm M))
    using sim_body by simp
  have qM_eq: mt_state cM = qM' using sim_body by simp
  show ?thesis
  proof
    assume hyp: mt_state cM = t_tm M
    hence qM'_eq_t: qM' = t_tm M using qM_eq by simp
    hence q_in_halt: qM' ∈ {t_tm M, r_tm M} by simp
    have shape: (ofs, buf, dest, idx) = init_stage (le_tm M)
      using disj q_in_halt by auto
    show mt_state c' = (t_tm M, init_stage (le_tm M))
      using state_comp qM'_eq_t shape by simp
```

```
  next
    assume mt_state c' = (t_tm M, init_stage (le_tm M))
    hence qM' = t_tm M using state_comp by simp
    thus mt_state cM = t_tm M using qM_eq by simp
  qed
qed
```

Initial setup: post-validation, the simulation holds between M's initial config on u and M''s post-validation config on *encode_input* (*bl_tm M*) *u*. Combines *ae_validation_post_state_canonical* with the encoder's correctness.

```
lemma ae_init_config_simulates_post_validation:
  fixes M :: ('q, 'a) mttm
    and u :: 'a list
  assumes vM: valid_mttm M
      and u_sub: set u ⊆ Sigma_tm M
      and le_neq_bl: le_tm M ≠ bl_tm M
  obtains n :: nat and c' where
      (ae_init_config M (encode_input (bl_tm M) u), c')
          ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
    and ae_simulates M
          (init_config_mttm M u)
          (c' :: ('c :: enum ⇒ 'a,
                  'q × ('a, 'c) ae_stage) mt_config)
    and ae_buffer_in_gamma_block M c'
    and ∀ kk<k_tm M. mt_tape c' kk 0 = LE_block (le_tm M)
    and ∀ i<n. ∀ d :: ('c :: enum ⇒ 'a,
                       'q × ('a, 'c) ae_stage) mt_config.
          (ae_init_config M (encode_input (bl_tm M) u), d)
             ∈ mttm_step (alphabet_enlarge_delta M) ^^ i
            ⟶ snd (snd (snd (snd (mt_state d))))
                 ∈ {VFwd, VFwdPad, VRet}
proof −
  obtain n c' where
      A: (ae_init_config M (encode_input (bl_tm M) u), c')
          ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
    and B: ae_simulates M
             (init_config_mttm M u)
             (c' :: ('c ⇒ 'a,
                     'q × ('a, 'c) ae_stage) mt_config)
    and C: ae_buffer_in_gamma_block M c'
    and D: ∀ kk<k_tm M. mt_tape c' kk 0 = LE_block (le_tm M)
    and E: ∀ i<n. ∀ d :: ('c ⇒ 'a,
                          'q × ('a, 'c) ae_stage) mt_config.
             (ae_init_config M (encode_input (bl_tm M) u), d)
                ∈ mttm_step (alphabet_enlarge_delta M) ^^ i
               ⟶ snd (snd (snd (snd (mt_state d))))
                    ∈ {VFwd, VFwdPad, VRet}
    by (rule ae_validation_post_state_canonical[OF vM u_sub le_neq_bl])
```

show *?thesis* **using** *A B C D E* **by** (*rule that*)
qed

2.27.2 Step-count machinery and chunked simulation engine

Step-counting lemmas: validation phase bounded by $2 \cdot n + f_v$; simulation phase bounded by $8 \cdot \lceil T(c \cdot n) \,/\, c \rceil$.

This is the general-input form of the validation-phase step count: for any well-formed AE-input w, the validation phase completes in $O(|w|)$ steps and lands either at the canonical SS1 configuration or in the reject state. It is not invoked by the headline time theorem *alphabet_enlarge_time* below for canonical encoder-image inputs that the base machine accepts, the narrower form *ae_validation_well_formed_to_SS1* in theory *AlphabetEnlargement_ValidationBound* gives the exact step count $2 \cdot |w| + 4$ with SS1 as the only outcome, which is what the linear-speedup proof needs. The general form is retained as a structural completeness result describing the AE machine's runtime behaviour on non-canonical or rejected inputs of potential use for downstream consumers that reason about reject paths, and for the nondeterministic-reverse research thread.

lemma *ae_validation_phase_step_count*:
fixes M :: $('q, 'a)$ *mttm*
and w :: $(('c :: enum) \Rightarrow 'a)$ *list*
assumes *vM*: *valid_mttm M*
and *w_sub*: *set w* $\subseteq$ *gamma_block* (*Sigma_tm M* $\cup$ {*bl_tm M*})
and *s_neq_t*: *s_tm M* $\neq$ *t_tm M*
and *s_neq_r*: *s_tm M* $\neq$ *r_tm M*
and *le_neq_bl*: *le_tm M* $\neq$ *bl_tm M*
obtains f_v :: *nat* **and** n :: *nat* **and** c' **where**
$n \leq 2 * length\ w + f_v$
and (*ae_init_config M w*, c')
$\in$ *mttm_step* (*alphabet_enlarge_delta M*) ^^ n
and *case mt_state* c' *of* (qM', _, _, _, *idx*) $\Rightarrow$
idx = *SS1* $\vee$ qM' = *r_tm M*
proof −
obtain f_v n c' **where**
A: $n \leq 2 * length\ w + f_v$
and *B*: (*ae_init_config M w*, c')
$\in$ *mttm_step* (*alphabet_enlarge_delta M*) ^^ n
and *C*: *case mt_state* c' *of* (qM', _, _, _, *idx*) $\Rightarrow$
idx = *SS1* $\vee$ qM' = *r_tm M*
using *ae_validation_steps_bound*[*OF vM w_sub s_neq_t s_neq_r le_neq_bl*]
by *metis*
show *?thesis* **using** *A B C* **by** (*rule that*)
qed

Chunked-induction engine for the simulation-phase step count. Given a specific (finite) accepting M-path of length n from a reachable *cM* with

a paired SS1 M'-config c' satisfying the invariants *buf_gamma*, *le_anchor*, exhibit a corresponding accepting M'-path of length at most $8 \cdot \lceil n \,/\, c \rceil$.

Proof structure (when discharged): induction on the M-path length n, taking the next chunk of up to $c = card\ (UNIV :: 'c\ set)$ M-steps per stage and invoking *ae_simulates_forward_stage_general*. The invariant *buf_gamma c'_j* $\wedge$ *le_anchor c'_j* is preserved by *forward_stage_general*'s output conjuncts (just strengthened in the previous commit). The M-side *no_le_per_tape* hypothesis for the per-tape unified forward stage is discharged from *valid_reach_LE_only_pos0_mttm* on the substrate, threaded through *reach_M*.

Wrapped by *ae_simulation_phase_step_count* below to produce the named-bound *obtains*-form.

```
lemma ae_simulation_phase_chunked:
  fixes M :: ('q, 'a) mttm
    and w :: 'a list
    and cM cM_final :: ('a, 'q) mt_config
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
    and n :: nat
  assumes vM:          valid_mttm M
      and lu:          le_unique M
      and w_sub:       set w ⊆ Sigma_tm M
      and le_neq_bl:   le_tm M ≠ bl_tm M
      and s_neq_t:     s_tm M ≠ t_tm M
      and s_neq_r:     s_tm M ≠ r_tm M
      and reach_M:     (init_config_mttm M w, cM)
                         ∈ (mttm_step (delta_tm M))*
      and trace:       (cM, cM_final) ∈ (mttm_step (delta_tm M))^^n
      and accept:      mt_state cM_final = t_tm M
      and sim:         ae_simulates M cM c'
      and buf_gamma:   ae_buffer_in_gamma_block M c'
      and le_anchor:   ∀kk<k_tm M. mt_tape c' kk 0 = LE_block (le_tm M)
  shows ∃m c''. m ≤ 8 * ((n + card (UNIV :: 'c set) - 1)
                           div card (UNIV :: 'c set))
              ∧ (c', c'') ∈ (mttm_step (alphabet_enlarge_delta M))^^m
              ∧ mt_state c'' = (t_tm M, init_stage (le_tm M))
  using reach_M trace accept sim buf_gamma le_anchor
proof (induction n arbitrary: cM cM_final c' rule: less_induct)
  case (less n)
  — Strong-induction case for path-length n. The hypotheses reach_M, trace,
accept, sim, buf_gamma, le_anchor have been re-quantified over cM, cM_final,
c' by the induction; less.prems restates them for this n, and less.IH gives the
conclusion for every strictly smaller n' (with its own fresh cM', cM_final', c'').
  show ?case
  proof (cases n)
    case 0
    — Base case: n = 0 means cM = cM_final (length-0 trace), so mt_state cM
```

$= t_tm\ M$ by *accept*. *ae_simulates_accept_iff* then forces $mt_state\ c' = (t_tm\ M, init_stage\ (le_tm\ M))$. Witness: $m = 0$, $c'' = c'$.

have *trace_zero*: $(cM, cM_final) \in (mttm_step\ (delta_tm\ M))^{\frown\frown}0$
using *less.prems*(2) ‹$n = 0$› **by** *simp*
have *cM_eq*: $cM_final = cM$
using *trace_zero* **by** *simp*
have *mt_cM*: $mt_state\ cM = t_tm\ M$
using *less.prems*(3) *cM_eq* **by** *simp*
have *state_c'*: $mt_state\ c' = (t_tm\ M, init_stage\ (le_tm\ M))$
using *ae_simulates_accept_iff*[*OF vM less.prems*(4)] *mt_cM* **by** *simp*
have *run_zero*:
$(c', c') \in (mttm_step\ (alphabet_enlarge_delta\ M))^{\frown\frown}0$
by *simp*
have *bound_zero*:
$(0 :: nat) \leq 8 * ((n + card\ (UNIV :: 'c\ set) - 1)$
$div\ card\ (UNIV :: 'c\ set))$
by *simp*
show *?thesis*
using *bound_zero run_zero state_c'* **by** *blast*

next

case (*Suc n_minus_1*)

— Inductive case: $n > 0$. Select the chunk size $k_chunk \leq c$: the earliest halt index in $[1..min\ n\ c]$, or c if no halt in that range. Split the trace as $cM \rightarrow k_chunk\ cM_k \rightarrow (n - k_chunk)\ cM_final$; apply *forward_stage_general* to get an M'-witness for the first chunk; apply *less.IH* on $n - k_chunk < n$ for the remaining trace.

— Substep A: extract the first M-step from the non-empty trace and derive cM's status (non-halt, in Q) from *mttm_step_src*.

have *trace_Suc*: (cM, cM_final)
$\in mttm_step\ (delta_tm\ M)\ ^{\frown\frown}\ Suc\ n_minus_1$
using *less.prems*(2) *Suc* **by** *simp*
obtain *cM_1* **where**
step_first: $(cM, cM_1) \in mttm_step\ (delta_tm\ M)$
and *rest_trace*: (cM_1, cM_final)
$\in mttm_step\ (delta_tm\ M)\ ^{\frown\frown}\ n_minus_1$
using *relpow_Suc_D2*[*OF trace_Suc*] **by** *blast*
have *cM_in_Q*: $mt_state\ cM \in Q_tm\ M$
using *mttm_step_src_in_Q*[*OF vM step_first*] **.**
have *cM_neq_t*: $mt_state\ cM \neq t_tm\ M$
using *mttm_step_src_neq_t*[*OF vM step_first*] **.**
have *cM_neq_r*: $mt_state\ cM \neq r_tm\ M$
using *mttm_step_src_neq_r*[*OF vM step_first*] **.**

— Substep B: name the alphabet-grouping size c and define the chunk size as $k_chunk = min\ n\ c$. Since the universal-form *prefix_nhalt* hypothesis has been retired from *forward_stage_general* (the AE-chain audit found it was dead weight propagated through every layer without being used substantively), the chunk just needs to satisfy *end_or_halt*, which holds via $k_chunk = n = halt$ when $n \leq c$, or via $k_chunk = c$ when $n > c$.

define c **where** $c \equiv card\ (UNIV :: 'c\ set)$

have *c_pos*: *0* < *c*
proof −
have (*c_first* :: *'c*) ∈ *UNIV* **by** *simp*
thus *?thesis* **unfolding** *c_def* **by** (*simp add*: *card_gt_0_iff*)
qed
define *k_chunk* **where** *k_chunk* ≡ *min n c*
have *k_chunk_pos*: *0* < *k_chunk*
using *c_pos Suc* **unfolding** *k_chunk_def* **by** *simp*
have *k_chunk_le_n*: *k_chunk* ≤ *n*
unfolding *k_chunk_def* **by** *simp*
have *k_chunk_le_c*: *k_chunk* ≤ *c*
unfolding *k_chunk_def* **by** *simp*
have *k_chunk_lt_n_or_eq_n*: *k_chunk* < *n* ∨ *k_chunk* = *n*
using *k_chunk_le_n* **by** *linarith*
— Substep C: split the M-trace at *k_chunk*. Since *k_chunk* ≤ *n*, *k_chunk* + (*n* − *k_chunk*) = *n*; *relpow_add* turns *rel* ^^ *n* into the composition *rel* ^^ *k_chunk* *O* *rel* ^^ (*n* − *k_chunk*), which a single unpacking exposes *cM_k*.
have *trace_split_rel*:
(*cM*, *cM_final*) ∈ (*mttm_step* (*delta_tm M*) ^^ *k_chunk*)
O (*mttm_step* (*delta_tm M*) ^^ (*n* − *k_chunk*))
proof −
have *sum_eq*: *k_chunk* + (*n* − *k_chunk*) = *n*
using *k_chunk_le_n* **by** *simp*
have (*cM*, *cM_final*) ∈ *mttm_step* (*delta_tm M*)
^^ (*k_chunk* + (*n* − *k_chunk*))
using *less.prems*(*2*) *sum_eq* **by** *simp*
thus *?thesis* **by** (*simp add*: *relpow_add*)
qed
obtain *cM_k* **where**
chunk_trace: (*cM*, *cM_k*) ∈ *mttm_step* (*delta_tm M*) ^^ *k_chunk*
and *rest_after_chunk*:
(*cM_k*, *cM_final*) ∈ *mttm_step* (*delta_tm M*) ^^ (*n* − *k_chunk*)
using *trace_split_rel* **by** *auto*
— Substep E: *end_or_halt*. Two cases: - *k_chunk* = *n*: *n* ≤ *c*, *cM_k* = *cM_final* (from length-zero remaining trace), state is *t_tm M* by *accept*. Second disjunct fires. - *k_chunk* < *n*: *n* > *c*, so *min n c* = *c*, i.e., *k_chunk* = *c*. First disjunct fires.
have *end_or_halt*:
k_chunk = *c* ∨ *mt_state cM_k* ∈ {*t_tm M*, *r_tm M*}
proof (*cases k_chunk* = *n*)
case *True*
hence *n* − *k_chunk* = *0* **by** *simp*
hence *cM_k* = *cM_final*
using *rest_after_chunk* **by** *simp*
hence *mt_state cM_k* = *t_tm M*
using *less.prems*(*3*) **by** *simp*
thus *?thesis* **by** *simp*
next
case *False*

```
    hence k_chunk < n using k_chunk_le_n by linarith
    hence min n c < n unfolding k_chunk_def by simp
    hence c < n by linarith
    hence min n c = c by simp
    hence k_chunk = c unfolding k_chunk_def by simp
    thus ?thesis by simp
  qed
```

— Substep F: derive *no_le_per_tape* for *cM* from substrate reachability. Each disjunct of *no_le_per_tape* asserts that an M-tape cell at some positive index is not the left-end marker. The substrate lemma *valid_reach_LE_only_pos0_mttm* exactly delivers this for every cell index $p \neq 0$ along a reachable trace from *init_config_mttm M w*; the three disjuncts' cell indices ($(mt_pos\ c'\ kk - 2) * c + 1 + i$ and *Suc i* twice) are all syntactically positive, so the $p \neq 0$ obligation is discharged by *by simp*.

```
  have reach_cM: (init_config_mttm M w, cM)
                   ∈ (mttm_step (delta_tm M))*
    using less.prems(1) .
  have no_le_per_tape:
     ∀ kk. (mt_pos c' kk ≥ 2
            ⟶ (∀ i. i < 3 * c
                  ⟶ mt_tape cM kk
                      ((mt_pos c' kk − 2) * c + 1 + i)
                    ≠ le_tm M))
          ∧ (mt_pos c' kk = 1
              ⟶ (∀ i. i < 2 * c
                    ⟶ mt_tape cM kk (Suc i) ≠ le_tm M))
          ∧ (mt_pos c' kk = 0
              ⟶ (∀ i. i < c
                    ⟶ mt_tape cM kk (Suc i) ≠ le_tm M))
  proof (intro allI conjI impI allI impI)
    fix kk :: nat and i :: nat
    assume 2 ≤ mt_pos c' kk and i < 3 * c
    have idx_nz: (mt_pos c' kk − 2) * c + 1 + i ≠ 0 by simp
    show mt_tape cM kk ((mt_pos c' kk − 2) * c + 1 + i) ≠ le_tm M
      using valid_reach_LE_only_pos0_mttm
            [OF vM lu w_sub reach_cM idx_nz le_neq_bl[symmetric]] .
  next
    fix kk :: nat and i :: nat
    assume mt_pos c' kk = 1 and i < 2 * c
    have idx_nz: Suc i ≠ 0 by simp
    show mt_tape cM kk (Suc i) ≠ le_tm M
      using valid_reach_LE_only_pos0_mttm
            [OF vM lu w_sub reach_cM idx_nz le_neq_bl[symmetric]] .
  next
    fix kk :: nat and i :: nat
    assume mt_pos c' kk = 0 and i < c
    have idx_nz: Suc i ≠ 0 by simp
    show mt_tape cM kk (Suc i) ≠ le_tm M
      using valid_reach_LE_only_pos0_mttm
```

[*OF vM lu w_sub reach_cM idx_nz le_neq_bl*[*symmetric*]] .

qed

— Substep G: invoke *forward_stage_general* on the first chunk *cM* →*k_chunk* *cM_k* to obtain an M'-witness *c8* of length 8 satisfying the invariants needed to recurse on the remaining trace. All twelve hypotheses are now in hand: *vM*, *sim* (from *less.prems*), *cM* in *Q* + non-halt (substep A), *buf_gamma* + *le_anchor* (from *less.prems*), *k_chunk_le_c* (substep B), *chunk_trace* (substep C), *end_or_halt* (substep E), *le_neq_bl* (outer assume), *no_le_per_tape* (substep F). Uses *obtain* ... *by* (*rule* ...) as the *obtains*-elim pattern (automation diverges on *obtains* rules with multiple output conjuncts).

have *k_chunk_le_card*: *k_chunk* ≤ *card* (*UNIV* :: *'c set*)
using *k_chunk_le_c* **unfolding** *c_def* .
have *end_or_halt_card*:
k_chunk = *card* (*UNIV* :: *'c set*)
∨ *mt_state cM_k* ∈ {*t_tm M*, *r_tm M*}
using *end_or_halt* **unfolding** *c_def* .
have *no_le_per_tape_card*:
∀ *kk*. (*mt_pos c' kk* ≥ *2*
⟶ (∀ *i*. *i* < *3* ∗ *card* (*UNIV* :: *'c set*)
⟶ *mt_tape cM kk*
((*mt_pos c' kk* − *2*) ∗ *card* (*UNIV* :: *'c set*) + *1* + *i*)
≠ *le_tm M*))
∧ (*mt_pos c' kk* = *1*
⟶ (∀ *i*. *i* < *2* ∗ *card* (*UNIV* :: *'c set*)
⟶ *mt_tape cM kk* (*Suc i*) ≠ *le_tm M*))
∧ (*mt_pos c' kk* = *0*
⟶ (∀ *i*. *i* < *card* (*UNIV* :: *'c set*)
⟶ *mt_tape cM kk* (*Suc i*) ≠ *le_tm M*))
using *no_le_per_tape* **unfolding** *c_def* .
obtain *c8* **where**
chain8: (*c'*, *c8*) ∈ *mttm_step* (*alphabet_enlarge_delta M*) ^^ *8*
and *sim_c8*: *ae_simulates M cM_k c8*
and *buf_gamma_c8*: *ae_buffer_in_gamma_block M c8*
and *le_anchor_c8*: ∀ *kk*<*k_tm M*. *mt_tape c8 kk 0* = *LE_block* (*le_tm M*)
by (*rule ae_simulates_forward_stage_general*
[*OF vM lu less.prems*(*4*) *cM_in_Q cM_neq_t cM_neq_r*
less.prems(*5*) *k_chunk_le_card chunk_trace*
end_or_halt_card less.prems(*6*) *le_neq_bl*
no_le_per_tape_card])

— Substep H: extend the substrate-reachability of *cM* to *cM_k*. Composes *reach_cM* (an *rtrancl* certificate) with *chunk_trace* (a *relpow* certificate of length *k_chunk*) by lifting the *relpow* to *rtrancl* via *relpow_imp_rtrancl*, then composing via *rtrancl_trans*. Needed because the IH recursion requires the substrate-reachability of the recursion's starting M-config (here, *cM_k*).

have *chunk_in_rtrancl*: (*cM*, *cM_k*) ∈ (*mttm_step* (*delta_tm M*))*
using *chunk_trace* **by** (*rule relpow_imp_rtrancl*)
have *reach_cM_k*: (*init_config_mttm M w*, *cM_k*)
∈ (*mttm_step* (*delta_tm M*))*
using *reach_cM chunk_in_rtrancl* **by** (*rule rtrancl_trans*)

— Substep I: apply *less.IH* for the remaining trace *cM_k* $\rightarrow(n - k_chunk)$ *cM_final*. Recursion is on $n - k_chunk < n$ (which holds because $k_chunk \geq 1$ and $n \geq 1$); the re-quantified hypotheses are *reach_cM_k* (substep H), *rest_after_chunk* (substep C), *less.prems*(*3*) (accept), *sim_c8*, *buf_gamma_c8*, *le_anchor_c8* (substep G). Yields the recursion-arm witness (*m_rec*, *c''*): an M'-path of length *m_rec* from *c8* ending in canonical halt, with *m_rec* bounded by $8 \cdot \lceil (n - k_chunk) \,/\, c \rceil$.

have *n_minus_lt_n*: $n - k_chunk < n$
using *k_chunk_pos Suc* **by** *linarith*
obtain *m_rec c''* **where**
bound_rec: $m_rec \leq 8 * ((n - k_chunk + card\ (UNIV :: 'c\ set) - 1)$
$div\ card\ (UNIV :: 'c\ set))$
and *chain_rec*: $(c8, c'') \in (mttm_step\ (alphabet_enlarge_delta\ M))^{\frown\frown} m_rec$
and *state_c''*: $mt_state\ c'' = (t_tm\ M, init_stage\ (le_tm\ M))$
using *less.IH*[*OF n_minus_lt_n reach_cM_k rest_after_chunk*
less.prems(*3*) *sim_c8 buf_gamma_c8 le_anchor_c8*]
by *blast*

— Substep J: compose the 8-step forward chunk *chain8* with the recursion-arm *chain_rec* via *relpow_add*: $rel^{\frown\frown} 8\ O\ rel^{\frown\frown} m_rec = rel^{\frown\frown}(8 + m_rec)$.

have *chain_compose*:
$(c', c'') \in (mttm_step\ (alphabet_enlarge_delta\ M))^{\frown\frown}(8 + m_rec)$
using *chain8 chain_rec* **by** (*auto simp*: *relpow_add*)

— Substep K: arithmetic bound $8 + m_rec \leq 8 \cdot \lceil n/c \rceil$. Case-split mirroring substep E's: - $k_chunk = n$: $n - k_chunk = 0$ so $m_rec = 0$ (IH base case yields zero-bound on a zero-length trace); and $n \geq 1$ gives $\lceil n/c \rceil \geq 1$, so the target $8 \leq 8 \cdot 1$ holds. - $k_chunk \neq n$: substep E forces $k_chunk = c$ (and $c < n$); the IH bound becomes $m_rec \leq 8 \cdot \lceil (n-1)/c \rceil$, and the identity $\lceil n/c \rceil = \lceil (n-1)/c \rceil + 1$ (which holds for $n \geq 1$, $c \geq 1$) gives the target with equality.

have *n_pos*: $0 < n$ **using** *Suc* **by** *simp*
have *bound_compose*:
$8 + m_rec$
$\leq 8 * ((n + card\ (UNIV :: 'c\ set) - 1)$
$div\ card\ (UNIV :: 'c\ set))$
proof (*cases* $k_chunk = n$)
case *True*
have *m_rec_zero*: $m_rec = 0$
proof −
have $n - k_chunk = 0$ **using** *True* **by** *simp*
hence $(n - k_chunk + card\ (UNIV :: 'c\ set) - 1)$
$div\ card\ (UNIV :: 'c\ set) = 0$
using *c_pos*[*unfolded c_def*] **by** *simp*
thus *?thesis* **using** *bound_rec* **by** *simp*
qed
have *one_le_ceil*:
$1 \leq (n + card\ (UNIV :: 'c\ set) - 1)$
$div\ card\ (UNIV :: 'c\ set)$
proof −
have $card\ (UNIV :: 'c\ set) \leq n + card\ (UNIV :: 'c\ set) - 1$
using *n_pos* **by** *simp*

```
    hence card (UNIV :: 'c set) div card (UNIV :: 'c set)
          ≤ (n + card (UNIV :: 'c set) − 1)
              div card (UNIV :: 'c set)
      using div_le_mono by blast
    thus ?thesis using c_pos[unfolded c_def] by simp
  qed
  show ?thesis using m_rec_zero one_le_ceil by simp
next
  case False
  have k_chunk_eq_c: k_chunk = c
  proof −
    have k_chunk < n using k_chunk_le_n False by linarith
    hence min n c < n unfolding k_chunk_def by simp
    hence c < n by linarith
    hence min n c = c by simp
    thus ?thesis unfolding k_chunk_def by simp
  qed
  have n_gt_c: c < n
    using False k_chunk_le_n k_chunk_eq_c by linarith
  have n_minus_eq: n − k_chunk + c − 1 = n − 1
    using k_chunk_eq_c n_gt_c by simp
  have rec_bound:
     m_rec ≤ 8 ∗ ((n − 1) div card (UNIV :: 'c set))
    using bound_rec n_minus_eq unfolding c_def by simp
  have ceil_step:
     (n + card (UNIV :: 'c set) − 1)
        div card (UNIV :: 'c set)
      = (n − 1) div card (UNIV :: 'c set) + 1
  proof −
    have c_nz: card (UNIV :: 'c set) ≠ 0
      using c_pos unfolding c_def by simp
    have sum_eq: n + card (UNIV :: 'c set) − 1
                = (n − 1) + card (UNIV :: 'c set)
      using n_pos by simp
    have div_step:
       ((n − 1) + card (UNIV :: 'c set))
          div card (UNIV :: 'c set)
        = (n − 1) div card (UNIV :: 'c set) + 1
      using div_add_self2[OF c_nz] .
    show ?thesis using sum_eq div_step by simp
  qed
  have 8 + m_rec
        ≤ 8 + 8 ∗ ((n − 1) div card (UNIV :: 'c set))
    using rec_bound by simp
  also have ... = 8 ∗ ((n − 1) div card (UNIV :: 'c set) + 1)
    by simp
  also have ... = 8 ∗ ((n + card (UNIV :: 'c set) − 1)
                    div card (UNIV :: 'c set))
    using ceil_step by simp
```

finally show *?thesis* **.**
qed
— Final assembly: witnesses (*8 + m_rec*, *c''*) satisfy the bound (*8 + m_rec*) $\leq 8 \cdot \lceil n/c \rceil$, the chain (*c'*, *c''*) $\in$ *alphabet_enlarge_delta*$^{\frown\frown}$(*8 + m_rec*), and the canonical halt state (*t_tm M*, *init_stage le_M*).
show *?thesis*
using *bound_compose chain_compose state_c''* **by** *blast*
qed
qed

Simulation-phase step count under weak acceptance. If M accepts u within time T (*length u*) (an accepting M-path of length at most T (*length u*) from *init_config_mttm M u* to a config in state *t_tm M*), then from any simulation-paired SS1 config c' there is an accepting M'-path of length at most $8 * \lceil T(\textit{length } u) \ / \ c \rceil$ ending at (*t_tm M*, *init_stage* (*le_tm M*)).

Under the weak time-bounded acceptance convention (*accepts_in_time_mttm*). Hypothesis *accepts_in_time_mttm M u* (*T* (*length u*)) replaces the universal-path-bound *upperb_time_mttm M T*; conclusion drops the *r_tm*-arm (under weak acceptance, "reject" just means "no accepting path"; no canonical *r_tm* config is tracked explicitly). Thin wrapper around *ae_simulation_phase_chunked*: unpacks the weak-acceptance witness, lifts the tight bound (*n0 + c − 1*) *div c* to the *T* (*length u*) bound via monotonicity of division, and adapts to the *obtains*-form.

lemma *ae_simulation_phase_step_count*:
fixes *M* :: ('*q*, '*a*) *mttm*
and *T* :: *nat* ⇒ *nat*
and *u* :: '*a list*
and *c'* :: ('*c* :: *enum* ⇒ '*a*,
'*q* × ('*a*, '*c*) *ae_stage*) *mt_config*
assumes *vM*: *valid_mttm M*
and *lu*: *le_unique M*
and *m_accepts*: *accepts_in_time_mttm M u* (*T* (*length u*))
and *u_sub*: *set u* ⊆ *Sigma_tm M*
and *sim*: *ae_simulates M* (*init_config_mttm M u*) *c'*
and *buf_gamma_c'*: *ae_buffer_in_gamma_block M c'*
and *le_anchor_c'*: ∀ *kk*<*k_tm M*. *mt_tape c' kk 0* = *LE_block* (*le_tm M*)
and *s_neq_t*: *s_tm M* ≠ *t_tm M*
and *s_neq_r*: *s_tm M* ≠ *r_tm M*
and *le_neq_bl*: *le_tm M* ≠ *bl_tm M*
obtains *n_steps* :: *nat* **and** *c''* **where**
n_steps ≤ *8* * ((*T* (*length u*) + *card* (*UNIV* :: '*c set*) − *1*)
div card (*UNIV* :: '*c set*))
and (*c'*, *c''*) ∈ *mttm_step* (*alphabet_enlarge_delta M*) ^^ *n_steps*
and *mt_state c''* = (*t_tm M*, *init_stage* (*le_tm M*))
proof −
— Step 1: unpack the weak-acceptance witness the specific accepting M-path of length *n0* ≤ *T* (*length u*) from *init_config_mttm M u* to a config in state *t_tm M*.

```
  from m_accepts obtain n0 cM_n0 where
      n0_bd:  n0 ≤ T (length u)
    and n0_run: (init_config_mttm M u, cM_n0)
                  ∈ mttm_step (delta_tm M) ^^ n0
    and n0_acc: mt_state cM_n0 = t_tm M
    unfolding accepts_in_time_mttm_def by blast
```

— Step 2: invoke the chunked-induction helper. Reachability of the starting M-config (*init_config_mttm M u*) from itself is reflexive. The helper returns an M'-witness path with the tight bound $8 \cdot \lceil n0/c \rceil$; the wrapper relaxes this to $8 \cdot \lceil T(|u|)/c \rceil$ via monotonicity of $(_ + c - 1)\ div\ c$ in the dividend.

```
  have init_reach:
      (init_config_mttm M u, init_config_mttm M u)
        ∈ (mttm_step (delta_tm M))*
    by simp
  obtain n_steps c'' where
      tight_bound: n_steps ≤ 8 * ((n0 + card (UNIV :: 'c set) - 1)
                                    div card (UNIV :: 'c set))
    and run:   (c', c'') ∈ mttm_step (alphabet_enlarge_delta M) ^^ n_steps
    and halt:  mt_state c'' = (t_tm M, init_stage (le_tm M))
    using ae_simulation_phase_chunked[OF vM lu u_sub le_neq_bl s_neq_t
s_neq_r
                                        init_reach n0_run n0_acc sim
                                        buf_gamma_c' le_anchor_c']
    by blast
```

— Step 3: arithmetic relax the tight $n0$-bound to the loose $T(|u|)$-bound using $n0 \le T\ (length\ u)$ and monotonicity of *div* on the dividend.

```
  have div_mono:
      (n0 + card (UNIV :: 'c set) - 1) div card (UNIV :: 'c set)
        ≤ (T (length u) + card (UNIV :: 'c set) - 1)
            div card (UNIV :: 'c set)
    using n0_bd by (intro div_le_mono add_le_mono) auto
  have loose_bound:
      n_steps ≤ 8 * ((T (length u) + card (UNIV :: 'c set) - 1)
                       div card (UNIV :: 'c set))
    using tight_bound div_mono by linarith
  show ?thesis
  proof (rule that)
    show n_steps ≤ 8 * ((T (length u) + card (UNIV :: 'c set) - 1)
                          div card (UNIV :: 'c set))
      using loose_bound .
    show (c', c'') ∈ mttm_step (alphabet_enlarge_delta M) ^^ n_steps
      using run .
    show mt_state c'' = (t_tm M, init_stage (le_tm M))
      using halt .
  qed
qed

end
theory AlphabetEnlargement
```

imports *AlphabetEnlargement_Acceptance*
begin

The three top-level theorems characterising *alphabet_enlarge*, at the end of the forward-simulation chain that begins in *AlphabetEnlargement_ComputeCorrect*: the linear time bound *alphabet_enlarge_time*, well-formedness preservation *alphabet_enlarge_wf*, and forward language preservation *alphabet_enlarge_language_forward*. The reverse direction of the language biconditional (*alphabet_enlarge_language*) and the nondeterministic corollary live in *AlphabetEnlargement_Reverse*.

2.28 Top-level theorems

Time bound (Form 2 / encoded form): $M' = \mathit{alphabet_enlarge}\ M$ on input $\mathit{encode_input}\ (\mathit{bl_tm}\ M)\ w$ runs in time $\alpha \cdot \lceil n/c \rceil + 8 \cdot \lceil T(n)\ /\ c \rceil + f$ for structural additive constants α, f independent of M, where $c = \mathit{card}\ (\mathit{UNIV} :: {'c}\ \mathit{set})$ is the grouping factor and n is M's input length $\mathit{length}\ w$. M''s input is the consolidated block-encoding $\mathit{encode_input}\ (\mathit{bl_tm}\ M)\ w$ of length $\lceil n/c \rceil$.

This is the linear-speedup theorem in **encoded form** (Form 2): the input bijection *encode_input* is exposed externally; M''s job is to validate the encoded input's shape and simulate M. The classical same-alphabet statement (Form 1 — Hartmanis and Stearns [4, Theorem 2], modernised as Hopcroft and Ullman [6, Theorem 12.3]) follows as a corollary by composing this with a generic substrate-level wrap combinator that prepends an inline encoder pass.

Cost breakdown:

- $\alpha \cdot \lceil n/c \rceil$: validation-phase pass over the encoded input of length $\lceil n/c \rceil$ (forward scan + return scan; $\alpha = 2$).
- $8 \cdot \lceil T(n)\ /\ c \rceil$: 8 M'-substeps per simulated c-fold M-step group; $\lceil T(n)/c \rceil$ such groups suffice to cover M's $T(n)$-step accepting path.
- f: validation-phase setup additive (*f_v*).

  The $T(n)$ argument (not $T(c \cdot n)$) reflects that M's and M''s input represent the same problem instance of size n; M''s tape just compresses it by a factor of c.

  Weak acceptance shape. Hypothesis and conclusion both at *accepts_in_time_mttm* (the existential accepting-path predicate). The constants α, f depend only on M (and the type-level $'c$), not on w; the universal-w form inside *obtains* encodes this.

  Form 1 follows as a corollary by composing with the inline-encoder wrap combinator (*encoding_wrap* in *Wrap_Defs.thy*). Theorem 12.3's

setup-phase cost $n + \lceil n/m \rceil$ appears in the wrapped form as the inline encoder's cost ($O(n)$) plus this lemma's $\lceil n/c \rceil$ validation cost. Together with the speedup factor $1/c$ applied to $T(n)$, this gives the textbook bound $c_0 \cdot T(n)$ for any $c_0 > 0$ when $\mathit{inf}\ T(n)/n = \infty$ (Theorem 12.3); the companion [6, Theorem 12.4] patches the linear case $T(n) = \Theta(n)$ via a different choice of c.

```
theorem alphabet_enlarge_time_explicit:
  fixes M :: ('q, 'a) mttm
    and T :: nat ⇒ nat
  assumes wf:       well_formed_mttm M
  shows ∀ w. set w ⊆ Sigma_tm M
              ⟶ accepts_in_time_mttm M w (T (length w))
              ⟶ accepts_in_time_mttm
                  (alphabet_enlarge M
                     :: ('q × ('a, ('c :: enum)) ae_stage,
                         'c ⇒ 'a) mttm)
                  (encode_input (bl_tm M) w)
                  (2 * ((length w + card (UNIV :: 'c set) − 1)
                          div card (UNIV :: 'c set))
                   + 8 * ((T (length w) + card (UNIV :: 'c set) − 1)
                            div card (UNIV :: 'c set))
                   + 4)
proof −
  from wf have vM:       valid_mttm M
         and lu:       le_unique M
         and s_neq_t:   s_tm M ≠ t_tm M
         and s_neq_r:   s_tm M ≠ r_tm M
         and le_neq_bl: le_tm M ≠ bl_tm M
    by auto

  let ?c = card (UNIV :: 'c set)
  let ?M' = alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm

    show ∀ w. set w ⊆ Sigma_tm M
              ⟶ accepts_in_time_mttm M w (T (length w))
              ⟶ accepts_in_time_mttm ?M' (encode_input (bl_tm M) w)
                  (2 * ((length w + ?c − 1) div ?c)
                   + 8 * ((T (length w) + ?c − 1) div ?c)
                   + 4)
    proof (intro allI impI)
      fix w :: 'a list
      assume w_sub: set w ⊆ Sigma_tm M
      assume m_acc: accepts_in_time_mttm M w (T (length w))

      — Step 1: encoded-input setup  gamma-block containment, well-formedness,
and s_tm M ∈ Q_tm M.
      let ?ew = encode_input (bl_tm M) w
```

```
        :: ('c ⇒ 'a) list
  have ew_sub: set ?ew ⊆ gamma_block (Sigma_tm M ∪ {bl_tm M})
    by (rule encode_input_in_gamma_block[OF w_sub])
  have ew_wf: ae_input_well_formed (bl_tm M) ?ew
    by (rule encode_input_well_formed[OF vM w_sub])
  have s_in_Q: s_tm M ∈ Q_tm M
    by (rule s_tm_in_Q_tm[OF vM])
```

— Step 2: validation chain exact length $2 \cdot |?ew| + 4$, landing at the canonical SS1 configuration with the tape unchanged from *ae_init_config M ?ew*.

```
  let ?c1 = Config_M (s_tm M, init_offset,
                      init_buffer (le_tm M), init_dest, SS1)
                     (mt_tape (ae_init_config M ?ew))
                     (λ_ :: nat. 0)
            :: ('c ⇒ 'a,
                'q × ('a, 'c) ae_stage) mt_config
  have val_chain:
     (ae_init_config M ?ew, ?c1)
       ∈ mttm_step (alphabet_enlarge_delta M)
           ^^ (2 * length ?ew + 4)
    by (rule ae_validation_well_formed_to_SS1
          [OF vM ew_sub ew_wf s_in_Q s_neq_t s_neq_r le_neq_bl])
```

— Step 3: post-validation invariants on *?c1*.

(a) *le_anchor*: tape position 0 is the LE-block. *mt_tape ?c1 = mt_tape (ae_init_config M ?ew)* by construction, and *ae_init_config_tape_le* gives the LE-anchor.

(b) *buf_gamma*: every component of the buffer *init_buffer (le_tm M) = (LE_block le, LE_block le, LE_block le)* is in *gamma_block (Γ_tm M)*; follows from *le_tm M ∈ Γ_tm M* via *LE_block_in_gamma_block*.

(c) *ae_simulates*: SS1-branch with the canonical start state. All five conjuncts derived directly from the start-config shape and the tape-side invariants on *ae_init_config*.

```
  have le_anchor: ∀ kk<k_tm M. mt_tape ?c1 kk 0 = LE_block (le_tm M)
  proof (intro allI impI)
    fix kk assume kk_lt: kk < k_tm M
    have mt_tape ?c1 kk 0 = mt_tape (ae_init_config M ?ew) kk 0
      by simp
    also have ... = LE_block (le_tm M)
      by (rule ae_init_config_tape_le[OF kk_lt])
    finally show mt_tape ?c1 kk 0 = LE_block (le_tm M) .
  qed

  have buf_gamma: ae_buffer_in_gamma_block M ?c1
    unfolding ae_buffer_in_gamma_block_def init_buffer_def
      using LE_block_in_gamma_block[OF valid_mttm_LE_in_Gamma[OF
vM]]
    by simp
```

```
have tape_corr:
   ∀k<k_tm M. ae_tape_correspondence (le_tm M)
         (mt_tape (init_config_mttm M w) k)
         (mt_tape (ae_init_config M ?ew) k)
  by (rule ae_tape_correspondence_init[OF vM w_sub])
have subst_pos_zero:
   ∀k. mt_pos (init_config_mttm M w) k = 0
  by (cases M) simp
have c1_pos_zero: ∀k. mt_pos ?c1 k = 0 by simp
have pos_corr_c1:
   ∀k. mt_pos (init_config_mttm M w) k
         = ae_decode_pos (mt_pos ?c1 k)
                  ((init_offset :: nat ⇒ 'c) k)
  using c1_pos_zero subst_pos_zero
  unfolding ae_decode_pos_def by simp
have qM_subst: mt_state (init_config_mttm M w) = s_tm M
  by (cases M) simp
have c1_state:
   mt_state ?c1 = (s_tm M, init_offset,
                  init_buffer (le_tm M), init_dest, SS1)
  by simp
have gamma_block_c1: ae_tape_in_gamma_block M ?c1
  unfolding ae_tape_in_gamma_block_def
proof (intro conjI)
  show ∀k p. mt_tape ?c1 k p ∈ gamma_block (Γ_tm M)
    using ae_init_config_in_gamma_block[OF vM ew_sub] by simp
  show ∀j≥k_tm M. ∀p. mt_tape ?c1 j p = bl_block (bl_tm M)
    by (simp add: ae_init_config_tape_blank_tail)
qed
have sim: ae_simulates M (init_config_mttm M w) ?c1
  unfolding ae_simulates_def Let_def c1_state
  using qM_subst tape_corr pos_corr_c1 gamma_block_c1
       s_neq_t s_neq_r by simp
```

— Step 4: simulation-phase step count via *ae_simulation_phase_step_count*: yields an M'-chain from *?c1* to the canonical halt configuration of length at most $8 \cdot \lceil T(|w|) \,/\, c \rceil$.

```
obtain n_sim c'' where
   n_sim_bd:
      n_sim ≤ 8 * ((T (length w) + ?c - 1) div ?c)
  and sim_chain:
     (?c1, c'') ∈ mttm_step (alphabet_enlarge_delta M) ^^ n_sim
  and c''_state: mt_state c'' = (t_tm M, init_stage (le_tm M))
  using ae_simulation_phase_step_count
        [where M = M and T = T and u = w and c' = ?c1,
         OF vM lu m_acc w_sub sim buf_gamma le_anchor
            s_neq_t s_neq_r le_neq_bl]
  by metis
```

```
    — Step 5: compose validation and simulation chains.
    have chain_compose:
      (ae_init_config M ?ew, c'')
        ∈ mttm_step (alphabet_enlarge_delta M)
            ^^ ((2 * length ?ew + 4) + n_sim)
      using val_chain sim_chain by (auto simp: relpow_add)

    — Step 6: arithmetic bound. length ?ew = ⌈|w|/c⌉ by length_encode_input;
n_sim already bounded by 8 · ⌈T(|w|)/c⌉.
    have len_ew: length ?ew = (length w + ?c − 1) div ?c
      by (rule length_encode_input)
    let ?total = (2 * length ?ew + 4) + n_sim
    let ?target = 2 * ((length w + ?c − 1) div ?c)
                + 8 * ((T (length w) + ?c − 1) div ?c)
                + 4
    have bound: ?total ≤ ?target
      using n_sim_bd len_ew by simp

        — Step 7: bridge to substrate-level expressions and conclude accepts_in_time_mttm.
    have init_bridge:
      init_config_mttm ?M' ?ew = ae_init_config M ?ew
      by (rule init_config_alphabet_enlarge)
    have t_bridge: t_tm ?M' = (t_tm M, init_stage (le_tm M))
      by (rule t_tm_alphabet_enlarge)
    have delta_bridge:
      delta_tm ?M' = alphabet_enlarge_delta M
      by (rule delta_tm_alphabet_enlarge)

    show accepts_in_time_mttm ?M' ?ew ?target
      unfolding accepts_in_time_mttm_def
    proof (intro exI conjI)
      show ?total ≤ ?target using bound .
      show (init_config_mttm ?M' ?ew, c'')
          ∈ (mttm_step (delta_tm ?M')) ^^ ?total
        using chain_compose init_bridge delta_bridge by simp
      show mt_state c'' = t_tm ?M'
        using c''_state t_bridge by simp
    qed
  qed
qed
```

The classical HU-form of the linear-speedup time bound: the additive constants α, f instantiated at $\alpha = 2$, $f = 4$ from *alphabet_enlarge_time_explicit*. The per-block simulation constant 8 and the speedup divisor *card* (*UNIV* :: 'c *set*) are already explicit in the statement.

```
theorem alphabet_enlarge_time:
  fixes M :: ('q, 'a) mttm
    and T :: nat ⇒ nat
```

```
  assumes wf: well_formed_mttm M
  obtains α f :: nat
    where ∀ w. set w ⊆ Sigma_tm M
              ⟶ accepts_in_time_mttm M w (T (length w))
              ⟶ accepts_in_time_mttm
                  (alphabet_enlarge M
                    :: ('q × ('a, ('c :: enum)) ae_stage,
                        'c ⇒ 'a) mttm)
                  (encode_input (bl_tm M) w)
                  (α * ((length w + card (UNIV :: 'c set) - 1)
                         div card (UNIV :: 'c set))
                   + 8 * ((T (length w) + card (UNIV :: 'c set) - 1)
                          div card (UNIV :: 'c set))
                   + f)
proof (rule that[of 2 4])
  show ∀ w. set w ⊆ Sigma_tm M
              ⟶ accepts_in_time_mttm M w (T (length w))
              ⟶ accepts_in_time_mttm
                  (alphabet_enlarge M
                    :: ('q × ('a, ('c :: enum)) ae_stage,
                        'c ⇒ 'a) mttm)
                  (encode_input (bl_tm M) w)
                  (2 * ((length w + card (UNIV :: 'c set) - 1)
                         div card (UNIV :: 'c set))
                   + 8 * ((T (length w) + card (UNIV :: 'c set) - 1)
                          div card (UNIV :: 'c set))
                   + 4)
    by (rule alphabet_enlarge_time_explicit[OF wf])
qed
```

Forward language inclusion modulo input encoding: a word w with *set* $w \subseteq \Sigma_M$ that is in M's language has its canonical block-encoding (using M's blank for padding) in $M' =$ *alphabet_enlarge* M's language. This is the forward leg of the language-equivalence claim; the reverse leg (*encode_input* $w \in$ *Lang_mttm* $M' \Longrightarrow w \in$ *Lang_mttm* M) holds for every well-formed M — with no determinism hypothesis, the original *det_mttm* M dependency having been removed in refactoring — and lives in *AlphabetEnlargement_Reverse.thy* as the biconditional *alphabet_enlarge_language*.

Hypotheses align with *alphabet_enlarge_time*: $s \neq t$, $s \neq r$, $le \neq bl$. The first two would be redundant if we manually handled the degenerate always-accept ($s = t$) and always-reject ($s = r$) cases, but matching the *_time* signature keeps the call sites uniform. $le \neq bl$ is genuinely necessary: the simulation infrastructure requires it (compute substep's buffer- write composition), and the *Sigma_tm* containment for encoded inputs uses it to exclude *LE_block*.

The *set* $w \subseteq$ *Sigma_tm* M antecedent inside the $\forall\, w$ is required: without it the inclusion fails for w containing blank symbols (such w are outside *Lang_mttm* M by the substrate's *Lang_mttm* definition, but their encodings

can still pass M'-validation and be M'-accepted).

Strategy: extract an accepting M-path of length *n0*; instantiate *alphabet_enlarge_time* with $T = (\lambda_.\ n0)$ to obtain a bounded M'-witness; drop the bound and conclude.

theorem *alphabet_enlarge_language_forward*:
  fixes $M :: ('q, 'a)\ mttm$
  assumes *wf*: *well_formed_mttm M*
  shows $\forall w.\ set\ w \subseteq Sigma_tm\ M$
    $\longrightarrow w \in Lang_mttm\ M$
    $\longrightarrow encode_input\ (bl_tm\ M)\ w \in Lang_mttm$
      $(alphabet_enlarge\ M$
        $:: ('q \times ('a, ('c :: enum))\ ae_stage,$
          $'c \Rightarrow 'a)\ mttm)$
proof −
  from *wf* **have** *vM*: *valid_mttm M*
    and *lu*: *le_unique M*
    and *s_neq_t*: $s_tm\ M \neq t_tm\ M$
    and *s_neq_r*: $s_tm\ M \neq r_tm\ M$
    and *le_neq_bl*: $le_tm\ M \neq bl_tm\ M$
   by *auto*

  let $?c = card\ (UNIV :: 'c\ set)$
  let $?M' = alphabet_enlarge\ M$
      $:: ('q \times ('a, 'c)\ ae_stage, 'c \Rightarrow 'a)\ mttm$

  show *?thesis*
  proof (*intro allI impI*)
   fix $w :: 'a\ list$
   assume *w_sub*: $set\ w \subseteq Sigma_tm\ M$
   assume *w_in_M*: $w \in Lang_mttm\ M$

— Set-containment for the encoded input under *alphabet_enlarge M*'s Σ: gamma-block membership minus the two excluded markers.
   have *enc_in_gamma*:
     $set\ (encode_input\ (bl_tm\ M)\ w :: ('c \Rightarrow 'a)\ list)$
       $\subseteq gamma_block\ (Sigma_tm\ M \cup \{bl_tm\ M\})$
    by (*rule encode_input_in_gamma_block*[*OF w_sub*])
   have *enc_no_bl*:
     $bl_block\ (bl_tm\ M)$
       $\notin set\ (encode_input\ (bl_tm\ M)\ w :: ('c \Rightarrow 'a)\ list)$
    by (*rule encode_input_no_bl_block*[*OF vM w_sub*])
   have *enc_no_LE*:
     $LE_block\ (le_tm\ M)$
       $\notin set\ (encode_input\ (bl_tm\ M)\ w :: ('c \Rightarrow 'a)\ list)$
    by (*rule encode_input_no_LE_block*[*OF vM w_sub*])
   have *enc_sub*:
     $set\ (encode_input\ (bl_tm\ M)\ w :: ('c \Rightarrow 'a)\ list)$
       $\subseteq Sigma_tm\ ?M'$
    unfolding *Sigma_tm_alphabet_enlarge*

using *enc_in_gamma enc_no_bl enc_no_LE* **by** *blast*

— Strategy: extract a specific accepting M-path of length *n0*; instantiate *alphabet_enlarge_time* with $T = (\lambda_.\ n0)$ to obtain a bounded M'-witness; drop the bound and conclude.

from *w_in_M* **obtain** *wM' nM* **where**
m_path: (*init_config_mttm M w*,
Config$_M$ (*t_tm M*) *wM' nM*)
∈ (*mttm_step* (*delta_tm M*))*
unfolding *Lang_mttm_def* **by** *blast*
obtain *n0* **where** *m_pow*:
(*init_config_mttm M w*, *Config*$_M$ (*t_tm M*) *wM' nM*)
∈ (*mttm_step* (*delta_tm M*)) ^^ *n0*
using *m_path rtrancl_imp_relpow* **by** *metis*
have *m_acc*: *accepts_in_time_mttm M w n0*
unfolding *accepts_in_time_mttm_def*
proof (*intro exI conjI*)
show (*n0* :: *nat*) ≤ *n0* **by** *simp*
show (*init_config_mttm M w*, *Config*$_M$ (*t_tm M*) *wM' nM*)
∈ *mttm_step* (*delta_tm M*) ^^ *n0*
using *m_pow* **.**
show *mt_state* (*Config*$_M$ (*t_tm M*) *wM' nM*) = *t_tm M*
by *simp*
qed

obtain α *f* :: *nat* **where** *AE_time*:
∀ *w_arg*. *set w_arg* ⊆ *Sigma_tm M*
⟶ *accepts_in_time_mttm M w_arg n0*
⟶ *accepts_in_time_mttm ?M'*
(*encode_input* (*bl_tm M*) *w_arg*)
(α ∗ ((*length w_arg* + *?c* − *1*) *div ?c*)
+ *8* ∗ ((*n0* + *?c* − *1*) *div ?c*) + *f*)
by (*rule alphabet_enlarge_time*
[**where** *T* = $\lambda_.$ *n0*, *OF wf*])
from *AE_time w_sub m_acc* **have** *m'_acc*:
accepts_in_time_mttm ?M' (*encode_input* (*bl_tm M*) *w*)
(α ∗ ((*length w* + *?c* − *1*) *div ?c*)
+ *8* ∗ ((*n0* + *?c* − *1*) *div ?c*) + *f*)
by *blast*
from *m'_acc* **obtain** *n_m' c_acc* **where**
run: (*init_config_mttm ?M'* (*encode_input* (*bl_tm M*) *w*),
c_acc)
∈ (*mttm_step* (*delta_tm ?M'*)) ^^ *n_m'*
and *acc*: *mt_state c_acc* = *t_tm ?M'*
unfolding *accepts_in_time_mttm_def* **by** *blast*
obtain *wM_acc' nM_acc* **where** *c_acc_eq*:
c_acc = *Config*$_M$ (*t_tm ?M'*) *wM_acc' nM_acc*
using *acc* **by** (*cases c_acc*) *simp*
have *run_star*:

```
      (init_config_mttm ?M' (encode_input (bl_tm M) w),
        Config_M (t_tm ?M') wM_acc' nM_acc)
          ∈ (mttm_step (delta_tm ?M'))*
    using run c_acc_eq relpow_imp_rtrancl by metis
  show encode_input (bl_tm M) w ∈ Lang_mttm ?M'
    unfolding Lang_mttm_def
    using enc_sub run_star by blast
 qed
qed

end
theory AlphabetEnlargement_Reverse
 imports AlphabetEnlargement
begin
```

This theory holds the reverse-arm machinery for the *alphabet_enlarge* simulation chain, together with the biconditional language theorem *alphabet_enlarge_language* — which closes via that machinery for every well-formed M, with *no determinism hypothesis* — and the nondeterministic corollary *alphabet_enlarge_nae*.

Contents, in dependency order:

- Three reverse-arm lifts of the buffered compute: *m_step_buffered_lift_to_tape_general*, *m_step_buffered_chain_lift_to_tape*, *m_steps_buffered_lift_to_tape*.
- *ae_step_ss4_ss5_destruct_to_buffered*: reverse-arm SS4→SS5 step destructor.
- The four reverse-arm chain peel lemmas *ae_backward_stage_load_chain*, *ae_backward_stage_compute_extract*, *ae_backward_stage_writeback_chain*, and *ae_backward_stage_decompose*, and the reverse-arm top-level wrapper *ae_backward_stage*.
- A chain-uniqueness lemma for *alphabet_enlarge_delta* derived from the 16 per-substep functionality lemmas in *AlphabetEnlargement_Delta*.
- The biconditional language theorem *alphabet_enlarge_language*, for every well-formed M (no determinism hypothesis). The forward direction delegates to *alphabet_enlarge_language_forward* (in *AlphabetEnlargement*); the reverse direction is proved here determinism-free, via the validation-phase uniqueness and the window-inversion kernel.
- The nondeterministic corollary *alphabet_enlarge_nae*: the biconditional and the time bound packaged for an arbitrary, possibly nondeterministic, well-formed M.

The reverse direction does not require *det_mttm M*. The natural route would close by chain uniqueness of *alphabet_enlarge_delta*, which a nondeterministic M does not supply (its compute substep is multi-valued); the determinism-bound step is instead discharged by the unconditional window-inversion kernel (the *det*-free SS5→SS8 subsection below). The biconditional and the corollary *alphabet_enlarge_nae* therefore hold for nondeterministic M as well.

2.29 Reverse direction: buffered → tape lift

2.29.1 Single-step lift to per-tape

Per-tape regime-dispatching single-step lift. Takes the per-tape unified *ae_window_invariant_general* (with the regime selector *pos_n* :: *nat* ⇒ *nat* that can differ across tapes) and produces a substrate-side *mttm_step* with the per-tape unified invariant preserved on the post-step config.

The proof factors out the regime-independent structure (*cM* unpack, δ-tuple extraction, per-tape read-match via *read_bp_via_window_general*, *mttm_step.step* packaging) from the regime-specific window-preservation step, which dispatches per tape on *pos_n k* into *ae_window_invariant_step* / *_le1_step* / *_le0_step*.

The hypothesis bundle for the regime-specific side conditions is the same per-regime form as the forward direction's *ae_m_steps_buffered_correct_trace_general* uses, so Step 2b can pass these through directly.

lemma *m_step_buffered_lift_to_tape_general*:
 fixes *M* :: (*'q*, *'a*) *mttm*
  and *cM* :: (*'a*, *'q*) *mt_config*
  and *q q'* :: *'q*
  and *blocks blocks'* ::
   nat ⇒ ((*'c* :: *enum* ⇒ *'a*) × (*'c* ⇒ *'a*) × (*'c* ⇒ *'a*))
  and *pos_bp pos_bp'* :: *nat* ⇒ *'c bp*
  and *pos_n* :: *nat* ⇒ *nat*
 assumes *vM*: *valid_mttm M*
  and *lu*: *le_unique M*
  and *cM_valid*: *valid_config_mttm M cM*
  and *step*: ((*q*, *blocks*, *pos_bp*), (*q'*, *blocks'*, *pos_bp'*))
   ∈ *m_step_buffered M*
  and *st_eq*: *mt_state cM* = *q*
  and *window*: ∀ *k*<*k_tm M*. *ae_window_invariant_general*
   (*mt_tape cM k*) (*mt_pos cM k*)
   (*pos_bp k*) (*blocks k*) (*pos_n k*)
   (*le_tm M*)
  and *bp_lt*: ∀ *k*. *bp_linear* (*pos_bp k*)
   < *3* ∗ *card* (*UNIV* :: *'c set*) − *1*
  and *bp_pos*: ∀ *k*. *0* < *bp_linear* (*pos_bp k*)

```
    and no_le:    ∀k<k_tm M. (pos_n k = 0
                      ⟶ (∀i. 0 < i
                              ∧ i ≤ card (UNIV :: 'c set)
                             ⟶ mt_tape cM k i ≠ le_tm M))
                   ∧ (pos_n k = 1
                      ⟶ (∀i. 0 < i
                              ∧ i ≤ 2 * card (UNIV :: 'c set)
                             ⟶ mt_tape cM k i ≠ le_tm M))
                   ∧ (pos_n k ≥ 2
                      ⟶ mt_tape cM k (mt_pos cM k) ≠ le_tm M)
  shows ∃cM'.
          (cM, cM') ∈ mttm_step (delta_tm M)
          ∧ mt_state cM' = q'
          ∧ (∀k<k_tm M. ae_window_invariant_general
                  (mt_tape cM' k) (mt_pos cM' k)
                  (pos_bp' k) (blocks' k) (pos_n k) (le_tm M))
proof −
  obtain ts_pre n_pre where
     cM_eq: cM = Config_M q ts_pre n_pre
  proof −
    obtain q0 ts0 n0 where cM_form: cM = Config_M q0 ts0 n0
      by (cases cM) auto
    from st_eq cM_form have q0 = q by simp
    with cM_form show thesis using that by simp
  qed
  have tp_eq: mt_tape cM = ts_pre using cM_eq by simp
  have ps_eq: mt_pos cM = n_pre   using cM_eq by simp
  from step obtain a a' d where
     delta_mem: (q, a, q', a', d) ∈ delta_tm M
    and read_eq:  ∀k. a k = read_bp (blocks k) (pos_bp k)
    and write_eq: ∀k. blocks' k = write_bp (blocks k) (pos_bp k) (a' k)
    and adv_eq:   ∀k. bp_advance_le (le_tm M) (a k) (pos_bp k) (d k)
                    = Some (pos_bp' k)
    unfolding m_step_buffered_def by auto
  have read_match: ∀k. a k = ts_pre k (n_pre k)
  proof (intro allI)
    fix k
    show a k = ts_pre k (n_pre k)
    proof (cases k < k_tm M)
      case True
      from window True have wi_k:
          ae_window_invariant_general (mt_tape cM k) (mt_pos cM k)
            (pos_bp k) (blocks k) (pos_n k) (le_tm M) by blast
      hence wi_k': ae_window_invariant_general (ts_pre k) (n_pre k)
                     (pos_bp k) (blocks k) (pos_n k) (le_tm M)
        using tp_eq ps_eq by simp
      have read_bp (blocks k) (pos_bp k) = ts_pre k (n_pre k)
        by (rule read_bp_via_window_general[OF wi_k'])
      thus a k = ts_pre k (n_pre k)
```

using *read_eq* **by** *simp*
next
case *False*
hence *kge*: $k \geq k_tm\ M$ **by** *simp*
— Padding tape: no window invariant available; the read-match holds because both sides are the substrate blank $bl_tm\ M$ — $a\ k$ by δ-support, the actual tape read by cM's blank-tail.
have *a_bl*: $a\ k = bl_tm\ M$
using *valid_mttm_delta_support*[*OF vM delta_mem kge*] **by** *simp*
have *ts_bl*: $ts_pre\ k\ (n_pre\ k) = bl_tm\ M$
using *valid_config_mttm_blank_tail*[*OF cM_valid kge*] *tp_eq* **by** *simp*
show $a\ k = ts_pre\ k\ (n_pre\ k)$ **using** *a_bl ts_bl* **by** *simp*
qed
qed
have *read_match_fun*: $(\lambda k.\ ts_pre\ k\ (n_pre\ k)) = a$
using *read_match* **by** (*auto intro*!: *ext*)
let *?cM'* = $Config_M\ q'\ (\lambda k.\ (ts_pre\ k)(n_pre\ k := a'\ k))$
$(\lambda k.\ go_dir\ (d\ k)\ (n_pre\ k))$
have *delta_tuple*: $(q,\ \lambda k.\ ts_pre\ k\ (n_pre\ k),\ q',\ a',\ d)$
$\in delta_tm\ M$
using *delta_mem read_match_fun* **by** *simp*
have *step_mttm*: $(cM,\ ?cM') \in mttm_step\ (delta_tm\ M)$
proof −
have $(Config_M\ q\ ts_pre\ n_pre,\ ?cM') \in mttm_step\ (delta_tm\ M)$
by (*rule mttm_step.step*
[**where** *ts*=*ts_pre* **and** *n*=*n_pre* **and** *a*=a' **and** *dir*=*d*,
OF delta_tuple])
thus *?thesis* **using** *cM_eq* **by** *simp*
qed
have *st_new*: $mt_state\ ?cM' = q'$ **by** *simp*
have *new_window*:
$\forall k{<}k_tm\ M.\ ae_window_invariant_general$
$(mt_tape\ ?cM'\ k)\ (mt_pos\ ?cM'\ k)$
$(pos_bp'\ k)\ (blocks'\ k)\ (pos_n\ k)\ (le_tm\ M)$
proof (*intro allI impI*)
fix k **assume** *klt*: $k < k_tm\ M$
let *?c* = *card* (*UNIV* :: *'c set*)
have *tape_k*: $mt_tape\ ?cM'\ k = (ts_pre\ k)(n_pre\ k := a'\ k)$ **by** *simp*
have *pos_k*: $mt_pos\ ?cM'\ k = go_dir\ (d\ k)\ (n_pre\ k)$ **by** *simp*
have *blocks'_k*: $blocks'\ k = write_bp\ (blocks\ k)\ (pos_bp\ k)\ (a'\ k)$
using *write_eq* **by** *blast*
have *adv_k_buf*: $bp_advance_le\ (le_tm\ M)\ (a\ k)\ (pos_bp\ k)\ (d\ k)$
$= Some\ (pos_bp'\ k)$
using *adv_eq* **by** *blast*
have *read_a_k*: $a\ k = ts_pre\ k\ (n_pre\ k)$ **using** *read_match* **by** *blast*
have *adv_k*: $bp_advance_le\ (le_tm\ M)\ (ts_pre\ k\ (n_pre\ k))$
$(pos_bp\ k)\ (d\ k) = Some\ (pos_bp'\ k)$
using *adv_k_buf read_a_k* **by** *simp*
have *wi_gen_k*: *ae_window_invariant_general*

```
    (ts_pre k) (n_pre k) (pos_bp k) (blocks k) (pos_n k) (le_tm M)
   using window klt tp_eq ps_eq by simp
  have bp_lt_k: bp_linear (pos_bp k) < 3 * ?c - 1 using bp_lt by blast
  have bp_pos_k: 0 < bp_linear (pos_bp k) using bp_pos by blast
  — LE-discipline side conditions, derived from valid_mttm M; used in le0 and
le1 branches.
  have a_le_k: ts_pre k (n_pre k) = le_tm M
               ⟹ a' k = le_tm M ∧ d k ∈ {dir.N, dir.R}
  proof −
   assume LE: ts_pre k (n_pre k) = le_tm M
   have a_LE: a k = le_tm M using read_a_k LE by simp
   show a' k = le_tm M ∧ d k ∈ {dir.N, dir.R}
    by (rule valid_mttm_deltaLE[OF vM delta_mem a_LE])
  qed
  have a_not_le_k: ts_pre k (n_pre k) ≠ le_tm M ⟹ a' k ≠ le_tm M
  proof
   assume nLE: ts_pre k (n_pre k) ≠ le_tm M and LE': a' k = le_tm M
   have a_LE: a k = le_tm M
    by (rule valid_mttm_deltaLE_no_write[OF lu delta_mem LE'])
   from a_LE read_a_k nLE show False by simp
  qed
  — Dispatch on pos_n k regime.
  consider (steady) pos_n k ≥ 2 | (le1) pos_n k = 1 | (le0) pos_n k = 0
   by linarith
  then show ae_window_invariant_general
            (mt_tape ?cM' k) (mt_pos ?cM' k)
            (pos_bp' k) (blocks' k) (pos_n k) (le_tm M)
  proof cases
   case steady
   have wi_st: ae_window_invariant (ts_pre k) (n_pre k)
                (pos_bp k) (blocks k) ((pos_n k − 2) * ?c + 1)
    using wi_gen_k steady
    unfolding ae_window_invariant_general_def by simp
   have no_le_k: ts_pre k (n_pre k) ≠ le_tm M
    using no_le klt steady tp_eq ps_eq by force
   have wi_post: ae_window_invariant
                  ((ts_pre k)(n_pre k := a' k))
                  (go_dir (d k) (n_pre k))
                  (pos_bp' k)
                  (write_bp (blocks k) (pos_bp k) (a' k))
                  ((pos_n k − 2) * ?c + 1)
    by (rule ae_window_invariant_step
          [OF wi_st adv_k bp_lt_k bp_pos_k no_le_k])
   have wi_post_gen:
     ae_window_invariant_general
       ((ts_pre k)(n_pre k := a' k))
       (go_dir (d k) (n_pre k))
       (pos_bp' k)
       (write_bp (blocks k) (pos_bp k) (a' k))
```

```
        (pos_n k) (le_tm M)
    using wi_post steady
    unfolding ae_window_invariant_general_def by simp
  show ?thesis using wi_post_gen tape_k pos_k blocks'_k by simp
next
  case le1
  have wi_l1: ae_window_invariant_le1 (ts_pre k) (n_pre k)
                (pos_bp k) (blocks k) (le_tm M)
    using wi_gen_k le1
    unfolding ae_window_invariant_general_def by simp
  have no_le_win_k: ∀ i. 0 < i ∧ i ≤ 2 * ?c
                          ⟶ ts_pre k i ≠ le_tm M
    using no_le klt le1 tp_eq by simp
  have wi_post: ae_window_invariant_le1
                  ((ts_pre k)(n_pre k := a' k))
                  (go_dir (d k) (n_pre k))
                  (pos_bp' k)
                  (write_bp (blocks k) (pos_bp k) (a' k))
                  (le_tm M)
    by (rule ae_window_invariant_le1_step
          [OF wi_l1 adv_k bp_lt_k bp_pos_k
              a_le_k a_not_le_k no_le_win_k])
  have wi_post_gen:
     ae_window_invariant_general
        ((ts_pre k)(n_pre k := a' k))
        (go_dir (d k) (n_pre k))
        (pos_bp' k)
        (write_bp (blocks k) (pos_bp k) (a' k))
        (pos_n k) (le_tm M)
    using wi_post le1
    unfolding ae_window_invariant_general_def by simp
  show ?thesis using wi_post_gen tape_k pos_k blocks'_k by simp
next
  case le0
  have wi_l0: ae_window_invariant_le0 (ts_pre k) (n_pre k)
                (pos_bp k) (blocks k) (le_tm M)
    using wi_gen_k le0
    unfolding ae_window_invariant_general_def by simp
  have no_le_win_k: ∀ i. 0 < i ∧ i ≤ ?c
                          ⟶ ts_pre k i ≠ le_tm M
    using no_le klt le0 tp_eq by simp
  have wi_post: ae_window_invariant_le0
                  ((ts_pre k)(n_pre k := a' k))
                  (go_dir (d k) (n_pre k))
                  (pos_bp' k)
                  (write_bp (blocks k) (pos_bp k) (a' k))
                  (le_tm M)
    by (rule ae_window_invariant_le0_step
          [OF wi_l0 adv_k bp_lt_k bp_pos_k
```

```
            a_le_k a_not_le_k no_le_win_k])
    have wi_post_gen:
      ae_window_invariant_general
        ((ts_pre k)(n_pre k := a' k))
        (go_dir (d k) (n_pre k))
        (pos_bp' k)
        (write_bp (blocks k) (pos_bp k) (a' k))
        (pos_n k) (le_tm M)
      using wi_post le0
      unfolding ae_window_invariant_general_def by simp
    show ?thesis using wi_post_gen tape_k pos_k blocks'_k by simp
  qed
 qed
 show ?thesis using step_mttm st_new new_window by blast
qed
```

Auxiliary induction for the reverse lift: given an n-step *m_step_buffered* chain (with $n \le c$) starting from a canonical SS4-entry buffered state and the per-tape *ae_window_invariant_general* + side conditions on a substrate-side initial config *cM*, produces an n-step substrate *mttm_step* trace from *cM* to some *cM_n* with matching state, preserved window invariant, preserved no-LE windows, preserved *left_not_le_pos0*, and the *bp_linear*-bound conjunct ($c \le$ *bp_linear* $+ n$, *bp_linear* $<$ *2c* $+ n$).

Mirror of the forward induction *ae_coupled_run_aux_general* with trace $\longleftrightarrow$ chain flipped. The single-step lift *m_step_buffered_lift_to_tape_general* does the per-step substrate-side construction; this lemma threads the invariants across the chain.

Used inside the smaller wrapper *m_steps_buffered_lift_to_tape*, which exposes the result via *m_steps_buffered*.

```
lemma m_step_buffered_chain_lift_to_tape:
 fixes M :: ('q, 'a) mttm
   and qM :: 'q
   and ofs :: nat ⇒ 'c :: enum
   and buf_full ::
         nat ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
   and pos :: nat ⇒ nat
   and n :: nat
   and qM_n :: 'q
   and buf_n ::
         nat ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
   and end_pos_n :: nat ⇒ 'c bp
   and cM :: ('a, 'q) mt_config
 assumes vM:    valid_mttm M
     and lu:    le_unique M
     and cM_valid: valid_config_mttm M cM
     and qM_in: qM ∈ Q_tm M
     and chain: ((qM, buf_full, λk. (AE_Home, ofs k)),
                 (qM_n, buf_n, end_pos_n))
```

```
                ∈ (m_step_buffered M) ^^ n
    and st_eq: mt_state cM = qM
    and window:
        ∀ k<k_tm M. ae_window_invariant_general
              (mt_tape cM k) (mt_pos cM k)
              (AE_Home, ofs k) (buf_full k) (pos k) (le_tm M)
    and n_bound: n ≤ card (UNIV :: 'c set)
    and no_le:
        ∀ k. (pos k = 0
              ⟶ (∀ i. i < card (UNIV :: 'c set)
                    ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
            ∧ (pos k = 1
                ⟶ (∀ i. i < 2 * card (UNIV :: 'c set)
                      ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
            ∧ (pos k ≥ 2
                ⟶ (∀ i. i < 3 * card (UNIV :: 'c set)
                      ⟶ mt_tape cM k
                          ((pos k − 2) * card (UNIV :: 'c set)
                            + 1 + i)
                        ≠ le_tm M))
    and left_not_le_pos0:
        ∀ k<k_tm M. pos k = 0 ⟶ fst (buf_full k) ≠ LE_block (le_tm M)
  shows ∃ cM_n.
        (cM, cM_n) ∈ mttm_step (delta_tm M) ^^ n
        ∧ mt_state cM_n = qM_n
        ∧ (∀ k<k_tm M. ae_window_invariant_general
              (mt_tape cM_n k) (mt_pos cM_n k)
              (end_pos_n k) (buf_n k) (pos k) (le_tm M))
        ∧ (∀ k. (pos k = 0
                ⟶ (∀ i. i < card (UNIV :: 'c set)
                      ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M))
              ∧ (pos k = 1
                  ⟶ (∀ i. i < 2 * card (UNIV :: 'c set)
                        ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M))
              ∧ (pos k ≥ 2
                  ⟶ (∀ i. i < 3 * card (UNIV :: 'c set)
                        ⟶ mt_tape cM_n k
                            ((pos k − 2) * card (UNIV :: 'c set)
                              + 1 + i)
                          ≠ le_tm M)))
        ∧ (∀ k<k_tm M. pos k = 0
                ⟶ fst (buf_n k) ≠ LE_block (le_tm M))
        ∧ (∀ k. card (UNIV :: 'c set) ≤ bp_linear (end_pos_n k) + n
              ∧ bp_linear (end_pos_n k)
                  < 2 * card (UNIV :: 'c set) + n)
  using chain n_bound
proof (induction n arbitrary: qM_n buf_n end_pos_n)
  case 0
  — Base case: zero buffered steps means the chain endpoint (qM_n, buf_n,
```

end_pos_n) equals the start (*qM*, *buf_full*, *AE_Home offsets*); witness *cM_n* = *cM* and the substrate trace of length 0 is trivial. All six output conjuncts lift directly from the lemma's input hypotheses.

```
  from 0.prems(1) have eq:
     (qM_n, buf_n, end_pos_n) = (qM, buf_full, λk. (AE_Home, ofs k))
    by simp
  have qM_n_eq:     qM_n = qM              using eq by simp
  have buf_n_eq:    buf_n = buf_full       using eq by simp
  have end_pos_eq:  end_pos_n = (λk. (AE_Home, ofs k)) using eq by simp
  show ?case
  proof (intro exI [where x = cM] conjI)
    show (cM, cM) ∈ mttm_step (delta_tm M) ^^ 0 by simp
    show mt_state cM = qM_n using st_eq qM_n_eq by simp
    show ∀k<k_tm M. ae_window_invariant_general
                (mt_tape cM k) (mt_pos cM k)
                (end_pos_n k) (buf_n k) (pos k) (le_tm M)
      using window buf_n_eq end_pos_eq by simp
    show ∀k. (pos k = 0
              ⟶ (∀i. i < card (UNIV :: 'c set)
                      ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
           ∧ (pos k = 1
              ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
                      ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
           ∧ (pos k ≥ 2
              ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
                      ⟶ mt_tape cM k
                          ((pos k − 2) * card (UNIV :: 'c set)
                            + 1 + i)
                        ≠ le_tm M))
      using no_le by simp
    show ∀k<k_tm M. pos k = 0 ⟶ fst (buf_n k) ≠ LE_block (le_tm M)
      using left_not_le_pos0 buf_n_eq by simp
    show ∀k. card (UNIV :: 'c set) ≤ bp_linear (end_pos_n k) + 0
             ∧ bp_linear (end_pos_n k) < 2 * card (UNIV :: 'c set) + 0
    proof (intro allI conjI)
      fix k
      have lin_eq: bp_linear (end_pos_n k)
                     = card (UNIV :: 'c set) + c_idx (ofs k)
        using end_pos_eq unfolding bp_linear_def by simp
      have idx_lt: c_idx (ofs k) < card (UNIV :: 'c set)
        by (rule c_idx_lt_card)
      show card (UNIV :: 'c set) ≤ bp_linear (end_pos_n k) + 0
        using lin_eq by simp
      show bp_linear (end_pos_n k) < 2 * card (UNIV :: 'c set) + 0
        using lin_eq idx_lt by simp
    qed
  qed
next
  case (Suc n')
```

— Inductive step. Peel the LAST *m_step_buffered* step via *relpow_Suc_E* to an intermediate (*qM_n'*, *buf_n'*, *end_pos_n'*). Apply the IH at n' to obtain an intermediate substrate config *cM_n'* with *mt_state cM_n'* = *qM_n'*, the window invariant, and the four threaded conjuncts. Use Step 2a's *m_step_buffered_lift_to_tape_general* to extract one substrate *mttm_step* from *cM_n'* to a fresh *cM_n*; compose via *relpow_Suc_I*. Re-derive the four threaded conjuncts at length *Suc n'*.

```
  have n'_bound: n' ≤ card (UNIV :: 'c set)
    using Suc.prems(2) by linarith
  from Suc.prems(1) obtain s_n' where
      chain_n': ((qM, buf_full, λk. (AE_Home, ofs k)), s_n')
                  ∈ (m_step_buffered M) ^^ n'
    and last_step_raw: (s_n', (qM_n, buf_n, end_pos_n))
                         ∈ m_step_buffered M
    by (rule relpow_Suc_E)
  obtain qM_n' buf_n' end_pos_n' where
      s_n'_eq: s_n' = (qM_n', buf_n', end_pos_n')
    by (cases s_n') auto
  have chain_n'_unp:
      ((qM, buf_full, λk. (AE_Home, ofs k)),
        (qM_n', buf_n', end_pos_n')) ∈ (m_step_buffered M) ^^ n'
    using chain_n' s_n'_eq by simp
  have last_step:
      ((qM_n', buf_n', end_pos_n'), (qM_n, buf_n, end_pos_n))
        ∈ m_step_buffered M
    using last_step_raw s_n'_eq by simp
— Invoke the IH at length n'.
  from Suc.IH[OF chain_n'_unp n'_bound]
  obtain cM_n' where
      trace_n': (cM, cM_n') ∈ mttm_step (delta_tm M) ^^ n'
    and st_n':  mt_state cM_n' = qM_n'
    and window_n':
        ∀k<k_tm M. ae_window_invariant_general
              (mt_tape cM_n' k) (mt_pos cM_n' k)
              (end_pos_n' k) (buf_n' k) (pos k) (le_tm M)
    and no_le_n':
        ∀k. (pos k = 0
              ⟶ (∀i. i < card (UNIV :: 'c set)
                    ⟶ mt_tape cM_n' k (Suc i) ≠ le_tm M))
            ∧ (pos k = 1
                ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
                      ⟶ mt_tape cM_n' k (Suc i) ≠ le_tm M))
            ∧ (pos k ≥ 2
                ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
                      ⟶ mt_tape cM_n' k
                          ((pos k − 2) * card (UNIV :: 'c set)
                            + 1 + i)
                        ≠ le_tm M))
    and left_n':
```

```
    ∀k<k_tm M. pos k = 0
        ⟶ fst (buf_n' k) ≠ LE_block (le_tm M)
  and bp_bound_n':
    ∀k. card (UNIV :: 'c set) ≤ bp_linear (end_pos_n' k) + n'
      ∧ bp_linear (end_pos_n' k)
          < 2 * card (UNIV :: 'c set) + n'
  by blast
```

— Validity of the intermediate config cM_n', threaded along the IH's n'-step trace; supplies the leaf lift's blank-tail read-match on padding tapes.

```
  have cM_n'_valid: valid_config_mttm M cM_n'
    by (rule valid_reach_relpow_mttm[OF vM cM_valid trace_n'])
```

— Per-tape strict bounds at n': $n' \le c - 1$ gives $0 < bp_linear < 3c - 1$.

```
  have c_pos: 1 ≤ card (UNIV :: 'c set)
    using Suc.prems(2) by linarith
  have bp_lt_n':
     ∀k. bp_linear (end_pos_n' k) < 3 * card (UNIV :: 'c set) - 1
  proof (intro allI)
    fix k
    have lin_lt: bp_linear (end_pos_n' k) < 2 * card (UNIV :: 'c set) + n'
      using bp_bound_n' by blast
    show bp_linear (end_pos_n' k) < 3 * card (UNIV :: 'c set) - 1
      using lin_lt Suc.prems(2) c_pos by linarith
  qed
  have bp_pos_n':
     ∀k. 0 < bp_linear (end_pos_n' k)
  proof (intro allI)
    fix k
    have low: card (UNIV :: 'c set) ≤ bp_linear (end_pos_n' k) + n'
      using bp_bound_n' by blast
    show 0 < bp_linear (end_pos_n' k)
      using low Suc.prems(2) c_pos by linarith
  qed
```

— Per-tape no-LE-at-head at cM_n', regime-dispatched (steady: read at head is not LE; le0/le1: window-wide no-LE on $[1, c]$ or $[1, 2c]$). Derived from the IH's no_le_n' conjunct. The steady-regime case requires one extra step: $mt_pos\ cM_n'\ k = (pos\ k - 2)c + 1 + bp_linear\ (end_pos_n'\ k)$ (from the window invariant), and $bp_linear < 3c$ places the head inside the no-LE window.

— Per-tape no-LE-at-head at cM_n', regime-dispatched, derived from the IH's no_le_n' conjunct. Split into three regime sublemmas, combined via *blast*.

```
  have head_le0:
     ∀k. pos k = 0
         ⟶ (∀i. 0 < i ∧ i ≤ card (UNIV :: 'c set)
                 ⟶ mt_tape cM_n' k i ≠ le_tm M)
  proof (intro allI impI allI impI)
    fix k i
    assume pos_0: pos k = 0
      and i_bnd: 0 < i ∧ i ≤ card (UNIV :: 'c set)
    then obtain j where i_eq: i = Suc j
                    and j_lt: j < card (UNIV :: 'c set)
```

```
    by (metis Suc_le_eq Suc_pred')
  from no_le_n' pos_0 j_lt
  have mt_tape cM_n' k (Suc j) ≠ le_tm M by blast
  thus mt_tape cM_n' k i ≠ le_tm M using i_eq by simp
qed
have head_le1:
    ∀k. pos k = 1
          ⟶ (∀i. 0 < i ∧ i ≤ 2 * card (UNIV :: 'c set)
                    ⟶ mt_tape cM_n' k i ≠ le_tm M)
proof (intro allI impI allI impI)
  fix k i
  assume pos_1: pos k = 1
     and i_bnd: 0 < i ∧ i ≤ 2 * card (UNIV :: 'c set)
  then obtain j where i_eq: i = Suc j
                  and j_lt: j < 2 * card (UNIV :: 'c set)
    by (metis Suc_le_eq Suc_pred')
  from no_le_n' pos_1 j_lt
  have mt_tape cM_n' k (Suc j) ≠ le_tm M by blast
  thus mt_tape cM_n' k i ≠ le_tm M using i_eq by simp
qed
have head_steady:
    ∀k<k_tm M. pos k ≥ 2
          ⟶ mt_tape cM_n' k (mt_pos cM_n' k) ≠ le_tm M
proof (intro allI impI impI)
  fix k
  assume klt: k < k_tm M and pos_ge: pos k ≥ 2
  let ?c = card (UNIV :: 'c set)
  have wi_k: ae_window_invariant_general
                (mt_tape cM_n' k) (mt_pos cM_n' k)
                (end_pos_n' k) (buf_n' k) (pos k) (le_tm M)
    using window_n' klt by blast
  have wi_steady: ae_window_invariant
                    (mt_tape cM_n' k) (mt_pos cM_n' k)
                    (end_pos_n' k) (buf_n' k)
                    ((pos k - 2) * ?c + 1)
    using wi_k pos_ge
    unfolding ae_window_invariant_general_def by simp
  have head_eq: mt_pos cM_n' k
                  = (pos k - 2) * ?c + 1
                    + bp_linear (end_pos_n' k)
    using wi_steady unfolding ae_window_invariant_def by simp
  have lin_lt_3c: bp_linear (end_pos_n' k) < 3 * ?c
  proof -
    have step: bp_linear (end_pos_n' k) < 3 * ?c - 1
      using bp_lt_n' by blast
    show ?thesis using step c_pos by linarith
  qed
  from no_le_n' pos_ge lin_lt_3c
  have mt_tape cM_n' k
```

```
        ((pos k − 2) ∗ ?c + 1 + bp_linear (end_pos_n′ k))
       ≠ le_tm M by blast
  thus mt_tape cM_n′ k (mt_pos cM_n′ k) ≠ le_tm M
    using head_eq by simp
qed
have no_le_step2a:
   ∀ k<k_tm M. (pos k = 0
        ⟶ (∀ i. 0 < i ∧ i ≤ card (UNIV :: 'c set)
                ⟶ mt_tape cM_n′ k i ≠ le_tm M))
      ∧ (pos k = 1
         ⟶ (∀ i. 0 < i ∧ i ≤ 2 ∗ card (UNIV :: 'c set)
                 ⟶ mt_tape cM_n′ k i ≠ le_tm M))
      ∧ (pos k ≥ 2
         ⟶ mt_tape cM_n′ k (mt_pos cM_n′ k) ≠ le_tm M)
  using head_le0 head_le1 head_steady by blast
— Apply Step 2a (single-step lift) to extract the substrate step from last_step.
obtain cM_n where
    step_extracted:
     (cM_n′, cM_n) ∈ mttm_step (delta_tm M)
      ∧ mt_state cM_n = qM_n
      ∧ (∀ k<k_tm M. ae_window_invariant_general
              (mt_tape cM_n k) (mt_pos cM_n k)
              (end_pos_n k) (buf_n k) (pos k) (le_tm M))
  using m_step_buffered_lift_to_tape_general
      [OF vM lu cM_n′_valid last_step st_n′ window_n′ bp_lt_n′ bp_pos_n′
          no_le_step2a] by blast
have step_one: (cM_n′, cM_n) ∈ mttm_step (delta_tm M)
  using step_extracted by simp
have st_n_eq:  mt_state cM_n = qM_n
  using step_extracted by simp
have window_n:
   ∀ k<k_tm M. ae_window_invariant_general
         (mt_tape cM_n k) (mt_pos cM_n k)
         (end_pos_n k) (buf_n k) (pos k) (le_tm M)
  using step_extracted by simp
have trace_n:
   (cM, cM_n) ∈ mttm_step (delta_tm M) ^^ (Suc n′)
  using trace_n′ step_one by (rule relpow_Suc_I)
— Re-derive the cM_n / cM_n′ relation in mttm_step-shape: mttm_step.cases
on step_one unpacks the delta tuple and the tape update.
obtain ts_pre n_pre q_post a_step dir_step where
    cM_n′_eq: cM_n′ = Config_M qM_n′ ts_pre n_pre
  and cM_n_eq:
     cM_n = Config_M q_post
            (λk. (ts_pre k)(n_pre k := a_step k))
            (λk. go_dir (dir_step k) (n_pre k))
  and delta_mem:
     (qM_n′, λk. ts_pre k (n_pre k),
        q_post, a_step, dir_step) ∈ delta_tm M
```

proof −
from *step_one* **obtain** *q_pre′ ts_pre′ n_pre′ q_post′ a_step′ dir_step′*
where *step_unp*:
cM_n′ = *Config*$_M$ *q_pre′ ts_pre′ n_pre′*
cM_n = *Config*$_M$ *q_post′*
(λ*k*. (*ts_pre′ k*)(*n_pre′ k* := *a_step′ k*))
(λ*k*. *go_dir* (*dir_step′ k*) (*n_pre′ k*))
(*q_pre′*, λ*k*. *ts_pre′ k* (*n_pre′ k*),
q_post′, *a_step′*, *dir_step′*) ∈ *delta_tm M*
by (*rule mttm_step.cases*)
have *q_pre′_eq*: *q_pre′* = *qM_n′*
using *step_unp*(*1*) *st_n′* **by** *simp*
show *thesis* **using** *step_unp q_pre′_eq that* **by** *simp*
qed
have *tp_n′*: *mt_tape cM_n′* = *ts_pre* **using** *cM_n′_eq* **by** *simp*
have *ps_n′*: *mt_pos cM_n′* = *n_pre* **using** *cM_n′_eq* **by** *simp*
have *tp_n*: ∀ *k*. *mt_tape cM_n k*
= (*ts_pre k*)(*n_pre k* := *a_step k*)
using *cM_n_eq* **by** *auto*
have *q_post_eq*: *q_post* = *qM_n*
using *cM_n_eq st_n_eq* **by** *simp*
— Conjunct 3 threading: new *no_le* at *cM_n*. Mirrors the forward direction. Three regime branches; for each cell in the window, case-split on whether the M-step's update site *n_pre k* coincides; if it does, use the LE-write contrapositive *valid_mttm_deltaLE_no_write*.
have *no_le_contra*:
∀ *k*. *ts_pre k* (*n_pre k*) ≠ *le_tm M*
⟶ *a_step k* ≠ *le_tm M*
proof (*intro allI impI*)
fix *k*
assume *rd_not_le*: *ts_pre k* (*n_pre k*) ≠ *le_tm M*
show *a_step k* ≠ *le_tm M*
proof
assume *a_le*: *a_step k* = *le_tm M*
have (λ*k*. *ts_pre k* (*n_pre k*)) *k* = *le_tm M*
by (*rule valid_mttm_deltaLE_no_write*[*OF lu delta_mem a_le*])
hence *ts_pre k* (*n_pre k*) = *le_tm M* **by** *simp*
with *rd_not_le* **show** *False* **..**
qed
qed
have *no_le0_n*:
∀ *k*. *pos k* = *0*
⟶ (∀ *i*. *i* < *card* (*UNIV* :: *′c set*)
⟶ *mt_tape cM_n k* (*Suc i*) ≠ *le_tm M*)
proof (*intro allI impI allI impI*)
fix *k i*
assume *pos_0*: *pos k* = *0*
and *i_bd*: *i* < *card* (*UNIV* :: *′c set*)
have *ih_at*: *ts_pre k* (*Suc i*) ≠ *le_tm M*

```
  proof −
    from no_le_n' pos_0 i_bd
    have mt_tape cM_n' k (Suc i) ≠ le_tm M by blast
    thus ?thesis using tp_n' by simp
  qed
  show mt_tape cM_n k (Suc i) ≠ le_tm M
  proof (cases Suc i = n_pre k)
    case False
    hence mt_tape cM_n k (Suc i) = ts_pre k (Suc i)
      using tp_n by simp
    thus ?thesis using ih_at by simp
  next
    case True
    hence tape_eq: mt_tape cM_n k (Suc i) = a_step k
      using tp_n by simp
    from True have ts_pre k (n_pre k) = ts_pre k (Suc i)
      by simp
    with ih_at have ts_pre k (n_pre k) ≠ le_tm M by simp
    hence a_step k ≠ le_tm M using no_le_contra by blast
    thus ?thesis using tape_eq by simp
  qed
qed
have no_le1_n:
   ∀ k. pos k = 1
         ⟶ (∀ i. i < 2 * card (UNIV :: 'c set)
                ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M)
proof (intro allI impI allI impI)
  fix k i
  assume pos_1: pos k = 1
    and i_bd: i < 2 * card (UNIV :: 'c set)
  have ih_at: ts_pre k (Suc i) ≠ le_tm M
  proof −
    from no_le_n' pos_1 i_bd
    have mt_tape cM_n' k (Suc i) ≠ le_tm M by blast
    thus ?thesis using tp_n' by simp
  qed
  show mt_tape cM_n k (Suc i) ≠ le_tm M
  proof (cases Suc i = n_pre k)
    case False
    hence mt_tape cM_n k (Suc i) = ts_pre k (Suc i)
      using tp_n by simp
    thus ?thesis using ih_at by simp
  next
    case True
    hence tape_eq: mt_tape cM_n k (Suc i) = a_step k
      using tp_n by simp
    from True have ts_pre k (n_pre k) = ts_pre k (Suc i)
      by simp
    with ih_at have ts_pre k (n_pre k) ≠ le_tm M by simp
```

```
    hence a_step k ≠ le_tm M using no_le_contra by blast
    thus ?thesis using tape_eq by simp
  qed
qed
have no_le2_n:
    ∀ k. pos k ≥ 2
          ⟶ (∀ i. i < 3 * card (UNIV :: 'c set)
                ⟶ mt_tape cM_n k
                    ((pos k − 2) * card (UNIV :: 'c set) + 1 + i)
                  ≠ le_tm M)
proof (intro allI impI allI impI)
  fix k i
  assume pos_ge: pos k ≥ 2
     and i_bd: i < 3 * card (UNIV :: 'c set)
  let ?j = (pos k − 2) * card (UNIV :: 'c set) + 1 + i
  have ih_at: ts_pre k ?j ≠ le_tm M
  proof −
    from no_le_n' pos_ge i_bd
    have mt_tape cM_n' k ?j ≠ le_tm M by blast
    thus ?thesis using tp_n' by simp
  qed
  show mt_tape cM_n k ?j ≠ le_tm M
  proof (cases ?j = n_pre k)
    case False
    hence mt_tape cM_n k ?j = ts_pre k ?j
      using tp_n by simp
    thus ?thesis using ih_at by simp
  next
    case True
    hence tape_eq: mt_tape cM_n k ?j = a_step k
      using tp_n by simp
    from True have ts_pre k (n_pre k) = ts_pre k ?j
      by simp
    with ih_at have ts_pre k (n_pre k) ≠ le_tm M by simp
    hence a_step k ≠ le_tm M using no_le_contra by blast
    thus ?thesis using tape_eq by simp
  qed
qed
have no_le_n:
    ∀ k. (pos k = 0
          ⟶ (∀ i. i < card (UNIV :: 'c set)
                ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M))
        ∧ (pos k = 1
            ⟶ (∀ i. i < 2 * card (UNIV :: 'c set)
                  ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M))
        ∧ (pos k ≥ 2
            ⟶ (∀ i. i < 3 * card (UNIV :: 'c set)
                  ⟶ mt_tape cM_n k
                      ((pos k − 2) * card (UNIV :: 'c set)
```

```
                                + 1 + i)
                              ≠ le_tm M))
    using no_le0_n no_le1_n no_le2_n by blast
```

— Conjunct 4 threading: *left_not_le_pos0* at *buf_n*. The buffered write at *end_pos_n′ k* changes *buf_n′ k* at the *AE_Left* / *AE_Home* / *AE_Right* block depending on *fst* (*end_pos_n′ k*). For *pos k = 0*, the regime invariant *ae_window_invariant_le0* forces the buffered head to be in the Home block (so the Left block is unchanged), giving *fst* (*buf_n k*) = *fst* (*buf_n′ k*); apply the IH (*left_n′*).

```
  from last_step obtain a_buf a'_buf d_buf where
      delta_buf: (qM_n', a_buf, qM_n, a'_buf, d_buf) ∈ delta_tm M
    and read_buf: ∀k. a_buf k = read_bp (buf_n' k) (end_pos_n' k)
    and write_buf: ∀k. buf_n k
                         = write_bp (buf_n' k) (end_pos_n' k) (a'_buf k)
    and adv_buf: ∀k. bp_advance_le (le_tm M) (a_buf k)
                           (end_pos_n' k) (d_buf k)
                       = Some (end_pos_n k)
    unfolding m_step_buffered_def by auto
  have left_n:
      ∀k<k_tm M. pos k = 0 ⟶ fst (buf_n k) ≠ LE_block (le_tm M)
  proof (intro allI impI impI)
    fix k
    assume klt: k < k_tm M and pos_0: pos k = 0
    have wi_k: ae_window_invariant_general
                 (mt_tape cM_n' k) (mt_pos cM_n' k)
                 (end_pos_n' k) (buf_n' k) (pos k) (le_tm M)
      using window_n' klt by blast
    have wi_l0: ae_window_invariant_le0
                 (mt_tape cM_n' k) (mt_pos cM_n' k)
                 (end_pos_n' k) (buf_n' k) (le_tm M)
      using wi_k pos_0
      unfolding ae_window_invariant_general_def by simp
    have head_home_or_right:
        fst (end_pos_n' k) ∈ {AE_Home, AE_Right}
    proof (cases fst (end_pos_n' k))
      case AE_Left
      have False
        using wi_l0 AE_Left
        unfolding ae_window_invariant_le0_def by simp
      thus ?thesis ..
    qed auto
    obtain l h r where buf_n'_eq:
        buf_n' k = (l, h, r) by (cases buf_n' k) auto
    obtain b off where end_eq:
        end_pos_n' k = (b, off) by (cases end_pos_n' k) auto
    from head_home_or_right end_eq have b_in: b ∈ {AE_Home, AE_Right}
by simp
    have fst_buf_n:
        fst (buf_n k) = fst (buf_n' k)
```

```
  proof −
    have wb_eq: buf_n k
                = write_bp (buf_n' k) (end_pos_n' k) (a'_buf k)
      using write_buf by blast
    consider (home) b = AE_Home | (right) b = AE_Right
      using b_in by blast
    thus ?thesis
    proof cases
      case home
      have write_bp (buf_n' k) (end_pos_n' k) (a'_buf k)
           = (l, h(off := a'_buf k), r)
        using buf_n'_eq end_eq home
        unfolding write_bp_def by simp
      thus ?thesis using wb_eq buf_n'_eq by simp
    next
      case right
      have write_bp (buf_n' k) (end_pos_n' k) (a'_buf k)
           = (l, h, r(off := a'_buf k))
        using buf_n'_eq end_eq right
        unfolding write_bp_def by simp
      thus ?thesis using wb_eq buf_n'_eq by simp
    qed
  qed
  show fst (buf_n k) ≠ LE_block (le_tm M)
    using fst_buf_n left_n' klt pos_0 by simp
qed
```

— Conjunct 5 threading: *bp_bound* at *end_pos_n*. One *m_step_buffered* step shifts *bp_linear* by at most 1 (R: +1, L: -1, N: 0), via *bp_advance_le*. The new bounds follow from the IH bounds and the advance equation. Helper: *bp_advance_le_lin_bounded* (in Setup).

```
have bp_bound_n:
  ∀k. card (UNIV :: 'c set) ≤ bp_linear (end_pos_n k) + Suc n'
      ∧ bp_linear (end_pos_n k)
          < 2 * card (UNIV :: 'c set) + Suc n'
proof (intro allI)
  fix k
  have low_n':
    card (UNIV :: 'c set) ≤ bp_linear (end_pos_n' k) + n'
    using bp_bound_n' by blast
  have high_n':
    bp_linear (end_pos_n' k) < 2 * card (UNIV :: 'c set) + n'
    using bp_bound_n' by blast
  have pos_n': 0 < bp_linear (end_pos_n' k)
    using bp_pos_n' by blast
  have adv_k: bp_advance_le (le_tm M) (a_buf k) (end_pos_n' k) (d_buf k)
              = Some (end_pos_n k)
    using adv_buf by blast
  show card (UNIV :: 'c set) ≤ bp_linear (end_pos_n k) + Suc n'
       ∧ bp_linear (end_pos_n k)
```

```
            < 2 * card (UNIV :: 'c set) + Suc n'
    by (rule bp_advance_le_lin_bounded
          [OF adv_k pos_n' high_n' low_n'])
  qed
  — Assemble the six output conjuncts.
  show ?case
  proof (intro exI [where x = cM_n] conjI)
    show (cM, cM_n) ∈ mttm_step (delta_tm M) ^^ Suc n'
      using trace_n .
    show mt_state cM_n = qM_n using st_n_eq .
    show ∀ k<k_tm M. ae_window_invariant_general
              (mt_tape cM_n k) (mt_pos cM_n k)
              (end_pos_n k) (buf_n k) (pos k) (le_tm M)
      using window_n .
    show ∀ k. (pos k = 0
              ⟶ (∀ i. i < card (UNIV :: 'c set)
                       ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M))
           ∧ (pos k = 1
              ⟶ (∀ i. i < 2 * card (UNIV :: 'c set)
                       ⟶ mt_tape cM_n k (Suc i) ≠ le_tm M))
           ∧ (pos k ≥ 2
              ⟶ (∀ i. i < 3 * card (UNIV :: 'c set)
                       ⟶ mt_tape cM_n k
                            ((pos k - 2) * card (UNIV :: 'c set)
                              + 1 + i)
                          ≠ le_tm M))
      using no_le_n .
    show ∀ k<k_tm M. pos k = 0
              ⟶ fst (buf_n k) ≠ LE_block (le_tm M)
      using left_n .
    show ∀ k. card (UNIV :: 'c set) ≤ bp_linear (end_pos_n k) + Suc n'
            ∧ bp_linear (end_pos_n k)
                < 2 * card (UNIV :: 'c set) + Suc n'
      using bp_bound_n .
  qed
qed
```

User-facing reverse lift on *m_steps_buffered*. Wraps Step 2b-aux (*m_step_buffered_chain_lift_to_tape*) by unfolding the existential n in *m_steps_buffered M*, applying the aux induction, and exposing the result via an *obtains*-form interface. Preserves the halt-exit disjunction ($n = c \vee$ *final state* $\in \{t, r\}$) on the substrate side.

Consumed by *ae_backward_stage* to extract the substrate M-side trace from an SS4→SS5 buffered chunk.

```
lemma m_steps_buffered_lift_to_tape:
  fixes M :: ('q, 'a) mttm
    and qM :: 'q
    and ofs :: nat ⇒ 'c :: enum
    and buf_full ::
```

$nat \Rightarrow (('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a))$
and $pos :: nat \Rightarrow nat$
and $qM_end :: 'q$
and $buf_end ::$
$nat \Rightarrow (('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a))$
and $end_pos_bp :: nat \Rightarrow 'c\ bp$
and $cM :: ('a, 'q)\ mt_config$
assumes vM: $valid_mttm\ M$
and lu: $le_unique\ M$
and cM_valid: $valid_config_mttm\ M\ cM$
and qM_in: $qM \in Q_tm\ M$
and $steps$: $((qM, buf_full, \lambda k.\ (AE_Home, ofs\ k)),$
$(qM_end, buf_end, end_pos_bp))$
$\in m_steps_buffered\ M$
and st_eq: $mt_state\ cM = qM$
and $window$:
$\forall k{<}k_tm\ M.\ ae_window_invariant_general$
$(mt_tape\ cM\ k)\ (mt_pos\ cM\ k)$
$(AE_Home, ofs\ k)\ (buf_full\ k)\ (pos\ k)\ (le_tm\ M)$
and no_le:
$\forall k.\ (pos\ k = 0$
$\longrightarrow (\forall i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow mt_tape\ cM\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k = 1$
$\longrightarrow (\forall i.\ i < 2 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt_tape\ cM\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k \geq 2$
$\longrightarrow (\forall i.\ i < 3 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt_tape\ cM\ k$
$((pos\ k - 2) * card\ (UNIV :: 'c\ set)$
$+ 1 + i)$
$\neq le_tm\ M))$
and $left_not_le_pos0$:
$\forall k{<}k_tm\ M.\ pos\ k = 0 \longrightarrow fst\ (buf_full\ k) \neq LE_block\ (le_tm\ M)$
obtains $cM_end\ n$ **where**
$(cM, cM_end) \in mttm_step\ (delta_tm\ M) \frown\frown n$
and $n \leq card\ (UNIV :: 'c\ set)$
and $n = card\ (UNIV :: 'c\ set) \vee mt_state\ cM_end \in \{t_tm\ M, r_tm\ M\}$
and $mt_state\ cM_end = qM_end$
and $\forall k{<}k_tm\ M.\ ae_window_invariant_general$
$(mt_tape\ cM_end\ k)\ (mt_pos\ cM_end\ k)$
$(end_pos_bp\ k)\ (buf_end\ k)\ (pos\ k)\ (le_tm\ M)$
and $\forall k.\ (pos\ k = 0$
$\longrightarrow (\forall i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow mt_tape\ cM_end\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k = 1$
$\longrightarrow (\forall i.\ i < 2 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt_tape\ cM_end\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k \geq 2$

$\longrightarrow (\forall\, i.\ i < 3 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt_tape\ cM_end\ k$
$((pos\ k - 2) * card\ (UNIV :: 'c\ set)$
$+ 1 + i)$
$\neq le_tm\ M))$
and $\forall\, k{<}k_tm\ M.\ pos\ k = 0$
$\longrightarrow fst\ (buf_end\ k) \neq LE_block\ (le_tm\ M)$
proof −
— Unfold *m_steps_buffered*'s existential.
from *steps* **obtain** *n* **where**
chain_n: $((qM,\ buf_full,\ \lambda k.\ (AE_Home,\ ofs\ k)),$
$(qM_end,\ buf_end,\ end_pos_bp))$
$\in (m_step_buffered\ M)\ ^{\frown\frown}\ n$
and *n_bnd*: $n \leq card\ (UNIV :: 'c\ set)$
and *halt_or_full*:
$n = card\ (UNIV :: 'c\ set) \vee qM_end \in \{t_tm\ M,\ r_tm\ M\}$
unfolding *m_steps_buffered_def* **by** *auto*
— Invoke Step 2b-aux to obtain the substrate trace plus the threaded invariants at *cM_n*.
from *m_step_buffered_chain_lift_to_tape*
[*OF vM lu cM_valid qM_in chain_n st_eq window n_bnd no_le left_not_le_pos0*]
obtain *cM_end* **where**
trace_n: $(cM,\ cM_end) \in mttm_step\ (delta_tm\ M)\ ^{\frown\frown}\ n$
and *st_eq_n*: $mt_state\ cM_end = qM_end$
and *window_n*:
$\forall\, k{<}k_tm\ M.\ ae_window_invariant_general$
$(mt_tape\ cM_end\ k)\ (mt_pos\ cM_end\ k)$
$(end_pos_bp\ k)\ (buf_end\ k)\ (pos\ k)\ (le_tm\ M)$
and *no_le_n*:
$\forall\, k.\ (pos\ k = 0$
$\longrightarrow (\forall\, i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow mt_tape\ cM_end\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k = 1$
$\longrightarrow (\forall\, i.\ i < 2 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt_tape\ cM_end\ k\ (Suc\ i) \neq le_tm\ M))$
$\wedge\ (pos\ k \geq 2$
$\longrightarrow (\forall\, i.\ i < 3 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt_tape\ cM_end\ k$
$((pos\ k - 2) * card\ (UNIV :: 'c\ set)$
$+ 1 + i)$
$\neq le_tm\ M))$
and *left_n*:
$\forall\, k{<}k_tm\ M.\ pos\ k = 0 \longrightarrow fst\ (buf_end\ k) \neq LE_block\ (le_tm\ M)$
by *blast*
have *halt_or_full_substrate*:
$n = card\ (UNIV :: 'c\ set) \vee mt_state\ cM_end \in \{t_tm\ M,\ r_tm\ M\}$
using *halt_or_full st_eq_n* **by** *simp*
show *?thesis*

by (*rule that*[*OF trace_n n_bnd halt_or_full_substrate*
st_eq_n window_n no_le_n left_n])
qed

2.29.2 SS4→SS5 backward destruct

Reverse-arm counterpart of *ae_step_ss4_ss5_construct_from_buffered*. Given an SS4→SS5 substrate step, destructure it to expose the SS4 source structure, the SS5 target structure, and the inner buffered chain. Consumed by *ae_backward_stage_compute_extract*.

No *ae_inv_ss4* hypothesis is required: the relation *ae_delta_ss4_ss5* itself pins the source state shape (*substep_idx* = SS4, $q \in Q_tm\ M$, $q \neq t_tm\ M$, $q \neq r_tm\ M$) and exposes the buffered chain. The destruct is therefore a purely structural elimination of *mttm_step.cases* followed by an unfold of *ae_delta_ss4_ss5_def*.

lemma *ae_step_ss4_ss5_destruct_to_buffered*:
fixes $M :: ('q, 'a)\ mttm$
and $c\ c' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae_stage)\ mt_config$
assumes *step*: $(c, c') \in mttm_step\ (ae_delta_ss4_ss5\ M)$
obtains *q ofs buf dest_old ts n q_out buf' bufC end_pos* **where**
$c = Config_M\ (q, ofs, buf, dest_old, SS4)\ ts\ n$
and $mt_state\ c' = (q_out,$
$\lambda k.\ if\ k < k_tm\ M\ then\ snd\ (end_pos\ k)$
$else\ init_offset\ k,$
$buf',$
$\lambda k.\ if\ k < k_tm\ M\ then\ fst\ (end_pos\ k)$
$else\ init_dest\ k, SS5)$
and $buf' = (\lambda k.\ if\ k < k_tm\ M\ then\ bufC\ k$
$else\ init_buffer\ (le_tm\ M)\ k)$
and $mt_tape\ c' = ts$
and $mt_pos\ c'$
$= (\lambda k.\ go_dir\ (if\ k < k_tm\ M$
$then\ (if\ ts\ k\ (n\ k) = LE_block\ (le_tm\ M)$
$then\ dir.N\ else\ dir.L)$
$else\ dir.N)$
$(n\ k))$
and $q \in Q_tm\ M$
and $q \neq t_tm\ M$
and $q \neq r_tm\ M$
and $\forall j{\geq}k_tm\ M.\ ts\ j\ (n\ j) = bl_block\ (bl_tm\ M)$
and $((q, \lambda k.\ (fst\ (buf\ k),$
$if\ k < k_tm\ M\ then\ fst\ (snd\ (buf\ k))\ else\ ts\ k\ (n\ k),$
$ts\ k\ (n\ k)),$
$\lambda k.\ (AE_Home, ofs\ k)),$
$(q_out, bufC, end_pos)) \in m_steps_buffered\ M$
proof −
from *step* **obtain** *s tsP nP s' aP dP* **where**

```
    cP_eq: c = Config_M s tsP nP
    and c'_eq: c' = Config_M s' (λk. (tsP k)(nP k := aP k))
                                (λk. go_dir (dP k) (nP k))
    and rel: (s, (λk. tsP k (nP k)), s', aP, dP)
             ∈ ae_delta_ss4_ss5 M
  by (auto elim: mttm_step.cases)
from rel obtain q ofs buf dest_old q_out ofs' buf' dest'
               buf_full end_pos bufC
  where s_eq: s = (q, ofs, buf, dest_old, SS4)
    and s'_eq: s' = (q_out, ofs', buf', dest', SS5)
    and a_eq: aP = (λk. tsP k (nP k))
    and q_in_Q: q ∈ Q_tm M
    and q_neq_t_M: q ≠ t_tm M
    and q_neq_r_M: q ≠ r_tm M
    and supp: ∀ j≥k_tm M. tsP j (nP j) = bl_block (bl_tm M)
    and buf_full_eq: buf_full
                   = (λk. (fst (buf k),
                           if k < k_tm M then fst (snd (buf k))
                             else tsP k (nP k),
                           tsP k (nP k)))
    and m_steps_raw: ((q, buf_full, λk. (AE_Home, ofs k)),
                      (q_out, bufC, end_pos))
                       ∈ m_steps_buffered M
    and ofs'_eq: ofs' = (λk. if k < k_tm M then snd (end_pos k)
                             else init_offset k)
    and buf'_eq: buf' = (λk. if k < k_tm M then bufC k
                             else init_buffer (le_tm M) k)
    and dest'_eq: dest' = (λk. if k < k_tm M then fst (end_pos k)
                               else init_dest k)
    and d_eq: dP = (λk. if k < k_tm M
                        then (if tsP k (nP k) = LE_block (le_tm M)
                                then dir.N else dir.L)
                        else dir.N)
  by (auto simp: ae_delta_ss4_ss5_def)
have ts_write_unchanged: (λk. (tsP k)(nP k := aP k)) = tsP
  using a_eq by (auto intro!: ext)
show thesis
proof (rule that[where q=q and ofs=ofs and buf=buf
                 and dest_old=dest_old and ts=tsP and n=nP
                 and q_out=q_out and buf'=buf' and bufC=bufC
                 and end_pos=end_pos])
  show c = Config_M (q, ofs, buf, dest_old, SS4) tsP nP
    using cP_eq s_eq by simp
  show mt_state c' = (q_out,
                      λk. if k < k_tm M then snd (end_pos k)
                            else init_offset k,
                      buf',
                      λk. if k < k_tm M then fst (end_pos k)
                            else init_dest k, SS5)
```

```
    using c'_eq s'_eq ofs'_eq dest'_eq by simp
  show buf' = (λk. if k < k_tm M then bufC k
                   else init_buffer (le_tm M) k)
    by (rule buf'_eq)
  show mt_tape c' = tsP
    using c'_eq ts_write_unchanged by simp
  show mt_pos c'
       = (λk. go_dir (if k < k_tm M
                      then (if tsP k (nP k) = LE_block (le_tm M)
                            then dir.N else dir.L)
                      else dir.N)
                     (nP k))
    using c'_eq d_eq by simp
  show q ∈ Q_tm M by (rule q_in_Q)
  show q ≠ t_tm M by (rule q_neq_t_M)
  show q ≠ r_tm M by (rule q_neq_r_M)
  show ∀j≥k_tm M. tsP j (nP j) = bl_block (bl_tm M) by (rule supp)
  show ((q, λk. (fst (buf k),
                 if k < k_tm M then fst (snd (buf k)) else tsP k (nP k),
                 tsP k (nP k)),
         λk. (AE_Home, ofs k)),
        (q_out, bufC, end_pos)) ∈ m_steps_buffered M
    using m_steps_raw buf_full_eq by simp
 qed
qed
```

2.29.3 Backward stage: load, compute, writeback

Reverse-arm counterpart of *ae_forward_stage_load_chain.* Given a 3-step *alphabet_enlarge_delta*-chain $c' \rightarrow\hat{}3\ c3$ with c' at SS1 with the forward-arm invariants, identifies the three buffer-load substeps (SS1→SS2, SS2→SS3, SS3→SS4), commits each peeled step to its canonical substep relation via *mttm_step_ae_delta_ssN_only*, and propagates the invariant bundle (*ae_inv_ss4*, *ae_tape_in_gamma_block*, *ae_buffer_in_gamma_block*, state preservation, *ae_position_link*) to *c3*. Side-band preservation reuses the direction-agnostic lemmas (*ae_step_alphabet_enlarge_gamma_preserve* etc.) from the forward arm.

```
lemma ae_backward_stage_load_chain:
  fixes M :: ('q, 'a) mttm
    and c' c3 :: ('c :: enum ⇒ 'a,
                  'q × ('a, 'c) ae_stage) mt_config
    and qM' :: 'q
    and ofs :: nat ⇒ 'c
    and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
    and dest :: nat ⇒ ae_dest
  assumes vM:          valid_mttm M
      and c'_state:    mt_state c' = (qM', ofs, buf, dest, SS1)
      and inv_ss1:     ae_inv_ss1 M c'
```

and *gamma_c'*: *ae_tape_in_gamma_block M c'*
and *buf_gamma*: *ae_buffer_in_gamma_block M c'*
and *pos_link_c'*: *ae_position_link M c'*
and *qM'_neq_t*: $qM' \neq t_tm\ M$
and *qM'_neq_r*: $qM' \neq r_tm\ M$
and *run*: $(c', c3) \in mttm_step\ (alphabet_enlarge_delta\ M) \hat{\ }\hat{\ } 3$
obtains *c1 c2* **where**
$(c', c1) \in mttm_step\ (ae_delta_ss1_ss2\ M)$
and $(c', c1) \in mttm_step\ (alphabet_enlarge_delta\ M)$
and $(c1, c2) \in mttm_step\ (ae_delta_ss2_ss3\ M)$
and $(c1, c2) \in mttm_step\ (alphabet_enlarge_delta\ M)$
and $(c2, c3) \in mttm_step\ (ae_delta_ss3_ss4\ M)$
and $(c2, c3) \in mttm_step\ (alphabet_enlarge_delta\ M)$
and *ae_inv_ss4 M c3*
and *ae_tape_in_gamma_block M c3*
and *ae_buffer_in_gamma_block M c3*
and $fst\ (mt_state\ c3) = qM'$
and *ae_position_link M c3*
proof −
— Peel the 3-step chain into three alphabet-enlarge steps through intermediates *c1*, *c2*.
from *run* **obtain** *c1 c2* **where**
step1: $(c', c1) \in mttm_step\ (alphabet_enlarge_delta\ M)$
and *step2*: $(c1, c2) \in mttm_step\ (alphabet_enlarge_delta\ M)$
and *step3*: $(c2, c3) \in mttm_step\ (alphabet_enlarge_delta\ M)$
by (*auto dest*!: *relpow_Suc_D2 simp*: *numeral_3_eq_3*)

— Substep 1: SS1→SS2. Commit step1 to *ae_delta_ss1_ss2* via *mttm_step_ae_delta_ss1_only*, then propagate the per-substep invariant and sidebands.
have *c'_idx_ss1*: $snd\ (snd\ (snd\ (snd\ (mt_state\ c')))) = SS1$
using *c'_state* **by** *simp*
have *step1_sub*: $(c', c1) \in mttm_step\ (ae_delta_ss1_ss2\ M)$
by (*rule mttm_step_ae_delta_ss1_only*[*OF step1 c'_idx_ss1*])
have *inv_ss2*: *ae_inv_ss2 M c1*
using *ae_step_ss1_ss2_invariant*[*OF vM inv_ss1 step1_sub*] .
have *gamma_c1*: *ae_tape_in_gamma_block M c1*
using *ae_step_alphabet_enlarge_gamma_preserve*[*OF gamma_c' step1*] .
have *buf_gamma_c1*: *ae_buffer_in_gamma_block M c1*
using *ae_step_alphabet_enlarge_buffer_gamma_preserve*[*OF vM buf_gamma gamma_c' step1*] .
have *c1_fst_eq*: $fst\ (mt_state\ c1) = qM'$
using *step1_sub c'_state*
by (*auto simp*: *ae_delta_ss1_ss2_def elim*: *mttm_step.cases*)
have *aux_45_at_c'*: $(c', c1) \in mttm_step\ (ae_delta_ss4_ss5\ M) \Longrightarrow ae_position_link\ M\ c1$
using *ae_pos_link_aux_void_ss4_ss5*[**where** $M = M$ **and** *c_pre* $= c'$ **and** *c_post* $= c1$]

```
      c'_idx_ss1 by simp
  have aux_56_at_c': (c', c1) ∈ mttm_step (ae_delta_ss5_ss6 M)
                  ⟹ ae_position_link M c1
    using ae_pos_link_aux_void_ss5_ss6[where M = M and c_pre = c' and
c_post = c1]
      c'_idx_ss1 by simp
  have aux_78_at_c': (c', c1) ∈ mttm_step (ae_delta_ss7_ss8 M)
                  ⟹ ae_position_link M c1
    using ae_pos_link_aux_void_ss7_ss8[where M = M and c_pre = c' and
c_post = c1]
      c'_idx_ss1 by simp
  have pos_link_c1: ae_position_link M c1
    using ae_step_alphabet_enlarge_position_link_preserve[OF pos_link_c'
        step1 aux_45_at_c' aux_56_at_c' aux_78_at_c'] .

  — Substep 2: SS2→SS3.
  have c1_idx_ss2: snd (snd (snd (snd (mt_state c1)))) = SS2
    using inv_ss2 unfolding ae_inv_ss2_def by (cases mt_state c1) auto
  have step2_sub: (c1, c2) ∈ mttm_step (ae_delta_ss2_ss3 M)
    by (rule mttm_step_ae_delta_ss2_only[OF step2 c1_idx_ss2])
  have inv_ss3: ae_inv_ss3 M c2
    using ae_step_ss2_ss3_invariant[OF vM inv_ss2 step2_sub] .
  have gamma_c2: ae_tape_in_gamma_block M c2
    using ae_step_alphabet_enlarge_gamma_preserve[OF gamma_c1 step2] .
  have buf_gamma_c2: ae_buffer_in_gamma_block M c2
          using   ae_step_alphabet_enlarge_buffer_gamma_preserve[OF   vM
buf_gamma_c1
                                                    gamma_c1 step2] .
  have c2_fst_eq: fst (mt_state c2) = qM'
    using step2_sub c1_fst_eq
    by (cases mt_state c1)
       (auto simp: ae_delta_ss2_ss3_def elim: mttm_step.cases)
  have aux_45_at_c1: (c1, c2) ∈ mttm_step (ae_delta_ss4_ss5 M)
                  ⟹ ae_position_link M c2
    using ae_pos_link_aux_void_ss4_ss5[where M = M and c_pre = c1 and
c_post = c2]
      c1_idx_ss2 by simp
  have aux_56_at_c1: (c1, c2) ∈ mttm_step (ae_delta_ss5_ss6 M)
                  ⟹ ae_position_link M c2
    using ae_pos_link_aux_void_ss5_ss6[where M = M and c_pre = c1 and
c_post = c2]
      c1_idx_ss2 by simp
  have aux_78_at_c1: (c1, c2) ∈ mttm_step (ae_delta_ss7_ss8 M)
                  ⟹ ae_position_link M c2
    using ae_pos_link_aux_void_ss7_ss8[where M = M and c_pre = c1 and
c_post = c2]
      c1_idx_ss2 by simp
  have pos_link_c2: ae_position_link M c2
    using ae_step_alphabet_enlarge_position_link_preserve[OF pos_link_c1
```

```
    step2 aux_45_at_c1 aux_56_at_c1 aux_78_at_c1] .

  — Substep 3: SS3→SS4.
  have c2_idx_ss3: snd (snd (snd (snd (mt_state c2)))) = SS3
    using inv_ss3 unfolding ae_inv_ss3_def by (cases mt_state c2) auto
  have step3_sub: (c2, c3) ∈ mttm_step (ae_delta_ss3_ss4 M)
    by (rule mttm_step_ae_delta_ss3_only[OF step3 c2_idx_ss3])
  have inv_ss4: ae_inv_ss4 M c3
    using ae_step_ss3_ss4_invariant[OF vM inv_ss3 step3_sub] .
  have gamma_c3: ae_tape_in_gamma_block M c3
    using ae_step_alphabet_enlarge_gamma_preserve[OF gamma_c2 step3] .
  have buf_gamma_c3: ae_buffer_in_gamma_block M c3
        using ae_step_alphabet_enlarge_buffer_gamma_preserve[OF vM
buf_gamma_c2
                                                gamma_c2 step3] .
  have c3_fst_eq: fst (mt_state c3) = qM'
    using step3_sub c2_fst_eq
    by (cases mt_state c2)
       (auto simp: ae_delta_ss3_ss4_def elim: mttm_step.cases)
  have aux_45_at_c2: (c2, c3) ∈ mttm_step (ae_delta_ss4_ss5 M)
                  ⟹ ae_position_link M c3
    using ae_pos_link_aux_void_ss4_ss5[where M = M and c_pre = c2 and
c_post = c3]
          c2_idx_ss3 by simp
  have aux_56_at_c2: (c2, c3) ∈ mttm_step (ae_delta_ss5_ss6 M)
                  ⟹ ae_position_link M c3
    using ae_pos_link_aux_void_ss5_ss6[where M = M and c_pre = c2 and
c_post = c3]
          c2_idx_ss3 by simp
  have aux_78_at_c2: (c2, c3) ∈ mttm_step (ae_delta_ss7_ss8 M)
                  ⟹ ae_position_link M c3
    using ae_pos_link_aux_void_ss7_ss8[where M = M and c_pre = c2 and
c_post = c3]
          c2_idx_ss3 by simp
  have pos_link_c3: ae_position_link M c3
    using ae_step_alphabet_enlarge_position_link_preserve[OF pos_link_c2
          step3 aux_45_at_c2 aux_56_at_c2 aux_78_at_c2] .

  show ?thesis
    by (rule that[OF step1_sub step1 step2_sub step2 step3_sub step3
                  inv_ss4 gamma_c3 buf_gamma_c3 c3_fst_eq pos_link_c3])
qed
```

Reverse-arm SS4→SS5 compute extract. Counterpart of the forward arm's SS4→SS5 step construction. Given the load-chain output (the three buffer-load steps c'→c3 plus standard SS1-boundary invariants) and a substrate step $(c3,\ c4) \in mttm_step\ (alphabet_enlarge_delta\ M)$ issuing from c3 at SS4, this lemma produces:

1. Step commitment: $(c3,\ c4)$ lies in *ae_delta_ss4_ss5* (via Step 3a's

mttm_step_ae_delta_ss4_only).

2. Lifted *cM* trace: (*cM*, *cM_new*) ∈ *mttm_step* (*delta_tm M*) ^^ *n* for some *n* ≤ |*c*|, with halt-or-full termination produced by *m_steps_buffered_lift_to_tape* (Step 2b-wrapper) from the *m_steps_buffered* chain exposed by *ae_step_ss4_ss5_destruct_to_buffered*.

3. Threaded invariants at *cM_new*/*c4*: window invariant carrying through, regime-aware no-LE, and the *fst* (*buf'* *k*) ≠ *LE_block* guard for pos=0 tapes.

   Composes the direction-agnostic helpers from the forward arm (*ae_home_classification*, *ae_ss1_to_ss4_buffer_chars_general*, *ae_ss4_window_from_correspondence_general*) with the reverse-arm helpers (*mttm_step_ae_delta_ss4_only*, *ae_step_ss4_ss5_destruct_to_buffered*) and Step 2b-wrapper. Consumer: *ae_backward_stage*'s body.

```
lemma ae_backward_stage_compute_extract:
  fixes M :: ('q, 'a) mttm
    and cM :: ('a, 'q) mt_config
    and c' c1 c2 c3 c4 :: ('c :: enum ⇒ 'a,
                           'q × ('a, 'c) ae_stage) mt_config
    and qM' :: 'q
    and ofs :: nat ⇒ 'c
    and buf :: nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
    and dest :: nat ⇒ ae_dest
  assumes vM:          valid_mttm M
      and lu:          le_unique M
      and cM_valid:    valid_config_mttm M cM
      and sim:         ae_simulates M cM c'
      and c'_state:    mt_state c' = (qM', ofs, buf, dest, SS1)
      and le_anchor:   ∀k<k_tm M. mt_tape c' k 0 = LE_block (le_tm M)
      and le_neq_bl:   le_tm M ≠ bl_tm M
      and no_le_per_tape:
          ∀k. (mt_pos c' k ≥ 2
                ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
                      ⟶ mt_tape cM k
                          ((mt_pos c' k − 2) * card (UNIV :: 'c set)
                            + 1 + i)
                        ≠ le_tm M))
            ∧ (mt_pos c' k = 1
                ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
                      ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
            ∧ (mt_pos c' k = 0
                ⟶ (∀i. i < card (UNIV :: 'c set)
                      ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
```

```
    and step1_sub:  (c', c1) ∈ mttm_step (ae_delta_ss1_ss2 M)
    and step2_sub:  (c1, c2) ∈ mttm_step (ae_delta_ss2_ss3 M)
    and step3_sub:  (c2, c3) ∈ mttm_step (ae_delta_ss3_ss4 M)
    and step4_full: (c3, c4) ∈ mttm_step (alphabet_enlarge_delta M)
    and c3_idx_ss4: snd (snd (snd (snd (mt_state c3)))) = SS4
  obtains cM_new buf' end_pos n where
    (c3, c4) ∈ mttm_step (ae_delta_ss4_ss5 M)
   and (cM, cM_new) ∈ mttm_step (delta_tm M) ^^ n
   and n ≤ card (UNIV :: 'c set)
   and n = card (UNIV :: 'c set)
         ∨ mt_state cM_new ∈ {t_tm M, r_tm M}
   and mt_state c4 = (mt_state cM_new,
                      λk. if k < k_tm M then snd (end_pos k)
                          else init_offset k,
                      buf',
                      λk. if k < k_tm M then fst (end_pos k)
                          else init_dest k, SS5)
   and mt_tape c4 = mt_tape c'
   and ∀k<k_tm M. ae_window_invariant_general
             (mt_tape cM_new k) (mt_pos cM_new k)
             (end_pos k) (buf' k) (mt_pos c' k) (le_tm M)
   and ∀k. (mt_pos c' k = 0
              ⟶ (∀i. i < card (UNIV :: 'c set)
                     ⟶ mt_tape cM_new k (Suc i) ≠ le_tm M))
           ∧ (mt_pos c' k = 1
              ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
                     ⟶ mt_tape cM_new k (Suc i) ≠ le_tm M))
           ∧ (mt_pos c' k ≥ 2
              ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
                     ⟶ mt_tape cM_new k
                         ((mt_pos c' k − 2) * card (UNIV :: 'c set)
                           + 1 + i)
                        ≠ le_tm M))
   and ∀k<k_tm M. mt_pos c' k = 0
             ⟶ fst (buf' k) ≠ LE_block (le_tm M)
proof −
  — Commit step4 to ae_delta_ss4_ss5 via Step 3a's cross-phase exclusion lift.
  have step4_sub: (c3, c4) ∈ mttm_step (ae_delta_ss4_ss5 M)
    by (rule mttm_step_ae_delta_ss4_only[OF step4_full c3_idx_ss4])

  — Extract tape correspondence from the simulation invariant.
  have tape_corr: ∀k<k_tm M. ae_tape_correspondence (le_tm M)
                       (mt_tape cM k) (mt_tape c' k)
    using sim c'_state unfolding ae_simulates_def by auto

  — Home-cell LE classification (Step 3c.1 helper).
  have home_classification:
     ∀k<k_tm M. (mt_pos c' k = 0
           ⟶ mt_tape c' k (mt_pos c' k) = LE_block (le_tm M))
```

```
      ∧ (mt_pos c' k ≥ 1
        ⟶ mt_tape c' k (mt_pos c' k) ≠ LE_block (le_tm M))
  by (rule ae_home_classification[OF le_anchor no_le_per_tape tape_corr])
```

— SS4-entry buffer / tape / position characterisation (direction-agnostic, shared with forward arm).

```
  have c3_state:
    mt_state c3 = (qM', ofs,
                   (λk. if k < k_tm M
                        then (if mt_pos c' k = 0
                              then (bl_block (bl_tm M),
                                    LE_block (le_tm M),
                                    snd (snd (buf k)))
                              else (mt_tape c' k (mt_pos c' k − 1),
                                    mt_tape c' k (mt_pos c' k),
                                    snd (snd (buf k))))
                        else init_buffer (le_tm M) k),
                   dest, SS4)
    using ae_ss1_to_ss4_buffer_chars_general(1)[OF c'_state step1_sub
                                                   step2_sub step3_sub
                                                   home_classification] .
  have c3_tape: mt_tape c3 = mt_tape c'
    using ae_ss1_to_ss4_buffer_chars_general(2)[OF c'_state step1_sub
                                                   step2_sub step3_sub
                                                   home_classification] .
  have c3_pos: ⋀k. k < k_tm M ⟹ mt_pos c3 k = mt_pos c' k + 1
    using ae_ss1_to_ss4_buffer_chars_general(3)[OF c'_state step1_sub
                                                   step2_sub step3_sub
                                                   home_classification] .
```

— Per-tape SS4 window invariant at cM via the unified from-correspondence helper.

```
  have window_at_cM:
    ∀k<k_tm M. ae_window_invariant_general
          (mt_tape cM k) (mt_pos cM k)
          (AE_Home, ofs k)
          (if mt_pos c' k = 0
             then (bl_block (bl_tm M),
                   LE_block (le_tm M),
                   mt_tape c' k 1)
           else if mt_pos c' k = 1
             then (mt_tape c' k 0,
                   mt_tape c' k 1,
                   mt_tape c' k 2)
           else (mt_tape c' k (mt_pos c' k − 1),
                 mt_tape c' k (mt_pos c' k),
                 mt_tape c' k (mt_pos c' k + 1)))
          (mt_pos c' k)
          (le_tm M)
```

```
  using ae_ss4_window_from_correspondence_general[OF sim c'_state
                                                      le_anchor
                                                      no_le_per_tape] .

— Destructure the SS4→SS5 step into its m_steps_buffered chain. Bind every obtained variable and every shows-conjunct, then discharge by direct rule-application to avoid blast divergence on the large obtains.
  obtain qD ofsD bufD destD tsD nD q_out buf' bufC end_pos where
      c3_eq: c3 = Config_M (qD, ofsD, bufD, destD, SS4) tsD nD
    and c4_state_raw:
      mt_state c4 = (q_out,
                     λk. if k < k_tm M then snd (end_pos k)
                         else init_offset k,
                     buf',
                     λk. if k < k_tm M then fst (end_pos k)
                         else init_dest k, SS5)
    and buf'_def: buf' = (λk. if k < k_tm M then bufC k
                              else init_buffer (le_tm M) k)
    and c4_tape_raw: mt_tape c4 = tsD
    and c4_pos_raw:
      mt_pos c4
        = (λk. go_dir (if k < k_tm M
                       then (if tsD k (nD k) = LE_block (le_tm M)
                             then dir.N else dir.L)
                       else dir.N)
                      (nD k))
    and qD_in: qD ∈ Q_tm M
    and qD_neq_t: qD ≠ t_tm M
    and qD_neq_r: qD ≠ r_tm M
    and tsD_supp: ∀j≥k_tm M. tsD j (nD j) = bl_block (bl_tm M)
    and m_steps:
      ((qD, λk. (fst (bufD k),
                 if k < k_tm M then fst (snd (bufD k)) else tsD k (nD k),
                 tsD k (nD k)),
        λk. (AE_Home, ofsD k)),
       (q_out, bufC, end_pos)) ∈ m_steps_buffered M
    by (rule ae_step_ss4_ss5_destruct_to_buffered[OF step4_sub])

— Identify destructed names with our load-chain view.
  have qD_eq: qD = qM'
    using c3_eq c3_state by simp
  have ofsD_eq: ofsD = ofs
    using c3_eq c3_state by simp
  have bufD_eq:
      bufD = (λk. if k < k_tm M
                  then (if mt_pos c' k = 0
                        then (bl_block (bl_tm M),
                              LE_block (le_tm M),
                              snd (snd (buf k)))
```

```
                    else (mt_tape c' k (mt_pos c' k − 1),
                          mt_tape c' k (mt_pos c' k),
                          snd (snd (buf k))))
               else init_buffer (le_tm M) k)
  using c3_eq c3_state by simp
have tsD_eq: tsD = mt_tape c'
  using c3_eq c3_tape by simp
have nD_eq: ⋀k. k < k_tm M ⟹ nD k = mt_pos c' k + 1
  using c3_eq c3_pos by simp
```

— The destruct's *m_steps* source is the guarded *buf_full* home block; name it *buf_full_src* so the Step 2b-wrapper unification is first-order. On active tapes ($k < k_tm\ M$) it collapses to the window helper's regime-aware buffer shape; padding tapes carry the frozen empty-tape shadow and never enter the wrapper's $\forall k{<}k_tm\ M$ obligations.

```
define buf_full_src ::
    nat ⇒ ('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
  where buf_full_src =
        (λk. (fst (bufD k),
              if k < k_tm M then fst (snd (bufD k)) else tsD k (nD k),
              tsD k (nD k)))
```

— Active-tape alignment: for $k < k_tm\ M$ the guarded source coincides with the window helper's regime shape.

```
have src_align:
    ∀k<k_tm M. buf_full_src k
          = (if mt_pos c' k = 0
             then (bl_block (bl_tm M), LE_block (le_tm M),
                   mt_tape c' k 1)
             else if mt_pos c' k = 1
             then (mt_tape c' k 0, mt_tape c' k 1, mt_tape c' k 2)
             else (mt_tape c' k (mt_pos c' k − 1),
                   mt_tape c' k (mt_pos c' k),
                   mt_tape c' k (mt_pos c' k + 1)))
proof (intro allI impI)
  fix k assume klt: k < k_tm M
  have bufD_k: bufD k
                = (if mt_pos c' k = 0
                   then (bl_block (bl_tm M), LE_block (le_tm M),
                         snd (snd (buf k)))
                   else (mt_tape c' k (mt_pos c' k − 1),
                         mt_tape c' k (mt_pos c' k),
                         snd (snd (buf k))))
    using bufD_eq klt by simp
  have tsD_k: tsD k (nD k) = mt_tape c' k (mt_pos c' k + 1)
    using tsD_eq nD_eq[OF klt] by simp
  have src_k: buf_full_src k
              = (fst (bufD k), fst (snd (bufD k)), tsD k (nD k))
    using klt unfolding buf_full_src_def by simp
```

```
    show buf_full_src k
          = (if mt_pos c' k = 0
               then (bl_block (bl_tm M), LE_block (le_tm M),
                     mt_tape c' k 1)
             else if mt_pos c' k = 1
               then (mt_tape c' k 0, mt_tape c' k 1, mt_tape c' k 2)
             else (mt_tape c' k (mt_pos c' k − 1),
                   mt_tape c' k (mt_pos c' k),
                   mt_tape c' k (mt_pos c' k + 1)))
    proof (cases mt_pos c' k)
      case 0
      thus ?thesis using src_k bufD_k tsD_k by simp
    next
      case (Suc m)
      show ?thesis
      proof (cases m)
        case 0
        with Suc src_k bufD_k tsD_k show ?thesis
          by (simp add: eval_nat_numeral)
      next
        case (Suc mm)
        with ‹mt_pos c' k = Suc m› src_k bufD_k tsD_k show ?thesis by simp
      qed
    qed
  qed

  — State alignment for cM: simulation gives mt_state cM = qM', and destruct
gives qD = qM'.
  have st_eq_cM: mt_state cM = qD
    using sim c'_state qD_eq unfolding ae_simulates_def by auto

  — m_steps chain in the form expected by the Step 2b-wrapper, with the guarded
source named and ofs substituted for ofsD.
  have m_steps_src:
      ((qD, buf_full_src, λk. (AE_Home, ofs k)),
          (q_out, bufC, end_pos)) ∈ m_steps_buffered M
    using m_steps ofsD_eq unfolding buf_full_src_def by simp

  — Window invariant at cM for the guarded source, on active tapes, via the regime
alignment.
  have window_at_cM_src:
      ∀k<k_tm M. ae_window_invariant_general
            (mt_tape cM k) (mt_pos cM k)
            (AE_Home, ofs k) (buf_full_src k) (mt_pos c' k) (le_tm M)
  proof (intro allI impI)
    fix k assume klt: k < k_tm M
    show ae_window_invariant_general (mt_tape cM k) (mt_pos cM k)
          (AE_Home, ofs k) (buf_full_src k) (mt_pos c' k) (le_tm M)
      using window_at_cM[rule_format, OF klt] src_align[rule_format, OF klt]
```

```
    by simp
  qed

  — Invoke Step 2b-wrapper. The window invariant at cM and the regime-keyed
  no_le match the wrapper's hypotheses by construction. Left-not-LE for pos=0
  tapes is the bl_block ≠ LE_block argument.
  have bl_neq_le:
      (bl_block (bl_tm M) :: 'c ⇒ 'a) ≠ LE_block (le_tm M)
  proof
    assume eq: (bl_block (bl_tm M) :: 'c ⇒ 'a) = LE_block (le_tm M)
    have bl_block (bl_tm M) (SOME x :: 'c. True)
          = LE_block (le_tm M) (SOME x :: 'c. True)
      using eq by simp
    hence bl_tm M = le_tm M
      unfolding bl_block_def LE_block_def by simp
    with le_neq_bl show False by simp
  qed
  have left_not_le_pos0_buf_full:
      ∀k<k_tm M. mt_pos c' k = 0
            ⟶ fst (buf_full_src k) ≠ LE_block (le_tm M)
  proof (intro allI impI impI)
    fix k assume klt: k < k_tm M and pos0: mt_pos c' k = 0
    have buf_full_src k
          = (bl_block (bl_tm M), LE_block (le_tm M), mt_tape c' k 1)
      using src_align[rule_format, OF klt] pos0 by simp
    thus fst (buf_full_src k) ≠ LE_block (le_tm M)
      using bl_neq_le by simp
  qed

  — Reorder the regime conjunction so it matches the Step 2b-wrapper's hypothesis
  shape (pos=0, pos=1, pos≥2). The outer no_le_per_tape uses the opposite order
  (pos≥2, pos=1, pos=0) inherited from the forward-arm assumption convention.
  have no_le_per_tape_reordered:
      ∀k. (mt_pos c' k = 0
            ⟶ (∀i. i < card (UNIV :: 'c set)
                  ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
        ∧ (mt_pos c' k = 1
            ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
                  ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
        ∧ (mt_pos c' k ≥ 2
            ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
                  ⟶ mt_tape cM k
                      ((mt_pos c' k − 2) * card (UNIV :: 'c set)
                        + 1 + i)
                    ≠ le_tm M))
  using no_le_per_tape by blast

  obtain cM_new n where
      cM_trace_n: (cM, cM_new) ∈ mttm_step (delta_tm M) ^^ n
```

```
  and n_bnd: n ≤ card (UNIV :: 'c set)
  and halt_or_full:
    n = card (UNIV :: 'c set)
      ∨ mt_state cM_new ∈ {t_tm M, r_tm M}
  and cM_new_st: mt_state cM_new = q_out
  and new_window_C:
    ∀k<k_tm M. ae_window_invariant_general
        (mt_tape cM_new k) (mt_pos cM_new k)
        (end_pos k) (bufC k) (mt_pos c' k) (le_tm M)
  and new_no_le:
    ∀k. (mt_pos c' k = 0
        ⟶ (∀i. i < card (UNIV :: 'c set)
            ⟶ mt_tape cM_new k (Suc i) ≠ le_tm M))
      ∧ (mt_pos c' k = 1
        ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
            ⟶ mt_tape cM_new k (Suc i) ≠ le_tm M))
      ∧ (mt_pos c' k ≥ 2
        ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
            ⟶ mt_tape cM_new k
                ((mt_pos c' k − 2) * card (UNIV :: 'c set)
                  + 1 + i)
              ≠ le_tm M))
  and new_left_C:
    ∀k<k_tm M. mt_pos c' k = 0
      ⟶ fst (bufC k) ≠ LE_block (le_tm M)
  by (rule m_steps_buffered_lift_to_tape
      [OF vM lu cM_valid qD_in m_steps_src st_eq_cM window_at_cM_src
        no_le_per_tape_reordered left_not_le_pos0_buf_full])
```

— The SS5 buffer *buf'* agrees with the chain target *bufC* on active tapes; transport the per-active-tape window and left-not-LE facts.

```
  have buf'_active: ∀k<k_tm M. buf' k = bufC k
    using buf'_def by simp
  have new_window:
    ∀k<k_tm M. ae_window_invariant_general
        (mt_tape cM_new k) (mt_pos cM_new k)
        (end_pos k) (buf' k) (mt_pos c' k) (le_tm M)
    using new_window_C buf'_active by simp
  have new_left_not_le_pos0:
    ∀k<k_tm M. mt_pos c' k = 0
        ⟶ fst (buf' k) ≠ LE_block (le_tm M)
    using new_left_C buf'_active by simp
```

— Tape preservation through SS4→SS5: the substep writes the same letter it reads ($a' = a$), so *mt_tape c4 = mt_tape c3 = mt_tape c'*.

```
  have c4_tape: mt_tape c4 = mt_tape c'
    using c4_tape_raw tsD_eq by simp
```

— Rewrite *c4_state_raw* to use *mt_state cM_new*; offset / dest fields are guarded (frozen at *init_∗* beyond the tape count), mirroring the SS4→SS5 δ.

```
  have c4_state:
     mt_state c4 = (mt_state cM_new,
                    λk. if k < k_tm M then snd (end_pos k)
                          else init_offset k,
                    buf',
                    λk. if k < k_tm M then fst (end_pos k)
                          else init_dest k, SS5)
    using c4_state_raw cM_new_st by simp

  show thesis
  proof (rule that[where cM_new = cM_new and buf' = buf'
                     and end_pos = end_pos and n = n])
    show (c3, c4) ∈ mttm_step (ae_delta_ss4_ss5 M) by (rule step4_sub)
    show (cM, cM_new) ∈ mttm_step (delta_tm M) ^^ n
      by (rule cM_trace_n)
    show n ≤ card (UNIV :: 'c set) by (rule n_bnd)
    show n = card (UNIV :: 'c set)
          ∨ mt_state cM_new ∈ {t_tm M, r_tm M}
      by (rule halt_or_full)
    show mt_state c4 = (mt_state cM_new,
                        λk. if k < k_tm M then snd (end_pos k)
                              else init_offset k,
                        buf',
                        λk. if k < k_tm M then fst (end_pos k)
                              else init_dest k, SS5)
      by (rule c4_state)
    show mt_tape c4 = mt_tape c' by (rule c4_tape)
    show ∀k<k_tm M. ae_window_invariant_general
              (mt_tape cM_new k) (mt_pos cM_new k)
              (end_pos k) (buf' k) (mt_pos c' k) (le_tm M)
      by (rule new_window)
    show ∀k. (mt_pos c' k = 0
                ⟶ (∀i. i < card (UNIV :: 'c set)
                        ⟶ mt_tape cM_new k (Suc i) ≠ le_tm M))
           ∧ (mt_pos c' k = 1
                ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
                        ⟶ mt_tape cM_new k (Suc i) ≠ le_tm M))
           ∧ (mt_pos c' k ≥ 2
                ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
                        ⟶ mt_tape cM_new k
                            ((mt_pos c' k − 2) * card (UNIV :: 'c set)
                              + 1 + i)
                          ≠ le_tm M))
      by (rule new_no_le)
    show ∀k<k_tm M. mt_pos c' k = 0
           ⟶ fst (buf' k) ≠ LE_block (le_tm M)
      by (rule new_left_not_le_pos0)
  qed
qed
```

Reverse-arm SS5→SS1 writeback chain peel. Mirror of *ae_backward_stage_load_chain* for the four write-back substeps (SS5→SS6, SS6→SS7, SS7→SS8, SS8→SS1).

Given an SS5-boundary config *c4* with the standard SS5 invariants and a 4-step *alphabet_enlarge_delta* chain to c'', identifies each substep via Step 3a's *mttm_step_ae_delta_ssN_only*, propagates the per-substep invariants *ae_inv_ss* through the chain, carries *ae_tape_in_gamma_block* and *ae_buffer_in_gamma_block* via the direction-agnostic preserve lemmas, and exposes the halt-aware c" status:

- non-halt branch ($q_out \notin \{t_tm\ M,\ r_tm\ M\}$): c'' lands at SS1 with the same q-component as c4 (preserved through all 4 substeps).
- halt branch ($q_out \in \{t_tm\ M,\ r_tm\ M\}$): c'' lands at VFwd with halted q-component and *init_stage le*.

  Position-link propagation through writeback substeps is NOT performed here it requires the regime-aware buffer non-LE analysis that's better packaged with the final *ae_simulates* assembly in Step 3f.

lemma *ae_backward_stage_writeback_chain*:
 fixes $M :: ('q,\ 'a)\ mttm$
  and $c4\ c'' :: ('c :: enum \Rightarrow 'a,$
  $'q \times ('a,\ 'c)\ ae_stage)\ mt_config$
  and $q_out :: 'q$
 assumes *vM*: $valid_mttm\ M$
  and *c4_inv*: $ae_inv_ss5\ M\ c4$
  and *c4_q*: $fst\ (mt_state\ c4) = q_out$
  and *q_out_in_Q*: $q_out \in Q_tm\ M$
  and *gamma_c4*: $ae_tape_in_gamma_block\ M\ c4$
  and *buf_gamma_c4*: $ae_buffer_in_gamma_block\ M\ c4$
  and *run*: $(c4,\ c'')$
  $\in mttm_step\ (alphabet_enlarge_delta\ M)\ \widehat{}\ 4$
 obtains *c5 c6 c7* **where**
  $(c4,\ c5) \in mttm_step\ (ae_delta_ss5_ss6\ M)$
  and $(c4,\ c5) \in mttm_step\ (alphabet_enlarge_delta\ M)$
  and $(c5,\ c6) \in mttm_step\ (ae_delta_ss6_ss7\ M)$
  and $(c5,\ c6) \in mttm_step\ (alphabet_enlarge_delta\ M)$
  and $(c6,\ c7) \in mttm_step\ (ae_delta_ss7_ss8\ M)$
  and $(c6,\ c7) \in mttm_step\ (alphabet_enlarge_delta\ M)$
  and $(c7,\ c'') \in mttm_step\ (ae_delta_ss8_ss1\ M)$
  and $(c7,\ c'') \in mttm_step\ (alphabet_enlarge_delta\ M)$
  and $q_out \notin \{t_tm\ M,\ r_tm\ M\} \longrightarrow ae_inv_ss1\ M\ c''$
  and $q_out \in \{t_tm\ M,\ r_tm\ M\}$
  $\longrightarrow (case\ mt_state\ c''\ of\ (qH,\ _,\ _,\ _,\ idx)$
  $\Rightarrow qH \in \{t_tm\ M,\ r_tm\ M\} \wedge idx = VFwd)$
  and $fst\ (mt_state\ c'') = q_out$
  and $ae_tape_in_gamma_block\ M\ c''$

```
    and ae_buffer_in_gamma_block M c''
proof −
  — Peel the 4-step chain into four alphabet-enlarge steps through intermediates
c5, c6, c7.
  from run obtain c5 c6 c7 where
      step5: (c4, c5) ∈ mttm_step (alphabet_enlarge_delta M)
    and step6: (c5, c6) ∈ mttm_step (alphabet_enlarge_delta M)
    and step7: (c6, c7) ∈ mttm_step (alphabet_enlarge_delta M)
    and step8: (c7, c'') ∈ mttm_step (alphabet_enlarge_delta M)
    by (auto dest!: relpow_Suc_D2 simp: numeral_eq_Suc)

  — Substep 5: SS5→SS6.
  have c4_idx_ss5: snd (snd (snd (snd (mt_state c4)))) = SS5
    using c4_inv unfolding ae_inv_ss5_def by (cases mt_state c4) auto
  have step5_sub: (c4, c5) ∈ mttm_step (ae_delta_ss5_ss6 M)
    by (rule mttm_step_ae_delta_ss5_only[OF step5 c4_idx_ss5])
  have inv_ss6: ae_inv_ss6 M c5
    using ae_step_ss5_ss6_invariant[OF vM c4_inv step5_sub] .
  have gamma_c5: ae_tape_in_gamma_block M c5
    using ae_step_alphabet_enlarge_gamma_preserve[OF gamma_c4 step5] .
  have buf_gamma_c5: ae_buffer_in_gamma_block M c5
        using ae_step_alphabet_enlarge_buffer_gamma_preserve[OF vM
buf_gamma_c4
                                                gamma_c4 step5] .
  have c5_fst_eq: fst (mt_state c5) = q_out
    using step5_sub c4_q
    by (cases mt_state c4)
      (auto simp: ae_delta_ss5_ss6_def elim: mttm_step.cases)

  — Substep 6: SS6→SS7.
  have c5_idx_ss6: snd (snd (snd (snd (mt_state c5)))) = SS6
    using inv_ss6 unfolding ae_inv_ss6_def
    by (cases mt_state c5) auto
  have step6_sub: (c5, c6) ∈ mttm_step (ae_delta_ss6_ss7 M)
    by (rule mttm_step_ae_delta_ss6_only[OF step6 c5_idx_ss6])
  have inv_ss7: ae_inv_ss7 M c6
    using ae_step_ss6_ss7_invariant[OF vM inv_ss6 step6_sub] .
  have gamma_c6: ae_tape_in_gamma_block M c6
    using ae_step_alphabet_enlarge_gamma_preserve[OF gamma_c5 step6] .
  have buf_gamma_c6: ae_buffer_in_gamma_block M c6
        using ae_step_alphabet_enlarge_buffer_gamma_preserve[OF vM
buf_gamma_c5
                                                gamma_c5 step6] .
  have c6_fst_eq: fst (mt_state c6) = q_out
    using step6_sub c5_fst_eq
    by (cases mt_state c5)
      (auto simp: ae_delta_ss6_ss7_def elim: mttm_step.cases)

  — Substep 7: SS7→SS8.
```

have *c6_idx_ss7*: *snd* (*snd* (*snd* (*snd* (*mt_state c6*)))) = *SS7*
using *inv_ss7* **unfolding** *ae_inv_ss7_def*
by (*cases mt_state c6*) *auto*
have *step7_sub*: (*c6*, *c7*) ∈ *mttm_step* (*ae_delta_ss7_ss8 M*)
by (*rule mttm_step_ae_delta_ss7_only*[*OF step7 c6_idx_ss7*])
have *inv_ss8*: *ae_inv_ss8 M c7*
using *ae_step_ss7_ss8_invariant*[*OF vM inv_ss7 step7_sub*] .
have *gamma_c7*: *ae_tape_in_gamma_block M c7*
using *ae_step_alphabet_enlarge_gamma_preserve*[*OF gamma_c6 step7*] .
have *buf_gamma_c7*: *ae_buffer_in_gamma_block M c7*
using *ae_step_alphabet_enlarge_buffer_gamma_preserve*[*OF vM buf_gamma_c6 gamma_c6 step7*] .
have *c7_fst_eq*: *fst* (*mt_state c7*) = *q_out*
using *step7_sub c6_fst_eq*
by (*cases mt_state c6*)
(*auto simp*: *ae_delta_ss7_ss8_def elim*: *mttm_step.cases*)

— Substep 8: SS8→SS1 with halt-aware routing. The substep relation pins the q-component; the post-state *stage′* is *init_stage le* in the halt branch and (*ofs*, *buf*, *init_dest*, *SS1*) in the non-halt branch.
have *c7_idx_ss8*: *snd* (*snd* (*snd* (*snd* (*mt_state c7*)))) = *SS8*
using *inv_ss8* **unfolding** *ae_inv_ss8_def*
by (*cases mt_state c7*) *auto*
have *step8_sub*: (*c7*, *c′′*) ∈ *mttm_step* (*ae_delta_ss8_ss1 M*)
by (*rule mttm_step_ae_delta_ss8_only*[*OF step8 c7_idx_ss8*])
have *c′′_fst_eq*: *fst* (*mt_state c′′*) = *q_out*
using *step8_sub c7_fst_eq*
by (*cases mt_state c7*)
(*auto simp*: *ae_delta_ss8_ss1_def elim*: *mttm_step.cases*)
have *gamma_c′′*: *ae_tape_in_gamma_block M c′′*
using *ae_step_alphabet_enlarge_gamma_preserve*[*OF gamma_c7 step8*] .
have *buf_gamma_c′′*: *ae_buffer_in_gamma_block M c′′*
using *ae_step_alphabet_enlarge_buffer_gamma_preserve*[*OF vM buf_gamma_c7 gamma_c7 step8*] .

— Halt-aware c” invariant. In the non-halt branch, *ae_step_ss8_ss1_invariant* gives *ae_inv_ss1 M c′′*. In the halt branch, *ae_step_ss8_ss1_invariant_halt* gives the *qH* ∈ {*t_tm M*, *r_tm M*} ∧ *idx* = *VFwd* state shape at c”.
have *non_halt_inv*:
q_out ∉ {*t_tm M*, *r_tm M*} ⟶ *ae_inv_ss1 M c′′*
proof
assume *nh*: *q_out* ∉ {*t_tm M*, *r_tm M*}
have *non_halt_at_c7*:
case mt_state c7 of (*qM′*, _, _, _, _) ⇒
qM′ ≠ *t_tm M* ∧ *qM′* ≠ *r_tm M*
using *c7_fst_eq nh* **by** (*cases mt_state c7*) *auto*
show *ae_inv_ss1 M c′′*

```
        using ae_step_ss8_ss1_invariant[OF vM inv_ss8 non_halt_at_c7
step8_sub] .
  qed
  have halt_inv:
     q_out ∈ {t_tm M, r_tm M}
        ⟶ (case mt_state c'' of (qH, _, _, _, idx)
              ⇒ qH ∈ {t_tm M, r_tm M} ∧ idx = VFwd)
  proof
    assume h: q_out ∈ {t_tm M, r_tm M}
    have halt_at_c7:
       case mt_state c7 of (qM', _, _, _, _) ⇒
         qM' ∈ {t_tm M, r_tm M}
      using c7_fst_eq h by (cases mt_state c7) auto
    show case mt_state c'' of (qH, _, _, _, idx)
          ⇒ qH ∈ {t_tm M, r_tm M} ∧ idx = VFwd
     using ae_step_ss8_ss1_invariant_halt[OF inv_ss8 halt_at_c7 step8_sub] .
  qed

  show ?thesis
    by (rule that[OF step5_sub step5 step6_sub step6 step7_sub step7
                  step8_sub step8 non_halt_inv halt_inv c''_fst_eq
                  gamma_c'' buf_gamma_c''])
qed
```

Reverse-arm dispatcher: unpack *ae_simulates* *without* committing to the SS1-versus-halt disjunction. Companion to *ae_forward_stage_unpack_sim* (which requires the SS1 branch via the non-halt M-state hypotheses).

Where *ae_forward_stage_unpack_sim* serves forward-arm proofs (M-state is mid-simulation by hypothesis, so SS1 is forced), *ae_simulates_state_disjunction* serves reverse-arm proofs that walk an M'-chain to a canonical halt config: at each step the current paired c' might be mid-simulation (SS1) or already at halt (*init_stage le*), and the proof needs both branches exposed for case-analysis.

```
lemma ae_simulates_state_disjunction:
  fixes M :: ('q, 'a) mttm
    and cM :: ('a, 'q) mt_config
    and c' :: ('c :: enum ⇒ 'a,
               'q × ('a, 'c) ae_stage) mt_config
  assumes sim: ae_simulates M cM c'
  obtains qM' ofs buf dest idx where
     mt_state c' = (qM', ofs, buf, dest, idx)
    and mt_state cM = qM'
    and (idx = SS1 ∧ qM' ∉ {t_tm M, r_tm M})
        ∨ (qM' ∈ {t_tm M, r_tm M}
             ∧ (ofs, buf, dest, idx) = init_stage (le_tm M))
    and ∀k<k_tm M. ae_tape_correspondence (le_tm M)
             (mt_tape cM k) (mt_tape c' k)
```

```
    and ae_tape_in_gamma_block M c'
proof −
  obtain qM' ofs buf dest idx where
      state_comp: mt_state c' = (qM', ofs, buf, dest, idx)
    by (cases mt_state c')
  have sim_body:
      ((idx = SS1 ∧ qM' ∉ {t_tm M, r_tm M})
         ∨ (qM' ∈ {t_tm M, r_tm M}
              ∧ (ofs, buf, dest, idx) = init_stage (le_tm M)))
       ∧ mt_state cM = qM'
       ∧ (∀k<k_tm M. ae_tape_correspondence (le_tm M)
              (mt_tape cM k) (mt_tape c' k))
       ∧ (idx = SS1
            ⟶ (∀k<k_tm M. mt_pos cM k = ae_decode_pos (mt_pos c' k) (ofs
k)))
       ∧ ae_tape_in_gamma_block M c'
    using sim state_comp unfolding ae_simulates_def by simp
  have qM_eq: mt_state cM = qM' using sim_body by simp
  have state_disj:
      (idx = SS1 ∧ qM' ∉ {t_tm M, r_tm M})
       ∨ (qM' ∈ {t_tm M, r_tm M}
            ∧ (ofs, buf, dest, idx) = init_stage (le_tm M))
    using sim_body by simp
  have tape_corr:
      ∀k<k_tm M. ae_tape_correspondence (le_tm M)
            (mt_tape cM k) (mt_tape c' k)
    using sim_body by simp
  have gamma_c': ae_tape_in_gamma_block M c'
    using sim_body by simp
  show ?thesis
    by (rule that[OF state_comp qM_eq state_disj tape_corr gamma_c'])
qed
```

2.29.4 Window-inversion kernel and SS5→SS8 congruence (*det*-free)

The det-free replacement for the SS4→SS5 chain-uniqueness used in *ae_backward_stage*. The *det_mttm M* hypothesis was consumed only to identify the given backward chain's SS4→SS5 target with the forward arm's reconstruction. Both targets satisfy *ae_window_invariant_general* against the *same* extracted M-trace endpoint, so the identification is a functional inversion of the window invariant, not a use of determinism. The kernel is two unconditional injectivity facts about the linearisation: (A) *bp_linear* is injective — the boundary position *bp* is recovered from the head *nM*; (B) the *3·card*-cell window *buf_lin_at blocks* determines *blocks* — the buffer triple is recovered from the tape. Combined, they give the full-window (*pos* ≥ *2*) inversion below.

c_idx is injective: it has *enum_class.enum* ! *c_idx x* = *x* as a section (*c_idx_in_range*), so equal indices give equal elements.

```
lemma c_idx_inj:
  fixes x y :: 'c :: enum
  assumes c_idx x = c_idx y
  shows x = y
proof −
  have x = (enum_class.enum :: 'c list) ! c_idx x
    by (rule c_idx_in_range(2)[symmetric])
  also have ... = (enum_class.enum :: 'c list) ! c_idx y
    using assms by simp
  also have ... = y by (rule c_idx_in_range(2))
  finally show ?thesis .
qed
```

(A) *bp_linear* injective. The direction component lands the value in one of three disjoint length-*card* intervals, so equal values force equal directions; *c_idx*-injectivity then forces equal cells.

```
lemma bp_linear_inj:
  fixes p q :: ('c :: enum) bp
  assumes bp_linear p = bp_linear q
  shows p = q
proof −
  obtain d x where p: p = (d, x) by (cases p)
  obtain d' y where q: q = (d', y) by (cases q)
  have bx: c_idx x < card (UNIV :: 'c set) by (rule c_idx_lt_card)
  have by': c_idx y < card (UNIV :: 'c set) by (rule c_idx_lt_card)
  from assms have eq:
    (case d of AE_Left ⇒ 0 | AE_Home ⇒ card (UNIV :: 'c set)
       | AE_Right ⇒ 2 * card (UNIV :: 'c set)) + c_idx x
    = (case d' of AE_Left ⇒ 0 | AE_Home ⇒ card (UNIV :: 'c set)
       | AE_Right ⇒ 2 * card (UNIV :: 'c set)) + c_idx y
    by (simp add: p q bp_linear_def)
  have d = d' ∧ c_idx x = c_idx y
    using eq bx by' by (cases d; cases d'; simp; linarith)
  hence d = d' and x = y using c_idx_inj by auto
  thus ?thesis by (simp add: p q)
qed
```

(B) the window *buf_lin_at blocks* determines *blocks*. Each of the three blocks is read at every *'c*-cell as *i* sweeps its third of the window, because *enum_class.enum* enumerates the whole type.

```
lemma buf_window_determines:
  fixes b1 b2 :: (('c :: enum) ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
  assumes ∀ i < 3 * card (UNIV :: 'c set). buf_lin_at b1 i = buf_lin_at b2 i
  shows b1 = b2
proof −
  obtain l1 h1 r1 where b1: b1 = (l1, h1, r1) by (cases b1)
```

```
  obtain l2 h2 r2 where b2: b2 = (l2, h2, r2) by (cases b2)
  let ?c = card (UNIV :: 'c set)
  have lE: l1 = l2
  proof (rule ext)
    fix x :: 'c
    have idx: c_idx x < ?c by (rule c_idx_lt_card)
    hence lt: c_idx x < 3 * ?c by simp
    have v1: buf_lin_at b1 (c_idx x) = l1 x
      using idx by (simp add: b1 buf_lin_at_def Let_def c_idx_in_range(2))
    have v2: buf_lin_at b2 (c_idx x) = l2 x
      using idx by (simp add: b2 buf_lin_at_def Let_def c_idx_in_range(2))
    from assms lt have buf_lin_at b1 (c_idx x) = buf_lin_at b2 (c_idx x) by
blast
    with v1 v2 show l1 x = l2 x by simp
  qed
  have hE: h1 = h2
  proof (rule ext)
    fix x :: 'c
    have idx: c_idx x < ?c by (rule c_idx_lt_card)
    hence lt: ?c + c_idx x < 3 * ?c by linarith
    have v1: buf_lin_at b1 (?c + c_idx x) = h1 x
      using idx by (simp add: b1 buf_lin_at_def Let_def c_idx_in_range(2))
    have v2: buf_lin_at b2 (?c + c_idx x) = h2 x
      using idx by (simp add: b2 buf_lin_at_def Let_def c_idx_in_range(2))
    from assms lt have buf_lin_at b1 (?c + c_idx x) = buf_lin_at b2 (?c +
c_idx x) by blast
    with v1 v2 show h1 x = h2 x by simp
  qed
  have rE: r1 = r2
  proof (rule ext)
    fix x :: 'c
    have idx: c_idx x < ?c by (rule c_idx_lt_card)
    hence lt: 2 * ?c + c_idx x < 3 * ?c by linarith
    have v1: buf_lin_at b1 (2 * ?c + c_idx x) = r1 x
      using idx by (simp add: b1 buf_lin_at_def Let_def c_idx_in_range(2))
    have v2: buf_lin_at b2 (2 * ?c + c_idx x) = r2 x
      using idx by (simp add: b2 buf_lin_at_def Let_def c_idx_in_range(2))
    from assms lt have buf_lin_at b1 (2 * ?c + c_idx x) = buf_lin_at b2 (2 *
?c + c_idx x) by blast
    with v1 v2 show r1 x = r2 x by simp
  qed
  show ?thesis using b1 b2 lE hE rE by simp
qed
```

Per-block reads of *buf_lin_at*: the three thirds of the window read the left / home / right block respectively.

```
lemma buf_lin_at_l:
  fixes b :: (('c :: enum) ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
  assumes i < card (UNIV :: 'c set)
```

shows *buf_lin_at b i = fst b ((enum_class.enum :: 'c list) ! i)*
using *assms* **by** (*simp add*: *buf_lin_at_def case_prod_beta Let_def*)

lemma *buf_lin_at_h*:
fixes *b* :: (('*c* :: *enum*) ⇒ '*a*) × ('*c* ⇒ '*a*) × ('*c* ⇒ '*a*)
assumes *card* (*UNIV* :: '*c set*) ≤ *i* **and** *i* < *2* ∗ *card* (*UNIV* :: '*c set*)
shows *buf_lin_at b i = fst* (*snd b*) ((*enum_class.enum* :: '*c list*) ! (*i* − *card* (*UNIV* :: '*c set*)))
using *assms* **by** (*simp add*: *buf_lin_at_def case_prod_beta Let_def*)

lemma *buf_lin_at_r*:
fixes *b* :: (('*c* :: *enum*) ⇒ '*a*) × ('*c* ⇒ '*a*) × ('*c* ⇒ '*a*)
assumes *2* ∗ *card* (*UNIV* :: '*c set*) ≤ *i* **and** *i* < *3* ∗ *card* (*UNIV* :: '*c set*)
shows *buf_lin_at b i = snd* (*snd b*) ((*enum_class.enum* :: '*c list*) ! (*i* − *2* ∗ *card* (*UNIV* :: '*c set*)))
using *assms* **by** (*simp add*: *buf_lin_at_def case_prod_beta Let_def*)

Per-block determination from window reads over the block's third.

lemma *buf_left_window*:
fixes *b1 b2* :: (('*c* :: *enum*) ⇒ '*a*) × ('*c* ⇒ '*a*) × ('*c* ⇒ '*a*)
assumes ∀ *i* < *card* (*UNIV* :: '*c set*). *buf_lin_at b1 i = buf_lin_at b2 i*
shows *fst b1 = fst b2*
proof (*rule ext*)
fix *x* :: '*c*
have *idx*: *c_idx x* < *card* (*UNIV* :: '*c set*) **by** (*rule c_idx_lt_card*)
have *fst b1 x = buf_lin_at b1* (*c_idx x*)
using *idx* **by** (*simp add*: *buf_lin_at_l c_idx_in_range*(*2*))
also have ... = *buf_lin_at b2* (*c_idx x*) **using** *assms idx* **by** *blast*
also have ... = *fst b2 x*
using *idx* **by** (*simp add*: *buf_lin_at_l c_idx_in_range*(*2*))
finally show *fst b1 x = fst b2 x* **.**
qed

lemma *buf_home_window*:
fixes *b1 b2* :: (('*c* :: *enum*) ⇒ '*a*) × ('*c* ⇒ '*a*) × ('*c* ⇒ '*a*)
assumes ∀ *i*. *card* (*UNIV* :: '*c set*) ≤ *i* ∧ *i* < *2* ∗ *card* (*UNIV* :: '*c set*)
⟶ *buf_lin_at b1 i = buf_lin_at b2 i*
shows *fst* (*snd b1*) = *fst* (*snd b2*)
proof (*rule ext*)
fix *x* :: '*c*
let *?c* = *card* (*UNIV* :: '*c set*)
have *idx*: *c_idx x* < *?c* **by** (*rule c_idx_lt_card*)
have *lo*: *?c* ≤ *?c* + *c_idx x* **by** *simp*
have *hi*: *?c* + *c_idx x* < *2* ∗ *?c* **using** *idx* **by** *linarith*
have *fst* (*snd b1*) *x = buf_lin_at b1* (*?c* + *c_idx x*)
using *lo hi* **by** (*simp add*: *buf_lin_at_h c_idx_in_range*(*2*))
also have ... = *buf_lin_at b2* (*?c* + *c_idx x*) **using** *assms lo hi* **by** *blast*
also have ... = *fst* (*snd b2*) *x*
using *lo hi* **by** (*simp add*: *buf_lin_at_h c_idx_in_range*(*2*))

```
  finally show fst (snd b1) x = fst (snd b2) x .
qed

lemma buf_right_window:
  fixes b1 b2 :: (('c :: enum) ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a)
  assumes ∀ i. 2 * card (UNIV :: 'c set) ≤ i ∧ i < 3 * card (UNIV :: 'c set)
                ⟶ buf_lin_at b1 i = buf_lin_at b2 i
  shows snd (snd b1) = snd (snd b2)
proof (rule ext)
  fix x :: 'c
  let ?c = card (UNIV :: 'c set)
  have idx: c_idx x < ?c by (rule c_idx_lt_card)
  have lo: 2 * ?c ≤ 2 * ?c + c_idx x by simp
  have hi: 2 * ?c + c_idx x < 3 * ?c using idx by linarith
  have snd (snd b1) x = buf_lin_at b1 (2 * ?c + c_idx x)
    using lo hi by (simp add: buf_lin_at_r c_idx_in_range(2))
  also have ... = buf_lin_at b2 (2 * ?c + c_idx x) using assms lo hi by blast
  also have ... = snd (snd b2) x
    using lo hi by (simp add: buf_lin_at_r c_idx_in_range(2))
  finally show snd (snd b1) x = snd (snd b2) x .
qed
```

Full-window inversion ($pos \geq 2$): the head and the window together determine both the boundary position and the buffer.

```
lemma ae_window_invariant_determines:
  fixes tM :: nat ⇒ 'a
  assumes h1: ae_window_invariant tM nM bp1 b1 p_start
      and h2: ae_window_invariant tM nM bp2 b2 p_start
  shows bp1 = (bp2 :: ('c :: enum) bp) ∧ b1 = b2
proof
  from h1 h2 have nM = p_start + bp_linear bp1 nM = p_start + bp_linear
bp2
    unfolding ae_window_invariant_def by simp_all
  hence bp_linear bp1 = bp_linear bp2 by simp
  thus bp1 = bp2 by (rule bp_linear_inj)
next
  from h1 h2 have
      ∀ i < 3 * card (UNIV :: 'c set). tM (p_start + i) = buf_lin_at b1 i
      ∀ i < 3 * card (UNIV :: 'c set). tM (p_start + i) = buf_lin_at b2 i
    unfolding ae_window_invariant_def by simp_all
  hence ∀ i < 3 * card (UNIV :: 'c set). buf_lin_at b1 i = buf_lin_at b2 i by
simp
  thus b1 = b2 by (rule buf_window_determines)
qed
```

Boundary-position *bp* inversions: the head nM determines the position even in the *pos = 1 / pos = 0* cases, where the encoding is a direction-case rather than a single *bp_linear* equation. The three directions still land nM in disjoint ranges.

```
lemma bp_le1_inj:
  fixes bp1 bp2 :: ('c :: enum) bp and nM :: nat
  assumes A1: case fst bp1 of AE_Left ⇒ snd bp1 = c_last ∧ nM = 0
                | AE_Home ⇒ nM = Suc (c_idx (snd bp1))
                | AE_Right ⇒ nM = Suc (card (UNIV :: 'c set) + c_idx (snd bp1))
      and A2: case fst bp2 of AE_Left ⇒ snd bp2 = c_last ∧ nM = 0
                | AE_Home ⇒ nM = Suc (c_idx (snd bp2))
                | AE_Right ⇒ nM = Suc (card (UNIV :: 'c set) + c_idx (snd bp2))
  shows bp1 = bp2
proof −
  obtain d1 x1 where p1: bp1 = (d1, x1) by (cases bp1)
  obtain d2 x2 where p2: bp2 = (d2, x2) by (cases bp2)
  have b1: c_idx x1 < card (UNIV :: 'c set) by (rule c_idx_lt_card)
  have b2: c_idx x2 < card (UNIV :: 'c set) by (rule c_idx_lt_card)
  have d1 = d2 ∧ x1 = x2
    using A1[unfolded p1] A2[unfolded p2] b1 b2
    by (cases d1; cases d2; simp; (linarith | blast intro: c_idx_inj))
  thus ?thesis using p1 p2 by simp
qed
```

Full inversion at *pos = 1*: every direction constrains the cell here, so the head pins *bp*; the window plus the fixed left block (*LE_block le*) pin all three blocks.

```
lemma ae_window_invariant_le1_determines:
  fixes tM :: nat ⇒ 'a
  assumes h1: ae_window_invariant_le1 tM nM bp1 b1 le
      and h2: ae_window_invariant_le1 tM nM bp2 b2 le
  shows bp1 = (bp2 :: ('c :: enum) bp) ∧ b1 = b2
proof −
  let ?c = card (UNIV :: 'c set)
  have a1: case fst bp1 of AE_Left ⇒ snd bp1 = c_last ∧ nM = 0
             | AE_Home ⇒ nM = Suc (c_idx (snd bp1))
             | AE_Right ⇒ nM = Suc (?c + c_idx (snd bp1))
    using h1 unfolding ae_window_invariant_le1_def by simp
  have a2: case fst bp2 of AE_Left ⇒ snd bp2 = c_last ∧ nM = 0
             | AE_Home ⇒ nM = Suc (c_idx (snd bp2))
             | AE_Right ⇒ nM = Suc (?c + c_idx (snd bp2))
    using h2 unfolding ae_window_invariant_le1_def by simp
  have bp: bp1 = bp2 by (rule bp_le1_inj[OF a1 a2])
  have lefts: fst b1 = LE_block le fst b2 = LE_block le
    using h1 h2 unfolding ae_window_invariant_le1_def by simp_all
  have e1: ∀ i<2 * ?c. tM (Suc i) = buf_lin_at b1 (?c + i)
    using h1 unfolding ae_window_invariant_le1_def by simp
  have e2: ∀ i<2 * ?c. tM (Suc i) = buf_lin_at b2 (?c + i)
    using h2 unfolding ae_window_invariant_le1_def by simp
  have home: fst (snd b1) = fst (snd b2)
  proof (rule buf_home_window, intro allI impI)
    fix i assume ?c ≤ i ∧ i < 2 * ?c
    then have i1: ?c ≤ i and i2: i < 2 * ?c by auto
```

```
    from i2 have lt: i − ?c < 2 ∗ ?c by linarith
    from i1 have eq: ?c + (i − ?c) = i by linarith
    from e1 lt have tM (Suc (i − ?c)) = buf_lin_at b1 (?c + (i − ?c)) by blast
    with eq have A: tM (Suc (i − ?c)) = buf_lin_at b1 i by simp
    from e2 lt have tM (Suc (i − ?c)) = buf_lin_at b2 (?c + (i − ?c)) by blast
    with eq have B: tM (Suc (i − ?c)) = buf_lin_at b2 i by simp
    from A B show buf_lin_at b1 i = buf_lin_at b2 i by simp
  qed
  have right: snd (snd b1) = snd (snd b2)
  proof (rule buf_right_window, intro allI impI)
    fix i assume 2 ∗ ?c ≤ i ∧ i < 3 ∗ ?c
    then have i1: 2 ∗ ?c ≤ i and i2: i < 3 ∗ ?c by auto
    from i2 have lt: i − ?c < 2 ∗ ?c by linarith
    from i1 have eq: ?c + (i − ?c) = i by linarith
    from e1 lt have tM (Suc (i − ?c)) = buf_lin_at b1 (?c + (i − ?c)) by blast
    with eq have A: tM (Suc (i − ?c)) = buf_lin_at b1 i by simp
    from e2 lt have tM (Suc (i − ?c)) = buf_lin_at b2 (?c + (i − ?c)) by blast
    with eq have B: tM (Suc (i − ?c)) = buf_lin_at b2 i by simp
    from A B show buf_lin_at b1 i = buf_lin_at b2 i by simp
  qed
  have b1 = b2 using lefts home right by (cases b1; cases b2) auto
  with bp show ?thesis by simp
qed
```

General window inversion (all *pos*). The home and right blocks are determined by the tape window at every *pos*. For $pos \neq 0$ the boundary position *bp* and the whole buffer are determined. At $pos = 0$ the left block and the home offset of *bp* are off-trace (the absent left neighbour of the endmarker, and $AE_Home \Longrightarrow nM = 0$ says nothing of the cell); they are carried from the source by the buffered simulation, not pinned by the window.

```
lemma ae_window_invariant_general_determines:
  fixes tM :: nat ⇒ 'a
  assumes h1: ae_window_invariant_general tM nM bp1 b1 pos le
      and h2: ae_window_invariant_general tM nM bp2 b2 pos le
  shows fst (snd b1) = fst (snd (b2 :: (('c :: enum) ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒
'a)))
        ∧ snd (snd b1) = snd (snd b2)
        ∧ (pos ≠ 0 ⟶ bp1 = bp2 ∧ b1 = b2)
proof (cases pos ≥ 2)
  case True
  have w1: ae_window_invariant tM nM bp1 b1 ((pos − 2) ∗ card (UNIV :: 'c
set) + 1)
    using h1 True unfolding ae_window_invariant_general_def by simp
  have w2: ae_window_invariant tM nM bp2 b2 ((pos − 2) ∗ card (UNIV :: 'c
set) + 1)
    using h2 True unfolding ae_window_invariant_general_def by simp
  from ae_window_invariant_determines[OF w1 w2]
  have bp1 = bp2 and b1 = b2 by simp_all
```

```
  thus ?thesis by simp
next
  case False
  show ?thesis
  proof (cases pos = 1)
    case True
    have l1: ae_window_invariant_le1 tM nM bp1 b1 le
      using h1 True unfolding ae_window_invariant_general_def by simp
    have l2: ae_window_invariant_le1 tM nM bp2 b2 le
      using h2 True unfolding ae_window_invariant_general_def by simp
    from ae_window_invariant_le1_determines[OF l1 l2]
    have bp1 = bp2 and b1 = b2 by simp_all
    thus ?thesis by simp
  next
    case False
    with ‹¬ 2 ≤ pos› have pos0: pos = 0 by simp
    have l1: ae_window_invariant_le0 tM nM bp1 b1 le
      using h1 pos0 unfolding ae_window_invariant_general_def by simp
    have l2: ae_window_invariant_le0 tM nM bp2 b2 le
      using h2 pos0 unfolding ae_window_invariant_general_def by simp
    let ?c = card (UNIV :: 'c set)
    have homes: fst (snd b1) = LE_block le fst (snd b2) = LE_block le
      using l1 l2 unfolding ae_window_invariant_le0_def by simp_all
    have f1: ∀ i<?c. tM (Suc i) = buf_lin_at b1 (2 * ?c + i)
      using l1 unfolding ae_window_invariant_le0_def by simp
    have f2: ∀ i<?c. tM (Suc i) = buf_lin_at b2 (2 * ?c + i)
      using l2 unfolding ae_window_invariant_le0_def by simp
    have right: snd (snd b1) = snd (snd b2)
    proof (rule buf_right_window, intro allI impI)
      fix i assume 2 * ?c ≤ i ∧ i < 3 * ?c
      then have i1: 2 * ?c ≤ i and i2: i < 3 * ?c by auto
      from i2 have lt: i − 2 * ?c < ?c by linarith
      from i1 have eq: 2 * ?c + (i − 2 * ?c) = i by linarith
      from f1 lt have tM (Suc (i − 2 * ?c)) = buf_lin_at b1 (2 * ?c + (i − 2 *
?c)) by blast
      with eq have A: tM (Suc (i − 2 * ?c)) = buf_lin_at b1 i by simp
      from f2 lt have tM (Suc (i − 2 * ?c)) = buf_lin_at b2 (2 * ?c + (i − 2 *
?c)) by blast
      with eq have B: tM (Suc (i − 2 * ?c)) = buf_lin_at b2 i by simp
      from A B show buf_lin_at b1 i = buf_lin_at b2 i by simp
    qed
    from homes right pos0 show ?thesis by auto
  qed
qed
```

Per-tape lift of the window inversion: across the active tapes $k < K$, two buffered outputs that both window-correspond to the same per-tape trace endpoint agree on the home + right blocks at every *pos*, and fully (boundary position and whole buffer) where *pos k* $\neq$ *0*. This is the form consumed by

the config-level SS4→SS5 inversion: the SS4→SS5 state's per-tape buffer / boundary components are exactly such families.

lemma *ae_window_general_indexed_determines*:
  fixes *tM* :: *nat* ⇒ *nat* ⇒ *'a* **and** *nM pos* :: *nat* ⇒ *nat*
    and *bp1 bp2* :: *nat* ⇒ (*'c* :: *enum*) *bp*
    and *b1 b2* :: *nat* ⇒ ((*'c* ⇒ *'a*) × (*'c* ⇒ *'a*) × (*'c* ⇒ *'a*))
    and *le* :: *'a* **and** *K* :: *nat*
  assumes *h1*: ∀ *k*<*K*. *ae_window_invariant_general* (*tM k*) (*nM k*)
                    (*bp1 k*) (*b1 k*) (*pos k*) *le*
      and *h2*: ∀ *k*<*K*. *ae_window_invariant_general* (*tM k*) (*nM k*)
                    (*bp2 k*) (*b2 k*) (*pos k*) *le*
  shows ∀ *k*<*K*. *fst* (*snd* (*b1 k*)) = *fst* (*snd* (*b2 k*))
            ∧ *snd* (*snd* (*b1 k*)) = *snd* (*snd* (*b2 k*))
            ∧ (*pos k* ≠ *0* ⟶ *bp1 k* = *bp2 k* ∧ *b1 k* = *b2 k*)
proof (*intro allI impI*)
  fix *k* **assume** *k*: *k* < *K*
  from *h1 k* **have** *g1*: *ae_window_invariant_general* (*tM k*) (*nM k*)
                    (*bp1 k*) (*b1 k*) (*pos k*) *le* **by** *blast*
  from *h2 k* **have** *g2*: *ae_window_invariant_general* (*tM k*) (*nM k*)
                    (*bp2 k*) (*b2 k*) (*pos k*) *le* **by** *blast*
  show *fst* (*snd* (*b1 k*)) = *fst* (*snd* (*b2 k*))
        ∧ *snd* (*snd* (*b1 k*)) = *snd* (*snd* (*b2 k*))
        ∧ (*pos k* ≠ *0* ⟶ *bp1 k* = *bp2 k* ∧ *b1 k* = *b2 k*)
    by (*rule ae_window_invariant_general_determines*[*OF g1 g2*])
qed

2.29.5 SS4→SS5 trace-functional

The det-free replacement for *ae_delta_ss4_ss5_functional*. The SS4→SS5 macro-step's nondeterminism is confined to the state's buffer / boundary-position components (the *m_steps_buffered* witness); the tape and head movements are functions of the source alone. This first lemma isolates that: two SS4→SS5 steps from a *common* source have identical tape and identical head positions, unconditionally (no trace correspondence needed).

Reading the *ae_delta_ss4_ss5* definition: the write symbol equals the read symbol (the step rewrites nothing, it only moves heads), and the per-tape direction *d* is *if a k = LE_block le then N else L* on active tapes and *N* beyond the tape count — a function of the read tuple *a* = λ*k*. *ts k* (*n k*), which the shared source fixes.

lemma *ae_ss4_ss5_shared_source_tape*:
  fixes *M* :: (*'q*, *'a*) *mttm*
    and *c4 c5a c5b* :: (*'c* :: *enum* ⇒ *'a*,
                  'q × (*'a*, *'c*) *ae_stage*) *mt_config*
  assumes *stepa*: (*c4*, *c5a*) ∈ *mttm_step* (*ae_delta_ss4_ss5 M*)
      and *stepb*: (*c4*, *c5b*) ∈ *mttm_step* (*ae_delta_ss4_ss5 M*)
  shows *mt_tape c5a* = *mt_tape c5b* ∧ *mt_pos c5a* = *mt_pos c5b*
proof −

obtain *s ts n* **where** *c4_eq*: *c4* = $Config_M$ *s ts n* **by** (*cases c4*)
from *stepa c4_eq* **obtain** *s'a a'a da* **where**
a_eq: *c5a* = $Config_M$ *s'a* (λ*k*. (*ts k*)(*n k* := *a'a k*))
(λ*k*. *go_dir* (*da k*) (*n k*))
and *rela*: (*s*, (λ*k*. *ts k* (*n k*)), *s'a*, *a'a*, *da*) ∈ *ae_delta_ss4_ss5 M*
by (*auto elim*: *mttm_step.cases*)
from *stepb c4_eq* **obtain** *s'b a'b db* **where**
b_eq: *c5b* = $Config_M$ *s'b* (λ*k*. (*ts k*)(*n k* := *a'b k*))
(λ*k*. *go_dir* (*db k*) (*n k*))
and *relb*: (*s*, (λ*k*. *ts k* (*n k*)), *s'b*, *a'b*, *db*) ∈ *ae_delta_ss4_ss5 M*
by (*auto elim*: *mttm_step.cases*)
— Write equals read and direction is canonical in the read, for both steps; hence both match the shared source.
have *wa*: *a'a* = (λ*k*. *ts k* (*n k*))
and *dda*: *da* = (λ*k*. *if k* < *k_tm M*
then (*if ts k* (*n k*) = *LE_block* (*le_tm M*)
then dir.N else dir.L)
else dir.N)
using *rela* **unfolding** *ae_delta_ss4_ss5_def* **by** *auto*
have *wb*: *a'b* = (λ*k*. *ts k* (*n k*))
and *ddb*: *db* = (λ*k*. *if k* < *k_tm M*
then (*if ts k* (*n k*) = *LE_block* (*le_tm M*)
then dir.N else dir.L)
else dir.N)
using *relb* **unfolding** *ae_delta_ss4_ss5_def* **by** *auto*
show *?thesis* **using** *a_eq b_eq wa dda wb ddb* **by** *simp*
qed

2.29.6 SS5→SS8 blindness to the pos=0 residual

At *pos* = *0* the home block is uniformly *LE_block le* (= (λ_. *le*); window *le0*). Every SS5→SS8 action then routes through its *h* = *LE_block le* branch, which reads only the current cell *a*, the right block *r*, and the boundary direction *ds* — never the left block *l*. And the actions never take the boundary *offset* as an argument at all. So the two components the window inversion leaves free at *pos* = *0* (the left block and the *AE_Home* offset) are invisible to SS5→SS8: no source-carrying induction is needed.

lemma *ae_ss5_action_indep_left*:
ae_ss5_action le a (*l*, *LE_block le*, *r*) *ds*
= *ae_ss5_action le a* (*l'*, *LE_block le*, *r*) *ds*
by *simp*

lemma *ae_ss6_action_indep_left*:
ae_ss6_action le a (*l*, *LE_block le*, *r*) *ds*
= *ae_ss6_action le a* (*l'*, *LE_block le*, *r*) *ds*
by *simp*

lemma *ae_ss7_action_indep_left*:

```
  ae_ss7_action le a (l, LE_block le, r) ds
    = ae_ss7_action le a (l', LE_block le, r) ds
  by simp

lemma ae_ss8_action_indep_left:
  ae_ss8_action le a (l, LE_block le, r) ds
    = ae_ss8_action le a (l', LE_block le, r) ds
  by simp
```

The boundary *direction* (the *dest* the actions read) does agree at $pos = 0$: against a fixed head *nM*, *le0* determines *fst bp* (*AE_Home* iff $nM = 0$, else *AE_Right*; never *AE_Left*), and the whole *bp* except the *AE_Home* offset (the one genuine pos=0 don't-care).

```
lemma ae_window_le0_dest_agree:
  fixes bp1 bp2 :: ('c :: enum) bp
  assumes h1: ae_window_invariant_le0 tM nM bp1 b1 le
      and h2: ae_window_invariant_le0 tM nM bp2 b2 le
  shows fst bp1 = fst bp2 ∧ (fst bp1 ≠ AE_Home ⟶ bp1 = bp2)
proof −
  obtain d1 x1 where p1: bp1 = (d1, x1) by (cases bp1)
  obtain d2 x2 where p2: bp2 = (d2, x2) by (cases bp2)
  have key1: (d1 = AE_Home ⟶ nM = 0)
             ∧ (d1 = AE_Right ⟶ nM = Suc (c_idx x1)) ∧ d1 ≠ AE_Left
    using h1[unfolded ae_window_invariant_le0_def p1] by (cases d1) auto
  have key2: (d2 = AE_Home ⟶ nM = 0)
             ∧ (d2 = AE_Right ⟶ nM = Suc (c_idx x2)) ∧ d2 ≠ AE_Left
    using h2[unfolded ae_window_invariant_le0_def p2] by (cases d2) auto
  have dir: d1 = d2
  proof (cases nM = 0)
    case True
    have d1 = AE_Home using key1 True by (cases d1) auto
    moreover have d2 = AE_Home using key2 True by (cases d2) auto
    ultimately show ?thesis by simp
  next
    case False
    have d1 = AE_Right using key1 False by (cases d1) auto
    moreover have d2 = AE_Right using key2 False by (cases d2) auto
    ultimately show ?thesis by simp
  qed
  have full: d1 ≠ AE_Home ⟶ x1 = x2
  proof
    assume nh: d1 ≠ AE_Home
    have r1: d1 = AE_Right using key1 nh by (cases d1) auto
    have nM = Suc (c_idx x1) using key1 r1 by simp
    moreover have nM = Suc (c_idx x2) using key2 dir r1 by simp
    ultimately have c_idx x1 = c_idx x2 by simp
    thus x1 = x2 by (rule c_idx_inj)
  qed
  show ?thesis using dir full p1 p2 by simp
```

qed

The per-block agreement the SS5→SS8 actions are congruent under: home and right blocks agree, and the left block agrees *where the home block is not LE_block le* (at the *LE_block* home — the pos=0 case — the actions never read the left block, so it may differ). Both clauses are delivered at every tape by the window inversion + *le0*.

definition *ae_blk_agree* ::
  $'a \Rightarrow (('c :: enum \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a))$
      $\Rightarrow (('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)) \Rightarrow bool$ **where**
  ae_blk_agree le b1 b2 $\longleftrightarrow$
    $fst\ (snd\ b1) = fst\ (snd\ b2)$
    $\wedge\ snd\ (snd\ b1) = snd\ (snd\ b2)$
    $\wedge\ (fst\ (snd\ b1) \neq LE_block\ le \longrightarrow fst\ b1 = fst\ b2)$

lemma *ae_blk_agree_action_eq*:
  assumes *ae_blk_agree le b1 b2*
  shows $b1 = b2 \vee (fst\ (snd\ b1) = LE_block\ le \wedge fst\ (snd\ b2) = LE_block\ le$
                $\wedge\ snd\ (snd\ b1) = snd\ (snd\ b2))$
proof (*cases* $fst\ (snd\ b1) = LE_block\ le$)
  case *False*
  hence $fst\ b1 = fst\ b2$ **using** *assms* **unfolding** *ae_blk_agree_def* **by** *simp*
  moreover have $fst\ (snd\ b1) = fst\ (snd\ b2)$ $snd\ (snd\ b1) = snd\ (snd\ b2)$
    using *assms* **unfolding** *ae_blk_agree_def* **by** *simp_all*
  ultimately have $b1 = b2$ **by** (*cases b1*; *cases b2*) *auto*
  thus *?thesis* **by** *simp*
next
  case *True*
  thus *?thesis* **using** *assms* **unfolding** *ae_blk_agree_def* **by** *simp*
qed

lemma *ae_ss5_action_cong*:
  assumes *ae_blk_agree le b1 b2*
  shows *ae_ss5_action le a b1 ds* = *ae_ss5_action le a b2 ds*
  using *ae_blk_agree_action_eq*[*OF assms*]
proof
  assume $b1 = b2$ **thus** *?thesis* **by** *simp*
next
  assume $fst\ (snd\ b1) = LE_block\ le \wedge fst\ (snd\ b2) = LE_block\ le$
          $\wedge\ snd\ (snd\ b1) = snd\ (snd\ b2)$
  thus *?thesis*
    by (*cases b1*; *cases b2*)
      (*simp add*: *ae_ss5_action_indep_left*)
qed

lemma *ae_ss6_action_cong*:
  assumes *ae_blk_agree le b1 b2*
  shows *ae_ss6_action le a b1 ds* = *ae_ss6_action le a b2 ds*
  using *ae_blk_agree_action_eq*[*OF assms*]

```
proof
  assume b1 = b2 thus ?thesis by simp
next
  assume fst (snd b1) = LE_block le ∧ fst (snd b2) = LE_block le
         ∧ snd (snd b1) = snd (snd b2)
  thus ?thesis
    by (cases b1; cases b2)
       (simp add: ae_ss6_action_indep_left)
qed

lemma ae_ss7_action_cong:
  assumes ae_blk_agree le b1 b2
  shows ae_ss7_action le a b1 ds = ae_ss7_action le a b2 ds
  using ae_blk_agree_action_eq[OF assms]
proof
  assume b1 = b2 thus ?thesis by simp
next
  assume fst (snd b1) = LE_block le ∧ fst (snd b2) = LE_block le
         ∧ snd (snd b1) = snd (snd b2)
  thus ?thesis
    by (cases b1; cases b2)
       (simp add: ae_ss7_action_indep_left)
qed

lemma ae_ss8_action_cong:
  assumes ae_blk_agree le b1 b2
  shows ae_ss8_action le a b1 ds = ae_ss8_action le a b2 ds
  using ae_blk_agree_action_eq[OF assms]
proof
  assume b1 = b2 thus ?thesis by simp
next
  assume fst (snd b1) = LE_block le ∧ fst (snd b2) = LE_block le
         ∧ snd (snd b1) = snd (snd b2)
  thus ?thesis
    by (cases b1; cases b2)
       (simp add: ae_ss8_action_indep_left)
qed
```

Config-level SS5→SS6 congruence: two configs that agree on tape, head positions, state q and *dest*, and whose buffers are *ae_blk_agree* per tape (offsets / buffers may otherwise differ), take an SS5→SS6 step to configs that again agree on tape and head positions, with the state advanced to SS6 and q / *buf* / *dest* carried unchanged. The reads coincide (tape + pos equal), so the action-congruence makes the writes and moves coincide.

```
lemma ae_ss5_ss6_cong:
  fixes M :: ('q, 'a) mttm
    and c1 c2 c1' c2' :: ('c :: enum ⇒ 'a,
                          'q × ('a, 'c) ae_stage) mt_config
  assumes tape: mt_tape c1 = mt_tape c2
```

and *pos*: *mt_pos c1 = mt_pos c2*
and *st1*: *mt_state c1 = (q, ofs1, buf1, dest, SS5)*
and *st2*: *mt_state c2 = (q, ofs2, buf2, dest, SS5)*
and *blk*: $\forall k.$ *ae_blk_agree (le_tm M) (buf1 k) (buf2 k)*
and *s1*: *(c1, c1′) ∈ mttm_step (ae_delta_ss5_ss6 M)*
and *s2*: *(c2, c2′) ∈ mttm_step (ae_delta_ss5_ss6 M)*
shows *mt_tape c1′ = mt_tape c2′ ∧ mt_pos c1′ = mt_pos c2′*
∧ mt_state c1′ = (q, ofs1, buf1, dest, SS6)
∧ mt_state c2′ = (q, ofs2, buf2, dest, SS6)
proof −
obtain *ts n* **where** *c1d*: *c1 = Config$_M$ (q, ofs1, buf1, dest, SS5) ts n*
using *st1* **by** *(cases c1) auto*
have *c2d*: *c2 = Config$_M$ (q, ofs2, buf2, dest, SS5) ts n*
using *st2 tape pos c1d* **by** *(cases c2) auto*
from *s1*[*unfolded c1d*] **obtain** *s1′ a′1 d1* **where**
c1′d: *c1′ = Config$_M$ s1′ (λk. (ts k)(n k := a′1 k))*
(λk. go_dir (d1 k) (n k))
and *rel1*: *((q, ofs1, buf1, dest, SS5), (λk. ts k (n k)),*
s1′, a′1, d1) ∈ ae_delta_ss5_ss6 M
by *(auto elim: mttm_step.cases)*
from *s2*[*unfolded c2d*] **obtain** *s2′ a′2 d2* **where**
c2′d: *c2′ = Config$_M$ s2′ (λk. (ts k)(n k := a′2 k))*
(λk. go_dir (d2 k) (n k))
and *rel2*: *((q, ofs2, buf2, dest, SS5), (λk. ts k (n k)),*
s2′, a′2, d2) ∈ ae_delta_ss5_ss6 M
by *(auto elim: mttm_step.cases)*
have *e1*: *s1′ = (q, ofs1, buf1, dest, SS6)*
and *a1*: *a′1 = (λk. if k < k_tm M*
then fst (ae_ss5_action (le_tm M) (ts k (n k))
(buf1 k) (dest k))
else bl_block (bl_tm M))
and *d1*: *d1 = (λk. if k < k_tm M*
then snd (ae_ss5_action (le_tm M) (ts k (n k))
(buf1 k) (dest k))
else dir.N)
using *rel1* **unfolding** *ae_delta_ss5_ss6_def* **by** *auto*
have *e2*: *s2′ = (q, ofs2, buf2, dest, SS6)*
and *a2*: *a′2 = (λk. if k < k_tm M*
then fst (ae_ss5_action (le_tm M) (ts k (n k))
(buf2 k) (dest k))
else bl_block (bl_tm M))
and *d2*: *d2 = (λk. if k < k_tm M*
then snd (ae_ss5_action (le_tm M) (ts k (n k))
(buf2 k) (dest k))
else dir.N)
using *rel2* **unfolding** *ae_delta_ss5_ss6_def* **by** *auto*
have *act*: $\bigwedge k.$ *ae_ss5_action (le_tm M) (ts k (n k)) (buf1 k) (dest k)*
= ae_ss5_action (le_tm M) (ts k (n k)) (buf2 k) (dest k)
by *(rule ae_ss5_action_cong*[*OF blk*[*rule_format*]])

```
  have a'1 = a'2
    by (rule ext) (simp add: a1 a2 act)
  moreover have d1 = d2
    by (rule ext) (simp add: d1 d2 act)
  ultimately show ?thesis using c1'd c2'd e1 e2 by simp
qed

lemma ae_ss6_ss7_cong:
  fixes M :: ('q, 'a) mttm
    and c1 c2 c1' c2' :: ('c :: enum ⇒ 'a,
                          'q × ('a, 'c) ae_stage) mt_config
  assumes tape: mt_tape c1 = mt_tape c2
      and pos:  mt_pos c1 = mt_pos c2
      and st1:  mt_state c1 = (q, ofs1, buf1, dest, SS6)
      and st2:  mt_state c2 = (q, ofs2, buf2, dest, SS6)
      and blk:  ∀ k. ae_blk_agree (le_tm M) (buf1 k) (buf2 k)
      and s1:   (c1, c1') ∈ mttm_step (ae_delta_ss6_ss7 M)
      and s2:   (c2, c2') ∈ mttm_step (ae_delta_ss6_ss7 M)
  shows mt_tape c1' = mt_tape c2' ∧ mt_pos c1' = mt_pos c2'
        ∧ mt_state c1' = (q, ofs1, buf1, dest, SS7)
        ∧ mt_state c2' = (q, ofs2, buf2, dest, SS7)
proof −
  obtain ts n where c1d: c1 = Config_M (q, ofs1, buf1, dest, SS6) ts n
    using st1 by (cases c1) auto
  have c2d: c2 = Config_M (q, ofs2, buf2, dest, SS6) ts n
    using st2 tape pos c1d by (cases c2) auto
  from s1[unfolded c1d] obtain s1' a'1 d1 where
      c1'd: c1' = Config_M s1' (λk. (ts k)(n k := a'1 k))
                               (λk. go_dir (d1 k) (n k))
    and rel1: ((q, ofs1, buf1, dest, SS6), (λk. ts k (n k)),
               s1', a'1, d1) ∈ ae_delta_ss6_ss7 M
    by (auto elim: mttm_step.cases)
  from s2[unfolded c2d] obtain s2' a'2 d2 where
      c2'd: c2' = Config_M s2' (λk. (ts k)(n k := a'2 k))
                               (λk. go_dir (d2 k) (n k))
    and rel2: ((q, ofs2, buf2, dest, SS6), (λk. ts k (n k)),
               s2', a'2, d2) ∈ ae_delta_ss6_ss7 M
    by (auto elim: mttm_step.cases)
  have e1: s1' = (q, ofs1, buf1, dest, SS7)
    and a1: a'1 = (λk. if k < k_tm M
                       then fst (ae_ss6_action (le_tm M) (ts k (n k))
                                               (buf1 k) (dest k))
                       else bl_block (bl_tm M))
    and d1: d1 = (λk. if k < k_tm M
                      then snd (ae_ss6_action (le_tm M) (ts k (n k))
                                              (buf1 k) (dest k))
                      else dir.N)
    using rel1 unfolding ae_delta_ss6_ss7_def by auto
  have e2: s2' = (q, ofs2, buf2, dest, SS7)
```

```
    and a2: a'2 = (λk. if k < k_tm M
                       then fst (ae_ss6_action (le_tm M) (ts k (n k))
                                 (buf2 k) (dest k))
                       else bl_block (bl_tm M))
    and d2: d2 = (λk. if k < k_tm M
                       then snd (ae_ss6_action (le_tm M) (ts k (n k))
                                 (buf2 k) (dest k))
                       else dir.N)
    using rel2 unfolding ae_delta_ss6_ss7_def by auto
  have act: ⋀k. ae_ss6_action (le_tm M) (ts k (n k)) (buf1 k) (dest k)
              = ae_ss6_action (le_tm M) (ts k (n k)) (buf2 k) (dest k)
    by (rule ae_ss6_action_cong[OF blk[rule_format]])
  have a'1 = a'2
    by (rule ext) (simp add: a1 a2 act)
  moreover have d1 = d2
    by (rule ext) (simp add: d1 d2 act)
  ultimately show ?thesis using c1'd c2'd e1 e2 by simp
qed

lemma ae_ss7_ss8_cong:
  fixes M :: ('q, 'a) mttm
    and c1 c2 c1' c2' :: ('c :: enum ⇒ 'a,
                          'q × ('a, 'c) ae_stage) mt_config
  assumes tape: mt_tape c1 = mt_tape c2
      and pos:  mt_pos c1 = mt_pos c2
      and st1:  mt_state c1 = (q, ofs1, buf1, dest, SS7)
      and st2:  mt_state c2 = (q, ofs2, buf2, dest, SS7)
      and blk:  ∀k. ae_blk_agree (le_tm M) (buf1 k) (buf2 k)
      and s1:   (c1, c1') ∈ mttm_step (ae_delta_ss7_ss8 M)
      and s2:   (c2, c2') ∈ mttm_step (ae_delta_ss7_ss8 M)
  shows mt_tape c1' = mt_tape c2' ∧ mt_pos c1' = mt_pos c2'
        ∧ mt_state c1' = (q, ofs1, buf1, dest, SS8)
        ∧ mt_state c2' = (q, ofs2, buf2, dest, SS8)
proof −
  obtain ts n where c1d: c1 = Config_M (q, ofs1, buf1, dest, SS7) ts n
    using st1 by (cases c1) auto
  have c2d: c2 = Config_M (q, ofs2, buf2, dest, SS7) ts n
    using st2 tape pos c1d by (cases c2) auto
  from s1[unfolded c1d] obtain s1' a'1 d1 where
      c1'd: c1' = Config_M s1' (λk. (ts k)(n k := a'1 k))
                               (λk. go_dir (d1 k) (n k))
    and rel1: ((q, ofs1, buf1, dest, SS7), (λk. ts k (n k)),
               s1', a'1, d1) ∈ ae_delta_ss7_ss8 M
    by (auto elim: mttm_step.cases)
  from s2[unfolded c2d] obtain s2' a'2 d2 where
      c2'd: c2' = Config_M s2' (λk. (ts k)(n k := a'2 k))
                               (λk. go_dir (d2 k) (n k))
    and rel2: ((q, ofs2, buf2, dest, SS7), (λk. ts k (n k)),
               s2', a'2, d2) ∈ ae_delta_ss7_ss8 M
```

```
  by (auto elim: mttm_step.cases)
 have e1: s1' = (q, ofs1, buf1, dest, SS8)
  and a1: a'1 = (λk. if k < k_tm M
                     then fst (ae_ss7_action (le_tm M) (ts k (n k))
                                 (buf1 k) (dest k))
                     else bl_block (bl_tm M))
  and d1: d1 = (λk. if k < k_tm M
                     then snd (ae_ss7_action (le_tm M) (ts k (n k))
                                 (buf1 k) (dest k))
                     else dir.N)
  using rel1 unfolding ae_delta_ss7_ss8_def by auto
 have e2: s2' = (q, ofs2, buf2, dest, SS8)
  and a2: a'2 = (λk. if k < k_tm M
                     then fst (ae_ss7_action (le_tm M) (ts k (n k))
                                 (buf2 k) (dest k))
                     else bl_block (bl_tm M))
  and d2: d2 = (λk. if k < k_tm M
                     then snd (ae_ss7_action (le_tm M) (ts k (n k))
                                 (buf2 k) (dest k))
                     else dir.N)
  using rel2 unfolding ae_delta_ss7_ss8_def by auto
 have act: ⋀k. ae_ss7_action (le_tm M) (ts k (n k)) (buf1 k) (dest k)
             = ae_ss7_action (le_tm M) (ts k (n k)) (buf2 k) (dest k)
  by (rule ae_ss7_action_cong[OF blk[rule_format]])
 have a'1 = a'2
  by (rule ext) (simp add: a1 a2 act)
 moreover have d1 = d2
  by (rule ext) (simp add: d1 d2 act)
 ultimately show ?thesis using c1'd c2'd e1 e2 by simp
qed
```

The closing SS8→SS1 step also resets *dest* to *init_dest* and the substep index to *SS1* (non-halt branch, $q \notin \{t, r\}$ — the case the reverse arm runs in). Otherwise as before: the action is congruent under *ae_blk_agree*, so tape and head positions coincide, and *q* / *ofs* / *buf* are carried.

```
lemma ae_ss8_ss1_cong:
 fixes M :: ('q, 'a) mttm
  and c1 c2 c1' c2' :: ('c :: enum ⇒ 'a,
                        'q × ('a, 'c) ae_stage) mt_config
 assumes tape: mt_tape c1 = mt_tape c2
    and pos:  mt_pos c1 = mt_pos c2
    and st1:  mt_state c1 = (q, ofs1, buf1, dest, SS8)
    and st2:  mt_state c2 = (q, ofs2, buf2, dest, SS8)
    and blk:  ∀k. ae_blk_agree (le_tm M) (buf1 k) (buf2 k)
    and qnh:  q ∉ {t_tm M, r_tm M}
    and s1:   (c1, c1') ∈ mttm_step (ae_delta_ss8_ss1 M)
    and s2:   (c2, c2') ∈ mttm_step (ae_delta_ss8_ss1 M)
 shows mt_tape c1' = mt_tape c2' ∧ mt_pos c1' = mt_pos c2'
       ∧ mt_state c1' = (q, ofs1, buf1, init_dest, SS1)
```

```
      ∧ mt_state c2' = (q, ofs2, buf2, init_dest, SS1)
proof −
  obtain ts n where c1d: c1 = Config_M (q, ofs1, buf1, dest, SS8) ts n
    using st1 by (cases c1) auto
  have c2d: c2 = Config_M (q, ofs2, buf2, dest, SS8) ts n
    using st2 tape pos c1d by (cases c2) auto
  from s1[unfolded c1d] obtain s1' a'1 d1 where
      c1'd: c1' = Config_M s1' (λk. (ts k)(n k := a'1 k))
                               (λk. go_dir (d1 k) (n k))
    and rel1: ((q, ofs1, buf1, dest, SS8), (λk. ts k (n k)),
               s1', a'1, d1) ∈ ae_delta_ss8_ss1 M
    by (auto elim: mttm_step.cases)
  from s2[unfolded c2d] obtain s2' a'2 d2 where
      c2'd: c2' = Config_M s2' (λk. (ts k)(n k := a'2 k))
                               (λk. go_dir (d2 k) (n k))
    and rel2: ((q, ofs2, buf2, dest, SS8), (λk. ts k (n k)),
               s2', a'2, d2) ∈ ae_delta_ss8_ss1 M
    by (auto elim: mttm_step.cases)
  have e1: s1' = (q, ofs1, buf1, init_dest, SS1)
    and a1: a'1 = (λk. if k < k_tm M
                       then fst (ae_ss8_action (le_tm M) (ts k (n k))
                                               (buf1 k) (dest k))
                       else bl_block (bl_tm M))
    and d1: d1 = (λk. if k < k_tm M
                      then snd (ae_ss8_action (le_tm M) (ts k (n k))
                                              (buf1 k) (dest k))
                      else dir.N)
    using rel1 qnh unfolding ae_delta_ss8_ss1_def by auto
  have e2: s2' = (q, ofs2, buf2, init_dest, SS1)
    and a2: a'2 = (λk. if k < k_tm M
                       then fst (ae_ss8_action (le_tm M) (ts k (n k))
                                               (buf2 k) (dest k))
                       else bl_block (bl_tm M))
    and d2: d2 = (λk. if k < k_tm M
                      then snd (ae_ss8_action (le_tm M) (ts k (n k))
                                              (buf2 k) (dest k))
                      else dir.N)
    using rel2 qnh unfolding ae_delta_ss8_ss1_def by auto
  have act: ⋀k. ae_ss8_action (le_tm M) (ts k (n k)) (buf1 k) (dest k)
              = ae_ss8_action (le_tm M) (ts k (n k)) (buf2 k) (dest k)
    by (rule ae_ss8_action_cong[OF blk[rule_format]])
  have a'1 = a'2
    by (rule ext) (simp add: a1 a2 act)
  moreover have d1 = d2
    by (rule ext) (simp add: d1 d2 act)
  ultimately show ?thesis using c1'd c2'd e1 e2 by simp
qed
```

Chaining the four congruences through the macro-cycle tail SS5→SS6→SS7→SS8→SS1: two SS5 configs agreeing on tape, head

positions, q, *dest*, and *ae_blk_agree* buffers reach SS1 boundary configs that agree on tape and head positions, with q and the (still *ae_blk_agree*) buffers carried and *dest* reset to *init_dest*. The buffers are carried unchanged, so the per-tape *ae_blk_agree* hypothesis is reused verbatim at each step.

```
lemma ae_ss5_ss8_segment_cong:
  fixes M :: ('q, 'a) mttm
    and c5a c5b ca cb :: ('c :: enum ⇒ 'a,
                          'q × ('a, 'c) ae_stage) mt_config
  assumes tape: mt_tape c5a = mt_tape c5b
      and pos:  mt_pos c5a = mt_pos c5b
      and st1:  mt_state c5a = (q, ofs1, buf1, dest, SS5)
      and st2:  mt_state c5b = (q, ofs2, buf2, dest, SS5)
      and blk:  ∀ k. ae_blk_agree (le_tm M) (buf1 k) (buf2 k)
      and qnh:  q ∉ {t_tm M, r_tm M}
      and a56:  (c5a, d6a) ∈ mttm_step (ae_delta_ss5_ss6 M)
      and a67:  (d6a, d7a) ∈ mttm_step (ae_delta_ss6_ss7 M)
      and a78:  (d7a, d8a) ∈ mttm_step (ae_delta_ss7_ss8 M)
      and a81:  (d8a, ca)  ∈ mttm_step (ae_delta_ss8_ss1 M)
      and b56:  (c5b, d6b) ∈ mttm_step (ae_delta_ss5_ss6 M)
      and b67:  (d6b, d7b) ∈ mttm_step (ae_delta_ss6_ss7 M)
      and b78:  (d7b, d8b) ∈ mttm_step (ae_delta_ss7_ss8 M)
      and b81:  (d8b, cb)  ∈ mttm_step (ae_delta_ss8_ss1 M)
  shows mt_tape ca = mt_tape cb ∧ mt_pos ca = mt_pos cb
        ∧ mt_state ca = (q, ofs1, buf1, init_dest, SS1)
        ∧ mt_state cb = (q, ofs2, buf2, init_dest, SS1)
proof −
  from ae_ss5_ss6_cong[OF tape pos st1 st2 blk a56 b56]
  have t6: mt_tape d6a = mt_tape d6b and p6: mt_pos d6a = mt_pos d6b
   and s6a: mt_state d6a = (q, ofs1, buf1, dest, SS6)
   and s6b: mt_state d6b = (q, ofs2, buf2, dest, SS6) by simp_all
  from ae_ss6_ss7_cong[OF t6 p6 s6a s6b blk a67 b67]
  have t7: mt_tape d7a = mt_tape d7b and p7: mt_pos d7a = mt_pos d7b
   and s7a: mt_state d7a = (q, ofs1, buf1, dest, SS7)
   and s7b: mt_state d7b = (q, ofs2, buf2, dest, SS7) by simp_all
  from ae_ss7_ss8_cong[OF t7 p7 s7a s7b blk a78 b78]
  have t8: mt_tape d8a = mt_tape d8b and p8: mt_pos d8a = mt_pos d8b
   and s8a: mt_state d8a = (q, ofs1, buf1, dest, SS8)
   and s8b: mt_state d8b = (q, ofs2, buf2, dest, SS8) by simp_all
  from ae_ss8_ss1_cong[OF t8 p8 s8a s8b blk qnh a81 b81]
  show ?thesis by simp
qed
```

The *ae_simulates* transfer (the congruence that closes the pos=0 don't-care). A reconstructed boundary config c''' with *ae_simulates M cM c'''* hands its simulation to a given c'' that agrees on tape, head positions, and state *except* the offset *ofs*, which need agree only where the simulated head *mt_pos cM k* is non-zero. At *mt_pos cM k = 0* the position decode

ae_decode_pos ignores the offset (it returns *0* exactly when its AE-head argument is *0*), so the offset residual is invisible. Both configs are at non-halt SS1, where *ae_simulates* reads neither *buf* nor *dest*.

```
lemma ae_decode_pos_eq_0:
  ae_decode_pos s i = 0 ⟹ s = 0
  by (simp add: ae_decode_pos_def split: if_splits)

lemma ae_decode_pos_zero_eq: ae_decode_pos 0 i = 0
  by (simp add: ae_decode_pos_def)

lemma ae_simulates_transfer:
  fixes M :: ('q, 'a) mttm and cM :: ('a, 'q) mt_config
    and c'' c''' :: ('c :: enum ⇒ 'a, 'q × ('a, 'c) ae_stage) mt_config
  assumes sim':   ae_simulates M cM c'''
      and cM_inQ: mt_state cM ∈ Q_tm M
      and cM_nt:  mt_state cM ≠ t_tm M
      and cM_nr:  mt_state cM ≠ r_tm M
      and tape:   mt_tape c'' = mt_tape c'''
      and pos:    mt_pos c'' = mt_pos c'''
      and st2:    mt_state c'' = (q, ofs1, buf1, init_dest, SS1)
      and st3:    mt_state c''' = (q, ofs2, buf2, init_dest, SS1)
      and qnh:    q ∉ {t_tm M, r_tm M}
      and ofsh:   ∀k<k_tm M. mt_pos cM k = 0 ∨ ofs1 k = ofs2 k
      and gam:    ae_tape_in_gamma_block M c''
  shows ae_simulates M cM c''
proof −
  obtain qM' ofs buf dest where
      ust: mt_state c''' = (qM', ofs, buf, dest, SS1)
    and qeq: mt_state cM = qM'
    and corr: ∀k<k_tm M. ae_tape_correspondence (le_tm M)
                          (mt_tape cM k) (mt_tape c''' k)
    and dec0: ∀k<k_tm M. mt_pos cM k
                          = ae_decode_pos (mt_pos c''' k) (ofs k)
    by (rule ae_forward_stage_unpack_sim[OF sim' cM_inQ cM_nt cM_nr])
  from ust st3 have qM'eq: qM' = q and ofeq: ofs = ofs2 by auto
  have dec: ∀k<k_tm M. mt_pos cM k
                        = ae_decode_pos (mt_pos c''' k) (ofs2 k)
    using dec0 ofeq by simp
  have corr2: ∀k<k_tm M. ae_tape_correspondence (le_tm M)
                          (mt_tape cM k) (mt_tape c'' k)
    using corr tape by simp
  have qeq2: mt_state cM = q using qeq qM'eq by simp
  have dec2: ∀k<k_tm M. mt_pos cM k
                        = ae_decode_pos (mt_pos c'' k) (ofs1 k)
  proof (intro allI impI)
    fix k assume k: k < k_tm M
    have d3: mt_pos cM k = ae_decode_pos (mt_pos c''' k) (ofs2 k)
      by (rule dec[rule_format, OF k])
    have ppos: mt_pos c'' k = mt_pos c''' k using pos by simp
```

```
    show mt_pos cM k = ae_decode_pos (mt_pos c'' k) (ofs1 k)
    proof (cases mt_pos cM k = 0)
      case True
      have ae_decode_pos (mt_pos c''' k) (ofs2 k) = 0
        using d3[symmetric] True by simp
      from ae_decode_pos_eq_0[OF this] have mt_pos c''' k = 0 .
      hence ae_decode_pos (mt_pos c'' k) (ofs1 k) = 0
        using ppos by (simp add: ae_decode_pos_zero_eq)
      thus ?thesis using True by simp
    next
      case False
      from ofsh[rule_format, OF k] False have ofs1 k = ofs2 k by simp
      hence ae_decode_pos (mt_pos c'' k) (ofs1 k)
            = ae_decode_pos (mt_pos c''' k) (ofs2 k)
        using ppos by simp
      thus ?thesis using d3 by simp
    qed
  qed
  show ?thesis
    unfolding ae_simulates_def Let_def st2
    using qeq2 corr2 dec2 gam qnh by simp
qed
```

The bridge from the window inversion to the segment congruence's hypotheses. Two SS5 buffered families window-corresponding (per active tape) to the same trace endpoint give: their buffers are *ae_blk_agree* (home + right agree at every *pos* from the kernel; at *pos = 0* the home is *LE_block le* so the left clause is vacuous), and their boundary *directions* agree (full *bp* agreement where *pos ≠ 0*; *le0* direction agreement at *pos = 0*). These are exactly the *buf / dest* congruence inputs the SS5→SS8 chain consumes.

```
lemma ae_ss5_window_blk_dest_agree:
  fixes tM :: nat ⇒ nat ⇒ 'a and nM pos :: nat ⇒ nat
    and bp1 bp2 :: nat ⇒ ('c :: enum) bp
    and b1 b2 :: nat ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
    and le :: 'a and K :: nat
  assumes h1: ∀k<K. ae_window_invariant_general (tM k) (nM k)
                      (bp1 k) (b1 k) (pos k) le
      and h2: ∀k<K. ae_window_invariant_general (tM k) (nM k)
                      (bp2 k) (b2 k) (pos k) le
  shows (∀k<K. ae_blk_agree le (b1 k) (b2 k))
        ∧ (∀k<K. fst (bp1 k) = fst (bp2 k))
proof (intro conjI allI impI)
  fix k assume k: k < K
  from h1 k have g1: ae_window_invariant_general (tM k) (nM k)
                      (bp1 k) (b1 k) (pos k) le by blast
  from h2 k have g2: ae_window_invariant_general (tM k) (nM k)
                      (bp2 k) (b2 k) (pos k) le by blast
  from ae_window_invariant_general_determines[OF g1 g2]
  have hr1: fst (snd (b1 k)) = fst (snd (b2 k))
```

```
   and hr2: snd (snd (b1 k)) = snd (snd (b2 k))
   and full: pos k ≠ 0 ⟶ bp1 k = bp2 k ∧ b1 k = b2 k by simp_all
  show ae_blk_agree le (b1 k) (b2 k)
  proof (cases pos k = 0)
    case True
    have le0_1: ae_window_invariant_le0 (tM k) (nM k) (bp1 k) (b1 k) le
      using g1 True unfolding ae_window_invariant_general_def by simp
    have fst (snd (b1 k)) = LE_block le
      using le0_1 unfolding ae_window_invariant_le0_def by simp
    thus ?thesis using hr1 hr2 unfolding ae_blk_agree_def by simp
  next
    case False
    with full have b1 k = b2 k by simp
    thus ?thesis unfolding ae_blk_agree_def by simp
  qed
next
  fix k assume k: k < K
  from h1 k have g1: ae_window_invariant_general (tM k) (nM k)
                       (bp1 k) (b1 k) (pos k) le by blast
  from h2 k have g2: ae_window_invariant_general (tM k) (nM k)
                       (bp2 k) (b2 k) (pos k) le by blast
  show fst (bp1 k) = fst (bp2 k)
  proof (cases pos k = 0)
    case True
    have le0_1: ae_window_invariant_le0 (tM k) (nM k) (bp1 k) (b1 k) le
      using g1 True unfolding ae_window_invariant_general_def by simp
    have le0_2: ae_window_invariant_le0 (tM k) (nM k) (bp2 k) (b2 k) le
      using g2 True unfolding ae_window_invariant_general_def by simp
    from ae_window_le0_dest_agree[OF le0_1 le0_2] show ?thesis by simp
  next
    case False
    from ae_window_invariant_general_determines[OF g1 g2] False
    show ?thesis by simp
  qed
qed
```

Halt branch of the closing SS8→SS1 step. When $q \in \{t, r\}$ the builder resets the stage to the constant *init_stage* (*le_tm M*), *independent* of the buffers and offset — so the two destination states are *identical* (not merely *ae_blk_agree*), and only the tape write needs the action-congruence. This is the halt-case companion of *ae_ss8_ss1_cong*.

```
lemma ae_ss8_ss1_cong_halt:
  fixes M :: ('q, 'a) mttm
    and c1 c2 c1' c2' :: ('c :: enum ⇒ 'a,
                          'q × ('a, 'c) ae_stage) mt_config
  assumes tape: mt_tape c1 = mt_tape c2
      and pos:  mt_pos c1 = mt_pos c2
      and st1:  mt_state c1 = (q, ofs1, buf1, dest, SS8)
      and st2:  mt_state c2 = (q, ofs2, buf2, dest, SS8)
```

```
    and blk:  ∀k. ae_blk_agree (le_tm M) (buf1 k) (buf2 k)
    and qh:   q ∈ {t_tm M, r_tm M}
    and s1:   (c1, c1') ∈ mttm_step (ae_delta_ss8_ss1 M)
    and s2:   (c2, c2') ∈ mttm_step (ae_delta_ss8_ss1 M)
  shows mt_tape c1' = mt_tape c2' ∧ mt_pos c1' = mt_pos c2'
      ∧ mt_state c1' = (q, init_stage (le_tm M))
      ∧ mt_state c2' = (q, init_stage (le_tm M))
proof −
  obtain ts n where c1d: c1 = Config_M (q, ofs1, buf1, dest, SS8) ts n
    using st1 by (cases c1) auto
  have c2d: c2 = Config_M (q, ofs2, buf2, dest, SS8) ts n
    using st2 tape pos c1d by (cases c2) auto
  from s1[unfolded c1d] obtain s1' a'1 d1 where
      c1'd: c1' = Config_M s1' (λk. (ts k)(n k := a'1 k))
                               (λk. go_dir (d1 k) (n k))
    and rel1: ((q, ofs1, buf1, dest, SS8), (λk. ts k (n k)),
               s1', a'1, d1) ∈ ae_delta_ss8_ss1 M
    by (auto elim: mttm_step.cases)
  from s2[unfolded c2d] obtain s2' a'2 d2 where
      c2'd: c2' = Config_M s2' (λk. (ts k)(n k := a'2 k))
                               (λk. go_dir (d2 k) (n k))
    and rel2: ((q, ofs2, buf2, dest, SS8), (λk. ts k (n k)),
               s2', a'2, d2) ∈ ae_delta_ss8_ss1 M
    by (auto elim: mttm_step.cases)
  have e1: s1' = (q, init_stage (le_tm M))
    and a1: a'1 = (λk. if k < k_tm M
                       then fst (ae_ss8_action (le_tm M) (ts k (n k))
                                               (buf1 k) (dest k))
                       else bl_block (bl_tm M))
    and d1: d1 = (λk. if k < k_tm M
                       then snd (ae_ss8_action (le_tm M) (ts k (n k))
                                               (buf1 k) (dest k))
                       else dir.N)
    using rel1 qh unfolding ae_delta_ss8_ss1_def by auto
  have e2: s2' = (q, init_stage (le_tm M))
    and a2: a'2 = (λk. if k < k_tm M
                       then fst (ae_ss8_action (le_tm M) (ts k (n k))
                                               (buf2 k) (dest k))
                       else bl_block (bl_tm M))
    and d2: d2 = (λk. if k < k_tm M
                       then snd (ae_ss8_action (le_tm M) (ts k (n k))
                                               (buf2 k) (dest k))
                       else dir.N)
    using rel2 qh unfolding ae_delta_ss8_ss1_def by auto
  have act: ⋀k. ae_ss8_action (le_tm M) (ts k (n k)) (buf1 k) (dest k)
              = ae_ss8_action (le_tm M) (ts k (n k)) (buf2 k) (dest k)
    by (rule ae_ss8_action_cong[OF blk[rule_format]])
  have a'1 = a'2
    by (rule ext) (simp add: a1 a2 act)
```

```
  moreover have d1 = d2
    by (rule ext) (simp add: d1 d2 act)
  ultimately show ?thesis using c1'd c2'd e1 e2 by simp
qed
```

Halt-agnostic prefix of the macro-cycle tail: chaining the three non-branching congruences SS5→SS6→SS7→SS8. None of these three steps inspects the halt flag, so the conclusion carries *q* / *dest* / the *ae_blk_agree* buffers unchanged whether or not *q* is halting; the halt split re-enters only at the final SS8→SS1 step.

```
lemma ae_ss5_ss8_segment_to_ss8:
  fixes M :: ('q, 'a) mttm
    and c5a c5b c8a c8b :: ('c :: enum ⇒ 'a,
                            'q × ('a, 'c) ae_stage) mt_config
  assumes tape: mt_tape c5a = mt_tape c5b
      and pos:  mt_pos c5a = mt_pos c5b
      and st1:  mt_state c5a = (q, ofs1, buf1, dest, SS5)
      and st2:  mt_state c5b = (q, ofs2, buf2, dest, SS5)
      and blk:  ∀k. ae_blk_agree (le_tm M) (buf1 k) (buf2 k)
      and a56:  (c5a, d6a) ∈ mttm_step (ae_delta_ss5_ss6 M)
      and a67:  (d6a, d7a) ∈ mttm_step (ae_delta_ss6_ss7 M)
      and a78:  (d7a, c8a) ∈ mttm_step (ae_delta_ss7_ss8 M)
      and b56:  (c5b, d6b) ∈ mttm_step (ae_delta_ss5_ss6 M)
      and b67:  (d6b, d7b) ∈ mttm_step (ae_delta_ss6_ss7 M)
      and b78:  (d7b, c8b) ∈ mttm_step (ae_delta_ss7_ss8 M)
  shows mt_tape c8a = mt_tape c8b ∧ mt_pos c8a = mt_pos c8b
        ∧ mt_state c8a = (q, ofs1, buf1, dest, SS8)
        ∧ mt_state c8b = (q, ofs2, buf2, dest, SS8)
proof −
  from ae_ss5_ss6_cong[OF tape pos st1 st2 blk a56 b56]
  have t6: mt_tape d6a = mt_tape d6b and p6: mt_pos d6a = mt_pos d6b
   and s6a: mt_state d6a = (q, ofs1, buf1, dest, SS6)
   and s6b: mt_state d6b = (q, ofs2, buf2, dest, SS6) by simp_all
  from ae_ss6_ss7_cong[OF t6 p6 s6a s6b blk a67 b67]
  have t7: mt_tape d7a = mt_tape d7b and p7: mt_pos d7a = mt_pos d7b
   and s7a: mt_state d7a = (q, ofs1, buf1, dest, SS7)
   and s7b: mt_state d7b = (q, ofs2, buf2, dest, SS7) by simp_all
  from ae_ss7_ss8_cong[OF t7 p7 s7a s7b blk a78 b78]
  show ?thesis by simp
qed
```

Halt-case *ae_simulates* transfer. When the simulated M-config *cM* is halting, *ae_simulates* reads only the M-state, the tape correspondence, and (via the halt disjunct) that the stage equals *init_stage* (*le_tm M*) — never the pos-decode. So a given *c″* whose stage is exactly *init_stage* (*le_tm M*) and whose tape matches a reconstructed *c‴* with *ae_simulates M cM c‴* inherits the simulation. Uses the halt-agnostic *ae_simulates_state_disjunction* (not *ae_forward_stage_unpack_sim*, which needs non-halt *cM*).

```
lemma ae_simulates_transfer_halt:
  fixes M :: ('q, 'a) mttm and cM :: ('a, 'q) mt_config
    and c'' c''' :: ('c :: enum ⇒ 'a, 'q × ('a, 'c) ae_stage) mt_config
  assumes sim':    ae_simulates M cM c'''
      and cM_halt: mt_state cM ∈ {t_tm M, r_tm M}
      and tape:    mt_tape c'' = mt_tape c'''
      and st2:     mt_state c'' = (mt_state cM, init_stage (le_tm M))
      and gam:     ae_tape_in_gamma_block M c''
  shows ae_simulates M cM c''
proof −
  obtain qM' ofs buf dest idx where
      ust: mt_state c''' = (qM', ofs, buf, dest, idx)
    and qeq: mt_state cM = qM'
    and corr: ∀k<k_tm M. ae_tape_correspondence (le_tm M)
                         (mt_tape cM k) (mt_tape c''' k)
    by (rule ae_simulates_state_disjunction[OF sim'])
  have corr2: ∀k<k_tm M. ae_tape_correspondence (le_tm M)
                         (mt_tape cM k) (mt_tape c'' k)
    using corr tape by simp
  show ?thesis
    unfolding ae_simulates_def Let_def st2 init_stage_def
    using cM_halt corr2 gam by simp
qed
```

The offset-agreement bridge (the non-halt transfer's *ofsh*). Two SS5 window-correspondences to the same trace endpoint agree on the boundary *offset snd bp except* where the simulated head *nM* is *0*. Where *pos* ≠ *0* the kernel pins the full *bp*. Where *pos* = *0* the *le0* dest-agreement splits on the direction: at *AE_Home* the invariant forces *nM* = *0* (the left disjunct); otherwise (*AE_Right*) it pins the full *bp*, so the offset agrees. This is precisely the residual being confined to the *pos* = *0* / *AE_Home* corner, where *ae_decode_pos* ignores the offset.

```
lemma ae_ss5_window_ofs_agree:
  fixes tM :: nat ⇒ nat ⇒ 'a and nM pos :: nat ⇒ nat
    and bp1 bp2 :: nat ⇒ ('c :: enum) bp
    and b1 b2 :: nat ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
    and le :: 'a and K :: nat
  assumes h1: ∀k<K. ae_window_invariant_general (tM k) (nM k)
                       (bp1 k) (b1 k) (pos k) le
      and h2: ∀k<K. ae_window_invariant_general (tM k) (nM k)
                       (bp2 k) (b2 k) (pos k) le
  shows ∀k<K. nM k = 0 ∨ snd (bp1 k) = snd (bp2 k)
proof (intro allI impI)
  fix k assume k: k < K
  from h1 k have g1: ae_window_invariant_general (tM k) (nM k)
                       (bp1 k) (b1 k) (pos k) le by blast
  from h2 k have g2: ae_window_invariant_general (tM k) (nM k)
                       (bp2 k) (b2 k) (pos k) le by blast
  show nM k = 0 ∨ snd (bp1 k) = snd (bp2 k)
```

```
  proof (cases pos k = 0)
    case False
    from ae_window_invariant_general_determines[OF g1 g2] False
    have bp1 k = bp2 k by simp
    thus ?thesis by simp
  next
    case True
    have le0_1: ae_window_invariant_le0 (tM k) (nM k) (bp1 k) (b1 k) le
      using g1 True unfolding ae_window_invariant_general_def by simp
    have le0_2: ae_window_invariant_le0 (tM k) (nM k) (bp2 k) (b2 k) le
      using g2 True unfolding ae_window_invariant_general_def by simp
    from ae_window_le0_dest_agree[OF le0_1 le0_2]
    have full: fst (bp1 k) ≠ AE_Home ⟶ bp1 k = bp2 k
      by (rule conjunct2)
    have home_nM0: fst (bp1 k) = AE_Home ⟶ nM k = 0
      using le0_1[unfolded ae_window_invariant_le0_def]
      by (cases fst (bp1 k)) auto
    show ?thesis
    proof (cases fst (bp1 k) = AE_Home)
      case True
      thus ?thesis using home_nM0 by simp
    next
      case False
      with full have bp1 k = bp2 k by simp
      thus ?thesis by simp
    qed
  qed
qed
```

Integration prototype (non-halt branch). This is the *ae_sim_c″* derivation of *ae_backward_stage* with the *det* chain-uniqueness replaced by the det-free toolkit, stated against the hypotheses the strengthened *ae_backward_stage_decompose* (run arm) and *ae_simulates_forward_stage_general* (forward arm) will expose. Both SS4→SS5 steps leave a *common* SS4 source *c3* (the SS1→SS4 builders are functional, matched in the real integration); both SS5 configs window-correspond to the same trace endpoint *cM_new*. The shared source pins tape+head; the window bridge pins the *buf* (*ae_blk_agree*), the *dest* direction, and the offset-except-where-*nM=0*; the inactive-tape tails come from both SS5 states being *init* beyond *k_tm M*. The SS5→SS1 segment congruence then transports the agreement to the SS1 boundary, and *ae_simulates_transfer* hands the forward simulation across.

```
lemma nae_sim_proto_nonhalt:
  fixes M :: ('q, 'a) mttm and cM_new :: ('a, 'q) mt_config
    and c3 c4 c5 c6 c7 c'' :: ('c :: enum ⇒ 'a,
                               'q × ('a, 'c) ae_stage) mt_config
    and c4f c5f c6f c7f c''' :: ('c :: enum ⇒ 'a,
                                 'q × ('a, 'c) ae_stage) mt_config
```

and *ofs_r ofs_f* :: *nat* ⇒ *'c*
and *buf_r buf_f* :: *nat* ⇒ ((*'c* ⇒ *'a*) × (*'c* ⇒ *'a*) × (*'c* ⇒ *'a*))
and *dest_r dest_f* :: *nat* ⇒ *ae_dest* **and** *pos* :: *nat* ⇒ *nat*
assumes *q_in_Q*: *mt_state cM_new* ∈ *Q_tm M*
and *nonhalt*: *mt_state cM_new* ∉ {*t_tm M*, *r_tm M*}
— common SS4 source
and *run45*: (*c3*, *c4*) ∈ *mttm_step* (*ae_delta_ss4_ss5 M*)
and *fwd45*: (*c3*, *c4f*) ∈ *mttm_step* (*ae_delta_ss4_ss5 M*)
— SS5 states + window correspondences (run / fwd)
and *c4_state*: *mt_state c4* = (*mt_state cM_new*, *ofs_r*, *buf_r*, *dest_r*, *SS5*)
and *c4f_state*: *mt_state c4f* = (*mt_state cM_new*, *ofs_f*, *buf_f*, *dest_f*, *SS5*)
and *run_window*: ∀ *k*<*k_tm M*. *ae_window_invariant_general*
(*mt_tape cM_new k*) (*mt_pos cM_new k*)
(*dest_r k*, *ofs_r k*) (*buf_r k*) (*pos k*) (*le_tm M*)
and *fwd_window*: ∀ *k*<*k_tm M*. *ae_window_invariant_general*
(*mt_tape cM_new k*) (*mt_pos cM_new k*)
(*dest_f k*, *ofs_f k*) (*buf_f k*) (*pos k*) (*le_tm M*)
— inactive-tape tails (both *init* beyond *k_tm M*)
and *buf_tail*: ∀ *k*. ¬ *k* < *k_tm M*
⟶ *ae_blk_agree* (*le_tm M*) (*buf_r k*) (*buf_f k*)
and *dest_tail*: ∀ *k*. ¬ *k* < *k_tm M* ⟶ *dest_r k* = *dest_f k*
— run / forward SS5→SS1 substep chains
and *run5*: (*c4*, *c5*) ∈ *mttm_step* (*ae_delta_ss5_ss6 M*)
and *run6*: (*c5*, *c6*) ∈ *mttm_step* (*ae_delta_ss6_ss7 M*)
and *run7*: (*c6*, *c7*) ∈ *mttm_step* (*ae_delta_ss7_ss8 M*)
and *run8*: (*c7*, *c''*) ∈ *mttm_step* (*ae_delta_ss8_ss1 M*)
and *fwd5*: (*c4f*, *c5f*) ∈ *mttm_step* (*ae_delta_ss5_ss6 M*)
and *fwd6*: (*c5f*, *c6f*) ∈ *mttm_step* (*ae_delta_ss6_ss7 M*)
and *fwd7*: (*c6f*, *c7f*) ∈ *mttm_step* (*ae_delta_ss7_ss8 M*)
and *fwd8*: (*c7f*, *c'''*) ∈ *mttm_step* (*ae_delta_ss8_ss1 M*)
and *sim_fwd*: *ae_simulates M cM_new c'''*
and *gam*: *ae_tape_in_gamma_block M c''*
shows *ae_simulates M cM_new c''*
proof −
— Shared SS4 source: tape + head positions agree at SS5.
from *ae_ss4_ss5_shared_source_tape*[*OF run45 fwd45*]
have *tape45*: *mt_tape c4* = *mt_tape c4f*
and *pos45*: *mt_pos c4* = *mt_pos c4f* **by** *simp_all*
— Window bridge: buffers *ae_blk_agree*, *dest* direction agrees (active tapes).
from *ae_ss5_window_blk_dest_agree*[*OF run_window fwd_window*]
have *blk_act*: ∀ *k*<*k_tm M*. *ae_blk_agree* (*le_tm M*) (*buf_r k*) (*buf_f k*)
and *dir_act*: ∀ *k*<*k_tm M*. *dest_r k* = *dest_f k* **by** *simp_all*
— Offset bridge: *ofs* agrees except where the simulated head is *0* — this is the transfer's *ofsh*.
from *ae_ss5_window_ofs_agree*[*OF run_window fwd_window*]
have *ofsh*: ∀ *k*<*k_tm M*. *mt_pos cM_new k* = *0* ∨ *ofs_r k* = *ofs_f k* **by** *simp*
— Lift *blk* / *dest* agreement to all tapes via the tails.

```
  have blk_all: ∀k. ae_blk_agree (le_tm M) (buf_r k) (buf_f k)
  proof
    fix k show ae_blk_agree (le_tm M) (buf_r k) (buf_f k)
      by (cases k < k_tm M) (use blk_act buf_tail in blast)+
  qed
  have dest_eq: dest_r = dest_f
  proof
    fix k show dest_r k = dest_f k
      by (cases k < k_tm M) (use dir_act dest_tail in blast)+
  qed
  have c4f_state': mt_state c4f = (mt_state cM_new, ofs_f, buf_f, dest_r, SS5)
    using c4f_state dest_eq by simp
  — Transport agreement SS5→SS1 (non-halt branch).
  from ae_ss5_ss8_segment_cong[OF tape45 pos45 c4_state c4f_state' blk_all
                                  nonhalt run5 run6 run7 run8
                                  fwd5 fwd6 fwd7 fwd8]
  have tcc: mt_tape c'' = mt_tape c'''
   and pcc: mt_pos c'' = mt_pos c'''
   and st_c'':  mt_state c'' = (mt_state cM_new, ofs_r, buf_r, init_dest, SS1)
   and st_c''': mt_state c''' = (mt_state cM_new, ofs_f, buf_f, init_dest, SS1)
    by simp_all
  — Transfer the forward simulation across the agreement.
  have cM_nt: mt_state cM_new ≠ t_tm M using nonhalt by simp
  have cM_nr: mt_state cM_new ≠ r_tm M using nonhalt by simp
  show ?thesis
    by (rule ae_simulates_transfer[OF sim_fwd q_in_Q cM_nt cM_nr tcc pcc
                                      st_c'' st_c''' nonhalt ofsh gam])
qed
```

Integration prototype (halt branch). When the simulated M-config halts within the macro-cycle (*mt_state cM_new* ∈ {*t*, *r*}) the closing SS8→SS1 step resets the stage to *init_stage* instead of looping to SS1, so the non-halt segment congruence does not apply. The shared source + window bridge are unchanged; the SS5→SS8 prefix is halt-agnostic; the halt SS8→SS1 congruence lands both boundary configs in the identical *init_stage* state with agreeing tapes, and *ae_simulates_transfer_halt* hands the (halt-disjunct) simulation across. No offset reasoning: the halt disjunct never reads the pos-decode.

```
lemma nae_sim_proto_halt:
  fixes M :: ('q, 'a) mttm and cM_new :: ('a, 'q) mt_config
    and c3 c4 c5 c6 c7 c'' :: ('c :: enum ⇒ 'a,
                               'q × ('a, 'c) ae_stage) mt_config
    and c4f c5f c6f c7f c''' :: ('c :: enum ⇒ 'a,
                                 'q × ('a, 'c) ae_stage) mt_config
    and ofs_r ofs_f :: nat ⇒ 'c
    and buf_r buf_f :: nat ⇒ (('c ⇒ 'a) × ('c ⇒ 'a) × ('c ⇒ 'a))
    and dest_r dest_f :: nat ⇒ ae_dest and pos :: nat ⇒ nat
  assumes halt: mt_state cM_new ∈ {t_tm M, r_tm M}
```

```
    and run45: (c3, c4) ∈ mttm_step (ae_delta_ss4_ss5 M)
    and fwd45: (c3, c4f) ∈ mttm_step (ae_delta_ss4_ss5 M)
    and c4_state: mt_state c4 = (mt_state cM_new, ofs_r, buf_r, dest_r,
SS5)
    and c4f_state: mt_state c4f = (mt_state cM_new, ofs_f, buf_f, dest_f,
SS5)
    and run_window: ∀k<k_tm M. ae_window_invariant_general
          (mt_tape cM_new k) (mt_pos cM_new k)
          (dest_r k, ofs_r k) (buf_r k) (pos k) (le_tm M)
    and fwd_window: ∀k<k_tm M. ae_window_invariant_general
          (mt_tape cM_new k) (mt_pos cM_new k)
          (dest_f k, ofs_f k) (buf_f k) (pos k) (le_tm M)
    and buf_tail: ∀k. ¬ k < k_tm M
                  ⟶ ae_blk_agree (le_tm M) (buf_r k) (buf_f k)
    and dest_tail: ∀k. ¬ k < k_tm M ⟶ dest_r k = dest_f k
    and run5: (c4, c5) ∈ mttm_step (ae_delta_ss5_ss6 M)
    and run6: (c5, c6) ∈ mttm_step (ae_delta_ss6_ss7 M)
    and run7: (c6, c7) ∈ mttm_step (ae_delta_ss7_ss8 M)
    and run8: (c7, c'') ∈ mttm_step (ae_delta_ss8_ss1 M)
    and fwd5: (c4f, c5f) ∈ mttm_step (ae_delta_ss5_ss6 M)
    and fwd6: (c5f, c6f) ∈ mttm_step (ae_delta_ss6_ss7 M)
    and fwd7: (c6f, c7f) ∈ mttm_step (ae_delta_ss7_ss8 M)
    and fwd8: (c7f, c''') ∈ mttm_step (ae_delta_ss8_ss1 M)
    and sim_fwd: ae_simulates M cM_new c'''
    and gam:     ae_tape_in_gamma_block M c''
  shows ae_simulates M cM_new c''
proof −
  from ae_ss4_ss5_shared_source_tape[OF run45 fwd45]
  have tape45: mt_tape c4 = mt_tape c4f
   and pos45: mt_pos c4 = mt_pos c4f by simp_all
  from ae_ss5_window_blk_dest_agree[OF run_window fwd_window]
  have blk_act: ∀k<k_tm M. ae_blk_agree (le_tm M) (buf_r k) (buf_f k)
   and dir_act: ∀k<k_tm M. dest_r k = dest_f k by simp_all
  have blk_all: ∀k. ae_blk_agree (le_tm M) (buf_r k) (buf_f k)
  proof
    fix k show ae_blk_agree (le_tm M) (buf_r k) (buf_f k)
      by (cases k < k_tm M) (use blk_act buf_tail in blast)+
  qed
  have dest_eq: dest_r = dest_f
  proof
    fix k show dest_r k = dest_f k
      by (cases k < k_tm M) (use dir_act dest_tail in blast)+
  qed
  have c4f_state': mt_state c4f = (mt_state cM_new, ofs_f, buf_f, dest_r, SS5)
    using c4f_state dest_eq by simp
  — Halt-agnostic SS5→SS8 prefix.
  from ae_ss5_ss8_segment_to_ss8[OF tape45 pos45 c4_state c4f_state' blk_all
                                    run5 run6 run7 fwd5 fwd6 fwd7]
  have t8: mt_tape c7 = mt_tape c7f and p8: mt_pos c7 = mt_pos c7f
```

and *s8*: *mt_state c7* = (*mt_state cM_new*, *ofs_r*, *buf_r*, *dest_r*, *SS8*)
and *s8f*: *mt_state c7f* = (*mt_state cM_new*, *ofs_f*, *buf_f*, *dest_r*, *SS8*)
by *simp_all*
— Halt SS8→SS1: both boundary configs reach the identical *init_stage*, tapes agree.
from *ae_ss8_ss1_cong_halt*[*OF t8 p8 s8 s8f blk_all halt run8 fwd8*]
have *tcc*: *mt_tape c''* = *mt_tape c'''*
and *st_c''*: *mt_state c''* = (*mt_state cM_new*, *init_stage* (*le_tm M*))
by *simp_all*
show *?thesis*
by (*rule ae_simulates_transfer_halt*[*OF sim_fwd halt tcc st_c'' gam*])
qed

Combined integration prototype: the full *ae_sim_c''* of *ae_backward_stage*, *det*-free. Takes the union of the two branches' exposure and case-splits on whether *cM_new* halts. This is the literal template for the real rewrite in *AlphabetEnlargement_Reverse*: the only remaining integration work is to (a) strengthen *ae_backward_stage_decompose* to expose the run arm's SS5 state *c4_state* + *run_window* (already derived internally as *window_cM_new*) and the inactive tails, (b) strengthen *ae_simulates_forward_stage_general* to expose its SS5 config *c4f* + *fwd_window* + the SS5→SS1 substeps alongside the *c'''* it already returns, and (c) match the two SS4 sources *c3* = *c3_fwd* via the SS1→SS4 functional builders. No new mathematics in the reverse arm.

lemma *nae_sim_proto*:
fixes *M* :: ('*q*, '*a*) *mttm* **and** *cM_new* :: ('*a*, '*q*) *mt_config*
and *c3 c4 c5 c6 c7 c''* :: ('*c* :: *enum* ⇒ '*a*,
'*q* × ('*a*, '*c*) *ae_stage*) *mt_config*
and *c4f c5f c6f c7f c'''* :: ('*c* :: *enum* ⇒ '*a*,
'*q* × ('*a*, '*c*) *ae_stage*) *mt_config*
and *ofs_r ofs_f* :: *nat* ⇒ '*c*
and *buf_r buf_f* :: *nat* ⇒ (('*c* ⇒ '*a*) × ('*c* ⇒ '*a*) × ('*c* ⇒ '*a*))
and *dest_r dest_f* :: *nat* ⇒ *ae_dest* **and** *pos* :: *nat* ⇒ *nat*
assumes *q_in_Q*: *mt_state cM_new* ∈ *Q_tm M*
and *run45*: (*c3*, *c4*) ∈ *mttm_step* (*ae_delta_ss4_ss5 M*)
and *fwd45*: (*c3*, *c4f*) ∈ *mttm_step* (*ae_delta_ss4_ss5 M*)
and *c4_state*: *mt_state c4* = (*mt_state cM_new*, *ofs_r*, *buf_r*, *dest_r*, *SS5*)
and *c4f_state*: *mt_state c4f* = (*mt_state cM_new*, *ofs_f*, *buf_f*, *dest_f*, *SS5*)
and *run_window*: ∀ *k*<*k_tm M*. *ae_window_invariant_general*
(*mt_tape cM_new k*) (*mt_pos cM_new k*)
(*dest_r k*, *ofs_r k*) (*buf_r k*) (*pos k*) (*le_tm M*)
and *fwd_window*: ∀ *k*<*k_tm M*. *ae_window_invariant_general*
(*mt_tape cM_new k*) (*mt_pos cM_new k*)
(*dest_f k*, *ofs_f k*) (*buf_f k*) (*pos k*) (*le_tm M*)
and *buf_tail*: ∀ *k*. ¬ *k* < *k_tm M*
⟶ *ae_blk_agree* (*le_tm M*) (*buf_r k*) (*buf_f k*)

```
    and dest_tail: ∀ k. ¬ k < k_tm M ⟶ dest_r k = dest_f k
    and run5: (c4, c5) ∈ mttm_step (ae_delta_ss5_ss6 M)
    and run6: (c5, c6) ∈ mttm_step (ae_delta_ss6_ss7 M)
    and run7: (c6, c7) ∈ mttm_step (ae_delta_ss7_ss8 M)
    and run8: (c7, c'') ∈ mttm_step (ae_delta_ss8_ss1 M)
    and fwd5: (c4f, c5f) ∈ mttm_step (ae_delta_ss5_ss6 M)
    and fwd6: (c5f, c6f) ∈ mttm_step (ae_delta_ss6_ss7 M)
    and fwd7: (c6f, c7f) ∈ mttm_step (ae_delta_ss7_ss8 M)
    and fwd8: (c7f, c''') ∈ mttm_step (ae_delta_ss8_ss1 M)
    and sim_fwd: ae_simulates M cM_new c'''
    and gam:     ae_tape_in_gamma_block M c''
  shows ae_simulates M cM_new c''
proof (cases mt_state cM_new ∈ {t_tm M, r_tm M})
  case True
  show ?thesis
    by (rule nae_sim_proto_halt[OF True run45 fwd45 c4_state c4f_state
                                   run_window fwd_window buf_tail dest_tail
                                   run5 run6 run7 run8 fwd5 fwd6 fwd7 fwd8
                                   sim_fwd gam])
next
  case False
  show ?thesis
    by (rule nae_sim_proto_nonhalt[OF q_in_Q False run45 fwd45 c4_state
                                      c4f_state run_window fwd_window
                                      buf_tail dest_tail
                                      run5 run6 run7 run8 fwd5 fwd6 fwd7 fwd8
                                      sim_fwd gam])
qed
```

Config-level functionality of a substep relation from delta-level functionality: if the underlying δ-relation R is single-valued in (s', a', d) given the source (s, a), then the induced *mttm_step* R is a partial function on configurations. The *det*-free analogue of *mttm_step_alphabet_enlarge_functional*, applied to the functional SS1→SS4 substep builders to match the run and forward arms' shared SS4 source *c3*.

```
lemma mttm_step_functional_of_delta:
  assumes fnl: ⋀s a s1 a1 d1 s2 a2 d2.
                 (s, a, s1, a1, d1) ∈ R ⟹ (s, a, s2, a2, d2) ∈ R
                   ⟹ s1 = s2 ∧ a1 = a2 ∧ d1 = d2
    and h1: (c, c1) ∈ mttm_step R
    and h2: (c, c2) ∈ mttm_step R
  shows c1 = c2
proof −
  from h1 obtain q ts n q1 a1 dir1 where
    c_eq1:  c  = Config_M q ts n
  and c1_eq: c1 = Config_M q1 (λk. (ts k)(n k := a1 k))
                              (λk. go_dir (dir1 k) (n k))
  and tr1: (q, λk. ts k (n k), q1, a1, dir1) ∈ R
  by (auto elim: mttm_step.cases)
```

from *h2* **obtain** q' ts' n' *q2 a2 dir2* **where**
  c_eq2: $c = Config_M\ q'\ ts'\ n'$
 and *c2_eq*: $c2 = Config_M\ q2\ (\lambda k.\ (ts'\ k)(n'\ k := a2\ k))$
  $(\lambda k.\ go_dir\ (dir2\ k)\ (n'\ k))$
 and *tr2*: $(q', \lambda k.\ ts'\ k\ (n'\ k),\ q2,\ a2,\ dir2) \in R$
 by (*auto elim*: *mttm_step.cases*)
from *c_eq1 c_eq2* **have** *qq*: $q = q'$ **and** *tsts*: $ts = ts'$ **and** *nn*: $n = n'$
 by *auto*
from *tr2 qq tsts nn*
have *tr2'*: $(q, \lambda k.\ ts\ k\ (n\ k),\ q2,\ a2,\ dir2) \in R$ **by** *simp*
have $q1 = q2 \wedge a1 = a2 \wedge dir1 = dir2$ **by** (*rule fnl*[*OF tr1 tr2'*])
thus *?thesis* **using** *c1_eq c2_eq tsts nn* **by** *simp*
qed

2.29.7 Backward stage orchestration

Reverse-arm chain orchestration (Step 3e). Combines *ae_backward_stage_load_chain*, *ae_backward_stage_compute_extract*, and *ae_backward_stage_writeback_chain* into a single decomposition of an 8-step *alphabet_enlarge_delta*-chain from an SS1 boundary configuration. Exposes the 8 substep facts, the substrate trace, its termination disjunction, and halt-aware c" status.

Design choice: the outer *obtains* exposes only first-order conjuncts projection-level state shape ($fst\ (mt_state\ c'') = mt_state\ cM_new$) rather than 5-tuple state equalities to avoid the higher-order pattern unification that defeats an *OF*-style discharge. The c4 state-tuple is internal plumbing; the consumer only needs first-order projections.

lemma *ae_backward_stage_decompose*:
 fixes $M :: ('q, 'a)\ mttm$
  and $cM :: ('a, 'q)\ mt_config$
  and $c'\ c'' :: ('c :: enum \Rightarrow 'a,$
  $'q \times ('a, 'c)\ ae_stage)\ mt_config$
  and $qM' :: 'q$
  and $ofs :: nat \Rightarrow 'c$
  and $buf :: nat \Rightarrow ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$
  and $dest :: nat \Rightarrow ae_dest$
 assumes *vM*: $valid_mttm\ M$
  and *lu*: $le_unique\ M$
  and *sim*: $ae_simulates\ M\ cM\ c'$
  and *val_cM*: $valid_config_mttm\ M\ cM$
  and *c'_state*: $mt_state\ c' = (qM', ofs, buf, dest, SS1)$
  and *inv_ss1*: $ae_inv_ss1\ M\ c'$
  and *gamma_c'*: $ae_tape_in_gamma_block\ M\ c'$
  and *buf_gamma*: $ae_buffer_in_gamma_block\ M\ c'$
  and *pos_link_c'*: $ae_position_link\ M\ c'$
  and *le_anchor*: $\forall k{<}k_tm\ M.\ mt_tape\ c'\ k\ 0 = LE_block\ (le_tm\ M)$
  and *le_neq_bl*: $le_tm\ M \neq bl_tm\ M$
  and *qM'_neq_t*: $qM' \neq t_tm\ M$

```
    and qM'_neq_r:   qM' ≠ r_tm M
    and no_le_per_tape:
        ∀k. (mt_pos c' k ≥ 2
              ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
                    ⟶ mt_tape cM k
                        ((mt_pos c' k − 2) * card (UNIV :: 'c set)
                          + 1 + i)
                       ≠ le_tm M))
            ∧ (mt_pos c' k = 1
               ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
                     ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
            ∧ (mt_pos c' k = 0
               ⟶ (∀i. i < card (UNIV :: 'c set)
                     ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
    and run:        (c', c'')
                    ∈ mttm_step (alphabet_enlarge_delta M) ^^ 8
  obtains c1 c2 c3 c4 c5 c6 c7 cM_new n ofs_r buf_r dest_r where
    (c', c1) ∈ mttm_step (ae_delta_ss1_ss2 M)
   and (c1, c2) ∈ mttm_step (ae_delta_ss2_ss3 M)
   and (c2, c3) ∈ mttm_step (ae_delta_ss3_ss4 M)
   and (c3, c4) ∈ mttm_step (ae_delta_ss4_ss5 M)
   and (c4, c5) ∈ mttm_step (ae_delta_ss5_ss6 M)
   and (c5, c6) ∈ mttm_step (ae_delta_ss6_ss7 M)
   and (c6, c7) ∈ mttm_step (ae_delta_ss7_ss8 M)
   and (c7, c'') ∈ mttm_step (ae_delta_ss8_ss1 M)
   and (cM, cM_new) ∈ mttm_step (delta_tm M) ^^ n
   and n ≤ card (UNIV :: 'c set)
   and n = card (UNIV :: 'c set)
         ∨ mt_state cM_new ∈ {t_tm M, r_tm M}
   and fst (mt_state c'') = mt_state cM_new
   and mt_state cM_new ∉ {t_tm M, r_tm M}
         ⟶ ae_inv_ss1 M c''
   and mt_state cM_new ∈ {t_tm M, r_tm M}
         ⟶ snd (snd (snd (snd (mt_state c'')))) = VFwd
   and ae_tape_in_gamma_block M c''
   and ae_buffer_in_gamma_block M c''
   and mt_state c4 = (mt_state cM_new, ofs_r, buf_r, dest_r, SS5)
   and ∀k<k_tm M. ae_window_invariant_general
          (mt_tape cM_new k) (mt_pos cM_new k)
          (dest_r k, ofs_r k) (buf_r k) (mt_pos c' k) (le_tm M)
   and ae_buffer_in_gamma_block M c4
   and mt_state cM_new ∈ Q_tm M
proof −
  let ?R = mttm_step (alphabet_enlarge_delta M)

  — Peel the 8-step chain into 8 individual full steps via numeral_eq_Suc-driven
Suc unfolding. Then rebuild the Rs prefix and Rt suffix for the helper invocations.
  from run obtain c1' c2' c3 c4 c5' c6' c7' where
    p1:         (c', c1') ∈ ?R
```

and *p2*: $(c1', c2') \in ?R$
and *p3*: $(c2', c3) \in ?R$
and *step4_full*: $(c3, c4) \in ?R$
and *p5*: $(c4, c5') \in ?R$
and *p6*: $(c5', c6') \in ?R$
and *p7*: $(c6', c7') \in ?R$
and *p8*: $(c7', c'') \in ?R$
by (*auto dest*!: *relpow_Suc_D2 simp*: *numeral_eq_Suc*)

— Rebuild $(c', c3) \in R^{\frown\frown}3$ via explicit Suc-form chain.
have *a0*: $(c', c1') \in ?R \mathbin{^{\frown\frown}} Suc\ 0$ **using** *p1* **by** *simp*
have *a1*: $(c', c2') \in ?R \mathbin{^{\frown\frown}} Suc\ (Suc\ 0)$
using *relpow_Suc_I*[*OF a0 p2*] **.**
have *a2*: $(c', c3) \in ?R \mathbin{^{\frown\frown}} Suc\ (Suc\ (Suc\ 0))$
using *relpow_Suc_I*[*OF a1 p3*] **.**
have *eq3*: $(3::nat) = Suc\ (Suc\ (Suc\ 0))$ **by** (*simp add*: *eval_nat_numeral*)
have *load_run*: $(c', c3) \in ?R \mathbin{^{\frown\frown}} 3$
using *a2* **unfolding** *eq3* **.**

— Rebuild $(c4, c'') \in R^{\frown\frown}4$.
have *b0*: $(c4, c5') \in ?R \mathbin{^{\frown\frown}} Suc\ 0$ **using** *p5* **by** *simp*
have *b1*: $(c4, c6') \in ?R \mathbin{^{\frown\frown}} Suc\ (Suc\ 0)$
using *relpow_Suc_I*[*OF b0 p6*] **.**
have *b2*: $(c4, c7') \in ?R \mathbin{^{\frown\frown}} Suc\ (Suc\ (Suc\ 0))$
using *relpow_Suc_I*[*OF b1 p7*] **.**
have *b3*: $(c4, c'') \in ?R \mathbin{^{\frown\frown}} Suc\ (Suc\ (Suc\ (Suc\ 0)))$
using *relpow_Suc_I*[*OF b2 p8*] **.**
have *eq4*: $(4::nat) = Suc\ (Suc\ (Suc\ (Suc\ 0)))$ **by** (*simp add*: *eval_nat_numeral*)
have *writeback_run*: $(c4, c'') \in ?R \mathbin{^{\frown\frown}} 4$
using *b3* **unfolding** *eq4* **.**

— Invoke *load_chain* on $(c', c3) \in R^{\frown\frown}3$. Returns fresh c1, c2 (the substep intermediates) with the substep facts and c3 invariants. These c1, c2 may differ syntactically from c1', c2' above, but we propagate forward using only c1, c2.
obtain *c1 c2* **where**
step1_sub: $(c', c1) \in mttm_step\ (ae_delta_ss1_ss2\ M)$
and *step1_full*: $(c', c1) \in ?R$
and *step2_sub*: $(c1, c2) \in mttm_step\ (ae_delta_ss2_ss3\ M)$
and *step2_full*: $(c1, c2) \in ?R$
and *step3_sub*: $(c2, c3) \in mttm_step\ (ae_delta_ss3_ss4\ M)$
and *step3_full*: $(c2, c3) \in ?R$
and *inv_ss4_c3*: *ae_inv_ss4 M c3*
and *gamma_c3*: *ae_tape_in_gamma_block M c3*
and *buf_gamma_c3*: *ae_buffer_in_gamma_block M c3*
and *c3_q_eq*: $fst\ (mt_state\ c3) = qM'$
and *pos_link_c3*: *ae_position_link M c3*
by (*rule ae_backward_stage_load_chain*
[*OF vM c'_state inv_ss1 gamma_c' buf_gamma pos_link_c'*
qM'_neq_t qM'_neq_r load_run])

— Recover *c3_idx_ss4* for *compute_extract*.
have *c3_idx_ss4*: *snd* (*snd* (*snd* (*snd* (*mt_state c3*)))) = *SS4*
using *inv_ss4_c3* **unfolding** *ae_inv_ss4_def*
by (*cases mt_state c3*) *auto*

— Invoke *compute_extract*. Returns the cM trace, termination disjunction, c4 5-tuple state shape (used internally only), and tape/buffer/window propagation facts.
obtain *cM_new buf′ end_pos n* **where**
step4_sub: (*c3*, *c4*) ∈ *mttm_step* (*ae_delta_ss4_ss5 M*)
and *cM_trace*: (*cM*, *cM_new*) ∈ *mttm_step* (*delta_tm M*) ^^ *n*
and *n_bnd*: *n* ≤ *card* (*UNIV* :: *′c set*)
and *halt_or_full*:
n = *card* (*UNIV* :: *′c set*)
∨ *mt_state cM_new* ∈ {*t_tm M*, *r_tm M*}
and *c4_state*:
mt_state c4 = (*mt_state cM_new*,
λ*k*. *if k* < *k_tm M then snd* (*end_pos k*)
else init_offset k,
buf′,
λ*k*. *if k* < *k_tm M then fst* (*end_pos k*)
else init_dest k, *SS5*)
and *c4_tape*: *mt_tape c4* = *mt_tape c′*
and *window_cM_new*:
∀ *k*<*k_tm M*. *ae_window_invariant_general*
(*mt_tape cM_new k*) (*mt_pos cM_new k*)
(*end_pos k*) (*buf′ k*) (*mt_pos c′ k*) (*le_tm M*)
and *no_le_cM_new*:
∀ *k*. (*mt_pos c′ k* = *0*
⟶ (∀ *i*. *i* < *card* (*UNIV* :: *′c set*)
⟶ *mt_tape cM_new k* (*Suc i*) ≠ *le_tm M*))
∧ (*mt_pos c′ k* = *1*
⟶ (∀ *i*. *i* < *2* ∗ *card* (*UNIV* :: *′c set*)
⟶ *mt_tape cM_new k* (*Suc i*) ≠ *le_tm M*))
∧ (*mt_pos c′ k* ≥ *2*
⟶ (∀ *i*. *i* < *3* ∗ *card* (*UNIV* :: *′c set*)
⟶ *mt_tape cM_new k*
((*mt_pos c′ k* − *2*) ∗ *card* (*UNIV* :: *′c set*)
+ *1* + *i*)
≠ *le_tm M*))
and *buf′_le_pos0*:
∀ *k*<*k_tm M*. *mt_pos c′ k* = *0*
⟶ *fst* (*buf′ k*) ≠ *LE_block* (*le_tm M*)
by (*rule ae_backward_stage_compute_extract*
[*OF vM lu val_cM sim c′_state le_anchor le_neq_bl no_le_per_tape*
step1_sub step2_sub step3_sub step4_full c3_idx_ss4])

— First-order projections from *c4_state* these are the inputs to *writeback_chain*

(*q_out* = *mt_state cM_new*) and the bridge facts used in the final discharge.
have *c4_q*: *fst* (*mt_state c4*) = *mt_state cM_new*
using *c4_state* **by** *simp*
have *c4_idx_ss5*: *snd* (*snd* (*snd* (*mt_state c4*)))) = *SS5*
using *c4_state* **by** *simp*

— Propagate *valid_config_mttm M* through the n-step substrate trace to obtain *mt_state cM_new* ∈ *Q_tm M*. Inline induction since no packaged multi-step preservation lemma exists.
have *val_cM_new*: *valid_config_mttm M cM_new*
proof −
have *preserve*: ⋀*c_b n0*. (*cM*, *c_b*) ∈ *mttm_step* (*delta_tm M*) ^^ *n0*
⟹ *valid_config_mttm M c_b*
proof −
fix *c_b n0*
assume (*cM*, *c_b*) ∈ *mttm_step* (*delta_tm M*) ^^ *n0*
thus *valid_config_mttm M c_b*
proof (*induction n0 arbitrary*: *c_b*)
case *0* **thus** *?case* **using** *val_cM* **by** *simp*
next
case (*Suc n0*)
from *Suc.prems* **obtain** *c_mid* **where**
ih: (*cM*, *c_mid*) ∈ *mttm_step* (*delta_tm M*) ^^ *n0*
and *step_last*: (*c_mid*, *c_b*) ∈ *mttm_step* (*delta_tm M*)
by (*auto elim*: *relpow_Suc_E*)
have *val_mid*: *valid_config_mttm M c_mid*
using *Suc.IH*[*OF ih*] **.**
show *?case* **by** (*rule valid_step_mttm*[*OF vM step_last val_mid*])
qed
qed
show *?thesis* **using** *preserve*[*OF cM_trace*] **.**
qed
have *q_in_Q*: *mt_state cM_new* ∈ *Q_tm M*
proof −
obtain *Q Sg Gm bl le dt s t r kM* **where**
Mr: *M* = *MTTM Q Sg Gm bl le dt s t r kM* **by** (*cases M*)
obtain *q ts nn* **where** *cr*: *cM_new* = *Config*$_M$ *q ts nn* **by** (*cases cM_new*)
show *?thesis* **using** *val_cM_new Mr cr* **by** *simp*
qed

— Push gamma / buffer-gamma through step4 to c4.
have *gamma_c4*: *ae_tape_in_gamma_block M c4*
using *ae_step_alphabet_enlarge_gamma_preserve*[*OF gamma_c3 step4_full*]
.
have *buf_gamma_c4*: *ae_buffer_in_gamma_block M c4*
using *ae_step_alphabet_enlarge_buffer_gamma_preserve*
[*OF vM buf_gamma_c3 gamma_c3 step4_full*] **.**

— Assemble *ae_inv_ss5 M c4* for *writeback_chain*.

have *inv_ss5_c4*: *ae_inv_ss5 M c4*
unfolding *ae_inv_ss5_def* **using** *c4_q c4_idx_ss5 q_in_Q*
by (*cases mt_state c4*) *auto*

— Invoke *writeback_chain*. *q_out* = *mt_state cM_new* (via *c4_q*); writeback exposes c5, c6, c7 substep facts and the halt-aware c" status keyed on *q_out*.
obtain *c5 c6 c7* **where**
step5_sub: (*c4*, *c5*) ∈ *mttm_step* (*ae_delta_ss5_ss6 M*)
and *step5_full*: (*c4*, *c5*) ∈ *?R*
and *step6_sub*: (*c5*, *c6*) ∈ *mttm_step* (*ae_delta_ss6_ss7 M*)
and *step6_full*: (*c5*, *c6*) ∈ *?R*
and *step7_sub*: (*c6*, *c7*) ∈ *mttm_step* (*ae_delta_ss7_ss8 M*)
and *step7_full*: (*c6*, *c7*) ∈ *?R*
and *step8_sub*: (*c7*, *c''*) ∈ *mttm_step* (*ae_delta_ss8_ss1 M*)
and *step8_full*: (*c7*, *c''*) ∈ *?R*
and *c''_non_halt*:
mt_state cM_new ∉ {*t_tm M*, *r_tm M*}
⟶ *ae_inv_ss1 M c''*
and *c''_halt*:
mt_state cM_new ∈ {*t_tm M*, *r_tm M*}
⟶ (*case mt_state c'' of* (*qH*, _, _, _, *idx*)
⇒ *qH* ∈ {*t_tm M*, *r_tm M*} ∧ *idx* = *VFwd*)
and *c''_q_eq*: *fst* (*mt_state c''*) = *mt_state cM_new*
and *gamma_c''*: *ae_tape_in_gamma_block M c''*
and *buf_gamma_c''*: *ae_buffer_in_gamma_block M c''*
by (*rule ae_backward_stage_writeback_chain*
[*OF vM inv_ss5_c4 c4_q q_in_Q gamma_c4 buf_gamma_c4*
writeback_run])

— Refine *c''_halt*'s tuple-case form to a first-order projection on *snd^4* (*mt_state c''*).
have *halt_inv_idx*:
mt_state cM_new ∈ {*t_tm M*, *r_tm M*}
⟶ *snd* (*snd* (*snd* (*snd* (*mt_state c''*)))) = *VFwd*
proof
assume *h*: *mt_state cM_new* ∈ {*t_tm M*, *r_tm M*}
have *shape*:
case mt_state c'' of (*qH*, _, _, _, *idx*)
⇒ *qH* ∈ {*t_tm M*, *r_tm M*} ∧ *idx* = *VFwd*
using *mp*[*OF c''_halt h*] **.**
show *snd* (*snd* (*snd* (*snd* (*mt_state c''*)))) = *VFwd*
using *shape* **by** (*cases mt_state c''*) *auto*
qed

— Run-arm SS5 exposure for the det-free reverse arm. Name the three SS5 stage fields (offset, buffer, dest) so the consumer can feed them to *nae_sim_proto*; re-express the window correspondence in (*dest*, *ofs*) bp form (the proto's shape). The inactive-tape tails travel inside *buf_gamma_c4* (*ae_buffer_in_gamma_block* ≡ *ae_valid_stage*: *buf* / *dest* frozen to *init* beyond *k_tm M*).

```
  define ofs_r where
    ofs_r = (λk. if k < k_tm M then snd (end_pos k) else init_offset k)
  define buf_r where buf_r = buf'
  define dest_r where
    dest_r = (λk. if k < k_tm M then fst (end_pos k) else init_dest k)
  have R_c4_state: mt_state c4 = (mt_state cM_new, ofs_r, buf_r, dest_r,
SS5)
    using c4_state by (simp add: ofs_r_def buf_r_def dest_r_def)
  have R_run_window:
      ∀k<k_tm M. ae_window_invariant_general
            (mt_tape cM_new k) (mt_pos cM_new k)
            (dest_r k, ofs_r k) (buf_r k) (mt_pos c' k) (le_tm M)
  proof (intro allI impI)
    fix k assume klt: k < k_tm M
    have bp_eq: (dest_r k, ofs_r k) = end_pos k
      using klt by (simp add: dest_r_def ofs_r_def)
    show ae_window_invariant_general (mt_tape cM_new k) (mt_pos cM_new
k)
          (dest_r k, ofs_r k) (buf_r k) (mt_pos c' k) (le_tm M)
        using window_cM_new[rule_format, OF klt] bp_eq by (simp add:
buf_r_def)
  qed

  — Final discharge: 19 first-order conjuncts, no HOU patterns.
  show ?thesis
    by (rule that[OF step1_sub step2_sub step3_sub step4_sub
                     step5_sub step6_sub step7_sub step8_sub
                     cM_trace n_bnd halt_or_full c''_q_eq
                     c''_non_halt halt_inv_idx gamma_c'' buf_gamma_c''
                     R_c4_state R_run_window buf_gamma_c4 q_in_Q])
qed
```

Reverse-arm top-level wrapper (Step 3f). Given the simulation invariant at an SS1 boundary configuration c' and an 8-step *alphabet_enlarge_delta*-chain $(c', c'') \in R\widehat{\ }\widehat{\ }8$, produces the corresponding substrate trace $(cM, cM_new) \in step\widehat{\ }n$ with $n \leq c$ and asserts the simulation invariant at the new boundary configuration c''.

This is the reverse-arm counterpart to *ae_simulates_forward_stage_general*. The internal decomposition is handled by *ae_backward_stage_decompose* (Step 3e); this wrapper unpacks the simulation invariant via *ae_forward_stage_unpack_sim*, invokes the decompose lemma, and re-packages the conclusions in terms of the *ae_simulates* invariant for the consumer (Step 4 validation-trace-unique-to-SS1, ultimately *alphabet_enlarge_language*'s reverse implication).

The body has two substantive components beyond the decompose invocation:

- **le_anchor* preservation at c''**: each of the 8 substeps preserves

mt_tape c kk 0 = LE_block (le_tm M). Discharged directly via the substrate's *mttm_relpow_LE_pos0_preserve* applied to the enlarged machine (post [*ae−drop−a−finite*], this is a one-shot bridge with no per-substep derivation needed).

- ***ae_simulates M cM_new c''***: the full simulation invariant tape correspondence, position decoding in the non-halt branch, halt-shape in the halt branch. Derived using the substep facts from Step 3e plus the same regime-aware writeback machinery the forward arm uses (*pos_link* propagation, tape preservation across writeback writes, per-tape regime case-splits for position decoding).

  An earlier draft of this wrapper also exposed a position displacement disjunct (*mt_pos c'' kk* $\in \{p,\ p{+}1,\ p{-}1\}$ per tape). Audit showed the consuming wrapper (*ae_trace_decode*) doesn't need it; the forward arm's parallel *disp_c8* was bound at its only call site (*ae_simulation_phase_chunked*) but never referenced. Dropped as dead code.

```
lemma ae_backward_stage:
  fixes M :: ('q, 'a) mttm
    and cM :: ('a, 'q) mt_config
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes vM:          valid_mttm M
      and lu:          le_unique M
      and sim:         ae_simulates M cM c'
      and val_cM:      valid_config_mttm M cM
      and q_neq_t:     mt_state cM ≠ t_tm M
      and q_neq_r:     mt_state cM ≠ r_tm M
      and buf_gamma:   ae_buffer_in_gamma_block M c'
      and le_anchor:   ∀ kk<k_tm M. mt_tape c' kk 0 = LE_block (le_tm M)
      and le_neq_bl:   le_tm M ≠ bl_tm M
      and no_le_per_tape:
          ∀ k. (mt_pos c' k ≥ 2
                ⟶ (∀ i. i < 3 * card (UNIV :: 'c set)
                      ⟶ mt_tape cM k
                          ((mt_pos c' k − 2) * card (UNIV :: 'c set)
                            + 1 + i)
                        ≠ le_tm M))
              ∧ (mt_pos c' k = 1
                 ⟶ (∀ i. i < 2 * card (UNIV :: 'c set)
                       ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
              ∧ (mt_pos c' k = 0
                 ⟶ (∀ i. i < card (UNIV :: 'c set)
                       ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
      and run:         (c', c'') ∈ mttm_step (alphabet_enlarge_delta M) ^^ 8
  obtains cM_new n where
```

```
    (cM, cM_new) ∈ mttm_step (delta_tm M) ^^ n
  and n ≤ card (UNIV :: 'c set)
  and n = card (UNIV :: 'c set)
        ∨ mt_state cM_new ∈ {t_tm M, r_tm M}
  and ae_simulates M cM_new c''
  and ae_buffer_in_gamma_block M c''
  and ∀ kk<k_tm M. mt_tape c'' kk 0 = LE_block (le_tm M)
proof −
 — Derive mt_state cM ∈ Q_tm M from val_cM for the sim-unpack invocation.
 have qM_in_Q: mt_state cM ∈ Q_tm M
 proof −
  obtain Q Sg Gm bl le dt s t r kM where
     Mr: M = MTTM Q Sg Gm bl le dt s t r kM by (cases M)
  obtain q ts nn where cr: cM = Config_M q ts nn by (cases cM)
  show ?thesis using val_cM Mr cr by simp
 qed

 — Unpack the simulation invariant at c'.
 obtain qM' ofs buf dest where
    c'_state:     mt_state c' = (qM', ofs, buf, dest, SS1)
  and qM_eq:      mt_state cM = qM'
  and qM'_in_Q:   qM' ∈ Q_tm M
  and qM'_neq_t:  qM' ≠ t_tm M
  and qM'_neq_r:  qM' ≠ r_tm M
  and tape_corr:  ∀ k<k_tm M. ae_tape_correspondence (le_tm M)
                        (mt_tape cM k) (mt_tape c' k)
  and pos_decode: ∀ k<k_tm M. mt_pos cM k = ae_decode_pos (mt_pos c' k)
(ofs k)
  and gamma_c':   ae_tape_in_gamma_block M c'
  and inv_ss1:    ae_inv_ss1 M c'
  and pos_link_c': ae_position_link M c'
 by (rule ae_forward_stage_unpack_sim[OF sim qM_in_Q q_neq_t q_neq_r])

 — Invoke Step 3e to decompose the 8-step chain.
 obtain c1 c2 c3 c4 c5 c6 c7 cM_new n ofs_r buf_r dest_r where
    step1_sub: (c', c1) ∈ mttm_step (ae_delta_ss1_ss2 M)
  and step2_sub: (c1, c2) ∈ mttm_step (ae_delta_ss2_ss3 M)
  and step3_sub: (c2, c3) ∈ mttm_step (ae_delta_ss3_ss4 M)
  and step4_sub: (c3, c4) ∈ mttm_step (ae_delta_ss4_ss5 M)
  and step5_sub: (c4, c5) ∈ mttm_step (ae_delta_ss5_ss6 M)
  and step6_sub: (c5, c6) ∈ mttm_step (ae_delta_ss6_ss7 M)
  and step7_sub: (c6, c7) ∈ mttm_step (ae_delta_ss7_ss8 M)
  and step8_sub: (c7, c'') ∈ mttm_step (ae_delta_ss8_ss1 M)
  and cM_trace: (cM, cM_new) ∈ mttm_step (delta_tm M) ^^ n
  and n_bnd:    n ≤ card (UNIV :: 'c set)
  and halt_or_full:
        n = card (UNIV :: 'c set)
          ∨ mt_state cM_new ∈ {t_tm M, r_tm M}
  and c''_fst_eq: fst (mt_state c'') = mt_state cM_new
```

and *c''_non_halt*:
mt_state cM_new $\notin$ {*t_tm M*, *r_tm M*}
$\longrightarrow$ *ae_inv_ss1 M c''*
and *c''_halt_idx*:
mt_state cM_new $\in$ {*t_tm M*, *r_tm M*}
$\longrightarrow$ *snd* (*snd* (*snd* (*snd* (*mt_state c''*)))) = *VFwd*
and *gamma_c''*: *ae_tape_in_gamma_block M c''*
and *buf_gamma_c''*: *ae_buffer_in_gamma_block M c''*
and *run_c4_state*:
mt_state c4 = (*mt_state cM_new*, *ofs_r*, *buf_r*, *dest_r*, *SS5*)
and *run_window*:
$\forall$ *k*<*k_tm M*. *ae_window_invariant_general*
(*mt_tape cM_new k*) (*mt_pos cM_new k*)
(*dest_r k*, *ofs_r k*) (*buf_r k*) (*mt_pos c' k*) (*le_tm M*)
and *run_buf_gamma_c4*: *ae_buffer_in_gamma_block M c4*
and *run_q_in_Q*: *mt_state cM_new* $\in$ *Q_tm M*
by (*rule ae_backward_stage_decompose*
[*OF vM lu sim val_cM c'_state inv_ss1 gamma_c' buf_gamma*
pos_link_c' le_anchor le_neq_bl qM'_neq_t qM'_neq_r
no_le_per_tape run])

— The *le_anchor*-at-*c''* conclusion collapses to a direct invocation of the substrate-level *mttm_relpow_LE_pos0_preserve* applied to the enlarged machine, with the *delta_tm* / *le_tm* bridges discharging the *alphabet_enlarge_delta* $\leftrightarrow$ *delta_tm* (*alphabet_enlarge M*) and *LE_block* (*le_tm M*) $\leftrightarrow$ *le_tm* (*alphabet_enlarge M*) identities.
have *le_anchor_c''*: $\forall$ *kk*<*k_tm M*. *mt_tape c'' kk 0* = *LE_block* (*le_tm M*)
proof (*intro allI impI*)
fix *kk*
assume *kklt*: *kk* < *k_tm M*
let *?M'* = *alphabet_enlarge M*
:: (*'q* $\times$ (*'a*, *'c*) *ae_stage*,
'c $\Rightarrow$ *'a*) *mttm*
have *vM'*: *valid_mttm ?M'*
by (*rule alphabet_enlarge_wf*[*OF vM*])
have *run'*: (*c'*, *c''*) $\in$ *mttm_step* (*delta_tm ?M'*) ^^ *8*
using *run* **by** (*simp add*: *delta_tm_alphabet_enlarge*)
have *le_at_c'_kk*: *mt_tape c' kk 0* = *le_tm ?M'*
using *le_anchor*[*rule_format*, *OF kklt*] **by** (*simp add*: *le_tm_alphabet_enlarge*)
have *mt_tape c'' kk 0* = *le_tm ?M'*
by (*rule mttm_relpow_LE_pos0_preserve*[*OF vM' run' le_at_c'_kk*])
then show *mt_tape c'' kk 0* = *LE_block* (*le_tm M*)
by (*simp add*: *le_tm_alphabet_enlarge*)
qed
— Sub-step *ae_sim_c''* (*det*-free): rather than identify the backward boundary *c''* with the forward witness *c'''* by chain uniqueness (which needs *det_mttm M*), apply *nae_sim_proto*. The forward arm rebuilds a *c'''* simulating the *same* extracted M-trace endpoint *cM_new*, now exposing its SS5 config; the two arms

share the SS4 source *c3* (the functional SS1→SS4 builders force *c1 = c1f*, *c2 = c2f*, *c3 = c3f*); the window invariant against the common *cM_new* pins the SS5 outputs modulo the pos=0 offset don't-care; and the inactive-tape tails agree because *ae_buffer_in_gamma_block* freezes both arms' *buf* / *dest* to *init* beyond *k_tm M*. The forward arm's full *ae_simulates* transports across the SS5→SS1 congruence to c''.

have *ae_sim_c''*: *ae_simulates M cM_new c''*
proof −
— Forward arm: a c''' simulating the same *cM_new*, with its full substep chain + SS5 exposure.
obtain *c''' c1f c2f c3f c4f c5f c6f c7f ofs_f buf_f dest_f* **where**
fwd_chain: $(c', c''') \in$ *mttm_step* (*alphabet_enlarge_delta M*) $\widehat{\ }\widehat{\ }$ *8*
and *sim_c'''*: *ae_simulates M cM_new c'''*
and *fwd_bufg_c'''*: *ae_buffer_in_gamma_block M c'''*
and *fwd_le_c'''*: $\forall$ *kk*<*k_tm M*. *mt_tape c''' kk 0 = LE_block* (*le_tm M*)
and *f_step1*: $(c', c1f) \in$ *mttm_step* (*ae_delta_ss1_ss2 M*)
and *f_step2*: $(c1f, c2f) \in$ *mttm_step* (*ae_delta_ss2_ss3 M*)
and *f_step3*: $(c2f, c3f) \in$ *mttm_step* (*ae_delta_ss3_ss4 M*)
and *f_step4*: $(c3f, c4f) \in$ *mttm_step* (*ae_delta_ss4_ss5 M*)
and *f_step5*: $(c4f, c5f) \in$ *mttm_step* (*ae_delta_ss5_ss6 M*)
and *f_step6*: $(c5f, c6f) \in$ *mttm_step* (*ae_delta_ss6_ss7 M*)
and *f_step7*: $(c6f, c7f) \in$ *mttm_step* (*ae_delta_ss7_ss8 M*)
and *f_step8*: $(c7f, c''') \in$ *mttm_step* (*ae_delta_ss8_ss1 M*)
and *fwd_c4_state*:
mt_state c4f = (*mt_state cM_new*, *ofs_f*, *buf_f*, *dest_f*, *SS5*)
and *fwd_window*:
$\forall$ *kk*<*k_tm M*. *ae_window_invariant_general*
(*mt_tape cM_new kk*) (*mt_pos cM_new kk*)
(*dest_f kk*, *ofs_f kk*) (*buf_f kk*) (*mt_pos c' kk*) (*le_tm M*)
and *fwd_buf_gamma_c4f*: *ae_buffer_in_gamma_block M c4f*
by (*rule ae_simulates_forward_stage_general*
[*OF vM lu sim qM_in_Q q_neq_t q_neq_r buf_gamma n_bnd*
cM_trace halt_or_full le_anchor le_neq_bl
no_le_per_tape])

— Shared SS4 source: both arms load SS1→SS4 from c' through the functional substep builders.
have *c1_eq*: *c1 = c1f*
by (*rule mttm_step_functional_of_delta*
[*OF ae_delta_ss1_ss2_functional step1_sub f_step1*])
have *f_step2'*: (*c1*, *c2f*) ∈ *mttm_step* (*ae_delta_ss2_ss3 M*)
using *f_step2 c1_eq* **by** *simp*
have *c2_eq*: *c2 = c2f*
by (*rule mttm_step_functional_of_delta*
[*OF ae_delta_ss2_ss3_functional step2_sub f_step2'*])
have *f_step3'*: (*c2*, *c3f*) ∈ *mttm_step* (*ae_delta_ss3_ss4 M*)
using *f_step3 c2_eq* **by** *simp*
have *c3_eq*: *c3 = c3f*

```
    by (rule mttm_step_functional_of_delta
          [OF ae_delta_ss3_ss4_functional step3_sub f_step3'])
  have fwd45: (c3, c4f) ∈ mttm_step (ae_delta_ss4_ss5 M)
    using f_step4 c3_eq by simp

  — Inactive-tape tails: ae_buffer_in_gamma_block carries the ae_valid_stage
tail, freezing buf / dest to init beyond k_tm M on both arms, so they agree there.
  have run_tails:
      (∀j≥k_tm M. buf_r j = init_buffer (le_tm M) j)
    ∧ (∀j≥k_tm M. dest_r j = init_dest j)
      using run_buf_gamma_c4[unfolded ae_buffer_in_gamma_block_def
run_c4_state]
    by simp
  have fwd_tails:
      (∀j≥k_tm M. buf_f j = init_buffer (le_tm M) j)
    ∧ (∀j≥k_tm M. dest_f j = init_dest j)
      using fwd_buf_gamma_c4f[unfolded ae_buffer_in_gamma_block_def
fwd_c4_state]
    by simp
  have buf_tail:
      ∀k. ¬ k < k_tm M
        ⟶ ae_blk_agree (le_tm M) (buf_r k) (buf_f k)
  proof (intro allI impI)
    fix k assume ¬ k < k_tm M
    hence kge: k_tm M ≤ k by simp
    have buf_r k = buf_f k using run_tails fwd_tails kge by simp
    thus ae_blk_agree (le_tm M) (buf_r k) (buf_f k)
      by (simp add: ae_blk_agree_def)
  qed
  have dest_tail: ∀k. ¬ k < k_tm M ⟶ dest_r k = dest_f k
  proof (intro allI impI)
    fix k assume ¬ k < k_tm M
    hence kge: k_tm M ≤ k by simp
    show dest_r k = dest_f k using run_tails fwd_tails kge by simp
  qed

  show ?thesis
    by (rule nae_sim_proto[OF run_q_in_Q step4_sub fwd45 run_c4_state
                              fwd_c4_state run_window fwd_window
                              buf_tail dest_tail
                              step5_sub step6_sub step7_sub step8_sub
                              f_step5 f_step6 f_step7 f_step8
                              sim_c''' gamma_c'])
qed

show ?thesis
  by (rule that[OF cM_trace n_bnd halt_or_full ae_sim_c''
                   buf_gamma_c'' le_anchor_c''])
qed
```

2.29.8 Substep cycle progression

Substep-cycle progression: a single full-delta step from a config whose substep idx is *SS1* lands at *SS2*.

Mechanism: the full *alphabet_enlarge_delta* is the union of 16 per-substep deltas, each of whose source tuple fixes the substep idx to its own $SS\langle N\rangle$ / validation marker. From an *SS1*-source step, only *ae_delta_ss1_ss2*'s source matches. Combined with *ae_substep_state_shape*, the post-substep idx is forced to *SS2*.

Foundation for the substep-cycle tracking that forces $m \geq 8$ in the reverse chunked-phase induction; without this, $m \in \{\!|1,...,7|\!\}$-paths from *SS1* can't be ruled out as candidates for reaching *VFwd*-halt.

lemma *ae_step_from_SS1_lands_at_SS2*:
  fixes $M :: ('q, 'a)$ *mttm*
    and $c'\ c'' :: ('c :: enum \Rightarrow 'a,$
                $'q \times ('a, 'c)$ *ae_stage*$)$ *mt_config*
  assumes *idx*: *snd* (*snd* (*snd* (*snd* (*mt_state* c')))) = *SS1*
    and *step*: $(c', c'') \in$ *mttm_step* (*alphabet_enlarge_delta* M)
  shows *snd* (*snd* (*snd* (*snd* (*mt_state* c'')))) = *SS2*
proof −
  from *step* **obtain** $s\ ts\ n\ s'\ a'\ d$ **where**
    c'*_eq*: $c' = Config_M\ s\ ts\ n$
    and c''*_eq*: $c'' = Config_M\ s'\ (\lambda k.\ (ts\ k)(n\ k := a'\ k))$
                $(\lambda k.\ go_dir\ (d\ k)\ (n\ k))$
    and *rel*: $(s, (\lambda k.\ ts\ k\ (n\ k)), s', a', d) \in$ *alphabet_enlarge_delta* M
    by (*auto elim*: *mttm_step.cases*)
  obtain *qS oS bS dS sxS* **where** *s_split*: $s = (qS, oS, bS, dS, sxS)$
    by (*cases s*) *auto*
  from *idx c'_eq s_split* **have** *sxS_SS1*: $sxS = SS1$ **by** *simp*
  have *rel_in_ss1_ss2*:
      $(s, (\lambda k.\ ts\ k\ (n\ k)), s', a', d) \in$ *ae_delta_ss1_ss2* M
    using *rel s_split sxS_SS1*
    unfolding *alphabet_enlarge_delta_def*
          ae_delta_val_fwd_advance_def
          ae_delta_val_fwd_to_padded_def
          ae_delta_val_fwd_reject_def
          ae_delta_val_fwd_to_ret_def
          ae_delta_val_pad_to_ret_def
          ae_delta_val_pad_reject_def
          ae_delta_val_ret_step_def
          ae_delta_val_ret_to_sim_def
          ae_delta_ss2_ss3_def
          ae_delta_ss3_ss4_def
          ae_delta_ss4_ss5_def
          ae_delta_ss5_ss6_def
          ae_delta_ss6_ss7_def
          ae_delta_ss7_ss8_def
          ae_delta_ss8_ss1_def

```
    by auto
  obtain qN oN bN dN where s'_split: s' = (qN, oN, bN, dN, SS2)
    using rel_in_ss1_ss2 unfolding ae_delta_ss1_ss2_def by blast
  show ?thesis using c''_eq s'_split by simp
qed

lemma ae_step_from_SS2_lands_at_SS3:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes idx: snd (snd (snd (snd (mt_state c')))) = SS2
      and step: (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
  shows snd (snd (snd (snd (mt_state c')))) = SS3
proof −
  from step obtain s ts n s' a' d where
      c'_eq: c' = Config_M s ts n
    and c''_eq: c'' = Config_M s' (λk. (ts k)(n k := a' k))
                                  (λk. go_dir (d k) (n k))
    and rel: (s, (λk. ts k (n k)), s', a', d) ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  obtain qS oS bS dS sxS where s_split: s = (qS, oS, bS, dS, sxS)
    by (cases s) auto
  from idx c'_eq s_split have sxS_SS2: sxS = SS2 by simp
  have rel_in_ss2_ss3:
      (s, (λk. ts k (n k)), s', a', d) ∈ ae_delta_ss2_ss3 M
    using rel s_split sxS_SS2
    unfolding alphabet_enlarge_delta_def
              ae_delta_val_fwd_advance_def
              ae_delta_val_fwd_to_padded_def
              ae_delta_val_fwd_reject_def
              ae_delta_val_fwd_to_ret_def
              ae_delta_val_pad_to_ret_def
              ae_delta_val_pad_reject_def
              ae_delta_val_ret_step_def
              ae_delta_val_ret_to_sim_def
              ae_delta_ss1_ss2_def
              ae_delta_ss3_ss4_def
              ae_delta_ss4_ss5_def
              ae_delta_ss5_ss6_def
              ae_delta_ss6_ss7_def
              ae_delta_ss7_ss8_def
              ae_delta_ss8_ss1_def
    by auto
  obtain qN oN bN dN where s'_split: s' = (qN, oN, bN, dN, SS3)
    using rel_in_ss2_ss3 unfolding ae_delta_ss2_ss3_def by blast
  show ?thesis using c''_eq s'_split by simp
qed

lemma ae_step_from_SS3_lands_at_SS4:
```

```
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes idx: snd (snd (snd (snd (mt_state c')))) = SS3
      and step: (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
  shows snd (snd (snd (snd (mt_state c')))) = SS4
proof −
  from step obtain s ts n s' a' d where
      c'_eq: c' = Config_M s ts n
    and c''_eq: c'' = Config_M s' (λk. (ts k)(n k := a' k))
                                  (λk. go_dir (d k) (n k))
    and rel: (s, (λk. ts k (n k)), s', a', d) ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  obtain qS oS bS dS sxS where s_split: s = (qS, oS, bS, dS, sxS)
    by (cases s) auto
  from idx c'_eq s_split have sxS_SS3: sxS = SS3 by simp
  have rel_in_ss3_ss4:
      (s, (λk. ts k (n k)), s', a', d) ∈ ae_delta_ss3_ss4 M
    using rel s_split sxS_SS3
    unfolding alphabet_enlarge_delta_def
              ae_delta_val_fwd_advance_def
              ae_delta_val_fwd_to_padded_def
              ae_delta_val_fwd_reject_def
              ae_delta_val_fwd_to_ret_def
              ae_delta_val_pad_to_ret_def
              ae_delta_val_pad_reject_def
              ae_delta_val_ret_step_def
              ae_delta_val_ret_to_sim_def
              ae_delta_ss1_ss2_def
              ae_delta_ss2_ss3_def
              ae_delta_ss4_ss5_def
              ae_delta_ss5_ss6_def
              ae_delta_ss6_ss7_def
              ae_delta_ss7_ss8_def
              ae_delta_ss8_ss1_def
    by auto
  obtain qN oN bN dN where s'_split: s' = (qN, oN, bN, dN, SS4)
    using rel_in_ss3_ss4 unfolding ae_delta_ss3_ss4_def by blast
  show ?thesis using c''_eq s'_split by simp
qed

lemma ae_step_from_SS4_lands_at_SS5:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes idx: snd (snd (snd (snd (mt_state c')))) = SS4
      and step: (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
  shows snd (snd (snd (snd (mt_state c')))) = SS5
proof −
```

```
  from step obtain s ts n s' a' d where
      c'_eq: c' = Config_M s ts n
    and c''_eq: c'' = Config_M s' (λk. (ts k)(n k := a' k))
                                  (λk. go_dir (d k) (n k))
    and rel: (s, (λk. ts k (n k)), s', a', d) ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  obtain qS oS bS dS sxS where s_split: s = (qS, oS, bS, dS, sxS)
    by (cases s) auto
  from idx c'_eq s_split have sxS_SS4: sxS = SS4 by simp
  have rel_in_ss4_ss5:
      (s, (λk. ts k (n k)), s', a', d) ∈ ae_delta_ss4_ss5 M
    using rel s_split sxS_SS4
    unfolding alphabet_enlarge_delta_def
              ae_delta_val_fwd_advance_def
              ae_delta_val_fwd_to_padded_def
              ae_delta_val_fwd_reject_def
              ae_delta_val_fwd_to_ret_def
              ae_delta_val_pad_to_ret_def
              ae_delta_val_pad_reject_def
              ae_delta_val_ret_step_def
              ae_delta_val_ret_to_sim_def
              ae_delta_ss1_ss2_def
              ae_delta_ss2_ss3_def
              ae_delta_ss3_ss4_def
              ae_delta_ss5_ss6_def
              ae_delta_ss6_ss7_def
              ae_delta_ss7_ss8_def
              ae_delta_ss8_ss1_def
    by auto
  obtain qN oN bN dN where s'_split: s' = (qN, oN, bN, dN, SS5)
    using rel_in_ss4_ss5 unfolding ae_delta_ss4_ss5_def by blast
  show ?thesis using c''_eq s'_split by simp
qed

lemma ae_step_from_SS5_lands_at_SS6:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes idx: snd (snd (snd (snd (mt_state c')))) = SS5
      and step: (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
  shows snd (snd (snd (snd (mt_state c'')))) = SS6
proof −
  from step obtain s ts n s' a' d where
      c'_eq: c' = Config_M s ts n
    and c''_eq: c'' = Config_M s' (λk. (ts k)(n k := a' k))
                                  (λk. go_dir (d k) (n k))
    and rel: (s, (λk. ts k (n k)), s', a', d) ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  obtain qS oS bS dS sxS where s_split: s = (qS, oS, bS, dS, sxS)
```

```
    by (cases s) auto
  from idx c'_eq s_split have sxS_SS5: sxS = SS5 by simp
  have rel_in_ss5_ss6:
      (s, (λk. ts k (n k)), s', a', d) ∈ ae_delta_ss5_ss6 M
    using rel s_split sxS_SS5
    unfolding alphabet_enlarge_delta_def
              ae_delta_val_fwd_advance_def
              ae_delta_val_fwd_to_padded_def
              ae_delta_val_fwd_reject_def
              ae_delta_val_fwd_to_ret_def
              ae_delta_val_pad_to_ret_def
              ae_delta_val_pad_reject_def
              ae_delta_val_ret_step_def
              ae_delta_val_ret_to_sim_def
              ae_delta_ss1_ss2_def
              ae_delta_ss2_ss3_def
              ae_delta_ss3_ss4_def
              ae_delta_ss4_ss5_def
              ae_delta_ss6_ss7_def
              ae_delta_ss7_ss8_def
              ae_delta_ss8_ss1_def
    by auto
  obtain qN oN bN dN where s'_split: s' = (qN, oN, bN, dN, SS6)
    using rel_in_ss5_ss6 unfolding ae_delta_ss5_ss6_def by blast
  show ?thesis using c''_eq s'_split by simp
qed

lemma ae_step_from_SS6_lands_at_SS7:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes idx: snd (snd (snd (snd (mt_state c')))) = SS6
      and step: (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
  shows snd (snd (snd (snd (mt_state c'')))) = SS7
proof −
  from step obtain s ts n s' a' d where
      c'_eq: c' = Config_M s ts n
    and c''_eq: c'' = Config_M s' (λk. (ts k)(n k := a' k))
                                  (λk. go_dir (d k) (n k))
    and rel: (s, (λk. ts k (n k)), s', a', d) ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  obtain qS oS bS dS sxS where s_split: s = (qS, oS, bS, dS, sxS)
    by (cases s) auto
  from idx c'_eq s_split have sxS_SS6: sxS = SS6 by simp
  have rel_in_ss6_ss7:
      (s, (λk. ts k (n k)), s', a', d) ∈ ae_delta_ss6_ss7 M
    using rel s_split sxS_SS6
    unfolding alphabet_enlarge_delta_def
              ae_delta_val_fwd_advance_def
```

```
        ae_delta_val_fwd_to_padded_def
        ae_delta_val_fwd_reject_def
        ae_delta_val_fwd_to_ret_def
        ae_delta_val_pad_to_ret_def
        ae_delta_val_pad_reject_def
        ae_delta_val_ret_step_def
        ae_delta_val_ret_to_sim_def
        ae_delta_ss1_ss2_def
        ae_delta_ss2_ss3_def
        ae_delta_ss3_ss4_def
        ae_delta_ss4_ss5_def
        ae_delta_ss5_ss6_def
        ae_delta_ss7_ss8_def
        ae_delta_ss8_ss1_def
    by auto
  obtain qN oN bN dN where s'_split: s' = (qN, oN, bN, dN, SS7)
    using rel_in_ss6_ss7 unfolding ae_delta_ss6_ss7_def by blast
  show ?thesis using c''_eq s'_split by simp
qed

lemma ae_step_from_SS7_lands_at_SS8:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes idx: snd (snd (snd (snd (mt_state c')))) = SS7
      and step: (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
  shows snd (snd (snd (snd (mt_state c'')))) = SS8
proof −
  from step obtain s ts n s' a' d where
      c'_eq: c' = Config_M s ts n
    and c''_eq: c'' = Config_M s' (λk. (ts k)(n k := a' k))
                                  (λk. go_dir (d k) (n k))
    and rel: (s, (λk. ts k (n k)), s', a', d) ∈ alphabet_enlarge_delta M
    by (auto elim: mttm_step.cases)
  obtain qS oS bS dS sxS where s_split: s = (qS, oS, bS, dS, sxS)
    by (cases s) auto
  from idx c'_eq s_split have sxS_SS7: sxS = SS7 by simp
  have rel_in_ss7_ss8:
      (s, (λk. ts k (n k)), s', a', d) ∈ ae_delta_ss7_ss8 M
    using rel s_split sxS_SS7
    unfolding alphabet_enlarge_delta_def
        ae_delta_val_fwd_advance_def
        ae_delta_val_fwd_to_padded_def
        ae_delta_val_fwd_reject_def
        ae_delta_val_fwd_to_ret_def
        ae_delta_val_pad_to_ret_def
        ae_delta_val_pad_reject_def
        ae_delta_val_ret_step_def
        ae_delta_val_ret_to_sim_def
```

ae_delta_ss1_ss2_def
ae_delta_ss2_ss3_def
ae_delta_ss3_ss4_def
ae_delta_ss4_ss5_def
ae_delta_ss5_ss6_def
ae_delta_ss6_ss7_def
ae_delta_ss8_ss1_def
by *auto*
obtain *qN oN bN dN* **where** *s'_split*: $s' = (qN,\ oN,\ bN,\ dN,\ SS8)$
using *rel_in_ss7_ss8* **unfolding** *ae_delta_ss7_ss8_def* **by** *blast*
show *?thesis* **using** *c''_eq s'_split* **by** *simp*
qed

SS8 is the cycle-closure substep: post-state lands at *SS1* (steady-state continuation) or *VFwd* (halt-branch, when the simulated M-state at SS8 is in $\{\!|t_tm\ M,\ r_tm\ M|\!\}$). The looser $SS1 \vee VFwd$ disjunction is what the reverse-arm chain-progression argument needs; the precise branch resolution is handled by *ae_step_ss8_ss1_invariant* / *ae_step_ss8_ss1_invariant_halt*.

lemma *ae_step_from_SS8_lands_at_SS1_or_VFwd*:
fixes $M :: ('q,\ 'a)\ mttm$
and $c'\ c'' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a,\ 'c)\ ae_stage)\ mt_config$
assumes *idx*: $snd\ (snd\ (snd\ (snd\ (mt_state\ c')))) = SS8$
and *step*: $(c',\ c'') \in mttm_step\ (alphabet_enlarge_delta\ M)$
shows $snd\ (snd\ (snd\ (snd\ (mt_state\ c'')))) \in \{SS1,\ VFwd\}$
proof −
from *step* **obtain** $s\ ts\ n\ s'\ a'\ d$ **where**
c'_eq: $c' = Config_M\ s\ ts\ n$
and c''_eq: $c'' = Config_M\ s'\ (\lambda k.\ (ts\ k)(n\ k := a'\ k))$
$(\lambda k.\ go_dir\ (d\ k)\ (n\ k))$
and *rel*: $(s,\ (\lambda k.\ ts\ k\ (n\ k)),\ s',\ a',\ d) \in alphabet_enlarge_delta\ M$
by (*auto elim*: *mttm_step.cases*)
obtain *qS oS bS dS sxS* **where** *s_split*: $s = (qS,\ oS,\ bS,\ dS,\ sxS)$
by (*cases s*) *auto*
from *idx c'_eq s_split* **have** *sxS_SS8*: $sxS = SS8$ **by** *simp*
have *rel_in_ss8_ss1*:
$(s,\ (\lambda k.\ ts\ k\ (n\ k)),\ s',\ a',\ d) \in ae_delta_ss8_ss1\ M$
using *rel s_split sxS_SS8*
unfolding *alphabet_enlarge_delta_def*
ae_delta_val_fwd_advance_def
ae_delta_val_fwd_to_padded_def
ae_delta_val_fwd_reject_def
ae_delta_val_fwd_to_ret_def
ae_delta_val_pad_to_ret_def
ae_delta_val_pad_reject_def
ae_delta_val_ret_step_def
ae_delta_val_ret_to_sim_def
ae_delta_ss1_ss2_def

```
            ae_delta_ss2_ss3_def
            ae_delta_ss3_ss4_def
            ae_delta_ss4_ss5_def
            ae_delta_ss5_ss6_def
            ae_delta_ss6_ss7_def
            ae_delta_ss7_ss8_def
    by auto
  obtain q ofs buf dest stage' where
      s_eq_full: s = (q, ofs, buf, dest, SS8)
    and s'_eq: s' = (q, stage')
    and stage'_eq: stage' = (if q ∈ {t_tm M, r_tm M}
                             then init_stage (le_tm M)
                             else (ofs, buf, init_dest, SS1))
    using rel_in_ss8_ss1 unfolding ae_delta_ss8_ss1_def by auto
  have init_stage_substep_VFwd:
      snd (snd (snd (init_stage (le_tm M) :: ('a, 'c) ae_stage))) = VFwd
    unfolding init_stage_def by simp
  show ?thesis
  proof (cases q ∈ {t_tm M, r_tm M})
    case True
    hence stage'_init: stage' = init_stage (le_tm M)
      using stage'_eq by simp
    have snd (snd (snd (snd (mt_state c'')))) = VFwd
      using c''_eq s'_eq stage'_init init_stage_substep_VFwd by simp
    thus ?thesis by simp
  next
    case False
    hence stage'_cont: stage' = (ofs, buf, init_dest, SS1)
      using stage'_eq by simp
    have snd (snd (snd (snd (mt_state c'')))) = SS1
      using c''_eq s'_eq stage'_cont by simp
    thus ?thesis by simp
  qed
qed
```

Chain-progression: from a config at substep *SS1*, no chain of length $1 \leq m \leq 7$ reaches a configuration at substep *VFwd*. The substep cycle $SS1 \to SS2 \to ... \to SS8 \to SS1 \vee VFwd$ takes exactly 8 steps from *SS1* to reach *VFwd*; shorter chains stay within $\{\!|SS2,...,SS8|\!\}$.

Used by the reverse chunked-phase argument to force $m \geq 8$ when an SS1-paired c' has a chain ending in the canonical halt config (*t_tm M*, *init_stage le*).

```
lemma ae_chain_from_SS1_short_not_VFwd:
  fixes M :: ('q, 'a) mttm
    and c' c'' :: ('c :: enum ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
    and m :: nat
  assumes idx_SS1: snd (snd (snd (snd (mt_state c')))) = SS1
      and m_lt_8:  0 < m ∧ m < 8
```

```
    and chain: (c', c'') ∈ mttm_step (alphabet_enlarge_delta M) ^^ m
  shows snd (snd (snd (snd (mt_state c'')))) ≠ VFwd
proof −
  let ?R = mttm_step (alphabet_enlarge_delta M)
  let ?idx = λc. snd (snd (snd (snd (mt_state c))))
  consider (m1) m = 1 | (m2) m = 2 | (m3) m = 3 | (m4) m = 4
        | (m5) m = 5 | (m6) m = 6 | (m7) m = 7
    using m_lt_8 by linarith
  thus ?thesis
  proof cases
    case m1
    have step1: (c', c'') ∈ ?R using chain m1 by simp
    have ?idx c'' = SS2
      by (rule ae_step_from_SS1_lands_at_SS2[OF idx_SS1 step1])
    thus ?thesis by simp
  next
    case m2
    from chain m2 obtain c1 where
        e1: (c', c1) ∈ ?R and e2: (c1, c'') ∈ ?R
      by (auto dest!: relpow_Suc_D2 simp: numeral_eq_Suc)
    have ?idx c1 = SS2
      by (rule ae_step_from_SS1_lands_at_SS2[OF idx_SS1 e1])
    hence ?idx c'' = SS3
      by (rule ae_step_from_SS2_lands_at_SS3[OF _ e2])
    thus ?thesis by simp
  next
    case m3
    from chain m3 obtain c1 c2 where
        e1: (c', c1) ∈ ?R and e2: (c1, c2) ∈ ?R
        and e3: (c2, c'') ∈ ?R
      by (auto dest!: relpow_Suc_D2 simp: numeral_eq_Suc)
    have x1: ?idx c1 = SS2
      by (rule ae_step_from_SS1_lands_at_SS2[OF idx_SS1 e1])
    have x2: ?idx c2 = SS3
      by (rule ae_step_from_SS2_lands_at_SS3[OF x1 e2])
    hence ?idx c'' = SS4
      by (rule ae_step_from_SS3_lands_at_SS4[OF _ e3])
    thus ?thesis by simp
  next
    case m4
    from chain m4 obtain c1 c2 c3 where
        e1: (c', c1) ∈ ?R and e2: (c1, c2) ∈ ?R
        and e3: (c2, c3) ∈ ?R and e4: (c3, c'') ∈ ?R
      by (auto dest!: relpow_Suc_D2 simp: numeral_eq_Suc)
    have x1: ?idx c1 = SS2
      by (rule ae_step_from_SS1_lands_at_SS2[OF idx_SS1 e1])
    have x2: ?idx c2 = SS3
      by (rule ae_step_from_SS2_lands_at_SS3[OF x1 e2])
    have x3: ?idx c3 = SS4
```

```
    by (rule ae_step_from_SS3_lands_at_SS4[OF x2 e3])
  hence ?idx c'' = SS5
    by (rule ae_step_from_SS4_lands_at_SS5[OF _ e4])
  thus ?thesis by simp
next
  case m5
  from chain m5 obtain c1 c2 c3 c4 where
      e1: (c', c1) ∈ ?R and e2: (c1, c2) ∈ ?R
      and e3: (c2, c3) ∈ ?R and e4: (c3, c4) ∈ ?R
      and e5: (c4, c'') ∈ ?R
    by (auto dest!: relpow_Suc_D2 simp: numeral_eq_Suc)
  have x1: ?idx c1 = SS2
    by (rule ae_step_from_SS1_lands_at_SS2[OF idx_SS1 e1])
  have x2: ?idx c2 = SS3
    by (rule ae_step_from_SS2_lands_at_SS3[OF x1 e2])
  have x3: ?idx c3 = SS4
    by (rule ae_step_from_SS3_lands_at_SS4[OF x2 e3])
  have x4: ?idx c4 = SS5
    by (rule ae_step_from_SS4_lands_at_SS5[OF x3 e4])
  hence ?idx c'' = SS6
    by (rule ae_step_from_SS5_lands_at_SS6[OF _ e5])
  thus ?thesis by simp
next
  case m6
  from chain m6 obtain c1 c2 c3 c4 c5 where
      e1: (c', c1) ∈ ?R and e2: (c1, c2) ∈ ?R
      and e3: (c2, c3) ∈ ?R and e4: (c3, c4) ∈ ?R
      and e5: (c4, c5) ∈ ?R and e6: (c5, c'') ∈ ?R
    by (auto dest!: relpow_Suc_D2 simp: numeral_eq_Suc)
  have x1: ?idx c1 = SS2
    by (rule ae_step_from_SS1_lands_at_SS2[OF idx_SS1 e1])
  have x2: ?idx c2 = SS3
    by (rule ae_step_from_SS2_lands_at_SS3[OF x1 e2])
  have x3: ?idx c3 = SS4
    by (rule ae_step_from_SS3_lands_at_SS4[OF x2 e3])
  have x4: ?idx c4 = SS5
    by (rule ae_step_from_SS4_lands_at_SS5[OF x3 e4])
  have x5: ?idx c5 = SS6
    by (rule ae_step_from_SS5_lands_at_SS6[OF x4 e5])
  hence ?idx c'' = SS7
    by (rule ae_step_from_SS6_lands_at_SS7[OF _ e6])
  thus ?thesis by simp
next
  case m7
  from chain m7 obtain c1 c2 c3 c4 c5 c6 where
      e1: (c', c1) ∈ ?R and e2: (c1, c2) ∈ ?R
      and e3: (c2, c3) ∈ ?R and e4: (c3, c4) ∈ ?R
      and e5: (c4, c5) ∈ ?R and e6: (c5, c6) ∈ ?R
      and e7: (c6, c'') ∈ ?R
```

```
    by (auto dest!: relpow_Suc_D2 simp: numeral_eq_Suc)
  have x1: ?idx c1 = SS2
    by (rule ae_step_from_SS1_lands_at_SS2[OF idx_SS1 e1])
  have x2: ?idx c2 = SS3
    by (rule ae_step_from_SS2_lands_at_SS3[OF x1 e2])
  have x3: ?idx c3 = SS4
    by (rule ae_step_from_SS3_lands_at_SS4[OF x2 e3])
  have x4: ?idx c4 = SS5
    by (rule ae_step_from_SS4_lands_at_SS5[OF x3 e4])
  have x5: ?idx c5 = SS6
    by (rule ae_step_from_SS5_lands_at_SS6[OF x4 e5])
  have x6: ?idx c6 = SS7
    by (rule ae_step_from_SS6_lands_at_SS7[OF x5 e6])
  hence ?idx c'' = SS8
    by (rule ae_step_from_SS7_lands_at_SS8[OF _ e7])
  thus ?thesis by simp
 qed
qed
```

2.29.9 Reverse-arm dispatcher and substrate consequences

Substrate consequence: from reachability of *cM* from *init_config_mttm M w*, every *M*-tape cell at any positive index along that trace is not the left-end marker *le_tm M*. Packaged as the *no_le_per_tape* conjunct expected by *ae_backward_stage* (forward chunked-phase derivation pattern at lines 8688+ of *AlphabetEnlargement.thy*). Factored out so the main chunked-reverse induction body stays under elaboration budget.

```
lemma ae_no_le_per_tape_from_reach:
 fixes M :: ('q, 'a) mttm
   and w :: 'a list
   and cM :: ('a, 'q) mt_config
   and c' :: ('c :: enum ⇒ 'a,
              'q × ('a, 'c) ae_stage) mt_config
 assumes vM:       valid_mttm M
   and lu:       le_unique M
   and w_sub:    set w ⊆ Sigma_tm M
   and le_neq_bl: le_tm M ≠ bl_tm M
   and reach_M:  (init_config_mttm M w, cM)
                 ∈ (mttm_step (delta_tm M))*
 shows ∀k. (mt_pos c' k ≥ 2
            ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
                   ⟶ mt_tape cM k
                       ((mt_pos c' k − 2)
                          * card (UNIV :: 'c set)
                        + 1 + i)
                     ≠ le_tm M))
         ∧ (mt_pos c' k = 1
            ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
```

```
                    ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
          ∧ (mt_pos c' k = 0
              ⟶ (∀ i. i < card (UNIV :: 'c set)
                    ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
proof (intro allI conjI impI allI impI)
  fix k :: nat and i :: nat
  assume 2 ≤ mt_pos c' k
    and i < 3 * card (UNIV :: 'c set)
  have idx_nz:
     (mt_pos c' k − 2) * card (UNIV :: 'c set) + 1 + i ≠ 0
    by simp
  show mt_tape cM k
       ((mt_pos c' k − 2) * card (UNIV :: 'c set)
         + 1 + i) ≠ le_tm M
    using valid_reach_LE_only_pos0_mttm
          [OF vM lu w_sub reach_M idx_nz le_neq_bl[symmetric]] .
next
  fix k :: nat and i :: nat
  assume mt_pos c' k = 1
    and i < 2 * card (UNIV :: 'c set)
  have idx_nz: Suc i ≠ 0 by simp
  show mt_tape cM k (Suc i) ≠ le_tm M
    using valid_reach_LE_only_pos0_mttm
          [OF vM lu w_sub reach_M idx_nz le_neq_bl[symmetric]] .
next
  fix k :: nat and i :: nat
  assume mt_pos c' k = 0
    and i < card (UNIV :: 'c set)
  have idx_nz: Suc i ≠ 0 by simp
  show mt_tape cM k (Suc i) ≠ le_tm M
    using valid_reach_LE_only_pos0_mttm
          [OF vM lu w_sub reach_M idx_nz le_neq_bl[symmetric]] .
qed
```

Halt-branch contradiction: when *mt_state c′* is at a halt-coerced state (*qM′*, *init_stage le*) with *qM′* ∈ {*t_tm M*, *r_tm M*}, the alphabet-enlarged accept/reject bridges identify the configuration's state with *t_tm M′* / *r_tm M′*, and *mttm_step_src_neq_t* / *mttm_step_src_neq_r* rule out any outgoing step. Factored out so the chunked-reverse induction body's halt-branch dispatch is a single *have False* ....

```
lemma ae_halt_c'_no_step:
  fixes M :: ('q, 'a) mttm
    and qM' :: 'q
    and c' c'' :: (('c :: enum) ⇒ 'a,
                   'q × ('a, 'c) ae_stage) mt_config
  assumes vM:        valid_mttm M
      and qM_in_tr:  qM' ∈ {t_tm M, r_tm M}
      and state_c':  mt_state c' = (qM', init_stage (le_tm M))
      and step:      (c', c'')
```

```
                ∈ mttm_step (alphabet_enlarge_delta M)
  shows False
proof −
  let ?M' = alphabet_enlarge M
            :: ('q × ('a, 'c) ae_stage,
                'c ⇒ 'a) mttm
  have valM': valid_mttm ?M'
    by (rule alphabet_enlarge_wf[OF vM])
  have step_M': (c', c'') ∈ mttm_step (delta_tm ?M')
    using step by (simp add: delta_tm_alphabet_enlarge)
  from qM_in_tr have qM_cases: qM' = t_tm M ∨ qM' = r_tm M
    by blast
  thus False
  proof
    assume qM' = t_tm M
    hence mt_state c' = t_tm ?M'
      using state_c' by (simp add: t_tm_alphabet_enlarge)
    thus False using mttm_step_src_neq_t[OF valM' step_M']
      by simp
  next
    assume qM' = r_tm M
    hence mt_state c' = r_tm ?M'
      using state_c' by (simp add: r_tm_alphabet_enlarge)
    thus False using mttm_step_src_neq_r[OF valM' step_M']
      by simp
  qed
qed
```

2.29.10 Chunked-reverse engine and language equivalence

Chunked-induction engine for the reverse arm. Given a finite M'-chain (c', *c_acc*) of length m ending in the canonical halt configuration (*t_tm M*, *init_stage le*), plus an *ae_simulates* invariant at c' with the standard side-band invariants *buf_gamma* and *le_anchor*, deliver an accepting M-path from *cM*. Mirror of *ae_simulation_phase_chunked* (forward) but inducting on the M'-side chain length rather than the M-side: at each level we peel 8 M'-steps via *ae_backward_stage* and recurse.

Three induction cases: $\cdot m = 0$ base: $c' =$ *c_acc*; the simulation's accept correspondence forces *mt_state cM = t_tm M*. $\cdot 0 < m < 8$: contradiction via *ae_chain_from_SS1_short_not_VFwd* (or trivial substep mismatch when $c' =$ *c_acc*'s sister case applies). $\cdot m \geq 8$ peel: split $m = 8 + (m - 8)$ via *relpow_add*, apply *ae_backward_stage* to the first chunk, IH on $m - 8$.

The conclusion is in *rtrancl* form (the *n_M* step counter is forgotten via *relpow_imp_rtrancl*) to ease the downstream *Lang_mttm*-repacking in *alphabet_enlarge_language*.

lemma *ae_simulation_phase_chunked_reverse*:

```
  fixes M :: ('q, 'a) mttm
    and w :: 'a list
    and cM :: ('a, 'q) mt_config
    and c' c_acc :: ('c :: enum ⇒ 'a,
              'q × ('a, 'c) ae_stage) mt_config
    and m :: nat
  assumes vM:          valid_mttm M
      and lu:        le_unique M
      and w_sub:      set w ⊆ Sigma_tm M
      and le_neq_bl:  le_tm M ≠ bl_tm M
      and reach_M:    (init_config_mttm M w, cM)
                      ∈ (mttm_step (delta_tm M))*
      and trace:      (c', c_acc)
                      ∈ mttm_step (alphabet_enlarge_delta M) ^^ m
      and accept:     mt_state c_acc
                      = (t_tm M, init_stage (le_tm M))
      and sim:        ae_simulates M cM c'
      and buf_gamma:  ae_buffer_in_gamma_block M c'
      and le_anchor:  ∀ kk<k_tm M. mt_tape c' kk 0 = LE_block (le_tm M)
  shows ∃ cM_final. (cM, cM_final) ∈ (mttm_step (delta_tm M))*
                  ∧ mt_state cM_final = t_tm M
  using reach_M trace sim buf_gamma le_anchor
proof (induction m arbitrary: cM c' rule: less_induct)
  case (less m)
  obtain qM' ofs buf dest idx where
     state_c': mt_state c' = (qM', ofs, buf, dest, idx)
    and qM_eq:   mt_state cM = qM'
    and state_disj: (idx = SS1 ∧ qM' ∉ {t_tm M, r_tm M})
                  ∨ (qM' ∈ {t_tm M, r_tm M}
                     ∧ (ofs, buf, dest, idx)
                        = init_stage (le_tm M))
    and ∀ k<k_tm M. ae_tape_correspondence (le_tm M)
            (mt_tape cM k) (mt_tape c' k)
    and ae_tape_in_gamma_block M c'
    by (rule ae_simulates_state_disjunction[OF less.prems(3)])
  consider (halt) qM' ∈ {t_tm M, r_tm M}
                ∧ (ofs, buf, dest, idx) = init_stage (le_tm M)
      | (ss1)  idx = SS1 ∧ qM' ∉ {t_tm M, r_tm M}
    using state_disj by blast
  thus ?case
  proof cases
    case halt
    have state_full: mt_state c' = (qM', init_stage (le_tm M))
      using state_c' halt by simp
    show ?thesis
    proof (cases m)
      case 0
      from less.prems(2) 0 have c'_eq: c' = c_acc by simp
      have mt_state c' = (t_tm M, init_stage (le_tm M))
```

```
    using accept c'_eq by simp
  hence qM_t: qM' = t_tm M using state_full by simp
  have cM_state: mt_state cM = t_tm M using qM_eq qM_t by simp
  have refl: (cM, cM) ∈ (mttm_step (delta_tm M))* by simp
  show ?thesis using refl cM_state by blast
next
  case (Suc m')
  have trace_Suc:
      (c', c_acc) ∈ mttm_step (alphabet_enlarge_delta M)
                      ^^ Suc m'
    using less.prems(2) Suc by simp
  obtain c'_1 where
      first_step:
        (c', c'_1) ∈ mttm_step (alphabet_enlarge_delta M)
    and rest_trace:
        (c'_1, c_acc) ∈ mttm_step (alphabet_enlarge_delta M)
                          ^^ m'
    using relpow_Suc_D2[OF trace_Suc] by blast
  have qM_in_tr: qM' ∈ {t_tm M, r_tm M} using halt by simp
  have False
    by (rule ae_halt_c'_no_step
            [OF vM qM_in_tr state_full first_step])
  thus ?thesis by simp
qed
next
  case ss1
  from ss1 have idx_SS1: idx = SS1 by simp
  have idx_snd4: snd (snd (snd (snd (mt_state c')))) = SS1
    using state_c' idx_SS1 by simp
  have qM_neq_t: mt_state cM ≠ t_tm M using qM_eq ss1 by simp
  have qM_neq_r: mt_state cM ≠ r_tm M using qM_eq ss1 by simp
  consider (m_zero) m = 0
         | (m_short) 0 < m ∧ m < 8
         | (m_long) 8 ≤ m
    by linarith
  thus ?thesis
  proof cases
    case m_zero
    from less.prems(2) m_zero have c'_eq: c' = c_acc by simp
    have mt_state c' = (t_tm M, init_stage (le_tm M))
      using accept c'_eq by simp
    hence c'_VFwd: snd (snd (snd (snd (mt_state c')))) = VFwd
      unfolding init_stage_def by simp
    from c'_VFwd idx_snd4 have False by simp
    thus ?thesis by simp
  next
    case m_short
    have c_acc_VFwd:
      snd (snd (snd (snd (mt_state c_acc)))) = VFwd
```

```
    using accept by (simp add: init_stage_def)
  have not_VFwd:
     snd (snd (snd (snd (mt_state c_acc)))) ≠ VFwd
    by (rule ae_chain_from_SS1_short_not_VFwd
          [OF idx_snd4 m_short less.prems(2)])
  from c_acc_VFwd not_VFwd have False by simp
  thus ?thesis by simp
next
  case m_long
  define m_rem where m_rem ≡ m − 8
  have m_split: m = 8 + m_rem
    using m_long m_rem_def by simp
  have rem_lt: m_rem < m
    using m_long m_rem_def by simp
  from less.prems(2) m_split have trace_compose:
     (c', c_acc)
       ∈ (mttm_step (alphabet_enlarge_delta M) ^^ 8)
          O (mttm_step (alphabet_enlarge_delta M) ^^ m_rem)
    by (simp add: relpow_add)
  obtain c_8 where
     run_8:    (c', c_8)
                ∈ mttm_step (alphabet_enlarge_delta M) ^^ 8
    and rest_8: (c_8, c_acc)
                ∈ mttm_step (alphabet_enlarge_delta M) ^^ m_rem
    using trace_compose by auto
  have val_cM: valid_config_mttm M cM
    by (rule valid_reach_mttm[OF vM w_sub less.prems(1)])
  have no_le_per_tape:
     ∀k. (mt_pos c' k ≥ 2
           ⟶ (∀i. i < 3 * card (UNIV :: 'c set)
                 ⟶ mt_tape cM k
                     ((mt_pos c' k − 2)
                        * card (UNIV :: 'c set)
                       + 1 + i)
                    ≠ le_tm M))
         ∧ (mt_pos c' k = 1
             ⟶ (∀i. i < 2 * card (UNIV :: 'c set)
                   ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
         ∧ (mt_pos c' k = 0
             ⟶ (∀i. i < card (UNIV :: 'c set)
                   ⟶ mt_tape cM k (Suc i) ≠ le_tm M))
    by (rule ae_no_le_per_tape_from_reach
          [OF vM lu w_sub le_neq_bl less.prems(1)])
  obtain cM_new n_M where
     cM_run:    (cM, cM_new) ∈ mttm_step (delta_tm M) ^^ n_M
    and n_M_le:   n_M ≤ card (UNIV :: 'c set)
    and end_disj: n_M = card (UNIV :: 'c set)
                  ∨ mt_state cM_new ∈ {t_tm M, r_tm M}
    and sim_new:  ae_simulates M cM_new c_8
```

```
      and buf_new:  ae_buffer_in_gamma_block M c_8
      and le_new:   ∀ kk<k_tm M. mt_tape c_8 kk 0 = LE_block (le_tm M)
      by (rule ae_backward_stage[OF vM lu less.prems(3) val_cM
                                    qM_neq_t qM_neq_r
                                    less.prems(4) less.prems(5)
                                    le_neq_bl no_le_per_tape run_8])
    have cM_to_new_star: (cM, cM_new) ∈ (mttm_step (delta_tm M))*
      using cM_run by (rule relpow_imp_rtrancl)
    have reach_cM_new: (init_config_mttm M w, cM_new)
                         ∈ (mttm_step (delta_tm M))*
      using less.prems(1) cM_to_new_star by (rule rtrancl_trans)
    obtain cM_final where
       rest_run:  (cM_new, cM_final)
                    ∈ (mttm_step (delta_tm M))*
      and acc_final: mt_state cM_final = t_tm M
      using less.IH[OF rem_lt reach_cM_new rest_8 sim_new
                       buf_new le_new]
      by blast
    have cM_to_final:
       (cM, cM_final) ∈ (mttm_step (delta_tm M))*
      using cM_to_new_star rest_run by (rule rtrancl_trans)
    show ?thesis using cM_to_final acc_final by blast
  qed
 qed
qed
```

Language-equivalence statement (biconditional) for every well-formed M — *with no determinism hypothesis*. Pairs with the forward inclusion *alphabet_enlarge_language_forward* in *AlphabetEnlargement.thy*: this theorem combines that forward leg with a determinism-free reverse leg.

Both legs assume only *well_formed_mttm M*. The forward leg is unconditional in M's determinism by construction. The reverse leg originally required *det_mttm M*: the natural route identifies the post-validation point by chain uniqueness of *alphabet_enlarge_delta* (*mttm_step_alphabet_enlarge_relpow_functional*), which a nondeterministic M does not supply — its compute substep is multi-valued, so that full chain is not unique. The refactoring removed the dependency. The validation prefix is pinned instead by the *validation-phase* uniqueness *mttm_step_alphabet_enlarge_val_relpow_functional* (the AE bookkeeping is deterministic whatever M does), and the single determinism-supplied step in *ae_backward_stage* — the SS4→SS5 identification — is replaced by the unconditional window-inversion kernel (*bp_linear* injectivity plus buffer-window inversion; the *det*-free SS5→SS8 kernel above), with the remainder discharged by *ae_simulation_phase_chunked_reverse*. The biconditional therefore holds for nondeterministic M as well, which is what the nondeterministic corollary *alphabet_enlarge_nae* packages with the time bound. The Hopcroft–Ullman linear-speedup corollaries [6, Theorems 12.3, 12.4]

reinstate *det_mttm M* only for the same-alphabet wrap, not for this biconditional.

```
theorem alphabet_enlarge_language:
  fixes M :: ('q, 'a) mttm
  assumes wf:        well_formed_mttm M
  shows ∀ w. set w ⊆ Sigma_tm M
            ⟶ (encode_input (bl_tm M) w ∈ Lang_mttm
                 (alphabet_enlarge M
                   :: ('q × ('a, ('c :: enum)) ae_stage,
                       'c ⇒ 'a) mttm))
                = (w ∈ Lang_mttm M)
proof −
  from wf have vM:        valid_mttm M
          and lu:         le_unique M
          and s_neq_t:    s_tm M ≠ t_tm M
          and s_neq_r:    s_tm M ≠ r_tm M
          and le_neq_bl: le_tm M ≠ bl_tm M
    by auto

  let ?M' = alphabet_enlarge M
             :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm

  have fwd: ∀ w. set w ⊆ Sigma_tm M
                ⟶ w ∈ Lang_mttm M
                ⟶ encode_input (bl_tm M) w ∈ Lang_mttm ?M'
    by (rule alphabet_enlarge_language_forward[OF wf])

  show ?thesis
  proof (intro allI impI)
    fix w :: 'a list
    assume w_sub: set w ⊆ Sigma_tm M
    show (encode_input (bl_tm M) w ∈ Lang_mttm ?M')
          = (w ∈ Lang_mttm M)
    proof
      assume w_in_M: w ∈ Lang_mttm M
      from fwd w_sub w_in_M
      show encode_input (bl_tm M) w ∈ Lang_mttm ?M'
        by blast
    next
      — Reverse direction: encode_input w ∈ Lang_mttm M' ⟹ w ∈ Lang_mttm
M.
Strategy: unpack the M'-accepting trace; peel validation via the strengthened ae_init_config_simulates_post_validation; identify the post-validation point on the trace via validation-phase chain uniqueness (mttm_step_alphabet_enlarge_val_relpow_functional, det-free); invoke ae_simulation_phase_chunked_reverse on the remainder; repack the resulting M-trace as Lang_mttm M.
      assume enc_in_M':
          encode_input (bl_tm M) w ∈ Lang_mttm ?M'
```

— Step 1: Unpack the M'-acceptance.
from *enc_in_M'* **obtain** *wM' nM'* **where**
acc_path_M':
(*init_config_mttm ?M'* (*encode_input* (*bl_tm M*) *w*),
Config$_M$ (*t_tm ?M'*) *wM' nM'*)
∈ (*mttm_step* (*delta_tm ?M'*))*
unfolding *Lang_mttm_def* **by** *blast*
define *c_acc* :: (′*c* ⇒ ′*a*, ′*q* × (′*a*, ′*c*) *ae_stage*)
mt_config
where *c_acc* ≡ *Config*$_M$ (*t_tm ?M'*) *wM' nM'*
— Step 2: Bridge *?M'* components to AE-side and convert rtrancl to relpow.
have *c_acc_state*:
mt_state c_acc = (*t_tm M*, *init_stage* (*le_tm M*))
unfolding *c_acc_def* **by** (*simp add*: *t_tm_alphabet_enlarge*)
have *acc_path_AE*:
(*ae_init_config M* (*encode_input* (*bl_tm M*) *w*), *c_acc*)
∈ (*mttm_step* (*alphabet_enlarge_delta M*))*
using *acc_path_M'*
by (*simp add*: *init_config_alphabet_enlarge*
delta_tm_alphabet_enlarge c_acc_def)
obtain *N* **where** *acc_pow*:
(*ae_init_config M* (*encode_input* (*bl_tm M*) *w*), *c_acc*)
∈ *mttm_step* (*alphabet_enlarge_delta M*) ^^ *N*
using *acc_path_AE rtrancl_imp_relpow* **by** *metis*
— Step 3: Peel validation. Explicit type annotation on *c'_post* to share ′*c* with *c_acc* (the validation-peel lemma is polymorphic over ′*c*; without the annotation Isabelle freshens).
obtain *n_val* **and** *c'_post* :: (′*c* ⇒ ′*a*,
′*q* × (′*a*, ′*c*) *ae_stage*) *mt_config* **where**
val_run: (*ae_init_config M* (*encode_input* (*bl_tm M*) *w*),
c'_post)
∈ *mttm_step* (*alphabet_enlarge_delta M*)
^^ *n_val*
and *val_sim*: *ae_simulates M* (*init_config_mttm M w*) *c'_post*
and *val_buf*: *ae_buffer_in_gamma_block M c'_post*
and *val_le*: ∀ *kk*<*k_tm M*. *mt_tape c'_post kk 0* = *LE_block* (*le_tm M*)
and *val_reach*:
∀ *i*<*n_val*. ∀ *d* :: (′*c* ⇒ ′*a*,
′*q* × (′*a*, ′*c*) *ae_stage*) *mt_config*.
(*ae_init_config M* (*encode_input* (*bl_tm M*) *w*), *d*)
∈ *mttm_step* (*alphabet_enlarge_delta M*) ^^ *i*
⟶ *snd* (*snd* (*snd* (*snd* (*mt_state d*))))
∈ {*VFwd*, *VFwdPad*, *VRet*}
by (*rule ae_init_config_simulates_post_validation*
[*OF vM w_sub le_neq_bl*])
— Step 4: Identify chain remainder. By *mttm_step_alphabet_enlarge_val_relpow_functional* the validation endpoint sits on the unique chain from *ae_init_config* to *c_acc*. Two cases on *n_val* vs *N*: · *n_val* ≤ *N*: extract remainder (*c'_post*, *c_acc*) ∈ *step* ^^ (*N* − *n_val*). · *n_val*

$> N$: contradiction a step from c_acc is impossible since $mt_state\ c_acc = t_tm\ M'$.

have $n_val_le_N$: $n_val \leq N$
proof (*rule ccontr*)
assume $\neg\ n_val \leq N$
hence N_lt: $N < n_val$ **by** *simp*
have sum_eq: $N + (n_val - N) = n_val$
using N_lt **by** *simp*
have $val_run_combined$:
$(ae_init_config\ M\ (encode_input\ (bl_tm\ M)\ w),\ c'_post)$
$\in mttm_step\ (alphabet_enlarge_delta\ M)$
$\frown\frown (N + (n_val - N))$
using $val_run\ sum_eq$ **by** *simp*
hence $val_compose$:
$(ae_init_config\ M\ (encode_input\ (bl_tm\ M)\ w),\ c'_post)$
$\in (mttm_step\ (alphabet_enlarge_delta\ M) \frown\frown N)$
$O\ (mttm_step\ (alphabet_enlarge_delta\ M)$
$\frown\frown (n_val - N))$
by (*simp add*: *relpow_add*)
obtain c_mid **where**
$step_to_mid$:
$(ae_init_config\ M\ (encode_input\ (bl_tm\ M)\ w),\ c_mid)$
$\in mttm_step\ (alphabet_enlarge_delta\ M) \frown\frown N$
and $step_from_mid$:
$(c_mid,\ c'_post)$
$\in mttm_step\ (alphabet_enlarge_delta\ M)$
$\frown\frown (n_val - N)$
using $val_compose$ **by** *auto*
have inv_N: $\forall\, i{<}N.\ \forall\, d :: ('c \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae_stage)\ mt_config.$
$(ae_init_config\ M\ (encode_input\ (bl_tm\ M)\ w),\ d)$
$\in mttm_step\ (alphabet_enlarge_delta\ M) \frown\frown i$
$\longrightarrow snd\ (snd\ (snd\ (snd\ (mt_state\ d))))$
$\in \{VFwd,\ VFwdPad,\ VRet\}$
using $val_reach\ N_lt$ **by** (*meson less_trans*)
have c_mid_eq: $c_mid = c_acc$
by (*rule mttm_step_alphabet_enlarge_val_relpow_functional*
[*OF le_neq_bl step_to_mid acc_pow inv_N*])
have $step_from_acc$:
$(c_acc,\ c'_post)$
$\in mttm_step\ (alphabet_enlarge_delta\ M)$
$\frown\frown (n_val - N)$
using $step_from_mid\ c_mid_eq$ **by** *simp*
have $extra_pos$: $0 < n_val - N$ **using** N_lt **by** *simp*
obtain m_extra_pred **where** m_extra_Suc:
$n_val - N = Suc\ m_extra_pred$
using $extra_pos$ **by** (*cases* $n_val - N$) *auto*
have $step_from_acc_Suc$:
$(c_acc,\ c'_post)$

```
        ∈ mttm_step (alphabet_enlarge_delta M)
            ^^ Suc m_extra_pred
    using step_from_acc m_extra_Suc by simp
  obtain c_acc_1 where
      first_step:
        (c_acc, c_acc_1)
          ∈ mttm_step (alphabet_enlarge_delta M)
    and rest_step:
        (c_acc_1, c'_post)
          ∈ mttm_step (alphabet_enlarge_delta M)
              ^^ m_extra_pred
    using relpow_Suc_D2[OF step_from_acc_Suc] by blast
  have valM': valid_mttm ?M'
    by (rule alphabet_enlarge_wf[OF vM])
  have first_step_M':
      (c_acc, c_acc_1) ∈ mttm_step (delta_tm ?M')
    using first_step by (simp add: delta_tm_alphabet_enlarge)
  have c_acc_state_M': mt_state c_acc = t_tm ?M'
    using c_acc_state by (simp add: t_tm_alphabet_enlarge)
  from mttm_step_src_neq_t[OF valM' first_step_M']
      c_acc_state_M'
    show False by simp
qed
have remainder:
    (c'_post, c_acc) ∈ mttm_step (alphabet_enlarge_delta M)
                          ^^ (N - n_val)
proof -
  have sum_eq: n_val + (N - n_val) = N
    using n_val_le_N by simp
  have (ae_init_config M (encode_input (bl_tm M) w), c_acc)
        ∈ mttm_step (alphabet_enlarge_delta M)
            ^^ (n_val + (N - n_val))
    using acc_pow sum_eq by simp
  hence acc_compose:
      (ae_init_config M (encode_input (bl_tm M) w), c_acc)
        ∈ (mttm_step (alphabet_enlarge_delta M) ^^ n_val)
            O (mttm_step (alphabet_enlarge_delta M)
                ^^ (N - n_val))
    by (simp add: relpow_add)
  obtain c_mid where
      run_to_mid:
        (ae_init_config M (encode_input (bl_tm M) w), c_mid)
          ∈ mttm_step (alphabet_enlarge_delta M) ^^ n_val
    and run_from_mid:
        (c_mid, c_acc)
          ∈ mttm_step (alphabet_enlarge_delta M)
              ^^ (N - n_val)
    using acc_compose by auto
  have c_mid_eq: c_mid = c'_post
```

```
      by (rule mttm_step_alphabet_enlarge_val_relpow_functional
            [OF le_neq_bl run_to_mid val_run val_reach])
    show ?thesis using run_from_mid c_mid_eq by simp
  qed
  — Step 5: Invoke chunked-reverse to extract an M-side accepting path.
  have reach_refl:
      (init_config_mttm M w, init_config_mttm M w)
        ∈ (mttm_step (delta_tm M))*
    by simp
  obtain cM_final where
      M_path:   (init_config_mttm M w, cM_final)
                  ∈ (mttm_step (delta_tm M))*
    and M_acc:    mt_state cM_final = t_tm M
    using ae_simulation_phase_chunked_reverse
          [OF vM lu w_sub le_neq_bl reach_refl remainder
              c_acc_state val_sim val_buf val_le]
    by blast
  — Step 6: Repack as w ∈ Lang_mttm M.
  obtain wM_acc nM_acc where
      cM_final_eq:
        cM_final = Config_M (t_tm M) wM_acc nM_acc
    using M_acc by (cases cM_final) simp
  show w ∈ Lang_mttm M
    unfolding Lang_mttm_def
    using w_sub M_path cM_final_eq by blast
 qed
qed
qed
```

2.30 Nondeterministic alphabet enlargement

The determinism-free building block for the nondeterministic linear speedup — the *Form-2* statement (raw, no *encoding_wrap*, tape count unchanged, over the enlarged alphabet). For an arbitrary, *possibly nondeterministic*, well-formed M, the enlargement *alphabet_enlarge M* preserves the language (under the *encode_input* block map) and the running time, with *no det_mttm M* hypothesis and *no det_mttm*-output conclusion; the witness *alphabet_enlarge M* is nondeterministic exactly when M is.

Wrapping this in the inline plant-*le* encoder yields the same-alphabet nondeterministic speedup corollaries *linear_speedup_HU_12_3_nae* / *12_4_nae_B* — the Hopcroft–Ullman page-291 corollary [6, p. 291] — proved in *Wrap_Speedup* alongside their deterministic counterparts *linear_speedup_HU_12_3_dae* / *12_4_dae_B*, which assume *det_mttm M* and additionally conclude output determinism via *wrap_det*.

```
theorem alphabet_enlarge_nae:
  fixes M :: ('q, 'a) mttm
    and T :: nat ⇒ nat
```

```
  assumes wf: well_formed_mttm M
  obtains α f :: nat
  where ∀ w. set w ⊆ Sigma_tm M
            ⟶ (encode_input (bl_tm M) w ∈ Lang_mttm
                  (alphabet_enlarge M
                     :: ('q × ('a, ('c :: enum)) ae_stage,
                         'c ⇒ 'a) mttm))
              = (w ∈ Lang_mttm M)
    and ∀ w. set w ⊆ Sigma_tm M
            ⟶ accepts_in_time_mttm M w (T (length w))
            ⟶ accepts_in_time_mttm
                 (alphabet_enlarge M
                    :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
                 (encode_input (bl_tm M) w)
                 (α * ((length w + card (UNIV :: 'c set) - 1)
                        div card (UNIV :: 'c set))
                  + 8 * ((T (length w) + card (UNIV :: 'c set) - 1)
                         div card (UNIV :: 'c set))
                  + f)
proof -
  have lang:
    ∀ w. set w ⊆ Sigma_tm M
          ⟶ (encode_input (bl_tm M) w ∈ Lang_mttm
                (alphabet_enlarge M
                   :: ('q × ('a, ('c :: enum)) ae_stage,
                       'c ⇒ 'a) mttm))
            = (w ∈ Lang_mttm M)
    by (rule alphabet_enlarge_language[OF wf])
  obtain α f :: nat where time:
    ∀ w. set w ⊆ Sigma_tm M
          ⟶ accepts_in_time_mttm M w (T (length w))
          ⟶ accepts_in_time_mttm
               (alphabet_enlarge M
                  :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
               (encode_input (bl_tm M) w)
               (α * ((length w + card (UNIV :: 'c set) - 1)
                      div card (UNIV :: 'c set))
                + 8 * ((T (length w) + card (UNIV :: 'c set) - 1)
                       div card (UNIV :: 'c set))
                + f)
    by (rule alphabet_enlarge_time[OF wf])
  show ?thesis by (rule that[OF lang time])
qed


end
theory Wrap_Base
  imports Multitape_TM_Substrate.Multitape_Substrate
begin
```

3 Encoding-wrap combinator

The encoding-wrap combinator. Given a substrate machine M over an encoded alphabet $'b$ and a per-block packer *pack* :: $'a$ *list* $\Rightarrow$ $'b$ together with a block size c, the combinator produces a substrate machine over the union alphabet ($'a$, $'b$) *wrap_alphabet* = *Raw* $'a$ | *Enc* $'b$ at the textbook tape count *k_tm M* (no extra tape): the user's raw input starts on tape *0*, which the wrap reuses as a work tape rather than adding one (faithful k-tape section below).

The wrap operates in five reachable phases:

1. *W_Init* advance both *W_User* and *W_M 0* past their LE markers in a single super-step.

2. *W_Buf ws* accumulate up to c raw symbols from *W_User*; on filling the buffer, pack and write one block to *W_M 0*; on seeing an *Enc* _ symbol (blank past the input region), flush any partial buffer and proceed to reset.

3. *W_Reset* move *W_M 0*'s head left until LE, restoring the canonical "head at position 0 reading LE" position M expects at the start of its own computation.

4. *W_Disp* single transition handing over to M.

5. *W_Run q* simulate M directly: every M-δ entry lifts to a wrap-δ entry that reads M's tapes through the *Enc* embedding and leaves *W_User* untouched.

   Halt propagation rides directly on the *W_Run* constructor: the wrap's accept state is *W_Run* (*t_tm M*), inheriting M's acceptance. The wrap's reject state *W_Rej* is declared for substrate compliance (the $t \neq r$ requirement) but is unreachable from the start state.

Architecture note: the wrap is monolithic at the substrate level. The substrate has no generic machine-composition combinator, so the wrap's per-cell read, head-positioning, and per-cell write actions are written directly into the wrap's δ-table rather than as composed sub-machines.

3.1 Wrap alphabet

Disjoint union of user-facing and encoded alphabets. Constructors are injective and disjoint; the wrap's substrate-level blank and LE symbols are chosen from the *Enc*-image to keep lifted M-transitions transparent.

datatype ($'a$, $'b$) *wrap_alphabet* = *Raw* $'a$ | *Enc* $'b$

3.2 Wrap tape index

Value-level tape indices (*nat*). The wrap keeps the textbook tape count *k_tm M* — *no* extra tape. Tape *0* initially holds the user's raw input (matching the substrate convention); once the encoder has consumed it, that spent tape is reused as one of M's work tapes rather than a fresh tape being prepended (faithful *k*-tape section below).

3.3 Wrap state space

Five reachable phases plus a formal reject. The *W_Buf* payload is a partial encoder buffer of length strictly less than *c*; the *W_Run* payload is the current M-state lifted through *W_Run*.

```
datatype ('q, 'a, 'b) wrap_state =
    W_Init
  | W_Buf 'a list
  | W_Reset
  | W_Disp
  | W_Run 'q
  | W_Rej
```

The wrap's state set: *W_Init*, *W_Reset*, *W_Disp*, *W_Rej* as singletons; *W_Buf* variants ranging over partial buffers of length $<c$ with symbols drawn from the user-facing alphabet; and *W_Run* variants ranging over M's state set. Polynomial in $|\Sigma u| \hat{} c + |Q_tm\ M|$; finite when both factors are.

```
definition wrap_state_set ::
  ('q, 'b) mttm ⇒ nat ⇒ 'a set
   ⇒ ('q, 'a, 'b) wrap_state set
where
  wrap_state_set M c Σu =
    {W_Init, W_Reset, W_Disp, W_Rej}
    ∪ {W_Buf ws | ws. length ws < c ∧ set ws ⊆ Σu}
    ∪ W_Run ‘ Q_tm M
```

3.4 Generic single-tape rewind

The one piece of the wrap δ-relation kept in the base: a generic family for rewinding a single tape's head back to its *le* marker. The full wrap transition relations the phase-local encoder, reset, dispatch, and lifted-M families are assembled downstream, on the *_gen* builders shared with the plant wrap *wrap_delta*; only this rewind family, reused verbatim there, lives at the base.

wrap_rewind_loop_delta_gen qf j K M walks tape *j*'s head one cell left per step while it reads a non-*le* symbol, looping in state *qf* and leaving

every other tape idle. Parameterised over the tape index j and the blank-tail boundary K, so it rewinds *any* tape at either wrap tape count (*Suc* (*k_tm M*) or the faithful *k_tm M*); the loop state *qf* lets it splice into any phase. The plant wrap instantiates it at $j = Suc\ 0$ to rewind the storage tape (*wrap_delta*'s *rloop* family).

```
definition wrap_rewind_loop_delta_gen ::
  ('q, 'a, 'b) wrap_state ⇒ nat ⇒ nat ⇒ ('q, 'b) mttm ⇒ 'a set
  ⇒ (('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × ('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × (nat ⇒ dir)) set
where
  wrap_rewind_loop_delta_gen qf j K M Σu =
    { (qf, sym, qf, sym, λt. if t = j then dir.L else dir.N)
      | sym.
          sym j ≠ Enc (le_tm M)
          ∧ (∀ i≥K. sym i = Enc (bl_tm M))
          ∧ sym ∈ UNIV → Raw ' Σu ∪ Enc ' Γ_tm M }
```

3.5 User-facing language and time-bounded acceptance

The user-facing language of a wrap-shaped machine is the *Lang_mttm* preimage under the *Raw* embedding: a user input w is in the user-facing language iff its *Raw*-embedded form is in the wrap's substrate-level language. Same routing for time-bounded acceptance. These predicates are stated over arbitrary *mttm* values whose alphabet is a *wrap_alphabet*; their primary use is on the encoding-wrap machines built downstream (the plant wrap *encoding_wrap*), but the definitions are alphabet-shape-agnostic.

```
definition Lang_user_wrap ::
  ('q, ('a, 'b) wrap_alphabet) mttm ⇒ 'a list set
where
  Lang_user_wrap M'' = {w. map Raw w ∈ Lang_mttm M''}

definition accepts_in_time_user_wrap ::
  ('q, ('a, 'b) wrap_alphabet) mttm
  ⇒ 'a list ⇒ nat ⇒ bool
where
  accepts_in_time_user_wrap M'' w t =
    accepts_in_time_mttm M'' (map Raw w) t
```

3.6 Encoder image and state-set finiteness

Two shared base helpers of the encoding wrap: the encoder image function *wrap_enc* (the packed form of a user input, which *Lang_user_wrap* and *accepts_in_time_user_wrap* refer to) and the finiteness of the wrap state

set. Both feed the downstream wrap well-formedness, determinism, and language proofs; neither depends on the wrap's transition relation.

The encoder image of a user input: chunk w into blocks of size c, apply *pack* to each block. This is the full encoder *enc* :: *'a list* ⇒ *'b list*, derived from *pack* and c because the wrap constructions take the per-chunk packer plus chunk size rather than an opaque encoder argument. Recursion is over *drop c w*, not a direct subterm, hence **function+termination** rather than **fun**.

```
function wrap_enc ::
  ('a list ⇒ 'b) ⇒ nat ⇒ 'a list ⇒ 'b list
where
  wrap_enc pack 0 w = []
| wrap_enc pack (Suc k) [] = []
| wrap_enc pack (Suc k) (x # xs) =
    pack (take (Suc k) (x # xs))
    # wrap_enc pack (Suc k) (drop (Suc k) (x # xs))
  by pat_completeness auto
termination
  by (relation measure (λ(_, _, w). length w)) auto
```

Helper: the wrap state set is finite when M is valid and the user alphabet is finite. *W_Buf*'s payload range is finite because length is bounded by c and entries are drawn from the finite Σu; *W_Run*'s payload range is finite because *Q_tm M* is finite by *valid_mttm ?M* ⟹ *finite* (*Q_tm ?M*).

```
lemma finite_wrap_state_set:
  assumes valid_mttm M
    and finite Σu
  shows finite (wrap_state_set M c Σu)
proof −
  have fin_Q: finite (Q_tm M)
    by (rule valid_mttm_finite_Q[OF assms(1)])
  have fin_buf: finite {W_Buf ws | ws. length ws < c ∧ set ws ⊆ Σu}
  proof −
    have {ws. length ws < c ∧ set ws ⊆ Σu} ⊆ {ws. set ws ⊆ Σu ∧ length ws ≤ c}
      by auto
    moreover have finite {ws. set ws ⊆ Σu ∧ length ws ≤ c}
      using finite_lists_length_le[OF ‹finite Σu›] by simp
    ultimately have finite {ws. length ws < c ∧ set ws ⊆ Σu}
      by (rule finite_subset)
    hence finite (W_Buf ‘ {ws. length ws < c ∧ set ws ⊆ Σu})
      by (rule finite_imageI)
    moreover have {W_Buf ws | ws. length ws < c ∧ set ws ⊆ Σu}
                  = W_Buf ‘ {ws. length ws < c ∧ set ws ⊆ Σu}
      by blast
    ultimately show ?thesis by metis
  qed
  have fin_run: finite (W_Run ‘ Q_tm M)
```

```
    using fin_Q by (rule finite_imageI)
  have finite ({W_Init, W_Reset, W_Disp, W_Rej}
                ∪ {W_Buf ws | ws. length ws < c ∧ set ws ⊆ Σu}
                ∪ W_Run ` Q_tm M)
    using fin_buf fin_run by simp
  thus ?thesis
    unfolding wrap_state_set_def by simp
qed
```

3.7 Encoder-image support lemmas

Structural facts about the encoder image *wrap_enc*, which chunks the user input into blocks of size c and applies *pack* to each. These lemmas pin down that image cell-by-cell: the per-block content of a full block (*wrap_enc_nth*) and of the final short block (*wrap_enc_nth_partial*), the length of the packed sequence (*length_wrap_enc*), and the tape-alphabet membership of every packed symbol — each lies in the *Enc*-image of Γ_*tm M* (*wrap_enc_in_Gamma*), is distinct from the *le* marker (*wrap_enc_ne_le*), and lies in *Sigma_tm M* whenever the packer contracts into it (*wrap_enc_in_Sigma*, with that packing side-condition isolated in *pack_take_in_set_wrap_enc*).

This is shared base machinery. The faithful k-tape encoder simulation — the *_gen* / *_B* transition chains that write *wrap_enc* onto the storage tape — reads these invariants back to certify the encoded input; and the AE-wrap glue *wrap_enc_eq_encode_input* builds on them to match *wrap_enc* against the alphabet-enlargement *encode_input*.

Support lemma about *wrap_enc*: when a full block at offset $i * c$ fits within w, the i-th encoded entry is the pack of that block. Proof is by induction on i, peeling one block off the front per step via *wrap_enc*'s recursive equation.

```
lemma wrap_enc_nth:
  assumes 0 < c
      and (Suc i) * c ≤ length w
  shows wrap_enc pack c w ! i = pack (take c (drop (i * c) w))
  using assms
proof (induction i arbitrary: w)
  case 0
  from ‹0 < c› obtain k where c_eq: c = Suc k
    by (cases c) auto
  from ‹Suc 0 * c ≤ length w› have c_le: c ≤ length w
    by simp
  with c_eq have w ≠ []
    by (cases w) auto
  then obtain x xs where w_eq: w = x # xs
    by (cases w) auto
  have wrap_enc pack c w
```

```
      = pack (take c (x # xs)) # wrap_enc pack c (drop c (x # xs))
    using c_eq w_eq by simp
  hence wrap_enc pack c w ! 0 = pack (take c (x # xs))
    by simp
  also have ... = pack (take c (drop (0 * c) w))
    using w_eq by simp
  finally show ?case .
next
  case (Suc i)
  from ‹0 < c› obtain k where c_eq: c = Suc k
    by (cases c) auto
  from ‹Suc (Suc i) * c ≤ length w› have c_le: c ≤ length w
    by simp
  with c_eq have w ≠ []
    by (cases w) auto
  then obtain x xs where w_eq: w = x # xs
    by (cases w) auto
  have unfold:
    wrap_enc pack c w
      = pack (take c (x # xs)) # wrap_enc pack c (drop c (x # xs))
    using c_eq w_eq by simp
  have nth_step:
    wrap_enc pack c w ! Suc i = wrap_enc pack c (drop c w) ! i
    using unfold w_eq by simp
  have len_drop: Suc i * c ≤ length (drop c w)
    using ‹Suc (Suc i) * c ≤ length w› by simp
  have ih:
    wrap_enc pack c (drop c w) ! i = pack (take c (drop (i * c) (drop c w)))
    using Suc.IH[OF ‹0 < c› len_drop] .
  have drop_drop: drop (i * c) (drop c w) = drop (Suc i * c) w
    by (simp add: add.commute)
  from nth_step ih drop_drop
  show ?case by simp
qed
```

Length of *wrap_enc*: *length w div c* full blocks plus one partial block if *length w mod c ≠ 0*. Proof is by strong induction on *length w*: case-split on whether $c \leq$ *length w*; if not, the encoding is empty ($w = []$) or a single partial block; if so, peel one full block off the front and recurse on *drop c w*.

```
lemma length_wrap_enc:
  assumes 0 < c
  shows length (wrap_enc pack c w) =
        length w div c + (if length w mod c = 0 then 0 else 1)
  using assms
proof (induction length w arbitrary: w rule: less_induct)
  case less
  from ‹0 < c› obtain k where c_eq: c = Suc k by (cases c) auto
  show ?case
  proof (cases c ≤ length w)
```

```
  case False
  hence w_short: length w < c by simp
  show ?thesis
  proof (cases w)
    case Nil
    with c_eq show ?thesis by simp
  next
    case (Cons x xs)
    have unfold:
      wrap_enc pack c w = pack (take c w) # wrap_enc pack c (drop c w)
      using c_eq Cons by simp
    have drop_eq: drop c w = []
      using w_short by simp
    have wrap_eq: wrap_enc pack c w = [pack (take c w)]
      using unfold drop_eq c_eq by simp
    from w_short ‹0 < c› have div_eq: length w div c = 0 by simp
    from w_short Cons ‹0 < c› have mod_pos: length w mod c ≠ 0
      by (auto simp: mod_less)
    from wrap_eq have length (wrap_enc pack c w) = 1 by simp
    with div_eq mod_pos show ?thesis by simp
  qed
next
  case True
  hence c_le: c ≤ length w by simp
  have w_nonempty: w ≠ [] using c_le ‹0 < c› by auto
  then obtain x xs where w_cons: w = x # xs by (cases w) auto
  have unfold:
    wrap_enc pack c w = pack (take c w) # wrap_enc pack c (drop c w)
    using c_eq w_cons by simp
  have tail_len: length (drop c w) = length w - c by simp
  have tail_less: length (drop c w) < length w
    using c_le ‹0 < c› tail_len by simp
  from less.hyps[OF tail_less ‹0 < c›]
  have ih: length (wrap_enc pack c (drop c w)) =
          length (drop c w) div c +
          (if length (drop c w) mod c = 0 then 0 else 1) .
  have div_step: length w div c = (length w - c) div c + 1
    using c_le ‹0 < c› by (simp add: le_div_geq)
  have mod_step: length w mod c = (length w - c) mod c
    using c_le ‹0 < c› by (simp add: le_mod_geq)
  from unfold have length (wrap_enc pack c w) =
                   1 + length (wrap_enc pack c (drop c w)) by simp
  also have ... = 1 + length (drop c w) div c +
                  (if length (drop c w) mod c = 0 then 0 else 1)
    using ih by simp
  also have ... = length w div c +
                  (if length w mod c = 0 then 0 else 1)
    using div_step mod_step tail_len by simp
  finally show ?thesis .
```

```
  qed
qed
```

The last entry of *wrap_enc* when the input has a non-trivial trailing partial block: it is the *pack* of that trailing block *drop* (*length w div c* ∗ *c*) *w* (of length *length w mod c*). Proof by the same strong-induction shape as *length_wrap_enc*, with the partial-block case yielding the singleton directly and the full-block case recursing on *drop c w*.

```
lemma wrap_enc_nth_partial:
  assumes 0 < c
      and length w mod c ≠ 0
  shows wrap_enc pack c w ! (length w div c) =
        pack (drop ((length w div c) ∗ c) w)
  using assms
proof (induction length w arbitrary: w rule: less_induct)
  case less
  from ‹0 < c› obtain k where c_eq: c = Suc k by (cases c) auto
  show ?case
  proof (cases c ≤ length w)
    case False
    hence w_short: length w < c by simp
    from ‹length w mod c ≠ 0› w_short ‹0 < c›
    have w_nonempty: w ≠ [] by auto
    then obtain x xs where w_cons: w = x # xs by (cases w) auto
    have unfold:
      wrap_enc pack c w = pack (take c w) # wrap_enc pack c (drop c w)
      using c_eq w_cons by simp
    have drop_eq: drop c w = [] using w_short by simp
    have take_all: take c w = w using w_short by simp
    have wrap_eq: wrap_enc pack c w = [pack w]
      using unfold drop_eq take_all c_eq by simp
    from w_short ‹0 < c› have div_eq: length w div c = 0 by simp
    have lhs: wrap_enc pack c w ! (length w div c) = pack w
      using wrap_eq div_eq by simp
    have rhs: drop ((length w div c) ∗ c) w = w
      using div_eq by simp
    from lhs rhs show ?thesis by simp
  next
    case True
    hence c_le: c ≤ length w by simp
    have w_nonempty: w ≠ [] using c_le ‹0 < c› by auto
    then obtain x xs where w_cons: w = x # xs by (cases w) auto
    have unfold:
      wrap_enc pack c w = pack (take c w) # wrap_enc pack c (drop c w)
      using c_eq w_cons by simp
    have tail_less: length (drop c w) < length w
      using c_le ‹0 < c› by simp
    have tail_mod: length (drop c w) mod c = length w mod c
      using c_le ‹0 < c› by (simp add: le_mod_geq)
```

```
    have tail_div: length (drop c w) div c = length w div c − 1
      using c_le ‹0 < c› by (simp add: le_div_geq)
    have div_pos: length w div c > 0
      using c_le ‹0 < c› by (simp add: le_div_geq)
    have mod_pos_tail: length (drop c w) mod c ≠ 0
      using ‹length w mod c ≠ 0› tail_mod by simp
    note ih = less.hyps[OF tail_less ‹0 < c› mod_pos_tail]
    have idx_step:
      wrap_enc pack c w ! (length w div c) =
       wrap_enc pack c (drop c w) ! (length w div c − 1)
      using unfold div_pos by (simp add: nth_Cons')
    also have ... = wrap_enc pack c (drop c w) ! (length (drop c w) div c)
      using tail_div by simp
    also have ... = pack (drop ((length (drop c w) div c) ∗ c) (drop c w))
      using ih .
    also have drop ((length (drop c w) div c) ∗ c) (drop c w)
            = drop ((length w div c) ∗ c) w
      using tail_div c_le div_pos
      by (simp add: drop_drop algebra_simps)
    finally show ?thesis .
  qed
qed
```

Derived safety lemmas for entries of *wrap_enc*: under the pack contracts (*pack* (*take c* (*drop* (*i* ∗ *c*) *w*)) ∈ Γ_*tm M* and the ≠ *le_tm M* companion, universally quantified over indices *i* with *i* ∗ *c* < *length w*), every entry of *wrap_enc pack c w* lies in Γ_*tm M* and differs from *le_tm M*. Proof case-splits on whether the entry is a full block (*j* < *length w div c*, served by *wrap_enc_nth*) or the trailing partial block (*j* = *length w div c* forced by length, served by *wrap_enc_nth_partial*). For the partial case *take c* (*drop* (*j* ∗ *c*) *w*) = *drop* (*j* ∗ *c*) *w* because the drop has length *length w mod c* < *c*, so the universal pack contract still fires. These lemmas are the bridge from the pack contracts to the per-cell symbol safety conditions used by *wrap_reset_loop_delta* and *wrap_reset_done_delta*.

Bridge for the reverse direction (consumed by *wrap_language_reverse*): every pack output of the form *pack* (*take c* (*drop* (*i* ∗ *c*) *w*)) for *i* ∗ *c* < *length w* appears as some entry of *wrap_enc pack c w*. Case-splits on whether the *i*-th block is full (*Suc i* ∗ *c* ≤ *length w*, served by *wrap_enc_nth*) or the trailing partial block (forced when *length w* < *Suc i* ∗ *c* and *i* ∗ *c* < *length w*, served by *wrap_enc_nth_partial*). Composing with *set* (*wrap_enc pack c w*) ⊆ *Sigma_tm M* (from *wrap_enc pack c w* ∈ *Lang_mttm M*) yields the pack contracts that *encoder_phase_terminates* / *reset_phase_terminates* need.

```
lemma pack_take_in_set_wrap_enc:
  assumes c_pos: 0 < c
      and i_c_lt: i ∗ c < length w
    shows pack (take c (drop (i ∗ c) w)) ∈ set (wrap_enc pack c w)
```

```
proof −
  let ?L = length (wrap_enc pack c w)
  let ?q = length w div c
  let ?r = length w mod c
  have len_eq: ?L = ?q + (if ?r = 0 then 0 else 1)
    by (rule length_wrap_enc[OF c_pos])
  have q_c_le: ?q * c ≤ length w
    by (rule div_times_less_eq_dividend)
  have div_mod_eq: ?q * c + ?r = length w
    by (rule div_mult_mod_eq)
  show ?thesis
  proof (cases (Suc i) * c ≤ length w)
    case True
    — Full block at index i.
    have div_le_mono_step: (Suc i) * c div c ≤ length w div c
      using True div_le_mono by blast
    have Suc_i_le_q: Suc i ≤ ?q
      using div_le_mono_step c_pos by simp
    have i_lt_q: i < ?q using Suc_i_le_q by simp
    have i_lt_L: i < ?L using i_lt_q len_eq by simp
    have nth_eq: wrap_enc pack c w ! i = pack (take c (drop (i * c) w))
      by (rule wrap_enc_nth[OF c_pos True])
    show ?thesis using nth_eq i_lt_L by (metis nth_mem)
  next
    case False
    — Partial trailing block: i forced to equal length w div c.
    hence Sci_gt: length w < (Suc i) * c by simp
    — From i * c < length w: i ≤ length w div c.
    have div_le_mono_step: i * c div c ≤ length w div c
      using i_c_lt by (intro div_le_mono) simp
    have i_le_q: i ≤ ?q using div_le_mono_step c_pos by simp
    — From Sci_gt: length w div c ≤ i (else (Suc i) * c ≤ ?q * c ≤ length w).
    have q_le_i: ?q ≤ i
    proof (rule ccontr)
      assume ¬ ?q ≤ i
      hence i < ?q by simp
      hence Suc i ≤ ?q by simp
      hence (Suc i) * c ≤ ?q * c by (rule mult_le_mono1)
      with q_c_le have (Suc i) * c ≤ length w by linarith
      with Sci_gt show False by simp
    qed
    have i_eq: i = ?q using i_le_q q_le_i by simp
    have ic_eq_qc: i * c = ?q * c using i_eq by simp
    have mod_pos: 0 < ?r
      using i_c_lt ic_eq_qc div_mod_eq by linarith
    have mod_ne: ?r ≠ 0 using mod_pos by simp
    have len_drop_basic: length (drop (i * c) w) = length w − i * c
      by simp
    have len_drop: length (drop (i * c) w) = ?r
```

```
    using len_drop_basic ic_eq_qc div_mod_eq by linarith
  have mod_lt_c: ?r < c using c_pos by (rule mod_less_divisor)
  have take_eq: take c (drop (i * c) w) = drop (i * c) w
    using len_drop mod_lt_c by simp
  have partial_nth:
    wrap_enc pack c w ! ?q = pack (drop (?q * c) w)
    by (rule wrap_enc_nth_partial[OF c_pos mod_ne])
  have nth_eq:
    wrap_enc pack c w ! i = pack (take c (drop (i * c) w))
    using partial_nth i_eq take_eq ic_eq_qc by simp
  have i_lt_L: i < ?L using i_eq mod_ne len_eq by simp
  show ?thesis using nth_eq i_lt_L by (metis nth_mem)
 qed
qed

lemma wrap_enc_in_Gamma:
 assumes c_pos: 0 < c
    and pack_contract_in:
      ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ∈ Γ_tm M
    and j_lt: j < length (wrap_enc pack c w)
  shows wrap_enc pack c w ! j ∈ Γ_tm M
proof −
 have len_eq:
  length (wrap_enc pack c w)
    = length w div c + (if length w mod c = 0 then 0 else 1)
  by (rule length_wrap_enc[OF c_pos])
 have q_c_le: (length w div c) * c ≤ length w
  by (rule div_times_less_eq_dividend)
 show ?thesis
 proof (cases j < length w div c)
  case True
  hence Suc_j_le_q: Suc j ≤ length w div c by simp
  have Sjc_le_qc: (Suc j) * c ≤ (length w div c) * c
   by (rule mult_le_mono1[OF Suc_j_le_q])
  hence Sjc_le: (Suc j) * c ≤ length w using q_c_le by linarith
  have jc_lt_Sjc: j * c < Suc j * c using c_pos by simp
  have jc_lt: j * c < length w
   using jc_lt_Sjc Sjc_le by linarith
  have nth_eq:
   wrap_enc pack c w ! j = pack (take c (drop (j * c) w))
   by (rule wrap_enc_nth[OF c_pos Sjc_le])
  show ?thesis using nth_eq pack_contract_in[OF jc_lt] by simp
 next
  case False
  with j_lt len_eq have j_eq: j = length w div c
                    and mod_ne: length w mod c ≠ 0
   by (auto split: if_splits)
  have nth_eq:
   wrap_enc pack c w ! j = pack (drop ((length w div c) * c) w)
```

```
    using j_eq wrap_enc_nth_partial[OF c_pos mod_ne] by simp
  have div_mod_eq: (length w div c) * c + length w mod c = length w
    by (rule div_mult_mod_eq)
  have mod_pos: 0 < length w mod c using mod_ne by simp
  have jc_eq: j * c = (length w div c) * c using j_eq by simp
  have jc_lt: j * c < length w
    using jc_eq div_mod_eq mod_pos by linarith
  have len_drop_basic: length (drop (j * c) w) = length w - j * c
    by simp
  have len_drop: length (drop (j * c) w) = length w mod c
    using len_drop_basic jc_eq div_mod_eq by linarith
  have mod_lt_c: length w mod c < c
    using c_pos by (rule mod_less_divisor)
  have take_eq: take c (drop (j * c) w) = drop (j * c) w
    using len_drop mod_lt_c by simp
  have nth_eq': wrap_enc pack c w ! j = pack (take c (drop (j * c) w))
    using nth_eq take_eq j_eq by simp
  show ?thesis using nth_eq' pack_contract_in[OF jc_lt] by simp
 qed
qed

lemma wrap_enc_ne_le:
 assumes c_pos: 0 < c
    and pack_contract_ne_le:
      ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ≠ le_tm M
    and j_lt: j < length (wrap_enc pack c w)
  shows wrap_enc pack c w ! j ≠ le_tm M
proof −
 have len_eq:
   length (wrap_enc pack c w)
     = length w div c + (if length w mod c = 0 then 0 else 1)
   by (rule length_wrap_enc[OF c_pos])
 have q_c_le: (length w div c) * c ≤ length w
   by (rule div_times_less_eq_dividend)
 show ?thesis
 proof (cases j < length w div c)
   case True
   hence Suc_j_le_q: Suc j ≤ length w div c by simp
   have Sjc_le_qc: (Suc j) * c ≤ (length w div c) * c
     by (rule mult_le_mono1[OF Suc_j_le_q])
   hence Sjc_le: (Suc j) * c ≤ length w using q_c_le by linarith
   have jc_lt_Sjc: j * c < Suc j * c using c_pos by simp
   have jc_lt: j * c < length w
     using jc_lt_Sjc Sjc_le by linarith
   have nth_eq:
     wrap_enc pack c w ! j = pack (take c (drop (j * c) w))
     by (rule wrap_enc_nth[OF c_pos Sjc_le])
   show ?thesis using nth_eq pack_contract_ne_le[OF jc_lt] by simp
 next
```

```
    case False
    with j_lt len_eq have j_eq: j = length w div c
                      and mod_ne: length w mod c ≠ 0
      by (auto split: if_splits)
    have nth_eq:
      wrap_enc pack c w ! j = pack (drop ((length w div c) * c) w)
      using j_eq wrap_enc_nth_partial[OF c_pos mod_ne] by simp
    have div_mod_eq: (length w div c) * c + length w mod c = length w
      by (rule div_mult_mod_eq)
    have mod_pos: 0 < length w mod c using mod_ne by simp
    have jc_eq: j * c = (length w div c) * c using j_eq by simp
    have jc_lt: j * c < length w
      using jc_eq div_mod_eq mod_pos by linarith
    have len_drop_basic: length (drop (j * c) w) = length w - j * c
      by simp
    have len_drop: length (drop (j * c) w) = length w mod c
      using len_drop_basic jc_eq div_mod_eq by linarith
    have mod_lt_c: length w mod c < c
      using c_pos by (rule mod_less_divisor)
    have take_eq: take c (drop (j * c) w) = drop (j * c) w
      using len_drop mod_lt_c by simp
    have nth_eq': wrap_enc pack c w ! j = pack (take c (drop (j * c) w))
      using nth_eq take_eq j_eq by simp
    show ?thesis using nth_eq' pack_contract_ne_le[OF jc_lt] by simp
  qed
qed
```

Companion to *wrap_enc_in_Gamma* for the Sigma alphabet (the tightened pack-image discipline from the 2026-05-20 Sigma-tightening of the buffer-close delta-families). Used in the forward direction of *wrap_language* to conclude *set (wrap_enc pack c w) ⊆ Sigma_tm M* from the pack-contracts-in-Sigma derived from a wrap-accepting trace.

```
lemma wrap_enc_in_Sigma:
  assumes c_pos: 0 < c
      and pack_contract_in_Sigma:
        ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ∈ Sigma_tm M
      and j_lt: j < length (wrap_enc pack c w)
    shows wrap_enc pack c w ! j ∈ Sigma_tm M
proof −
  have len_eq:
    length (wrap_enc pack c w)
      = length w div c + (if length w mod c = 0 then 0 else 1)
    by (rule length_wrap_enc[OF c_pos])
  have q_c_le: (length w div c) * c ≤ length w
    by (rule div_times_less_eq_dividend)
  show ?thesis
  proof (cases j < length w div c)
    case True
    hence Suc_j_le_q: Suc j ≤ length w div c by simp
```

```
  have Sjc_le_qc: (Suc j) * c ≤ (length w div c) * c
    by (rule mult_le_mono1[OF Suc_j_le_q])
  hence Sjc_le: (Suc j) * c ≤ length w using q_c_le by linarith
  have jc_lt_Sjc: j * c < Suc j * c using c_pos by simp
  have jc_lt: j * c < length w
    using jc_lt_Sjc Sjc_le by linarith
  have nth_eq:
    wrap_enc pack c w ! j = pack (take c (drop (j * c) w))
    by (rule wrap_enc_nth[OF c_pos Sjc_le])
  show ?thesis using nth_eq pack_contract_in_Sigma[OF jc_lt] by simp
next
  case False
  with j_lt len_eq have j_eq: j = length w div c
                    and mod_ne: length w mod c ≠ 0
    by (auto split: if_splits)
  have nth_eq:
    wrap_enc pack c w ! j = pack (drop ((length w div c) * c) w)
    using j_eq wrap_enc_nth_partial[OF c_pos mod_ne] by simp
  have div_mod_eq: (length w div c) * c + length w mod c = length w
    by (rule div_mult_mod_eq)
  have mod_pos: 0 < length w mod c using mod_ne by simp
  have jc_eq: j * c = (length w div c) * c using j_eq by simp
  have jc_lt: j * c < length w
    using jc_eq div_mod_eq mod_pos by linarith
  have len_drop_basic: length (drop (j * c) w) = length w - j * c
    by simp
  have len_drop: length (drop (j * c) w) = length w mod c
    using len_drop_basic jc_eq div_mod_eq by linarith
  have mod_lt_c: length w mod c < c
    using c_pos by (rule mod_less_divisor)
  have take_eq: take c (drop (j * c) w) = drop (j * c) w
    using len_drop mod_lt_c by simp
  have nth_eq': wrap_enc pack c w ! j = pack (take c (drop (j * c) w))
    using nth_eq take_eq j_eq by simp
  show ?thesis using nth_eq' pack_contract_in_Sigma[OF jc_lt] by simp
qed
qed
```

4 Faithful (k-tape) encoding wrap: transpose primitives

The faithful k-tape encoding wrap runs an alphabet-enlarged machine M on exactly *k_tm M* physical tapes the textbook tape count, matching the textbook *k > 1*. The obvious encoding wrap would prepend a dedicated raw-input tape and run M on physical tapes *1 ... k*, costing *Suc (k_tm M)* tapes; the faithful variant instead reuses the spent input tape. After the encoder has written the encoded input onto physical tape *1* (= M's

tape *0*), physical tape *0* is blanked and serves as M's tape *1*, and the run phase *transposes* physical tapes *0* and *1* (M's tape i lives on physical tape *wrap_tau i*). This is Hopcroft and Ullman's "use the storage tape as the input tape and the old input tape as a storage tape" [6, p. 290], and needs $2 \leq k_tm\ M$ (the textbook $k > 1$). No substrate change: the leftward blanking walk stops at the existing *le* (LE discipline clause 1) and writes no fresh *le*.

This section introduces the tape transposition, the boundary-parameterised encoder families, and the transposed run relation purely additive definitions over the base wrap types above; they are assembled into the plant-*le* combinator *wrap_delta* downstream.

4.1 The tape transposition

wrap_tau swaps tape indices *0* and *1* and fixes every other index. It is the involution that places M's tape *0* (the encoded input) on physical tape *1* and M's tape *1* (the reused input tape) on physical tape *0*.

definition *wrap_tau* :: *nat* ⇒ *nat* **where**
wrap_tau p =
(*case p of 0* ⇒ *Suc 0* | *Suc 0* ⇒ *0* | *Suc* (*Suc k*) ⇒ *Suc* (*Suc k*))

lemma *wrap_tau_0* [*simp*]: *wrap_tau 0* = *Suc 0*
by (*simp add*: *wrap_tau_def*)

lemma *wrap_tau_1* [*simp*]: *wrap_tau* (*Suc 0*) = *0*
by (*simp add*: *wrap_tau_def*)

lemma *wrap_tau_ge2* [*simp*]: *wrap_tau* (*Suc* (*Suc k*)) = *Suc* (*Suc k*)
by (*simp add*: *wrap_tau_def*)

lemma *wrap_tau_invol* [*simp*]: *wrap_tau* (*wrap_tau p*) = *p*
by (*cases p*; *simp*; *rename_tac n*, *case_tac n*; *simp*)

lemma *wrap_tau_ge2_id*: *2* ≤ *p* ⟹ *wrap_tau p* = *p*
by (*cases p*; *simp*; *rename_tac n*, *case_tac n*; *simp*)

lemma *wrap_tau_lt2*: *wrap_tau p* < *2* ⟷ *p* < *2*
by (*cases p*; *simp*; *rename_tac n*, *case_tac n*; *simp*)

4.2 Boundary-parameterised encoder families

The six encoder phase families, generalised over the in-range tape boundary K and instantiated below at the faithful tape count $K = k_tm\ M$. Each read / write / head action addresses only physical tapes *0* and *1*; K marks the boundary above which tapes are out of range and blank. Stating the families over the boundary K keeps the encoder-correctness lemmas independent of

the exact tape count.

```
definition wrap_init_delta_gen ::
  nat ⇒ ('q, 'b) mttm
  ⇒ (('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × ('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × (nat ⇒ dir)) set
where
  wrap_init_delta_gen K M =
    { (W_Init,
       λt. if t < K then Enc (le_tm M) else Enc (bl_tm M),
       W_Buf [],
       λt. if t < K then Enc (le_tm M) else Enc (bl_tm M),
       λt. case t of 0 ⇒ dir.R
                   | Suc k ⇒ (if k = 0 then dir.R else dir.N)) }

definition wrap_buf_extend_delta_gen ::
  nat ⇒ ('q, 'b) mttm ⇒ nat ⇒ 'a set
  ⇒ (('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × ('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × (nat ⇒ dir)) set
where
  wrap_buf_extend_delta_gen K M c Σu =
    { (W_Buf ws, sym, W_Buf (ws @ [a]), sym,
       λt. case t of 0 ⇒ dir.R | _ ⇒ dir.N)
     | ws a sym.
         length ws + 1 < c ∧ a ∈ Σu ∧ set ws ⊆ Σu
         ∧ sym 0 = Raw a
         ∧ (∀ j≥K. sym j = Enc (bl_tm M))
         ∧ sym ∈ UNIV → Raw ‘ Σu ∪ Enc ‘ Γ_tm M }

definition wrap_buf_close_delta_gen ::
  nat ⇒ ('q, 'b) mttm ⇒ ('a list ⇒ 'b) ⇒ nat ⇒ 'a set
  ⇒ (('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × ('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × (nat ⇒ dir)) set
where
  wrap_buf_close_delta_gen K M pack c Σu =
    { (W_Buf ws, sym, W_Buf [],
       (λt. case t of Suc k ⇒ (if k = 0 then Enc (pack (ws @ [a]))
                                        else sym (Suc k))
                    | 0 ⇒ sym 0),
       λt. case t of 0 ⇒ dir.R
                   | Suc k ⇒ (if k = 0 then dir.R else dir.N))
```

```
      | ws a sym.
          length ws + 1 = c ∧ a ∈ Σu ∧ set ws ⊆ Σu
          ∧ sym 0 = Raw a
          ∧ sym (Suc 0) ≠ Enc (le_tm M)
          ∧ pack (ws @ [a]) ∈ Sigma_tm M
          ∧ (∀ j≥K. sym j = Enc (bl_tm M))
          ∧ sym ∈ UNIV → Raw ‘ Σu ∪ Enc ‘ Γ_tm M }

definition wrap_buf_empty_end_delta_gen ::
  nat ⇒ ('q, 'b) mttm ⇒ 'a set
  ⇒ (('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × ('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × (nat ⇒ dir)) set
where
  wrap_buf_empty_end_delta_gen K M Σu =
    { (W_Buf [], sym, W_Reset, sym, λ_. dir.N)
      | sym.
          sym 0 = Enc (bl_tm M)
          ∧ (∀ j≥K. sym j = Enc (bl_tm M))
          ∧ sym ∈ UNIV → Raw ‘ Σu ∪ Enc ‘ Γ_tm M }

definition wrap_buf_nonempty_end_delta_gen ::
  nat ⇒ ('q, 'b) mttm ⇒ ('a list ⇒ 'b) ⇒ nat ⇒ 'a set
  ⇒ (('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × ('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × (nat ⇒ dir)) set
where
  wrap_buf_nonempty_end_delta_gen K M pack c Σu =
    { (W_Buf ws, sym, W_Reset,
       (λt. case t of Suc k ⇒ (if k = 0 then Enc (pack ws)
                               else sym (Suc k))
                    | 0 ⇒ sym 0),
       λt. case t of Suc k ⇒ (if k = 0 then dir.R else dir.N)
                   | 0 ⇒ dir.N)
      | ws sym.
          ws ≠ [] ∧ length ws < c ∧ set ws ⊆ Σu
          ∧ sym 0 = Enc (bl_tm M)
          ∧ sym (Suc 0) ≠ Enc (le_tm M)
          ∧ pack ws ∈ Sigma_tm M
          ∧ (∀ j≥K. sym j = Enc (bl_tm M))
          ∧ sym ∈ UNIV → Raw ‘ Σu ∪ Enc ‘ Γ_tm M }

definition wrap_disp_delta_gen ::
  nat ⇒ ('q, 'b) mttm ⇒ 'a set
  ⇒ (('q, 'a, 'b) wrap_state
```

```
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × ('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × (nat ⇒ dir)) set
where
  wrap_disp_delta_gen K M Σu =
    { (W_Disp, sym, W_Run (s_tm M), sym, λ_. dir.N)
      | sym.
          (∀ j≥K. sym j = Enc (bl_tm M))
          ∧ sym ∈ UNIV → Raw ' Σu ∪ Enc ' Γ_tm M }
```

4.3 Transposed run relation

Phase 5, faithful form. Each M-δ entry $(q, \sigma, q', \sigma', d)$ lifts so that physical tape p carries M's tape *wrap_tau p*: it is read as *Enc* $(\sigma$ *(wrap_tau p))*, written as *Enc* $(\sigma'$ *(wrap_tau p))*, and moved by d *(wrap_tau p)*. No physical tape is frozen as a leftover input tape: physical tape *0* is M's tape *1*, an ordinary work tape.

```
definition wrap_run_delta ::
  ('q, 'b) mttm ⇒ 'a set
   ⇒ (('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × ('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × (nat ⇒ dir)) set
where
  wrap_run_delta M Σu =
    { (W_Run q, sym, W_Run q',
       (λp. Enc (σ' (wrap_tau p))),
       (λp. d (wrap_tau p)))
      | q σ q' σ' d sym.
          (q, σ, q', σ', d) ∈ delta_tm M
          ∧ (∀ p. sym p = Enc (σ (wrap_tau p)))
          ∧ sym ∈ UNIV → Raw ' Σu ∪ Enc ' Γ_tm M }
```

5 Faithful (k-tape) encoding wrap: encoder-phase threading

The boundary-parameterised encoder configurations that the six encoder families thread through. The families address only physical tape *0* (reading the raw input) and physical tape *1* (receiving the encoded input); the in-range boundary *k_tm M* fixes the content of every tape from index *k_tm M* up (out of range, blank). The encoder never touches those tapes, so the config-threading lemmas are stated generically over the boundary K and specialised at K = *k_tm M*.

The threading reuses the boundary-*independent* *wrap_enc* lemmas

(*length_wrap_enc*, *wrap_enc_nth*, *wrap_enc_in_Gamma*, ...); the three encoder configurations defined below are parameterised by the in-range boundary K.

5.1 Boundary-parameterised encoder configurations

The post-encoder waypoint: the encoder has finished, state *W_Reset*, physical tape *0* still holding the raw input, physical tape *1* holding the full *wrap_enc* encoding, both heads at the right end. Generalised over the in-range boundary K, specialised below at the faithful instance K = *k_tm M*.

```
definition post_encoder_config_gen ::
  nat ⇒ ('q, 'b) mttm ⇒ ('a list ⇒ 'b) ⇒ nat ⇒ 'a list
  ⇒ (('a, 'b) wrap_alphabet, ('q, 'a, 'b) wrap_state) mt_config
where
  post_encoder_config_gen K M pack c w =
    Config_M W_Reset
      (λt n. if t < K
          then (case t of
            0 ⇒ (if n = 0 then Enc (le_tm M)
                  else if n ≤ length w then Raw (w ! (n − 1))
                  else Enc (bl_tm M))
          | Suc k ⇒ (if n = 0 then Enc (le_tm M)
                  else if k = 0 ∧ n ≤ length (wrap_enc pack c w)
                    then Enc (wrap_enc pack c w ! (n − 1))
                  else Enc (bl_tm M)))
          else Enc (bl_tm M))
      (λt. case t of
          0 ⇒ length w + 1
        | Suc k ⇒ (if k = 0 then 1 + length (wrap_enc pack c w) else 0))
```

Intermediate encoder configuration after i complete block cycles: state *W_Buf* [], physical tape *1* holding the first i encoded blocks at positions *1*..i with head at *1* + i, physical tape *0* head at *1* + i ∗ c.

```
definition mid_encoder_config_gen ::
  nat ⇒ ('q, 'b) mttm ⇒ ('a list ⇒ 'b) ⇒ nat ⇒ 'a list ⇒ nat
  ⇒ (('a, 'b) wrap_alphabet, ('q, 'a, 'b) wrap_state) mt_config
where
  mid_encoder_config_gen K M pack c w i =
    Config_M (W_Buf []) 
      (λt n. if t < K
          then (case t of
            0 ⇒ (if n = 0 then Enc (le_tm M)
                  else if n ≤ length w then Raw (w ! (n − 1))
                  else Enc (bl_tm M))
          | Suc k ⇒ (if n = 0 then Enc (le_tm M)
                  else if k = 0 ∧ n ≤ i
```

```
                then Enc (wrap_enc pack c w ! (n − 1))
              else Enc (bl_tm M)))
      else Enc (bl_tm M))
  (λt. case t of
      0 ⇒ 1 + i ∗ c
    | Suc k ⇒ (if k = 0 then 1 + i else 0))
```

Finer-grained intermediate inside the i-th cycle, with j input symbols already accumulated in the buffer ($0 \le j \le c - 1$).

```
definition mid_buf_extending_config_gen ::
  nat ⇒ ('q, 'b) mttm ⇒ ('a list ⇒ 'b) ⇒ nat ⇒ 'a list
   ⇒ nat ⇒ nat
   ⇒ (('a, 'b) wrap_alphabet, ('q, 'a, 'b) wrap_state) mt_config
where
  mid_buf_extending_config_gen K M pack c w i j =
    Config_M (W_Buf (take j (drop (i ∗ c) w)))
      (λt n. if t < K
        then (case t of
          0 ⇒ (if n = 0 then Enc (le_tm M)
                else if n ≤ length w then Raw (w ! (n − 1))
                else Enc (bl_tm M))
        | Suc k ⇒ (if n = 0 then Enc (le_tm M)
                else if k = 0 ∧ n ≤ i
                  then Enc (wrap_enc pack c w ! (n − 1))
                else Enc (bl_tm M)))
        else Enc (bl_tm M))
      (λt. case t of
          0 ⇒ 1 + i ∗ c + j
        | Suc k ⇒ (if k = 0 then 1 + i else 0))
```

6 Faithful (k-tape) encoding wrap: combined reset and dispatch

After the encoder phase the faithful wrap is at *post_encoder_config_gen*: physical tape *0* still holds the raw input (the spent input tape), physical tape *1* holds the encoded input, both heads parked at the right end. The *combined reset* walks both heads left at once blanking physical tape *0* (which becomes M's blank work tape *1*) while *preserving* physical tape *1*'s encoded content each head halting at its own *le*; when both read *le* the dispatch fires (*W_Reset* → *W_Disp* → *W_Run* (*s_tm M*)). The resulting configuration is the τ-lift of M's initial configuration on the encoded input.

Physical tape *0*'s content evolves with its head as it is blanked, and the two heads walk at different speeds: the encoded tape is no longer than the input tape, so its head reaches *le* first and then sits there while the input tape finishes blanking.

6.1 The τ-lift of an M-configuration

Physical tape p carries M's tape *wrap_tau p*, encoded. No frozen input tape: every physical tape is an *Enc*-image of an M-tape, so the lift needs no *pack* / *c* / *w* parameters.

definition *lift_M_config* ::
  $('q, 'b)$ *mttm* $\Rightarrow$ $('b, 'q)$ *mt_config*
  $\Rightarrow$ $(('a, 'b)$ *wrap_alphabet*, $('q, 'a, 'b)$ *wrap_state*$)$ *mt_config*
where
  lift_M_config M cM =
    $Config_M$ (*W_Run* (*mt_state cM*))
      (λ*p n*. *if p* < *k_tm M then Enc* (*mt_tape cM* (*wrap_tau p*) *n*)
            else Enc (*bl_tm M*))
      (λ*p*. *mt_pos cM* (*wrap_tau p*))

6.2 The combined reset configuration

State *W_Reset*; physical tape *0* head at *m0* with the raw input blanked above it (cells > *m0* are *bl*); physical tape *1* head at *m1* with the encoded input preserved; tapes *2* ... *K−1* untouched. Parameterised by the in-range boundary K and the two head positions.

definition *mid_reset_combined_config_gen* ::
  nat $\Rightarrow$ $('q, 'b)$ *mttm* $\Rightarrow$ $('a$ *list* $\Rightarrow$ $'b) \Rightarrow$ *nat* $\Rightarrow$ $'a$ *list* $\Rightarrow$ *nat* $\Rightarrow$ *nat*
  $\Rightarrow$ $(('a, 'b)$ *wrap_alphabet*, $('q, 'a, 'b)$ *wrap_state*$)$ *mt_config*
where
  mid_reset_combined_config_gen K M pack c w m0 m1 =
    $Config_M$ *W_Reset*
      (λ*t n*. *if t* < *K*
            then (*case t of*
              0 $\Rightarrow$ (*if n* = *0 then Enc* (*le_tm M*)
                        else if n ≤ *m0* ∧ *n* ≤ *length w then Raw* (*w* ! (*n* − *1*))
                        else Enc (*bl_tm M*))
            | *Suc k* $\Rightarrow$ (*if n* = *0 then Enc* (*le_tm M*)
                        else if k = *0* ∧ *n* ≤ *length* (*wrap_enc pack c w*)
                          then Enc (*wrap_enc pack c w* ! (*n* − *1*))
                        else Enc (*bl_tm M*)))
            else Enc (*bl_tm M*))
      (λ*t*. *case t of*
            0 $\Rightarrow$ *m0*
          | *Suc k* $\Rightarrow$ (*if k* = *0 then m1 else 0*))

Boundary identity: *post_encoder_config_gen* is the top of the combined reset physical tape *0* head at *length w* + *1* (no cell blanked yet, since $n \leq$ *length w* + *1* ∧ $n \leq$ *length w* collapses to $n \leq$ *length w*), physical tape *1* head at *1* + *length* (*wrap_enc pack c w*).

lemma *post_encoder_eq_mid_reset_combined_top*:
  post_encoder_config_gen K M pack c w
    = *mid_reset_combined_config_gen K M pack c w*

```
    (length w + 1) (Suc (length (wrap_enc pack c w)))
  unfolding post_encoder_config_gen_def mid_reset_combined_config_gen_def
  by (simp add: fun_eq_iff split: nat.split)

end
theory AlphabetEnlargement_Pack
  imports AlphabetEnlargement_Reverse Wrap_Base
begin
```

7 AE-wrap glue: *wrap_enc* matches *encode_input*

The bridge between the encoding-wrap combinator's input shape and the alphabet enlargement's encoded input: the *ae_pack* packer, a ceiling-division identity, and the glue lemma *wrap_enc_eq_encode_input* identifying *wrap_enc* under *ae_pack* with the alphabet enlargement's *encode_input*. This bridge is independent of either wrap's tape layout, so it is shared by both the shift and the faithful (transpose) classical-speedup headline theories.

Ceiling-division identity for nat: $\lceil n/c \rceil$ expressed as *n div c + (if n mod c = 0 then 0 else 1)* equals *(n + c − 1) div c*. Used to bridge the closed forms of *length_wrap_enc* and *length_encode_input*.

```
lemma nat_ceil_div_eq:
  fixes n c :: nat
  assumes c_pos: 0 < c
  shows n div c + (if n mod c = 0 then 0 else 1) = (n + c − 1) div c
proof −
  let ?q = n div c
  let ?r = n mod c
  have decomp: n = ?q * c + ?r
    by (simp add: mult.commute)
  show ?thesis
  proof (cases ?r = 0)
    case True
    have lw_eq: n = ?q * c using True decomp by simp
    have step1: (n + c − 1) div c = (?q * c + (c − 1)) div c
      using lw_eq c_pos by (simp add: Nat.add_diff_assoc)
    have c_neq: c ≠ 0 using c_pos by simp
    have step2: (?q * c + (c − 1)) div c = ?q + (c − 1) div c
      using div_mult_self3[OF c_neq, of ?q c − 1] .
    have step3: ?q + (c − 1) div c = ?q
      using c_pos by simp
    have rhs_eq: (n + c − 1) div c = ?q
      using step1 step2 step3 by simp
    show ?thesis using True rhs_eq by simp
  next
    case False
    have r_pos: 0 < ?r using False by simp
```

```
    have r_lt: ?r < c using c_pos by simp
    have rewrite: n + c - 1 = (?q + 1) * c + (?r - 1)
    proof -
      have eq1: n + c - 1 = ?q * c + ?r + c - 1 using decomp by simp
      have eq2: ?q * c + ?r + c - 1 = ?q * c + c + (?r - 1)
        using r_pos by simp
      have eq3: ?q * c + c + (?r - 1) = (?q + 1) * c + (?r - 1)
        by (simp add: distrib_right)
      from eq1 eq2 eq3 show ?thesis by simp
    qed
    have r_m1_lt: ?r - 1 < c using r_lt by simp
    have c_neq: c ≠ 0 using c_pos by simp
    have step1: (n + c - 1) div c = ((?q + 1) * c + (?r - 1)) div c
      using rewrite by simp
    have step2: ((?q + 1) * c + (?r - 1)) div c = (?q + 1) + (?r - 1) div c
      using div_mult_self3[OF c_neq, of ?q + 1 ?r - 1] .
    have step3: (?q + 1) + (?r - 1) div c = ?q + 1
      using r_m1_lt by simp
    have rhs_eq: (n + c - 1) div c = ?q + 1
      using step1 step2 step3 by simp
    show ?thesis using False rhs_eq by simp
  qed
qed
```

The pack function that turns *wrap_enc* into AE's *encode_input*: given a list-shaped block of up to c symbols, build the block function $'c \Rightarrow 'a$ by reading off the block's *c_idx* x-th entry (or the blank *bl* if the block is shorter than *c_idx x + 1*). This is the per-block packer for the AE-wrap instance of *wrap_enc*.

```
definition ae_pack ::
  'a ⇒ 'a list ⇒ (('c :: enum) ⇒ 'a) where
  ae_pack bl block = (λx. if c_idx x < length block
                          then block ! c_idx x
                          else bl)
```

Glue lemma: when the wrap's grouping factor matches AE's $'c$-cardinality, *wrap_enc* with *ae_pack* produces exactly AE's *encode_input*. This rewrite lets an encoding-wrap correctness lemma compose with the alphabet-enlargement results *alphabet_enlarge_language* / *alphabet_enlarge_time*, identifying the wrap's encoded input with the AE-machine's expected input format.

```
lemma wrap_enc_eq_encode_input:
  fixes w :: 'a list and bl :: 'a
  defines c_def: c ≡ card (UNIV :: ('c :: enum) set)
  assumes c_pos: 0 < c
  shows wrap_enc (ae_pack bl :: 'a list ⇒ ('c ⇒ 'a)) c w
        = (encode_input bl w :: ('c ⇒ 'a) list)
proof (rule nth_equalityI)
```

let *?L* = (*length w* + *c* − *1*) *div c*
let *?lhs* = *wrap_enc* (*ae_pack bl* :: *'a list* ⇒ (*'c* ⇒ *'a*)) *c w*
let *?rhs* = *encode_input bl w* :: (*'c* ⇒ *'a*) *list*

— Lengths agree.
have *ceil_id*:
  length w div c + (*if length w mod c* = *0 then 0 else 1*) = *?L*
  by (*rule nat_ceil_div_eq*[*OF c_pos*])
have *len_lhs*: *length ?lhs* = *?L*
proof −
  have *length ?lhs*
        = *length w div c* + (*if length w mod c* = *0 then 0 else 1*)
    by (*rule length_wrap_enc*[*OF c_pos*])
  also have ... = *?L* **using** *ceil_id* **.**
  finally show *?thesis* **.**
qed
have *len_rhs*: *length ?rhs* = *?L*
  using *length_encode_input*[**where** *bl* = *bl* **and** *u* = *w*]
  by (*simp add*: *c_def*)
show *length ?lhs* = *length ?rhs*
  using *len_lhs len_rhs* **by** *simp*

— Per-index agreement.
fix *i* :: *nat*
assume *i_lt*: *i* < *length ?lhs*
with *len_lhs* **have** *i_L*: *i* < *?L* **by** *simp*

— Bound: *i*∗*c* < *length w*, from *Suc i* ≤ *?L* + the div-multiplication iff.
have *i_c_lt_lenw*: *i* ∗ *c* < *length w*
proof −
  from *i_L* **have** *Si_le*: *Suc i* ≤ (*length w* + *c* − *1*) *div c* **by** *simp*
  have *iff_lemma*:
    (*Suc i* ≤ (*length w* + *c* − *1*) *div c*)
      = (*Suc i* ∗ *c* ≤ *length w* + *c* − *1*)
    using *c_pos* **by** (*rule less_eq_div_iff_mult_less_eq*)
  from *Si_le iff_lemma* **have** *step1*: *Suc i* ∗ *c* ≤ *length w* + *c* − *1* **by** *simp*
  have *step2*: *i* ∗ *c* + *c* ≤ *length w* + *c* − *1* **using** *step1* **by** *simp*
  show *?thesis* **using** *step2 c_pos* **by** *linarith*
qed

— Element equality: the *i*-th block agrees pointwise on *'c*.
have *nth_lhs*:
  ?lhs ! *i* = *ae_pack bl* (*take c* (*drop* (*i* ∗ *c*) *w*))
proof (*cases* (*Suc i*) ∗ *c* ≤ *length w*)
  case *True* — full block
  show *?thesis*
    using *wrap_enc_nth*[*OF c_pos True*] **.**
next
  case *False*

```
  — Partial block: i = length w div c, length w mod c ≠ 0.
  have mod_ne: length w mod c ≠ 0
  proof
    assume mod_eq: length w mod c = 0
    from mod_eq have len_div: length w = (length w div c) * c
      using div_mult_mod_eq[of length w c] by simp
    have L_eq: ?L = length w div c
      using nat_ceil_div_eq[OF c_pos, of length w] mod_eq by simp
    from i_L L_eq have i_lt: i < length w div c by simp
    hence Suc i ≤ length w div c by simp
    hence Suc i * c ≤ (length w div c) * c
      by (rule mult_le_mono1)
    with len_div have Suc i * c ≤ length w by simp
    with False show False ..
  qed
  have i_eq: i = length w div c
  proof −
    have L_eq: ?L = length w div c + 1
      using nat_ceil_div_eq[OF c_pos, of length w] mod_ne by simp
    from i_L L_eq have i_le: i ≤ length w div c by simp
    from False have length w < Suc i * c by simp
    hence length w div c < Suc i
      using c_pos by (simp add: div_less_iff_less_mult mult.commute)
    hence length w div c ≤ i by simp
    with i_le show ?thesis by simp
  qed
  have nth_partial:
    ?lhs ! (length w div c) = ae_pack bl (drop ((length w div c) * c) w)
    by (rule wrap_enc_nth_partial[OF c_pos mod_ne])
  have drop_short:
    length (drop ((length w div c) * c) w) < c
  proof −
    have len_drop: length (drop ((length w div c) * c) w)
                   = length w − (length w div c) * c
      by simp
    have mod_eq: length w − (length w div c) * c = length w mod c
      using div_mult_mod_eq[of length w c] by linarith
    have mod_lt: length w mod c < c using c_pos by simp
    from len_drop mod_eq mod_lt show ?thesis by simp
  qed
  have take_eq:
    take c (drop ((length w div c) * c) w)
      = drop ((length w div c) * c) w
    using drop_short by simp
  show ?thesis
    using nth_partial i_eq take_eq by simp
qed

show ?lhs ! i = ?rhs ! i
```

```
proof (rule ext)
  fix x :: 'c
  have cx_lt: c_idx x < c
    unfolding c_def by (rule c_idx_lt_card)

  — The right-hand entry, point-x form, via encode_input_nth.
  have rhs_i_lt: i < length ?rhs using i_lt len_lhs len_rhs by simp
  have nth_rhs_x:
    (?rhs ! i) x = (if i * c + c_idx x < length w
                    then w ! (i * c + c_idx x) else bl)
    using encode_input_nth[OF rhs_i_lt, where x = x]
    by (simp add: c_def Let_def)

  show (?lhs ! i) x = (?rhs ! i) x
  proof (cases (Suc i) * c ≤ length w)
    case True  — full block
    have len_eq: length (take c (drop (i * c) w)) = c
      using True by simp
    have j_lt: i * c + c_idx x < length w
    proof −
      from True have i * c + c ≤ length w by simp
      with cx_lt show ?thesis by linarith
    qed
    have (?lhs ! i) x = (ae_pack bl (take c (drop (i * c) w))) x
      using nth_lhs by simp
    also have ... = (if c_idx x < length (take c (drop (i * c) w))
                     then (take c (drop (i * c) w)) ! c_idx x else bl)
      unfolding ae_pack_def by simp
    also have ... = (take c (drop (i * c) w)) ! c_idx x
      using cx_lt len_eq by simp
    also have ... = w ! (i * c + c_idx x)
      using True cx_lt by (simp add: nth_drop)
    also have ... = (?rhs ! i) x
      using nth_rhs_x j_lt by simp
    finally show ?thesis .
  next
    case False
    have len_eq: length (take c (drop (i * c) w)) = length w − i * c
      using False i_c_lt_lenw by simp
    have cond_iff:
      (c_idx x < length w − i * c) = (i * c + c_idx x < length w)
      using i_c_lt_lenw by linarith
    have lhs_x:
      (?lhs ! i) x = (if c_idx x < length w − i * c
                      then (take c (drop (i * c) w)) ! c_idx x
                      else bl)
    proof −
      have (?lhs ! i) x = (ae_pack bl (take c (drop (i * c) w))) x
        using nth_lhs by simp
```

```
      also have ... = (if c_idx x < length (take c (drop (i * c) w))
                       then (take c (drop (i * c) w)) ! c_idx x else bl)
        unfolding ae_pack_def by simp
      also have ... = (if c_idx x < length w - i * c
                       then (take c (drop (i * c) w)) ! c_idx x else bl)
        using len_eq by simp
      finally show ?thesis .
    qed
    show ?thesis
    proof (cases c_idx x < length w - i * c)
      case True
      from True cond_iff have j_lt: i * c + c_idx x < length w by simp
      have nth_take: (take c (drop (i * c) w)) ! c_idx x
                       = w ! (i * c + c_idx x)
        using True cx_lt i_c_lt_lenw by (simp add: nth_drop)
      have lhs_simp: (?lhs ! i) x = w ! (i * c + c_idx x)
        using lhs_x True nth_take by simp
      have rhs_simp: (?rhs ! i) x = w ! (i * c + c_idx x)
        using nth_rhs_x j_lt by simp
      show ?thesis using lhs_simp rhs_simp by simp
    next
      case False
      from False cond_iff have j_ge: ¬ i * c + c_idx x < length w by simp
      have lhs_simp: (?lhs ! i) x = bl
        using lhs_x False by simp
      have rhs_simp: (?rhs ! i) x = bl
        using nth_rhs_x j_ge by simp
      show ?thesis using lhs_simp rhs_simp by simp
    qed
  qed
 qed
qed

end
theory Wrap_Defs
 imports Wrap_Base Multitape_TM_Substrate.Multitape_Origin_Float
begin
```

8 Faithful (k-tape) encoding wrap: plant-*le* variant (HU 12.4)

Reaching faithful k tapes for the super-linear speed-up [6, Theorem 12.3] by *rewinding* the reused input tape costs a $\sim 2n$ setup that the super-linear growth hypothesis absorbs. Theorem 12.4's tight $(1+\varepsilon)n$ bound cannot absorb it, so this variant avoids the rewind of the input tape entirely: it plants a fresh *le* at the input head's final position, *floating* the origin (the spent raw input to its left becomes unreachable, penned off by clause 1).

Only the reset phase differs from the transpose wrap. The storage tape (physical *1*, carrying M's tape 0 = the encoded input) is still rewound to its *le* — but that is only $\lceil n/c \rceil$ cells (the encoded length), the $\varepsilon{\cdot}n$ term — via the generic single-tape rewind *wrap_rewind_loop_delta_gen*. The input tape (physical *0*, becoming M's tape 1) is *not* rewound: the rewind-done step plants *le* on it and hands over to *W_Disp*. Everything else — the six transpose encoder families, the transposed run, the dispatch — is reused verbatim.

The planted *le* makes the wrap *not le_unique* (it writes *le* where it did not read *le*); it remains *valid_mttm*, since clause 1 constrains only transitions that *read le*. The engine is then run from a floated-origin init config, bridged back to the proper *init_config_mttm* by the origin-float lemmas of *Multitape_TM_Substrate.Multitape_Origin_Float*.

8.1 The plant-*le* reset done step

Rewind-done for the plant variant: fires when the storage tape (index *Suc 0*) reads *le* (its cell 0, reached by the generic rewind loop), and in the same transition writes *le* on the input tape (index *0*) at its current head — the plant — handing over to *W_Disp*. No head moves. This is the sole wrap-B-specific reset family; the loop is the generic *wrap_rewind_loop_delta_gen*.

```
definition wrap_plant_done_delta ::
  ('q, 'b) mttm ⇒ 'a set
  ⇒ (('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × ('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × (nat ⇒ dir)) set
where
  wrap_plant_done_delta M Σu =
    { (W_Reset, sym, W_Disp,
       (λt. if t = 0 then Enc (le_tm M) else sym t),
       λ_. dir.N)
      | sym.
          sym (Suc 0) = Enc (le_tm M)
          ∧ (∀ i≥k_tm M. sym i = Enc (bl_tm M))
          ∧ sym ∈ UNIV → Raw ‘ Σu ∪ Enc ‘ Γ_tm M }
```

8.2 The plant-*le* combinator

The plant wrap's transition relation: the six transpose encoder families at $K = k_tm\ M$, the transposed run and dispatch, and — for the reset — the generic single-tape storage rewind (loop) plus the plant-done step.

```
definition wrap_delta ::
  ('q, 'b) mttm ⇒ ('a list ⇒ 'b) ⇒ nat ⇒ 'a set
  ⇒ (('q, 'a, 'b) wrap_state
```

```
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × ('q, 'a, 'b) wrap_state
      × (nat ⇒ ('a, 'b) wrap_alphabet)
      × (nat ⇒ dir)) set
where
  wrap_delta M pack c Σu =
    wrap_init_delta_gen (k_tm M) M
    ∪ wrap_buf_extend_delta_gen (k_tm M) M c Σu
    ∪ wrap_buf_close_delta_gen (k_tm M) M pack c Σu
    ∪ wrap_buf_empty_end_delta_gen (k_tm M) M Σu
    ∪ wrap_buf_nonempty_end_delta_gen (k_tm M) M pack c Σu
    ∪ wrap_rewind_loop_delta_gen W_Reset (Suc 0) (k_tm M) M Σu
    ∪ wrap_plant_done_delta M Σu
    ∪ wrap_disp_delta_gen (k_tm M) M Σu
    ∪ wrap_run_delta M Σu
```

The faithful k-tape plant-le encoding wrap: transition relation *wrap_delta* at tape count *k_tm M*, reusing the wrap state set (no new state — the plant folds into the reset-done step). Well-formed only when $2 \leq$ *k_tm M*, as for the transpose wrap.

```
fun encoding_wrap ::
  ('q, 'b) mttm ⇒ ('a list ⇒ 'b) ⇒ nat ⇒ 'a set
   ⇒ (('q, 'a, 'b) wrap_state, ('a, 'b) wrap_alphabet) mttm
where
  encoding_wrap M pack c Σu =
    MTTM
      (wrap_state_set M c Σu)
      (Raw ` Σu)
      (Raw ` Σu ∪ Enc ` Γ_tm M)
      (Enc (bl_tm M))
      (Enc (le_tm M))
      (wrap_delta M pack c Σu)
      W_Init
      (W_Run (t_tm M))
      W_Rej
      (k_tm M)
```

8.3 Well-formedness of the plant-*le* wrap

Phase-family case split for *wrap_delta*: a transition belongs to exactly one of the nine builders — the six boundary-parameterised encoder families at *k_tm M*, the generic single-tape storage rewind loop, the plant-le rewind-done step, and the transposed run.

```
lemma wrap_delta_cases:
  assumes (q, a, q', a', d) ∈ wrap_delta M pack c Σu
  obtains
    (init) (q, a, q', a', d) ∈ wrap_init_delta_gen (k_tm M) M
  | (ext) (q, a, q', a', d) ∈ wrap_buf_extend_delta_gen (k_tm M) M c Σu
```

```
  | (close) (q, a, q', a', d) ∈ wrap_buf_close_delta_gen (k_tm M) M pack c Σu
  | (eend) (q, a, q', a', d) ∈ wrap_buf_empty_end_delta_gen (k_tm M) M Σu
  | (nend) (q, a, q', a', d) ∈ wrap_buf_nonempty_end_delta_gen (k_tm M) M
pack c Σu
  | (rloop) (q, a, q', a', d)
              ∈ wrap_rewind_loop_delta_gen W_Reset (Suc 0) (k_tm M) M Σu
  | (rdone) (q, a, q', a', d) ∈ wrap_plant_done_delta M Σu
  | (disp) (q, a, q', a', d) ∈ wrap_disp_delta_gen (k_tm M) M Σu
  | (run) (q, a, q', a', d) ∈ wrap_run_delta M Σu
  using assms unfolding wrap_delta_def by blast
```

Obligation 1 — well-formedness of the faithful k-tape plant-*le* wrap at tape count *k_tm M*. Hypotheses: $2 \le$ *k_tm M*, *valid_mttm M*, and *le_tm M* $\neq$ *bl_tm M* (the storage rewind passes *bl* through unchanged, and the plant writes *le*, so the no-spurious-LE obligations need them distinct). It does *not* require *le_unique* of the input, and — crucially — the resulting wrap is itself *not le_unique*: the plant-done step writes *le* on physical tape *0* without reading it there. That is legal for *valid_mttm*, whose clause 1 (obligation 14 below) constrains only transitions that *read le*; the dropped clause 2 was exactly the no-planting rule.

```
lemma wrap_wf:
  assumes valM: valid_mttm M
      and le_ne_bl: le_tm M ≠ bl_tm M
      and finSu: finite Σu
      and c_pos: 0 < c
      and k2: 2 ≤ k_tm M
  shows valid_mttm (encoding_wrap M pack c Σu)
proof −
  have eq:
    encoding_wrap M pack c Σu =
      MTTM (wrap_state_set M c Σu)
           (Raw ' Σu)
           (Raw ' Σu ∪ Enc ' Γ_tm M)
           (Enc (bl_tm M))
           (Enc (le_tm M))
           (wrap_delta M pack c Σu)
           W_Init
           (W_Run (t_tm M))
           W_Rej
           (k_tm M)
    by simp
  show ?thesis
    unfolding eq valid_mttm.simps
  proof (intro conjI)
    — (1) finite Q'
    show finite (wrap_state_set M c Σu)
      by (rule finite_wrap_state_set[OF valM finSu])
    — (2) finite Γ'
```

show *finite* (*Raw* ‘ Σ*u* ∪ *Enc* ‘ Γ_*tm M*)
using *finSu valid_mttm_finite_Gamma*[*OF valM*] **by** *simp*
— (3) Σ’ ⊆ Γ’
show *Raw* ‘ Σ*u* ⊆ *Raw* ‘ Σ*u* ∪ *Enc* ‘ Γ_*tm M*
by *blast*
— (4) start in Q’
show *W_Init* ∈ *wrap_state_set M c* Σ*u*
unfolding *wrap_state_set_def* **by** *simp*
— (5) accept in Q’
show *W_Run* (*t_tm M*) ∈ *wrap_state_set M c* Σ*u*
using *valid_mttm_t_in_Q*[*OF valM*]
unfolding *wrap_state_set_def* **by** *simp*
— (6) reject in Q’
show *W_Rej* ∈ *wrap_state_set M c* Σ*u*
unfolding *wrap_state_set_def* **by** *simp*
— (7a) bl in Γ’
show *Enc* (*bl_tm M*) ∈ *Raw* ‘ Σ*u* ∪ *Enc* ‘ Γ_*tm M*
using *valid_mttm_blank_in_Gamma*[*OF valM*] **by** *blast*
— (7b) bl not in Σ’
show *Enc* (*bl_tm M*) ∉ *Raw* ‘ Σ*u*
by *auto*
— (8a) le in Γ’
show *Enc* (*le_tm M*) ∈ *Raw* ‘ Σ*u* ∪ *Enc* ‘ Γ_*tm M*
using *valid_mttm_LE_in_Gamma*[*OF valM*] **by** *blast*
— (8b) le not in Σ’
show *Enc* (*le_tm M*) ∉ *Raw* ‘ Σ*u*
by *auto*
— (11) accept ≠ reject
show *W_Run* (*t_tm M*) ≠ *W_Rej*
by *simp*
— (12) *0* < *k*: the plant wrap keeps M’s tape count.
show *0* < *k_tm M*
using *valid_mttm_k_pos*[*OF valM*] **by** *simp*
— (13) *δ*-shape: source / dest / read / write range typing. The plant writes *Enc le* on tape *0*, which is in range by *valid_mttm ?M* ⟹ *le_tm ?M* ∈ Γ_*tm ?M*.
show *wrap_delta M pack c* Σ*u*
⊆ (*wrap_state_set M c* Σ*u* − {*W_Run* (*t_tm M*), *W_Rej*})
× (*UNIV* → *Raw* ‘ Σ*u* ∪ *Enc* ‘ Γ_*tm M*)
× *wrap_state_set M c* Σ*u*
× (*UNIV* → *Raw* ‘ Σ*u* ∪ *Enc* ‘ Γ_*tm M*)
× (*UNIV* → *UNIV*)
unfolding *wrap_delta_def*
wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
wrap_buf_nonempty_end_delta_gen_def
wrap_rewind_loop_delta_gen_def
wrap_plant_done_delta_def
wrap_disp_delta_gen_def wrap_run_delta_def
wrap_state_set_def

using *c_pos valid_mttm_t_in_Q*[*OF valM*] *valid_mttm_s_in_Q*[*OF valM*]
valid_mttm_LE_in_Gamma[*OF valM*]
valid_mttm_blank_in_Gamma[*OF valM*]
valid_mttm_delta_set[*OF valM*]
valid_mttm_Sigma_sub_Gamma[*OF valM*]
by (*fastforce split*: *nat.split if_splits*)

— (14) δLE-preservation: read LE $\Longrightarrow$ write LE and dir $\in$ N, R. The plant-done step writes LE on tape *0* regardless of what it read there — legal, since this obligation only fires when the read cell already *is* LE, and then the plant writes LE anyway. The rewind loop only moves the storage head (*Suc 0*), whose read cannot be LE (its guard), so no LE-read cell moves.

show $\forall\, q\ a\ q'\ a'\ d\ k.$
$(q,\ a,\ q',\ a',\ d) \in \mathit{wrap_delta}\ M\ \mathit{pack}\ c\ \Sigma u \longrightarrow$
$a\ k = \mathit{Enc}\ (\mathit{le_tm}\ M) \longrightarrow$
$a'\ k = \mathit{Enc}\ (\mathit{le_tm}\ M) \wedge d\ k \in \{\mathit{dir.N},\ \mathit{dir.R}\}$
proof (*intro allI impI*)
fix $q\ a\ q'\ a'\ d\ k$
assume *in_δ*: $(q,\ a,\ q',\ a',\ d) \in \mathit{wrap_delta}\ M\ \mathit{pack}\ c\ \Sigma u$
and *LE*: $a\ k = \mathit{Enc}\ (\mathit{le_tm}\ M)$
from *in_δ* **show** $a'\ k = \mathit{Enc}\ (\mathit{le_tm}\ M) \wedge d\ k \in \{\mathit{dir.N},\ \mathit{dir.R}\}$
proof (*cases rule*: *wrap_delta_cases*)
case *init*
thus *?thesis* **using** *LE* **unfolding** *wrap_init_delta_gen_def*
by (*auto split*: *nat.splits if_splits*)
next
case *ext*
thus *?thesis* **using** *LE* **unfolding** *wrap_buf_extend_delta_gen_def*
by (*auto split*: *nat.splits*)
next
case *close*
thus *?thesis* **using** *LE* **unfolding** *wrap_buf_close_delta_gen_def*
by (*fastforce split*: *nat.splits if_splits*)
next
case *eend*
thus *?thesis* **using** *LE* **unfolding** *wrap_buf_empty_end_delta_gen_def*
by (*auto split*: *nat.splits*)
next
case *nend*
thus *?thesis* **using** *LE* **unfolding** *wrap_buf_nonempty_end_delta_gen_def*
by (*fastforce split*: *nat.splits if_splits*)
next
case *rloop*

— Storage rewind: write = read, so LE is preserved; the only moving head is *Suc 0*, whose guard forbids reading LE, so a LE-read cell (necessarily $k \neq$ *Suc 0*) stays stationary.

thus *?thesis* **using** *LE* **unfolding** *wrap_rewind_loop_delta_gen_def*
by (*auto split*: *if_splits*)
next
case *rdone*

— Plant-done: read = write on every tape except *0*, where the plant writes LE; either way the write at a LE-read cell is LE, and no head moves.
thus *?thesis* **using** *LE* **unfolding** *wrap_plant_done_delta_def*
by (*auto split*: *if_splits*)
next
case *disp*
thus *?thesis* **using** *LE* **unfolding** *wrap_disp_delta_gen_def*
by (*auto split*: *nat.splits*)
next
case *run*
then obtain *qq* σ q'' σ' *dd sym* **where**
aeq: $a = sym$
and $a'eq$: $a' = (\lambda p.\ Enc\ (\sigma'\ (wrap_tau\ p)))$
and *deq*: $d = (\lambda p.\ dd\ (wrap_tau\ p))$
and *trM*: $(qq, \sigma, q'', \sigma', dd) \in delta_tm\ M$
and *symc*: $\forall p.\ sym\ p = Enc\ (\sigma\ (wrap_tau\ p))$
unfolding *wrap_run_delta_def* **by** *auto*
have $sym\ k = Enc\ (le_tm\ M)$ **using** *LE aeq* **by** *simp*
moreover have $sym\ k = Enc\ (\sigma\ (wrap_tau\ k))$ **using** *symc* **by** *simp*
ultimately have $\sigma\ (wrap_tau\ k) = le_tm\ M$ **by** *simp*
from *valid_mttm_deltaLE*[*OF valM trM this*]
have $\sigma'\ (wrap_tau\ k) = le_tm\ M \wedge dd\ (wrap_tau\ k) \in \{dir.N, dir.R\}$ **.**
with $a'eq$ *deq* **show** *?thesis* **by** *simp*
qed
qed
— (15) δ-support: beyond the tape count each transition reads and writes blank and is stationary. Since $2 \le k_tm\ M \le j$ we have $2 \le j$: the storage head $Suc\ 0$ and the plant tape *0* are both $< j$, so both reset families are blank-stationary there, and the run inherits $[\![valid_mttm\ ?M; (?q, ?a, ?q', ?a', ?d) \in delta_tm\ ?M; k_tm\ ?M \le ?j]\!] \Longrightarrow ?a\ ?j = bl_tm\ ?M \wedge ?a'\ ?j = bl_tm\ ?M \wedge ?d\ ?j = dir.N$.
show $\forall q\ a\ q'\ a'\ d.$
$(q, a, q', a', d) \in wrap_delta\ M\ pack\ c\ \Sigma u \longrightarrow$
$(\forall j \ge k_tm\ M.$
$a\ j = Enc\ (bl_tm\ M) \wedge a'\ j = Enc\ (bl_tm\ M) \wedge d\ j = dir.N)$
proof (*intro allI impI*)
fix $q\ a\ q'\ a'\ d\ j$
assume *in_δ*: $(q, a, q', a', d) \in wrap_delta\ M\ pack\ c\ \Sigma u$
and *jge*: $k_tm\ M \le j$
from *k2 jge* **have** *j2*: $2 \le j$ **by** *simp*
from *in_δ*
show $a\ j = Enc\ (bl_tm\ M) \wedge a'\ j = Enc\ (bl_tm\ M) \wedge d\ j = dir.N$
proof (*cases rule*: *wrap_delta_cases*)
case *init*
thus *?thesis* **using** *jge j2* **unfolding** *wrap_init_delta_gen_def*
by (*auto split*: *nat.splits if_splits*)
next
case *ext*
thus *?thesis* **using** *jge j2* **unfolding** *wrap_buf_extend_delta_gen_def*
by (*auto split*: *nat.splits*)

```
      next
        case close
        thus ?thesis using jge j2 unfolding wrap_buf_close_delta_gen_def
          by (auto split: nat.splits if_splits)
      next
        case eend
       thus ?thesis using jge j2 unfolding wrap_buf_empty_end_delta_gen_def
          by (auto split: nat.splits)
      next
        case nend
                                thus ?thesis using jge j2 unfolding
wrap_buf_nonempty_end_delta_gen_def
          by (auto split: nat.splits if_splits)
      next
        case rloop
        thus ?thesis using jge j2 unfolding wrap_rewind_loop_delta_gen_def
          by (auto split: if_splits)
      next
        case rdone
        thus ?thesis using jge j2 unfolding wrap_plant_done_delta_def
          by (auto split: if_splits)
      next
        case disp
        thus ?thesis using jge j2 unfolding wrap_disp_delta_gen_def
          by (auto split: nat.splits)
      next
        case run
        then obtain qq σ q'' σ' dd sym where
            aeq: a = sym
            and a'eq: a' = (λp. Enc (σ' (wrap_tau p)))
            and deq: d = (λp. dd (wrap_tau p))
            and trM: (qq, σ, q'', σ', dd) ∈ delta_tm M
            and symc: ∀p. sym p = Enc (σ (wrap_tau p))
          unfolding wrap_run_delta_def by auto
        have tauj: wrap_tau j = j using j2 by (rule wrap_tau_ge2_id)
        from valid_mttm_delta_support[OF valM trM jge]
        have supp: σ j = bl_tm M ∧ σ' j = bl_tm M ∧ dd j = dir.N .
        have a j = Enc (bl_tm M) using aeq symc tauj supp by simp
        moreover have a' j = Enc (bl_tm M) using a'eq tauj supp by simp
        moreover have d j = dir.N using deq tauj supp by simp
        ultimately show ?thesis by simp
      qed
    qed
  qed
qed
```

8.4 Determinism of the plant-*le* wrap

Obligation 4 — determinism preservation, conditional on *det_mttm M*. The encoder families and dispatch are functional per read pattern, and the transposed run inherits M-determinism through *wrap_tau*. Only the *W_Reset* case differs: instead of the two combined-reset families it disambiguates the generic storage rewind loop from the plant-done step by their *le*-guards on the storage tape (index *Suc 0*) — the loop reads a non-*le* there, the plant-done reads *le* — each functional in the read.

```
lemma wrap_det:
  assumes valM: valid_mttm M
      and detM: det_mttm M
      and finSu: finite Σu
      and c_pos: 0 < c
  shows det_mttm (encoding_wrap M pack c Σu)
  unfolding det_mttm_def
proof (intro allI impI)
  fix q a p1 b1 d1 p2 b2 d2
  assume t1: (q, a, p1, b1, d1) ∈ delta_tm (encoding_wrap M pack c Σu)
    and t2: (q, a, p2, b2, d2) ∈ delta_tm (encoding_wrap M pack c Σu)
  hence t1w: (q, a, p1, b1, d1) ∈ wrap_delta M pack c Σu
   and t2w: (q, a, p2, b2, d2) ∈ wrap_delta M pack c Σu
   by simp_all
  show (p1, b1, d1) = (p2, b2, d2)
  proof (cases q)
    case W_Init
    with t1w t2w show ?thesis
      unfolding wrap_delta_def
                wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
              wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
                wrap_buf_nonempty_end_delta_gen_def
                wrap_rewind_loop_delta_gen_def
                wrap_plant_done_delta_def
                wrap_disp_delta_gen_def wrap_run_delta_def
      by auto
  next
    case (W_Buf ws)
    with t1w t2w show ?thesis
      unfolding wrap_delta_def
                wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
              wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
                wrap_buf_nonempty_end_delta_gen_def
                wrap_rewind_loop_delta_gen_def
                wrap_plant_done_delta_def
                wrap_disp_delta_gen_def wrap_run_delta_def
      by auto
  next
    case W_Reset
```

— Both storage-rewind families read *sym* (*Suc 0*); the loop fires when it is not *le* (looping in *W_Reset*, no head moves but *Suc 0* left), the plant-done when it is *le* (to *W_Disp*, planting *le* on tape *0*). Their guards are complementary, so at most one fires per read.

with *t1w t2w* **show** *?thesis*
unfolding *wrap_delta_def*
wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
wrap_buf_nonempty_end_delta_gen_def
wrap_rewind_loop_delta_gen_def
wrap_plant_done_delta_def
wrap_disp_delta_gen_def wrap_run_delta_def
by (*auto split*: *if_splits*)
next
case *W_Disp*
with *t1w t2w* **show** *?thesis*
unfolding *wrap_delta_def*
wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
wrap_buf_nonempty_end_delta_gen_def
wrap_rewind_loop_delta_gen_def
wrap_plant_done_delta_def
wrap_disp_delta_gen_def wrap_run_delta_def
by *auto*
next
case (*W_Run qM*)

— Only *wrap_run_delta* has a *W_Run* source. Extract the M-side δ-tuples linked to each wrap-tuple, then use M-determinism.

from *t1w W_Run* **obtain** *sig1 q1′ sig1′ dd1* **where**
m1: (*qM*, *sig1*, *q1′*, *sig1′*, *dd1*) ∈ *delta_tm M*
and *sym1*: ∀ *p*. *a p* = *Enc* (*sig1* (*wrap_tau p*))
and *p1_eq*: *p1* = *W_Run q1′*
and *b1_eq*: *b1* = (λ*p*. *Enc* (*sig1′* (*wrap_tau p*)))
and *d1_eq*: *d1* = (λ*p*. *dd1* (*wrap_tau p*))
unfolding *wrap_delta_def*
wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
wrap_buf_nonempty_end_delta_gen_def
wrap_rewind_loop_delta_gen_def
wrap_plant_done_delta_def
wrap_disp_delta_gen_def wrap_run_delta_def
by *auto*
from *t2w W_Run* **obtain** *sig2 q2′ sig2′ dd2* **where**
m2: (*qM*, *sig2*, *q2′*, *sig2′*, *dd2*) ∈ *delta_tm M*
and *sym2*: ∀ *p*. *a p* = *Enc* (*sig2* (*wrap_tau p*))
and *p2_eq*: *p2* = *W_Run q2′*
and *b2_eq*: *b2* = (λ*p*. *Enc* (*sig2′* (*wrap_tau p*)))
and *d2_eq*: *d2* = (λ*p*. *dd2* (*wrap_tau p*))
unfolding *wrap_delta_def*

```
      wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
    wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
      wrap_buf_nonempty_end_delta_gen_def
      wrap_rewind_loop_delta_gen_def
      wrap_plant_done_delta_def
      wrap_disp_delta_gen_def wrap_run_delta_def
    by auto
  have sigma_eq: sig1 = sig2
  proof (rule ext)
    fix x
    have e1: a (wrap_tau x) = Enc (sig1 (wrap_tau (wrap_tau x)))
      using sym1 by blast
    have e2: a (wrap_tau x) = Enc (sig2 (wrap_tau (wrap_tau x)))
      using sym2 by blast
   from e1 e2 have sig1 (wrap_tau (wrap_tau x)) = sig2 (wrap_tau (wrap_tau
x))
      by simp
    thus sig1 x = sig2 x by (simp add: wrap_tau_invol)
  qed
  have M_det: (q1', sig1', dd1) = (q2', sig2', dd2)
    using m1 m2 sigma_eq detM[unfolded det_mttm_def] by blast
  hence q'_eq: q1' = q2' and sig'_eq: sig1' = sig2' and dd_eq: dd1 = dd2
    by simp_all
  show ?thesis
    using p1_eq p2_eq b1_eq b2_eq d1_eq d2_eq q'_eq sig'_eq dd_eq
    by metis
 next
  case W_Rej
  with t1w show ?thesis
    unfolding wrap_delta_def
      wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
    wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
      wrap_buf_nonempty_end_delta_gen_def
      wrap_rewind_loop_delta_gen_def
      wrap_plant_done_delta_def
      wrap_disp_delta_gen_def wrap_run_delta_def
    by auto
 qed
qed

end
theory Wrap_Reset
 imports Wrap_Defs
begin
```

9 Faithful (k-tape) plant-*le* wrap: reset threading and the origin float

The plant wrap's reset differs from a combined blank-and-rewind reset in that it *does not* walk the input tape (physical *0*) back to its *le*. It rewinds only the storage tape (physical *1*, carrying M's tape *0* = the encoded input) — the $\lceil n/c \rceil$ cells that make up the $\varepsilon{\cdot}n$ term — freezing the input head at *length w + 1* throughout. When the storage head reaches its *le* at cell *0*, the plant-done step writes a fresh *le* on the input tape at the frozen head and hands to *W_Disp*.

The resulting engine configuration is the input tape *floated*: physical tape *0* carries a planted *le* at cell *length w + 1* with the spent raw input as an unreachable prefix below it, and blanks above — indistinguishable, from the engine's position-agnostic view, from a fresh blank work tape whose origin sits at *length w + 1*. This is exactly a *shift_rel* by *length w + 1* of the τ-lift of M's initial configuration, so the origin-float lemmas of *Multitape_TM_Substrate.Multitape_Origin_Float* bridge the floated run back to the proper *init_config_mttm* the transpose run machinery consumes.

9.1 The storage-tape rewind, input head frozen

One generic-rewind step retracts the storage head (physical *Suc 0*) from *Suc m1* to *m1*, the input head frozen at *length w + 1* and every tape's content preserved. This reuses *mid_reset_combined_config_gen* with the input head *m0* pinned at *length w + 1*: since that config only blanks cells $>$ *m0* and *length w* $<$ *length w + 1*, the raw input stays intact and the configuration depends on the storage head *m1* only through its position.

lemma *mid_plant_rewind_step*:
  assumes *vM*: *valid_mttm M*
    and *c_pos*: *0 < c*
    and *w_alpha*: *set w* $\subseteq$ Σu
    and *bl_ne_le*: *bl_tm M* $\neq$ *le_tm M*
    and *k2*: *2* $\leq$ *k_tm M*
    and *pack_in*:
      $\bigwedge i$. *i * c < length w* $\Longrightarrow$ *pack (take c (drop (i * c) w))* $\in$ Γ*_tm M*
    and *pack_ne_le*:
      $\bigwedge i$. *i * c < length w* $\Longrightarrow$ *pack (take c (drop (i * c) w))* $\neq$ *le_tm M*
  shows (*mid_reset_combined_config_gen (k_tm M) M pack c w (length w + 1) (Suc m1)*,
      mid_reset_combined_config_gen (k_tm M) M pack c w (length w + 1) m1)
      $\in$ *mttm_step (wrap_delta M pack c* Σu*)*
proof −
  let *?src = mid_reset_combined_config_gen (k_tm M) M pack c w (length w + 1) (Suc m1)*
  let *?L = length (wrap_enc pack c w)*

let *?sym* = λk. *mt_tape ?src k* (*mt_pos ?src k*)

have *bl_in*: *bl_tm M* ∈ Γ*_tm M* **by** (*rule valid_mttm_blank_in_Gamma*[*OF vM*])
have *le_in*: *le_tm M* ∈ Γ*_tm M* **by** (*rule valid_mttm_LE_in_Gamma*[*OF vM*])
have *kpos*: *0* < *k_tm M* **using** *k2* **by** *simp*

have *src_state*: *mt_state ?src* = *W_Reset*
by (*simp add*: *mid_reset_combined_config_gen_def*)

have *src_eq*: *?src* = $Config_M$ *W_Reset* (*mt_tape ?src*) (*mt_pos ?src*)
using *src_state* **by** (*cases ?src*) *simp*

— Storage head reads a non-*le* at *Suc m1* (never cell 0).
have *sym1_ne_le*: *?sym* (*Suc 0*) ≠ *Enc* (*le_tm M*)
proof (*cases Suc m1* ≤ *?L*)
case *True*
hence *m1_lt_L*: *m1* < *?L* **by** *simp*
have *ne*: *wrap_enc pack c w* ! *m1* ≠ *le_tm M*
by (*rule wrap_enc_ne_le*[*OF c_pos pack_ne_le m1_lt_L*])
have *read*: *?sym* (*Suc 0*) = *Enc* (*wrap_enc pack c w* ! *m1*)
using *True k2* **by** (*simp add*: *mid_reset_combined_config_gen_def split*: *nat.split*)
show *?thesis* **using** *read ne* **by** *simp*
next
case *False*
have *read*: *?sym* (*Suc 0*) = *Enc* (*bl_tm M*)
using *False k2* **by** (*simp add*: *mid_reset_combined_config_gen_def split*: *nat.split*)
show *?thesis* **using** *read bl_ne_le* **by** *simp*
qed

— Range typing and blank tail of the read function.
have *enc_in_Gamma*:
⋀*i*. *i* < *length* (*wrap_enc pack c w*) ⟹ *wrap_enc pack c w* ! *i* ∈ Γ*_tm M*
by (*rule wrap_enc_in_Gamma*[**where** *pack* = *pack*, *OF c_pos pack_in*])
have *wi0*: ⋀*i*. *Suc i* ≤ *length w* ⟹ *w* ! *i* ∈ Σ*u*
using *w_alpha* **by** (*meson Suc_le_eq nth_mem subsetD*)
have *rd_range*: *?sym* ∈ *UNIV* → *Raw* ‘ Σ*u* ∪ *Enc* ‘ Γ*_tm M*
using *bl_in le_in enc_in_Gamma wi0*
by (*auto simp*: *mid_reset_combined_config_gen_def split*: *nat.split*)

have *tail_bl*: ∀ *j*≥*k_tm M*. *?sym j* = *Enc* (*bl_tm M*)
by (*auto simp*: *mid_reset_combined_config_gen_def*)

— The generic-rewind tuple: read = write, storage head left, all else N.
let *?dir* = λt. *if t* = *Suc 0 then dir.L else dir.N*
have *tuple_in*: (*W_Reset*, *?sym*, *W_Reset*, *?sym*, *?dir*) ∈ *wrap_delta M pack c*

```
Σu
  proof −
    have (W_Reset, ?sym, W_Reset, ?sym, ?dir)
          ∈ wrap_rewind_loop_delta_gen W_Reset (Suc 0) (k_tm M) M Σu
      unfolding wrap_rewind_loop_delta_gen_def
      using sym1_ne_le tail_bl rd_range by blast
    thus ?thesis by (simp add: wrap_delta_def)
  qed

  have step_holds:
    (Config_M W_Reset (mt_tape ?src) (mt_pos ?src),
      Config_M W_Reset
        (λk. (mt_tape ?src k)((mt_pos ?src k) := ?sym k))
        (λk. go_dir (?dir k) (mt_pos ?src k)))
     ∈ mttm_step (wrap_delta M pack c Σu)
    by (rule mttm_step.step
          [where ts = mt_tape ?src and n = mt_pos ?src and a = ?sym and dir
= ?dir,
           OF tuple_in])

  — Target equality. Write = read, so tapes are preserved; only the storage head
moves left (Suc m1 to m1); the input head stays.
  have target_eq:
    Config_M W_Reset
      (λk. (mt_tape ?src k)((mt_pos ?src k) := ?sym k))
      (λk. go_dir (?dir k) (mt_pos ?src k))
    = mid_reset_combined_config_gen (k_tm M) M pack c w (length w + 1) m1
    by (auto simp: mid_reset_combined_config_gen_def fun_upd_triv fun_eq_iff
          split: nat.split)

  from step_holds target_eq src_eq show ?thesis by simp
qed
```

Iterated: s storage-rewind steps retract the storage head from $m1 + s$ to $m1$, the input head frozen throughout.

```
lemma mid_plant_rewind_iter:
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and w_alpha: set w ⊆ Σu
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
      and pack_in:
        ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ∈ Γ_tm M
      and pack_ne_le:
        ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ≠ le_tm M
    shows (mid_reset_combined_config_gen (k_tm M) M pack c w (length w +
1) (m1 + s),
           mid_reset_combined_config_gen (k_tm M) M pack c w (length w + 1)
m1)
```

```
        ∈ (mttm_step (wrap_delta M pack c Σu)) ^^ s
proof (induction s)
  case 0
  show ?case by simp
next
  case (Suc s)
  have step:
    (mid_reset_combined_config_gen (k_tm M) M pack c w (length w + 1) (Suc
(m1 + s)),
     mid_reset_combined_config_gen (k_tm M) M pack c w (length w + 1) (m1
+ s))
      ∈ mttm_step (wrap_delta M pack c Σu)
    apply (rule mid_plant_rewind_step[OF vM c_pos w_alpha bl_ne_le k2])
     apply (fact pack_in)
    apply (fact pack_ne_le)
    done
  from relpow_Suc_I2[OF step Suc.IH]
  show ?case by simp
qed
```

The storage rewind reaches the bottom *mid_reset_combined_config_gen (k_tm M) M pack c w (length w + 1) 0* (storage head at *le*, input head still frozen at *length w + 1*) from *post_encoder_config_gen* in exactly *Suc (length (wrap_enc pack c w))* rewind steps — the $\lceil n/c \rceil$ storage cells, the $\varepsilon{\cdot}n$ term. The input tape is *never* walked.

```
lemma plant_rewind_loop_relpow:
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and w_alpha: set w ⊆ Σu
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
      and pack_in:
        ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ∈ Γ_tm M
      and pack_ne_le:
        ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ≠ le_tm M
    shows (post_encoder_config_gen (k_tm M) M pack c w,
           mid_reset_combined_config_gen (k_tm M) M pack c w (length w + 1)
0)
          ∈ (mttm_step (wrap_delta M pack c Σu))
              ^^ (Suc (length (wrap_enc pack c w)))
proof −
  let ?L = length (wrap_enc pack c w)
  have iter:
    (mid_reset_combined_config_gen (k_tm M) M pack c w (length w + 1) (0 +
Suc ?L),
     mid_reset_combined_config_gen (k_tm M) M pack c w (length w + 1) 0)
      ∈ (mttm_step (wrap_delta M pack c Σu)) ^^ (Suc ?L)
    apply (rule mid_plant_rewind_iter[OF vM c_pos w_alpha bl_ne_le k2])
```

```
    apply (fact pack_in)
   apply (fact pack_ne_le)
   done
 show ?thesis
   using iter by (simp add: post_encoder_eq_mid_reset_combined_top)
qed
```

9.2 The floated engine configuration and the plant-done / dispatch steps

After the rewind bottoms out, the plant-done step writes a fresh *le* on the input tape (physical *0*) at its frozen head *length w + 1* and hands to *W_Disp*; the dispatch then lifts *W_Disp* → *W_Run* (*s_tm M*). Both leave heads fixed. The common tape function *plant_reset_tape* is the rewind bottom with the planted *le*: physical tape *0* carries *le* at cell *0*, the spent raw input at cells *1* … *length w*, the *planted le* at cell *length w + 1*, and blanks above.

```
definition plant_reset_tape ::
  ('q, 'b) mttm ⇒ ('a list ⇒ 'b) ⇒ nat ⇒ 'a list ⇒ nat ⇒ nat
   ⇒ ('a, 'b) wrap_alphabet
where
  plant_reset_tape M pack c w t n =
    (if t < k_tm M
     then (case t of
       0 ⇒ (if n = 0 then Enc (le_tm M)
            else if n ≤ length w then Raw (w ! (n − 1))
            else if n = length w + 1 then Enc (le_tm M)
            else Enc (bl_tm M))
     | Suc k ⇒ (if n = 0 then Enc (le_tm M)
                else if k = 0 ∧ n ≤ length (wrap_enc pack c w)
                  then Enc (wrap_enc pack c w ! (n − 1))
                else Enc (bl_tm M)))
     else Enc (bl_tm M))

definition plant_reset_pos :: 'a list ⇒ nat ⇒ nat where
  plant_reset_pos w t = (case t of 0 ⇒ length w + 1 | Suc _ ⇒ 0)

definition mid_plant_disp_config ::
  ('q, 'b) mttm ⇒ ('a list ⇒ 'b) ⇒ nat ⇒ 'a list
   ⇒ (('a, 'b) wrap_alphabet, ('q, 'a, 'b) wrap_state) mt_config
where
  mid_plant_disp_config M pack c w =
    Config_M W_Disp (plant_reset_tape M pack c w) (plant_reset_pos w)

definition post_plant_dispatch_config ::
  ('q, 'b) mttm ⇒ ('a list ⇒ 'b) ⇒ nat ⇒ 'a list
   ⇒ (('a, 'b) wrap_alphabet, ('q, 'a, 'b) wrap_state) mt_config
where
  post_plant_dispatch_config M pack c w =
```

Config$_M$ (*W_Run* (*s_tm M*)) (*plant_reset_tape M pack c w*) (*plant_reset_pos w*)

The plant-done step: from the rewind bottom (storage head at *le*) one *wrap_plant_done_delta* transition plants *le* on the input tape at its frozen head and lifts *W_Reset* → *W_Disp*.

lemma *plant_done_step*:
assumes *vM*: *valid_mttm M*
and *k2*: *2* ≤ *k_tm M*
shows (*mid_reset_combined_config_gen* (*k_tm M*) *M pack c w* (*length w* + *1*) *0*,
mid_plant_disp_config M pack c w)
∈ *mttm_step* (*wrap_delta M pack c* Σ*u*)
proof −
let *?src* = *mid_reset_combined_config_gen* (*k_tm M*) *M pack c w* (*length w* + *1*) *0*
let *?sym* = λ*k*. *mt_tape ?src k* (*mt_pos ?src k*)
let *?a* = λ*t*. *if t* = *0 then Enc* (*le_tm M*) *else ?sym t*
have *bl_in*: *bl_tm M* ∈ Γ*_tm M* **by** (*rule valid_mttm_blank_in_Gamma*[*OF vM*])
have *le_in*: *le_tm M* ∈ Γ*_tm M* **by** (*rule valid_mttm_LE_in_Gamma*[*OF vM*])

have *src_state*: *mt_state ?src* = *W_Reset*
by (*simp add*: *mid_reset_combined_config_gen_def*)
have *src_eq*: *?src* = *Config*$_M$ *W_Reset* (*mt_tape ?src*) (*mt_pos ?src*)
using *src_state* **by** (*cases ?src*) *simp*

have *sym1_le*: *?sym* (*Suc 0*) = *Enc* (*le_tm M*)
using *k2* **by** (*simp add*: *mid_reset_combined_config_gen_def split*: *nat.split*)
have *tail_bl*: ∀ *j*≥*k_tm M*. *?sym j* = *Enc* (*bl_tm M*)
by (*auto simp*: *mid_reset_combined_config_gen_def*)
have *rd_range*: *?sym* ∈ *UNIV* → *Raw* ‘ Σ*u* ∪ *Enc* ‘ Γ*_tm M*
using *bl_in le_in*
by (*auto simp*: *mid_reset_combined_config_gen_def split*: *nat.split*)

have *tuple_in*: (*W_Reset*, *?sym*, *W_Disp*, *?a*, λ*_*. *dir.N*)
∈ *wrap_delta M pack c* Σ*u*
proof −
have (*W_Reset*, *?sym*, *W_Disp*, *?a*, λ*_*. *dir.N*)
∈ *wrap_plant_done_delta M* Σ*u*
unfolding *wrap_plant_done_delta_def*
using *sym1_le tail_bl rd_range* **by** *blast*
thus *?thesis* **by** (*simp add*: *wrap_delta_def*)
qed

have *step_holds*:
(*Config*$_M$ *W_Reset* (*mt_tape ?src*) (*mt_pos ?src*),
Config$_M$ *W_Disp*

```
      (λk. (mt_tape ?src k)((mt_pos ?src k) := ?a k))
      (λk. go_dir dir.N (mt_pos ?src k)))
   ∈ mttm_step (wrap_delta M pack c Σu)
  by (rule mttm_step.step
       [where ts = mt_tape ?src and n = mt_pos ?src and a = ?a
          and dir = λ_. dir.N, OF tuple_in])

 have target_eq:
  Config_M W_Disp
    (λk. (mt_tape ?src k)((mt_pos ?src k) := ?a k))
    (λk. go_dir dir.N (mt_pos ?src k))
  = mid_plant_disp_config M pack c w
  using k2
       by (auto simp: mid_plant_disp_config_def plant_reset_tape_def
plant_reset_pos_def
            mid_reset_combined_config_gen_def fun_eq_iff
        split: nat.split)

 from step_holds target_eq src_eq show ?thesis by simp
qed
```

The dispatch step: one *wrap_disp_delta_gen* transition lifts *W_Disp* → *W_Run* (*s_tm M*), tapes and heads untouched — reaching the floated engine configuration *post_plant_dispatch_config*.

```
lemma plant_disp_step:
 assumes vM: valid_mttm M
    and k2: 2 ≤ k_tm M
    shows (mid_plant_disp_config M pack c w, post_plant_dispatch_config M
pack c w)
       ∈ mttm_step (wrap_delta M pack c Σu)
proof −
 let ?src = mid_plant_disp_config M pack c w
 let ?sym = λk. mt_tape ?src k (mt_pos ?src k)
 have bl_in: bl_tm M ∈ Γ_tm M by (rule valid_mttm_blank_in_Gamma[OF
vM])
 have le_in: le_tm M ∈ Γ_tm M by (rule valid_mttm_LE_in_Gamma[OF
vM])

 have src_state: mt_state ?src = W_Disp
  by (simp add: mid_plant_disp_config_def)
 have src_eq: ?src = Config_M W_Disp (mt_tape ?src) (mt_pos ?src)
  using src_state by (cases ?src) simp

 have tail_bl: ∀ j≥k_tm M. ?sym j = Enc (bl_tm M)
  by (auto simp: mid_plant_disp_config_def plant_reset_tape_def)
 have rd_range: ?sym ∈ UNIV → Raw ‘ Σu ∪ Enc ‘ Γ_tm M
  using bl_in le_in
       by (auto simp: mid_plant_disp_config_def plant_reset_tape_def
plant_reset_pos_def
```

```
        split: nat.split)

  have tuple_in:
    (W_Disp, ?sym, W_Run (s_tm M), ?sym, λ_. dir.N) ∈ wrap_delta M pack
c Σu
  proof −
    have (W_Disp, ?sym, W_Run (s_tm M), ?sym, λ_. dir.N)
          ∈ wrap_disp_delta_gen (k_tm M) M Σu
      unfolding wrap_disp_delta_gen_def
      using tail_bl rd_range by blast
    thus ?thesis by (simp add: wrap_delta_def)
  qed

  have step_holds:
    (Config_M W_Disp (mt_tape ?src) (mt_pos ?src),
      Config_M (W_Run (s_tm M))
        (λk. (mt_tape ?src k)((mt_pos ?src k) := ?sym k))
        (λk. go_dir dir.N (mt_pos ?src k)))
     ∈ mttm_step (wrap_delta M pack c Σu)
    by (rule mttm_step.step
          [where ts = mt_tape ?src and n = mt_pos ?src
                 and a = ?sym and dir = λ_. dir.N, OF tuple_in])

  have target_eq:
    Config_M (W_Run (s_tm M))
      (λk. (mt_tape ?src k)((mt_pos ?src k) := ?sym k))
      (λk. go_dir dir.N (mt_pos ?src k))
    = post_plant_dispatch_config M pack c w
     by (simp add: post_plant_dispatch_config_def mid_plant_disp_config_def
fun_eq_iff)

  from step_holds target_eq src_eq show ?thesis by simp
qed
```

9.3 The reset reaches the floated engine configuration

From *post_encoder_config_gen* the storage rewind, plant-done, and dispatch together reach the floated engine configuration *post_plant_dispatch_config* in *Suc* (*length* (*wrap_enc pack c w*)) + *2* wrap-steps: *Suc* (*length* (*wrap_enc pack c w*)) storage-rewind steps, the plant-done step, and the dispatch step. Only the storage tape is walked; the input tape is left in place with a planted *le*.

```
lemma plant_reset_dispatch_relpow:
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and w_alpha: set w ⊆ Σu
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
```

```
    and pack_in:
      ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ∈ Γ_tm M
    and pack_ne_le:
      ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ≠ le_tm M
  shows (post_encoder_config_gen (k_tm M) M pack c w,
         post_plant_dispatch_config M pack c w)
        ∈ (mttm_step (wrap_delta M pack c Σu))
            ^^ (Suc (length (wrap_enc pack c w)) + 2)
proof −
  let ?R = mttm_step (wrap_delta M pack c Σu)
  let ?L = length (wrap_enc pack c w)
  have loop: (post_encoder_config_gen (k_tm M) M pack c w,
              mid_reset_combined_config_gen (k_tm M) M pack c w (length w +
1) 0)
               ∈ ?R ^^ (Suc ?L)
    apply (rule plant_rewind_loop_relpow[OF vM c_pos w_alpha bl_ne_le k2])
     apply (fact pack_in)
    apply (fact pack_ne_le)
    done
   have done_step: (mid_reset_combined_config_gen (k_tm M) M pack c w
(length w + 1) 0,
                    mid_plant_disp_config M pack c w) ∈ ?R
    by (rule plant_done_step[OF vM k2])
  have disp: (mid_plant_disp_config M pack c w, post_plant_dispatch_config M
pack c w)
               ∈ ?R
    by (rule plant_disp_step[OF vM k2])

  from relpow_Suc_I[OF loop done_step]
  have (post_encoder_config_gen (k_tm M) M pack c w, mid_plant_disp_config
M pack c w)
         ∈ ?R ^^ Suc (Suc ?L) .
  from relpow_Suc_I[OF this disp]
  have (post_encoder_config_gen (k_tm M) M pack c w,
        post_plant_dispatch_config M pack c w) ∈ ?R ^^ Suc (Suc (Suc ?L)) .
  thus ?thesis by (simp add: numeral_2_eq_2)
qed

lemma plant_reset_dispatch_rtrancl:
  assumes vM: valid_mttm M
    and c_pos: 0 < c
    and w_alpha: set w ⊆ Σu
    and bl_ne_le: bl_tm M ≠ le_tm M
    and k2: 2 ≤ k_tm M
    and pack_in:
      ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ∈ Γ_tm M
    and pack_ne_le:
      ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ≠ le_tm M
  shows (post_encoder_config_gen (k_tm M) M pack c w,
```

```
          post_plant_dispatch_config M pack c w)
         ∈ (mttm_step (wrap_delta M pack c Σu))*
proof −
  have (post_encoder_config_gen (k_tm M) M pack c w,
        post_plant_dispatch_config M pack c w)
         ∈ (mttm_step (wrap_delta M pack c Σu))
             ^^ (Suc (length (wrap_enc pack c w)) + 2)
    apply (rule plant_reset_dispatch_relpow[OF vM c_pos w_alpha bl_ne_le k2])
     apply (fact pack_in)
    apply (fact pack_ne_le)
    done
  thus ?thesis by (rule relpow_imp_rtrancl)
qed
```

9.4 The floated config is a shift of the proper τ-lift

The floated engine configuration *post_plant_dispatch_config* is exactly a *shift_rel* of the τ-lift of M's initial configuration on the encoded input, shifting *only* physical tape *0* (M's tape *1*) right by *length w + 1*: the planted *le* sits at the shifted origin, the spent raw input is the discarded prefix below it, and blanks lie above. Every other physical tape (the storage tape *1* = M's tape *0*, carrying the encoded input, and the fresh work tapes *2* … *k−1*) is unshifted (*d p = 0*), so it matches the lift verbatim. This is the hinge: the origin-float lemmas of *Multitape_TM_Substrate.Multitape_Origin_Float* translate runs between the two configs, and only physical tape *0* needs *le* at its base origin — which the lift supplies (M's blank tape *1* has *le* at cell *0*).

```
lemma post_plant_dispatch_shift_lift:
  assumes k2: 2 ≤ k_tm M
  shows shift_rel (λp. if p = 0 then length w + 1 else 0)
          (lift_M_config M (init_config_mttm M (wrap_enc pack c w)))
          (post_plant_dispatch_config M pack c w)
proof (cases M)
  case (MTTM Q Σi Γ bl le δ s t r kM)
  show ?thesis
    using k2
    unfolding shift_rel_def MTTM
                              post_plant_dispatch_config_def  plant_reset_tape_def
plant_reset_pos_def
              lift_M_config_def
    by (auto simp: wrap_tau_def fun_eq_iff split: nat.split if_splits)
qed

end
theory Wrap_Run
  imports Wrap_Reset
begin
```

10 Faithful (k-tape) plant-*le* wrap: the transposed run

The run phase of the plant wrap dispatches to *wrap_run_delta*, the shared transposed-run family, on *W_Run* states. The correspondence lemmas below — an M-step lifts to a wrap-step and back — rest on two facts about *wrap_delta*: it *contains wrap_run_delta*, and its only *W_Run*-sourced transitions are in *wrap_run_delta* (the other eight families source from *W_Init* / *W_Buf* / *W_Reset* / *W_Disp*). This is the run correspondence for the faithful k-tape wrap.

10.1 Forward simulation: an M-step lifts to a wrap-step

lemma *run_step_forward*:
  assumes *vM*: *valid_mttm M*
    and *k2*: *2* ≤ *k_tm M*
    and *val_cM*: *valid_config_mttm M cM*
    and *step_M*: (*cM*, *cM′*) ∈ *mttm_step* (*delta_tm M*)
  shows (*lift_M_config M cM*, *lift_M_config M cM′*)
      ∈ *mttm_step* (*wrap_delta M pack c* Σ*u*)
proof −
  obtain *Q* Σ*i* Γ *bl le* δ*M sM tM rM kM* **where** *M_eq*:
    M = *MTTM Q* Σ*i* Γ *bl le* δ*M sM tM rM kM*
    by (*cases M*)
  obtain *q ts n* **where** *cM_eq*: *cM* = *Config*$_M$ *q ts n*
    by (*cases cM*)
  obtain *q′ a d* **where** *cM′_eq*:
    cM′ = *Config*$_M$ *q′* (λ*k*. (*ts k*)(*n k* := *a k*))
                (λ*k*. *go_dir* (*d k*) (*n k*))
    and *tr*: (*q*, (λ*k*. *ts k* (*n k*)), *q′*, *a*, *d*) ∈ *delta_tm M*
    using *step_M cM_eq* **by** (*auto elim*: *mttm_step.cases*)

  have *bl_in*: *bl_tm M* ∈ Γ*_tm M* **by** (*rule valid_mttm_blank_in_Gamma*[*OF vM*])
  have *le_in*: *le_tm M* ∈ Γ*_tm M* **by** (*rule valid_mttm_LE_in_Gamma*[*OF vM*])

  from *val_cM cM_eq M_eq* **have** *ts_Gamma*: ⋀*k m*. *ts k m* ∈ Γ*_tm M*
    by (*auto simp*: *M_eq*)
  have *ts_tail*: ∀ *i*≥*k_tm M*. ∀ *p*. *ts i p* = *bl_tm M*
    using *val_cM* **unfolding** *cM_eq* **by** (*cases M*) *auto*
  have *a_tail*: ∀ *j*≥*k_tm M*. *a j* = *bl_tm M*
  proof (*intro allI impI*)
    fix *j* **assume** *k_tm M* ≤ *j*
    thus *a j* = *bl_tm M* **using** *valid_mttm_delta_support*[*OF vM tr*] **by** *blast*
  qed

  let *?src* = *lift_M_config M cM*

```
  let ?dst = lift_M_config M cM'
  let ?lift_ts =
    λp m. if p < k_tm M then Enc (ts (wrap_tau p) m) else Enc (bl_tm M)
  let ?lift_n = λp. n (wrap_tau p)
  let ?sym = λp. ?lift_ts p (?lift_n p)
  let ?σ = λk. ts k (n k)
  let ?wrap_a = λp. Enc (a (wrap_tau p))
  let ?wrap_dir = λp. d (wrap_tau p)

  have src_unfold: ?src = Config_M (W_Run q) ?lift_ts ?lift_n
    unfolding lift_M_config_def cM_eq by (simp add: fun_eq_iff)

  have sym_eq: ∀ p. ?sym p = Enc (?σ (wrap_tau p))
  proof
    fix p
    show ?sym p = Enc (?σ (wrap_tau p))
    proof (cases p < k_tm M)
      case True
      thus ?thesis by simp
    next
      case False
      hence jge: k_tm M ≤ p by simp
      with k2 have 2 ≤ p by simp
      hence wtp: wrap_tau p = p by (rule wrap_tau_ge2_id)
      have ?sym p = Enc (bl_tm M) using False by simp
      moreover have ?σ (wrap_tau p) = bl_tm M
        using wtp jge ts_tail by simp
      ultimately show ?thesis by simp
    qed
  qed

  have sym_range: ?sym ∈ UNIV → Raw ‘ Σu ∪ Enc ‘ Γ_tm M
    using bl_in ts_Gamma by auto

  have tuple_in_run:
    (W_Run q, ?sym, W_Run q', ?wrap_a, ?wrap_dir) ∈ wrap_run_delta M Σu
    unfolding wrap_run_delta_def
    using tr sym_eq sym_range by blast
  hence tuple_in:
    (W_Run q, ?sym, W_Run q', ?wrap_a, ?wrap_dir) ∈ wrap_delta M pack c
Σu
    by (simp add: wrap_delta_def)

  have step_holds:
    (Config_M (W_Run q) ?lift_ts ?lift_n,
      Config_M (W_Run q')
        (λp. (?lift_ts p)(?lift_n p := ?wrap_a p))
        (λp. go_dir (?wrap_dir p) (?lift_n p)))
     ∈ mttm_step (wrap_delta M pack c Σu)
```

by (*rule mttm_step.step*
[**where** *ts* = *?lift_ts* **and** *n* = *?lift_n*
and *a* = *?wrap_a* **and** *dir* = *?wrap_dir*, *OF tuple_in*])

have *dst_unfold*:
?dst = $Config_M$ (*W_Run q'*)
(λ*p*. (*?lift_ts p*)(*?lift_n p* := *?wrap_a p*))
(λ*p*. *go_dir* (*?wrap_dir p*) (*?lift_n p*))
unfolding *lift_M_config_def cM'_eq*
using *ts_tail a_tail k2*
by (*auto simp*: *fun_eq_iff wrap_tau_ge2_id*)

show *?thesis*
unfolding *src_unfold dst_unfold* **by** (*rule step_holds*)
qed

10.2 State-routing exclusion

Only *wrap_run_delta* contains a tuple whose source state is *W_Run q*; the other eight families of *wrap_delta* source from *W_Init*, *W_Buf ws*, *W_Reset*, or *W_Disp* (the storage rewind loop and the plant-done step both from *W_Reset*).

lemma *wrap_delta_W_Run_source_only_run*:
fixes *M* :: ('*q*, '*b*) *mttm*
and *sym* :: *nat* ⇒ ('*a*, '*b*) *wrap_alphabet*
and *qs'* :: ('*q*, '*a*, '*b*) *wrap_state*
and *a'* :: *nat* ⇒ ('*a*, '*b*) *wrap_alphabet*
and *dir* :: *nat* ⇒ *dir*
assumes (*W_Run q*, *sym*, *qs'*, *a'*, *dir*) ∈ *wrap_delta M pack c* Σ*u*
shows (*W_Run q*, *sym*, *qs'*, *a'*, *dir*) ∈ *wrap_run_delta M* Σ*u*
using *assms*
unfolding *wrap_delta_def wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def*
wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
wrap_buf_nonempty_end_delta_gen_def
wrap_rewind_loop_delta_gen_def
wrap_plant_done_delta_def wrap_disp_delta_gen_def
by *auto*

10.3 Reverse simulation: a wrap-step projects to an M-step

lemma *run_step_reverse*:
assumes *vM*: *valid_mttm M*
and *k2*: *2* ≤ *k_tm M*
and *val_cM*: *valid_config_mttm M cM*
and *step_wrap*:
(*lift_M_config M cM*, *C'*) ∈ *mttm_step* (*wrap_delta M pack c* Σ*u*)
shows ∃ *cM'*. *C'* = *lift_M_config M cM'*
∧ (*cM*, *cM'*) ∈ *mttm_step* (*delta_tm M*)

```
proof −
  obtain q ts n where cM_eq: cM = Config_M q ts n
    by (cases cM)

  let ?lift_ts =
    λp m. if p < k_tm M then Enc (ts (wrap_tau p) m) else Enc (bl_tm M)
  let ?lift_n = λp. n (wrap_tau p)

  have ts_tail: ∀ i≥k_tm M. ∀ p. ts i p = bl_tm M
    using val_cM unfolding cM_eq by (cases M) auto

  have src_eq:
    lift_M_config M cM = Config_M (W_Run q) ?lift_ts ?lift_n
    unfolding lift_M_config_def cM_eq by (simp add: fun_eq_iff)

  note step_wrap' = step_wrap[unfolded src_eq]
  from step_wrap' obtain qs' a' dir' where
    C'_eq:
      C' = Config_M qs'
            (λx. (?lift_ts x)(?lift_n x := a' x))
            (λx. go_dir (dir' x) (?lift_n x))
    and tuple_in:
      (W_Run q, λx. ?lift_ts x (?lift_n x), qs', a', dir')
        ∈ wrap_delta M pack c Σu
    by (auto elim: mttm_step.cases)

  have tuple_run:
    (W_Run q, λx. ?lift_ts x (?lift_n x), qs', a', dir')
      ∈ wrap_run_delta M Σu
    by (rule wrap_delta_W_Run_source_only_run[OF tuple_in])

  from tuple_run[unfolded wrap_run_delta_def mem_Collect_eq]
  obtain qa σ q'_in σ'_M d sym_pat where
    raw: (W_Run q, λx. ?lift_ts x (?lift_n x), qs', a', dir')
          = (W_Run qa, sym_pat, W_Run q'_in,
             (λp. Enc (σ'_M (wrap_tau p))), (λp. d (wrap_tau p)))
    and tr_M_pre: (qa, σ, q'_in, σ'_M, d) ∈ delta_tm M
    and sym_pat_eq: ∀ p. sym_pat p = Enc (σ (wrap_tau p))
    and sym_range: sym_pat ∈ UNIV → Raw ‘ Σu ∪ Enc ‘ Γ_tm M
    by blast

  from raw have q_eq: qa = q by simp
  from raw have read_eq: (λx. ?lift_ts x (?lift_n x)) = sym_pat by simp
  from raw have qs'_eq: qs' = W_Run q'_in by simp
  from raw have a'_eq: a' = (λp. Enc (σ'_M (wrap_tau p))) by simp
  from raw have dir'_eq: dir' = (λp. d (wrap_tau p)) by simp

  have sigma_eq: σ = (λk. ts k (n k))
  proof (rule ext)
```

```
    fix k
    have rk: ?lift_ts (wrap_tau k) (?lift_n (wrap_tau k)) = sym_pat (wrap_tau
k)
      using fun_cong[OF read_eq, of wrap_tau k] by simp
    have sk: sym_pat (wrap_tau k) = Enc (σ k)
      using sym_pat_eq by simp
    show σ k = ts k (n k)
    proof (cases k < k_tm M)
      case True
      have wk_lt: wrap_tau k < k_tm M
      proof (cases k < 2)
        case True
        hence wrap_tau k < 2 using wrap_tau_lt2 by simp
        thus ?thesis using k2 by simp
      next
        case False
        hence 2 ≤ k by simp
        hence wrap_tau k = k by (rule wrap_tau_ge2_id)
        thus ?thesis using ‹k < k_tm M› by simp
      qed
      have Enc (σ k) = sym_pat (wrap_tau k) using sk by (rule sym)
      also have ... = ?lift_ts (wrap_tau k) (?lift_n (wrap_tau k))
        using rk by (rule sym)
      also have ... = Enc (ts k (n k)) using wk_lt by simp
      finally show ?thesis by simp
    next
      case False
      hence kge: k_tm M ≤ k by simp
      with k2 have 2 ≤ k by simp
      hence wk: wrap_tau k = k by (rule wrap_tau_ge2_id)
      have Enc (σ k) = sym_pat (wrap_tau k) using sk by (rule sym)
      also have ... = ?lift_ts (wrap_tau k) (?lift_n (wrap_tau k))
        using rk by (rule sym)
      also have ... = Enc (bl_tm M) using wk kge by simp
      finally have σ k = bl_tm M by simp
      moreover have ts k (n k) = bl_tm M using ts_tail kge by simp
      ultimately show ?thesis by simp
    qed
  qed

  have tr_M: (q, λk. ts k (n k), q'_in, σ'_M, d) ∈ delta_tm M
    using tr_M_pre q_eq sigma_eq by simp

  have sigma'_tail: ∀j≥k_tm M. σ'_M j = bl_tm M
  proof (intro allI impI)
    fix j assume k_tm M ≤ j
    thus σ'_M j = bl_tm M using valid_mttm_delta_support[OF vM tr_M] by
blast
  qed
```

```
  let ?cM' = Config_M q'_in (λk. (ts k)(n k := σ'_M k))
                            (λk. go_dir (d k) (n k))
  have M_step: (cM, ?cM') ∈ mttm_step (delta_tm M)
    unfolding cM_eq
    by (rule mttm_step.step
          [where ts = ts and n = n and a = σ'_M and dir = d, OF tr_M])

  have C'_lift: C' = lift_M_config M ?cM'
    unfolding C'_eq qs'_eq a'_eq dir'_eq lift_M_config_def
    using ts_tail sigma'_tail k2
    by (auto simp: fun_eq_iff wrap_tau_ge2_id)

  show ?thesis using C'_lift M_step by blast
qed
```

10.4 Iterated simulation

An entire M-trace lifts to a wrap-trace of the same length, and vice versa — iterating the single-step correspondence, threading *valid_config_mttm* through *valid_step_mttm*.

```
lemma run_steps_forward:
  assumes vM: valid_mttm M
      and k2: 2 ≤ k_tm M
      and val_cM: valid_config_mttm M cM
      and steps_M: (cM, cM') ∈ (mttm_step (delta_tm M))*
    shows (lift_M_config M cM, lift_M_config M cM')
          ∈ (mttm_step (wrap_delta M pack c Σu))*
proof −
  let ?R_W = mttm_step (wrap_delta M pack c Σu)
  have helper:
    valid_config_mttm M cM' ∧
    (lift_M_config M cM, lift_M_config M cM') ∈ ?R_W*
    using steps_M
  proof induction
    case base
    show ?case using val_cM by simp
  next
    case (step y z)
    from step.IH have val_y: valid_config_mttm M y
                  and ih_lift: (lift_M_config M cM, lift_M_config M y) ∈ ?R_W*
      by auto
    have val_z: valid_config_mttm M z
      by (rule valid_step_mttm[OF vM step.hyps(2) val_y])
    have step_lift: (lift_M_config M y, lift_M_config M z) ∈ ?R_W
      by (rule run_step_forward[OF vM k2 val_y step.hyps(2)])
    from ih_lift step_lift
    have lift_z: (lift_M_config M cM, lift_M_config M z) ∈ ?R_W*
      by (rule rtrancl_into_rtrancl)
```

```
    from val_z lift_z show ?case by simp
  qed
  thus ?thesis by simp
qed

lemma run_steps_forward_relpow:
  assumes vM: valid_mttm M
      and k2: 2 ≤ k_tm M
      and val_cM: valid_config_mttm M cM
      and steps_M: (cM, cM') ∈ (mttm_step (delta_tm M)) ^^ n
    shows (lift_M_config M cM, lift_M_config M cM')
        ∈ (mttm_step (wrap_delta M pack c Σu)) ^^ n
proof −
  let ?R_M = mttm_step (delta_tm M)
  let ?R_W = mttm_step (wrap_delta M pack c Σu)
  have helper:
    ⋀cM'. (cM, cM') ∈ ?R_M ^^ n ⟹
          valid_config_mttm M cM' ∧
          (lift_M_config M cM, lift_M_config M cM') ∈ ?R_W ^^ n
  proof (induction n)
    case 0
    fix cM'
    assume (cM, cM') ∈ ?R_M ^^ 0
    hence cM' = cM by simp
    thus valid_config_mttm M cM' ∧
         (lift_M_config M cM, lift_M_config M cM') ∈ ?R_W ^^ 0
      using val_cM by simp
  next
    case (Suc n)
    fix cM'
    assume chain: (cM, cM') ∈ ?R_M ^^ Suc n
    from chain obtain y where
       prefix: (cM, y) ∈ ?R_M ^^ n and last: (y, cM') ∈ ?R_M
      by (auto elim: relpow_Suc_E)
    from Suc.IH[OF prefix] have
       val_y: valid_config_mttm M y
      and ih_lift: (lift_M_config M cM, lift_M_config M y) ∈ ?R_W ^^ n
      by auto
    have val_cM': valid_config_mttm M cM'
      by (rule valid_step_mttm[OF vM last val_y])
    have step_lift: (lift_M_config M y, lift_M_config M cM') ∈ ?R_W
      by (rule run_step_forward[OF vM k2 val_y last])
    from ih_lift step_lift
    have suc_chain:
      (lift_M_config M cM, lift_M_config M cM') ∈ ?R_W ^^ Suc n
      by (rule relpow_Suc_I)
    from val_cM' suc_chain show
      valid_config_mttm M cM' ∧
      (lift_M_config M cM, lift_M_config M cM') ∈ ?R_W ^^ Suc n
```

```
    by simp
  qed
  show ?thesis using helper[OF steps_M] by simp
qed

lemma run_steps_reverse:
  assumes vM: valid_mttm M
      and k2: 2 ≤ k_tm M
      and val_cM: valid_config_mttm M cM
      and steps_wrap:
        (lift_M_config M cM, C') ∈ (mttm_step (wrap_delta M pack c Σu))*
    shows ∃ cM'. C' = lift_M_config M cM'
            ∧ (cM, cM') ∈ (mttm_step (delta_tm M))*
            ∧ valid_config_mttm M cM'
  using steps_wrap
proof induction
  case base
  show ?case using val_cM by blast
next
  case (step y z)
  from step.IH obtain cM_y where
      y_lift: y = lift_M_config M cM_y
    and y_M: (cM, cM_y) ∈ (mttm_step (delta_tm M))*
    and val_y: valid_config_mttm M cM_y
    by blast
  from step.hyps(2) y_lift have
    (lift_M_config M cM_y, z) ∈ mttm_step (wrap_delta M pack c Σu)
    by simp
  from run_step_reverse[OF vM k2 val_y this]
  obtain cM_z where
      z_lift: z = lift_M_config M cM_z
    and step_M: (cM_y, cM_z) ∈ mttm_step (delta_tm M)
    by blast
  have val_z: valid_config_mttm M cM_z
    by (rule valid_step_mttm[OF vM step_M val_y])
  from y_M step_M
  have steps_z: (cM, cM_z) ∈ (mttm_step (delta_tm M))*
    by (rule rtrancl_into_rtrancl)
  from z_lift steps_z val_z show ?case by blast
qed

end
theory Wrap_Encoder
  imports Wrap_Run
begin
```

11 Faithful (k-tape) plant-*le* wrap: encoder-phase threading

The encoder phase of *encoding_wrap* threads the six encoder families through the boundary-parameterised configurations (*post_encoder_config_gen*, *mid_encoder_config_gen*, *mid_buf_extending_config_gen*), reaching *post_encoder_config_gen* in state *W_Reset*. The encoder never touches the reset seam (it stops *before* the reset fires), and the configurations are reused, not redefined. These step-membership lemmas compose with *plant_reset_dispatch_rtrancl* at *post_encoder_config_gen*.

11.1 Helper (1a): the initial encoder step

One wrap-step from the initial configuration reaches *mid_encoder_config_gen* (*k_tm M*) *M pack c w 0*, via the unique *wrap_init_delta_gen* transition.

```
lemma init_to_mid_zero:
  assumes valid_mttm M
      and 0 < c
  shows (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
         mid_encoder_config_gen (k_tm M) M pack c w 0)
        ∈ mttm_step (wrap_delta M pack c Σu)
proof −
  let ?init = init_config_mttm (encoding_wrap M pack c Σu) (map Raw w)

  have init_state: mt_state ?init = W_Init
    by simp
  have init_pos: mt_pos ?init = (λ_. 0)
    by simp
  have init_tape:
    mt_tape ?init
      = (λk n. if k < k_tm M
            then (if n = 0 then Enc (le_tm M)
                  else if k = 0 ∧ n ≤ length (map Raw w)
                       then (map Raw w) ! (n − 1)
                       else Enc (bl_tm M))
            else Enc (bl_tm M))
    by (simp add: fun_eq_iff)

  have init_eq: ?init = Config_M W_Init (mt_tape ?init) (mt_pos ?init)
    using init_state init_pos init_tape
    by (cases ?init) simp

  have read_at_init:
    (λk. mt_tape ?init k (mt_pos ?init k))
      = (λt. if t < k_tm M then Enc (le_tm M) else Enc (bl_tm M))
```

```
  by (simp add: init_pos init_tape fun_eq_iff)

 have tuple_in:
  (W_Init, λk. mt_tape ?init k (mt_pos ?init k), W_Buf [],
   λt. if t < k_tm M then Enc (le_tm M) else Enc (bl_tm M),
   λt. case t of 0 ⇒ dir.R | Suc k ⇒ if k = 0 then dir.R else dir.N)
  ∈ wrap_delta M pack c Σu
  using read_at_init
  by (simp add: wrap_delta_def wrap_init_delta_gen_def)

 have step_holds:
  (Config_M W_Init (mt_tape ?init) (mt_pos ?init),
    Config_M (W_Buf [])
      (λk. (mt_tape ?init k)((mt_pos ?init k) :=
            (if k < k_tm M then Enc (le_tm M) else Enc (bl_tm M))))
      (λk. go_dir
           ((λt. case t of 0 ⇒ dir.R
                         | Suc k' ⇒ if k' = 0 then dir.R else dir.N) k)
           (mt_pos ?init k)))
   ∈ mttm_step (wrap_delta M pack c Σu)
  by (rule mttm_step.step
       [where ts = mt_tape ?init and n = mt_pos ?init
          and a = λt. if t < k_tm M then Enc (le_tm M) else Enc (bl_tm M)
           and dir = λt. case t of 0 ⇒ dir.R
                                 | Suc k' ⇒ if k' = 0 then dir.R else dir.N,
        OF tuple_in])

 have target_eq:
   Config_M (W_Buf [])
     (λk. (mt_tape ?init k)((mt_pos ?init k) :=
           (if k < k_tm M then Enc (le_tm M) else Enc (bl_tm M))))
     (λk. go_dir
          ((λt. case t of 0 ⇒ dir.R
                        | Suc k' ⇒ if k' = 0 then dir.R else dir.N) k)
          (mt_pos ?init k))
   = mid_encoder_config_gen (k_tm M) M pack c w 0
  unfolding mid_encoder_config_gen_def
  by (auto simp: init_pos init_tape fun_eq_iff
           split: nat.split if_splits)

 from step_holds target_eq init_eq
 show ?thesis by simp
qed
```

11.2 Helper (1b)(i): one buffer-extend step

A single *wrap_buf_extend_delta_gen* step within cycle i, advancing the buffer fill level from j to *Suc j*. The buffer-extend step only grows the buffer and advances *W_User* (physical tape *0*), touching no other tape, so it is

boundary-insensitive.

```
lemma mid_buf_extend_step:
  assumes valid_mttm M
      and 0 < c
      and set w ⊆ Σu
      and Suc j < c
      and i * c + j < length w
  shows (mid_buf_extending_config_gen (k_tm M) M pack c w i j,
         mid_buf_extending_config_gen (k_tm M) M pack c w i (Suc j))
        ∈ mttm_step (wrap_delta M pack c Σu)
proof −
  let ?src = mid_buf_extending_config_gen (k_tm M) M pack c w i j
  let ?ws = take j (drop (i * c) w)
  let ?a = w ! (i * c + j)

  — List-arithmetic facts about the buffer prefix.
  have j_lt_drop: j < length (drop (i * c) w)
    using ‹i * c + j < length w› by simp
  have len_ws: length ?ws = j
    using j_lt_drop by simp
  have set_ws: set ?ws ⊆ Σu
    using ‹set w ⊆ Σu›
    by (meson dual_order.trans set_drop_subset set_take_subset)
  have take_app: ?ws @ [?a] = take (Suc j) (drop (i * c) w)
    using j_lt_drop by (simp add: take_Suc_conv_app_nth)
  have a_in_Sigmau: ?a ∈ Σu
    using ‹set w ⊆ Σu› ‹i * c + j < length w›
    by (meson nth_mem subsetD)

  — Substrate facts about M's distinguished symbols.
  have bl_in: bl_tm M ∈ Γ_tm M
    by (rule valid_mttm_blank_in_Gamma[OF ‹valid_mttm M›])
  have le_in: le_tm M ∈ Γ_tm M
    by (rule valid_mttm_LE_in_Gamma[OF ‹valid_mttm M›])

  — Selector facts for the source config.
  have src_state: mt_state ?src = W_Buf ?ws
    by (simp add: mid_buf_extending_config_gen_def)
  have src_pos:
    mt_pos ?src
      = (λt. case t of 0 ⇒ 1 + i * c + j
                     | Suc k ⇒ (if k = 0 then 1 + i else 0))
    by (simp add: mid_buf_extending_config_gen_def fun_eq_iff)
  have src_tape:
    mt_tape ?src
      = (λt n. if t < k_tm M
               then (case t of
                 0 ⇒ (if n = 0 then Enc (le_tm M)
                          else if n ≤ length w then Raw (w ! (n − 1))
```

```
                else Enc (bl_tm M))
          | Suc k ⇒ (if n = 0 then Enc (le_tm M)
                else if k = 0 ∧ n ≤ i
                  then Enc (wrap_enc pack c w ! (n − 1))
                else Enc (bl_tm M)))
        else Enc (bl_tm M))
  by (simp add: mid_buf_extending_config_gen_def fun_eq_iff)

have src_eq: ?src = Config_M (W_Buf ?ws) (mt_tape ?src) (mt_pos ?src)
  using src_state src_pos src_tape
  by (cases ?src) simp

— The load-bearing read: W_User sees Raw ?a.
have read_W_User:
  mt_tape ?src 0 (mt_pos ?src 0) = Raw ?a
  using ‹i * c + j < length w› valid_mttm_k_pos[OF ‹valid_mttm M›]
  by (simp add: src_tape src_pos)

— Range typing of the read function.
have rd_range:
  (λk. mt_tape (mid_buf_extending_config_gen (k_tm M) M pack c w i j) k
       (mt_pos (mid_buf_extending_config_gen (k_tm M) M pack c w i j) k))
    ∈ UNIV → Raw ‘ Σu ∪ Enc ‘ Γ_tm M
  using a_in_Sigmau bl_in le_in ‹i * c + j < length w›
  by (auto simp: mid_buf_extending_config_gen_def split: nat.split)

— The wrap_buf_extend_delta_gen tuple.
have tuple_in:
  (W_Buf ?ws,
   λk. mt_tape ?src k (mt_pos ?src k),
   W_Buf (?ws @ [?a]),
   λk. mt_tape ?src k (mt_pos ?src k),
   λt. case t of 0 ⇒ dir.R | _ ⇒ dir.N)
  ∈ wrap_delta M pack c Σu
proof −
  have (W_Buf ?ws,
        λk. mt_tape ?src k (mt_pos ?src k),
        W_Buf (?ws @ [?a]),
        λk. mt_tape ?src k (mt_pos ?src k),
        λt. case t of 0 ⇒ dir.R | _ ⇒ dir.N)
       ∈ wrap_buf_extend_delta_gen (k_tm M) M c Σu
    unfolding wrap_buf_extend_delta_gen_def
    using len_ws ‹Suc j < c› a_in_Sigmau set_ws read_W_User rd_range
    by (auto simp: src_tape)
  thus ?thesis
    by (simp add: wrap_delta_def)
qed

— Apply mttm_step.step.
```

```
  have step_holds:
    (Config_M (W_Buf ?ws) (mt_tape ?src) (mt_pos ?src),
      Config_M (W_Buf (?ws @ [?a]))
        (λk. (mt_tape ?src k)((mt_pos ?src k) :=
                              mt_tape ?src k (mt_pos ?src k)))
        (λk. go_dir
             ((λt. case t of 0 ⇒ dir.R | _ ⇒ dir.N) k)
             (mt_pos ?src k)))
     ∈ mttm_step (wrap_delta M pack c Σu)
    by (rule mttm_step.step
         [where ts = mt_tape ?src and n = mt_pos ?src
                and a = λk. mt_tape ?src k (mt_pos ?src k)
                and dir = λt. case t of 0 ⇒ dir.R | _ ⇒ dir.N,
          OF tuple_in])

  — Target equality.
  have target_eq:
    Config_M (W_Buf (?ws @ [?a]))
      (λk. (mt_tape ?src k)((mt_pos ?src k) :=
                            mt_tape ?src k (mt_pos ?src k)))
      (λk. go_dir
           ((λt. case t of 0 ⇒ dir.R | _ ⇒ dir.N) k)
           (mt_pos ?src k))
     = mid_buf_extending_config_gen (k_tm M) M pack c w i (Suc j)
    unfolding mid_buf_extending_config_gen_def
    using take_app
    by (auto simp: src_pos src_tape fun_eq_iff
             split: nat.split)

  from step_holds target_eq src_eq
  show ?thesis by simp
qed
```

11.3 Helper (1b)(ii): one buffer-close step

A single *wrap_buf_close_delta_gen* step closing cycle i: packs the completed block, writes the encoded block to physical tape *1*, and advances both heads. Needs $2 \le k_tm\ M$: the encoded block is written to physical tape *1*, which must be in range for the target configuration to record the write.

```
lemma mid_buf_close_step:
  assumes valid_mttm M
      and 0 < c
      and set w ⊆ Σu
      and (Suc i) * c ≤ length w
      and pack (take c (drop (i * c) w)) ∈ Sigma_tm M
      and bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
```

```
  shows (mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1),
         mid_encoder_config_gen (k_tm M) M pack c w (Suc i))
        ∈ mttm_step (wrap_delta M pack c Σu)
proof −
  — Local opaque abbreviation cm = c − 1.
  define cm where cm = c − 1
  let ?src = mid_buf_extending_config_gen (k_tm M) M pack c w i cm
  let ?ws = take cm (drop (i ∗ c) w)
  let ?a = w ! (i ∗ c + cm)
  let ?packed = pack (take c (drop (i ∗ c) w))

  — Arithmetic facts about c.
  from ‹0 < c› have c_succ: Suc cm = c unfolding cm_def by simp
  from ‹(Suc i) ∗ c ≤ length w› have full_block:
    i ∗ c + c ≤ length w by simp
  with ‹0 < c› have a_idx: i ∗ c + cm < length w
    unfolding cm_def by linarith
  from c_succ have a_idx_succ: i ∗ c + cm + 1 = i ∗ c + c
    by simp

  — List-arithmetic facts about the completed block.
  have c1_lt_drop: cm < length (drop (i ∗ c) w)
    using a_idx by simp
  have len_ws: length ?ws = cm
    using c1_lt_drop by simp
  have len_ws_plus_one: length ?ws + 1 = c
    using len_ws c_succ by simp
  have set_ws: set ?ws ⊆ Σu
    using ‹set w ⊆ Σu›
    by (meson dual_order.trans set_drop_subset set_take_subset)
  have take_app: ?ws @ [?a] = take c (drop (i ∗ c) w)
  proof −
    have take c (drop (i ∗ c) w) = take (Suc cm) (drop (i ∗ c) w)
      using c_succ by simp
    also have ... = take cm (drop (i ∗ c) w) @ [drop (i ∗ c) w ! cm]
      using c1_lt_drop by (simp add: take_Suc_conv_app_nth)
    also have drop (i ∗ c) w ! cm = w ! (i ∗ c + cm)
      using a_idx by simp
    finally show ?thesis by simp
  qed
  have a_in_Sigmau: ?a ∈ Σu
    using ‹set w ⊆ Σu› a_idx
    by (meson nth_mem subsetD)
  have pack_app_eq: pack (?ws @ [?a]) = ?packed
    using take_app by simp

  — Substrate facts about M's distinguished symbols.
  have bl_in: bl_tm M ∈ Γ_tm M
    by (rule valid_mttm_blank_in_Gamma[OF ‹valid_mttm M›])
```

have *le_in*: *le_tm M* ∈ Γ*_tm M*
by (*rule valid_mttm_LE_in_Gamma*[*OF* ‹*valid_mttm M*›])

— Pack-value / *wrap_enc*-entry identity.
have *wenc_eq*: *?packed* = *wrap_enc pack c w* ! *i*
by (*rule wrap_enc_nth*[*OF* ‹*0* < *c*› ‹(*Suc i*) ∗ *c* ≤ *length w*›, *symmetric*])

— Selector facts for the source config.
have *src_state*: *mt_state ?src* = *W_Buf ?ws*
by (*simp add*: *mid_buf_extending_config_gen_def*)
have *src_pos*:
mt_pos ?src
= (λ*t*. *case t of 0* ⇒ *1* + *i* ∗ *c* + *cm*
| *Suc k* ⇒ (*if k* = *0 then 1* + *i else 0*))
by (*simp add*: *mid_buf_extending_config_gen_def fun_eq_iff*)
have *src_tape*:
mt_tape ?src
= (λ*t n*. *if t* < *k_tm M*
then (*case t of*
0 ⇒ (*if n* = *0 then Enc* (*le_tm M*)
else if n ≤ *length w then Raw* (*w* ! (*n* − *1*))
else Enc (*bl_tm M*))
| *Suc k* ⇒ (*if n* = *0 then Enc* (*le_tm M*)
else if k = *0* ∧ *n* ≤ *i*
then Enc (*wrap_enc pack c w* ! (*n* − *1*))
else Enc (*bl_tm M*)))
else Enc (*bl_tm M*))
by (*simp add*: *mid_buf_extending_config_gen_def fun_eq_iff*)

have *src_eq*: *?src* = *Config*$_M$ (*W_Buf ?ws*) (*mt_tape ?src*) (*mt_pos ?src*)
using *src_state src_pos src_tape*
by (*cases ?src*) *simp*

— Per-tape reads. *W_M 0* (physical tape *1*) reads blank in both boundary branches (head at *1* + *i*, beyond the *1..i* encoded prefix), so this read is boundary-insensitive.
have *read_W_User*:
mt_tape ?src 0 (*mt_pos ?src 0*) = *Raw ?a*
using *a_idx valid_mttm_k_pos*[*OF* ‹*valid_mttm M*›]
by (*simp add*: *src_tape src_pos*)
have *read_W_M_0*:
mt_tape ?src (*Suc 0*) (*mt_pos ?src* (*Suc 0*)) = *Enc* (*bl_tm M*)
by (*simp add*: *src_tape src_pos*)

— Range typing of the read function.
have *rd_range*:
(λ*k*. *mt_tape* (*mid_buf_extending_config_gen* (*k_tm M*) *M pack c w i cm*) *k*
(*mt_pos* (*mid_buf_extending_config_gen* (*k_tm M*) *M pack c w i cm*) *k*))

```
    ∈ UNIV → Raw ‘ Σu ∪ Enc ‘ Γ_tm M
  using a_in_Sigmau bl_in le_in a_idx
  by (auto simp: mid_buf_extending_config_gen_def split: nat.split)

— Static guard sym (Suc 0) ≠ Enc (le_tm M).
have sym_W_M_0_ne_LE:
  mt_tape ?src (Suc 0) (mt_pos ?src (Suc 0)) ≠ Enc (le_tm M)
  using read_W_M_0 ‹bl_tm M ≠ le_tm M› by simp

— Padded-tail blank fact for the faithful boundary ∀j ≥ k_tm M.
have tail_bl:
  ∀j≥k_tm M. mt_tape ?src j (mt_pos ?src j) = Enc (bl_tm M)
  by (simp add: src_tape)

— The wrap_buf_close_delta_gen tuple.
have tuple_in:
  (W_Buf ?ws,
    λk. mt_tape ?src k (mt_pos ?src k),
    W_Buf [],
    (λt. case t of
          Suc k ⇒ if k = 0 then Enc (pack (?ws @ [?a]))
                   else mt_tape ?src (Suc k) (mt_pos ?src (Suc k))
        | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0)),
    λt. case t of 0 ⇒ dir.R
                | Suc k ⇒ if k = 0 then dir.R else dir.N)
   ∈ wrap_delta M pack c Σu
proof −
  have (W_Buf ?ws,
        λk. mt_tape ?src k (mt_pos ?src k),
        W_Buf [],
        (λt. case t of
              Suc k ⇒ if k = 0 then Enc (pack (?ws @ [?a]))
                       else mt_tape ?src (Suc k) (mt_pos ?src (Suc k))
            | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0)),
        λt. case t of 0 ⇒ dir.R
                    | Suc k ⇒ if k = 0 then dir.R else dir.N)
       ∈ wrap_buf_close_delta_gen (k_tm M) M pack c Σu
    unfolding wrap_buf_close_delta_gen_def
              using len_ws_plus_one a_in_Sigmau set_ws read_W_User
sym_W_M_0_ne_LE
          ‹pack (take c (drop (i * c) w)) ∈ Sigma_tm M›
          pack_app_eq rd_range tail_bl
    by auto
  thus ?thesis
    by (simp add: wrap_delta_def)
qed

— Apply mttm_step.step.
have step_holds:
```

```
  (Config_M (W_Buf ?ws) (mt_tape ?src) (mt_pos ?src),
    Config_M (W_Buf [])
      (λk. (mt_tape ?src k)
           ((mt_pos ?src k) :=
             (case k of
                Suc k' ⇒ if k' = 0 then Enc (pack (?ws @ [?a]))
                           else mt_tape ?src (Suc k') (mt_pos ?src (Suc k'))
              | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0))))
      (λk. go_dir
           ((λt. case t of 0 ⇒ dir.R
                         | Suc k' ⇒ if k' = 0 then dir.R else dir.N) k)
           (mt_pos ?src k)))
   ∈ mttm_step (wrap_delta M pack c Σu)
  by (rule mttm_step.step
       [where ts = mt_tape ?src and n = mt_pos ?src
          and a = λt. case t of
                        Suc k ⇒ if k = 0 then Enc (pack (?ws @ [?a]))
                                  else mt_tape ?src (Suc k) (mt_pos ?src (Suc k))
                      | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0)
          and dir = λt. case t of 0 ⇒ dir.R
                                | Suc k' ⇒ if k' = 0 then dir.R else dir.N,
        OF tuple_in])
```

— Target equality. The block write to physical tape *1* lands in range (needs $2 \le$ *k_tm M*); the written *Enc* (*pack* (*?ws* @ [*?a*])) matches the target's *Enc* (*wrap_enc pack c w* ! *i*) via *pack* (*take c* (*drop* (*i* ∗ *c*) *w*)) = *wrap_enc pack c w* ! *i*.

```
  have target_eq:
    Config_M (W_Buf [])
      (λk. (mt_tape ?src k)
           ((mt_pos ?src k) :=
             (case k of
                Suc k' ⇒ if k' = 0 then Enc (pack (?ws @ [?a]))
                           else mt_tape ?src (Suc k') (mt_pos ?src (Suc k'))
              | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0))))
      (λk. go_dir
           ((λt. case t of 0 ⇒ dir.R
                         | Suc k' ⇒ if k' = 0 then dir.R else dir.N) k)
           (mt_pos ?src k))
    = mid_encoder_config_gen (k_tm M) M pack c w (Suc i)
    unfolding mid_encoder_config_gen_def
    using a_idx pack_app_eq wenc_eq c_succ a_idx_succ
          valid_mttm_k_pos[OF ‹valid_mttm M›] k2
    by (auto simp: src_pos src_tape fun_eq_iff
             split: nat.split if_splits)

  from step_holds target_eq src_eq
  show ?thesis unfolding cm_def by simp
qed
```

11.4 Helper (1b): one block cycle

One cycle composes $c - 1$ *mid_buf_extend_step*s with one *mid_buf_close_step*; carries $2 \le k_tm\ M$ for the closing step.

```
lemma mid_encoder_step:
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and w_alpha: set w ⊆ Σu
      and block_fits: (Suc i) * c ≤ length w
      and pack_in_Sigma: pack (take c (drop (i * c) w)) ∈ Sigma_tm M
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
  shows (mid_encoder_config_gen (k_tm M) M pack c w i,
         mid_encoder_config_gen (k_tm M) M pack c w (Suc i))
        ∈ (mttm_step (wrap_delta M pack c Σu))*
proof −
  let ?R = mttm_step (wrap_delta M pack c Σu)

  have base_eq:
    mid_buf_extending_config_gen (k_tm M) M pack c w i 0
      = mid_encoder_config_gen (k_tm M) M pack c w i
                    by (simp add: mid_encoder_config_gen_def
mid_buf_extending_config_gen_def
            fun_eq_iff split: nat.split)

  have block_pos: ⋀j. j < c ⟹ i * c + j < length w
    using block_fits by simp

  have iter:
    j ≤ c − 1 ⟹
     (mid_buf_extending_config_gen (k_tm M) M pack c w i 0,
      mid_buf_extending_config_gen (k_tm M) M pack c w i j) ∈ ?R*
    for j
  proof (induction j)
    case 0
    show ?case by simp
  next
    case (Suc j)
    from Suc.prems c_pos have j_le: j ≤ c − 1 by linarith
    note ih = Suc.IH[OF j_le]
    from Suc.prems c_pos have Suc_j_lt: Suc j < c by linarith
    from Suc.prems c_pos have j_lt: j < c by linarith
    have block_in: i * c + j < length w
      using block_pos[OF j_lt] .
    have step:
      (mid_buf_extending_config_gen (k_tm M) M pack c w i j,
       mid_buf_extending_config_gen (k_tm M) M pack c w i (Suc j)) ∈ ?R
      by (rule mid_buf_extend_step
            [OF vM c_pos w_alpha Suc_j_lt block_in])
```

```
    show ?case using ih step by (rule rtrancl_into_rtrancl)
  qed

  have extends_done:
    (mid_buf_extending_config_gen (k_tm M) M pack c w i 0,
      mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1)) ∈ ?R*
    using iter[of c − 1] by simp

  have close_step:
    (mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1),
      mid_encoder_config_gen (k_tm M) M pack c w (Suc i)) ∈ ?R
    by (rule mid_buf_close_step
          [where pack = pack,
           OF vM c_pos w_alpha block_fits pack_in_Sigma bl_ne_le k2])

  from extends_done close_step
  have (mid_buf_extending_config_gen (k_tm M) M pack c w i 0,
        mid_encoder_config_gen (k_tm M) M pack c w (Suc i)) ∈ ?R*
    by (rule rtrancl_into_rtrancl)
  with base_eq show ?thesis by simp
qed
```

L1+L2: relpow variant one cycle is exactly c wrap-steps.

```
lemma mid_encoder_step_relpow:
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and w_alpha: set w ⊆ Σu
      and block_fits: (Suc i) * c ≤ length w
      and pack_in_Sigma: pack (take c (drop (i * c) w)) ∈ Sigma_tm M
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
  shows (mid_encoder_config_gen (k_tm M) M pack c w i,
         mid_encoder_config_gen (k_tm M) M pack c w (Suc i))
        ∈ (mttm_step (wrap_delta M pack c Σu)) ^^ c
proof −
  let ?R = mttm_step (wrap_delta M pack c Σu)

  have base_eq:
    mid_buf_extending_config_gen (k_tm M) M pack c w i 0
      = mid_encoder_config_gen (k_tm M) M pack c w i
                    by (simp add: mid_encoder_config_gen_def
mid_buf_extending_config_gen_def
          fun_eq_iff split: nat.split)

  have block_pos: ⋀j. j < c ⟹ i * c + j < length w
    using block_fits by simp

  have iter:
    j ≤ c − 1 ⟹
```

```
    (mid_buf_extending_config_gen (k_tm M) M pack c w i 0,
     mid_buf_extending_config_gen (k_tm M) M pack c w i j) ∈ ?R ^^ j
   for j
 proof (induction j)
   case 0
   show ?case by simp
 next
   case (Suc j)
   from Suc.prems c_pos have j_le: j ≤ c − 1 by linarith
   note ih = Suc.IH[OF j_le]
   from Suc.prems c_pos have Suc_j_lt: Suc j < c by linarith
   from Suc.prems c_pos have j_lt: j < c by linarith
   have block_in: i * c + j < length w
     using block_pos[OF j_lt] .
   have step:
     (mid_buf_extending_config_gen (k_tm M) M pack c w i j,
       mid_buf_extending_config_gen (k_tm M) M pack c w i (Suc j)) ∈ ?R
     by (rule mid_buf_extend_step
           [OF vM c_pos w_alpha Suc_j_lt block_in])
   from ih step show ?case by (rule relpow_Suc_I)
 qed

 have extends_done:
   (mid_buf_extending_config_gen (k_tm M) M pack c w i 0,
     mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1)) ∈ ?R ^^
(c − 1)
   using iter[of c − 1] by simp

 have close_step:
   (mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1),
     mid_encoder_config_gen (k_tm M) M pack c w (Suc i)) ∈ ?R
   by (rule mid_buf_close_step
         [where pack = pack,
          OF vM c_pos w_alpha block_fits pack_in_Sigma bl_ne_le k2])

 from extends_done close_step
 have combined:
   (mid_buf_extending_config_gen (k_tm M) M pack c w i 0,
     mid_encoder_config_gen (k_tm M) M pack c w (Suc i)) ∈ ?R ^^ Suc (c −
1)
   by (rule relpow_Suc_I)

 have c_eq: Suc (c − 1) = c using c_pos by simp

 from combined base_eq c_eq show ?thesis by simp
qed
```

L3: encoder iter to *mid_encoder_config_gen* (*k_tm M*) *M pack c w i* for any $i \leq$ *length w div c* takes exactly $c * i$ wrap-steps.

```
lemma mid_encoder_iter_relpow:
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and w_alpha: set w ⊆ Σu
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
      and pack_contract_in_Sigma:
        ⋀i. i * c < length w ⟹
            pack (take c (drop (i * c) w)) ∈ Sigma_tm M
      and i_le: i ≤ length w div c
    shows (mid_encoder_config_gen (k_tm M) M pack c w 0,
           mid_encoder_config_gen (k_tm M) M pack c w i)
          ∈ (mttm_step (wrap_delta M pack c Σu)) ^^ (c * i)
proof −
  let ?R = mttm_step (wrap_delta M pack c Σu)
  let ?q = length w div c
  let ?r = length w mod c

  have div_mod_eq: ?q * c + ?r = length w
    by (rule div_mult_mod_eq)

  show ?thesis using i_le
  proof (induction i)
    case 0
    show ?case by simp
  next
    case (Suc i)
    from Suc.prems have i_le_q: i ≤ ?q by simp
    from Suc.prems have Suc_i_le_q: Suc i ≤ ?q by simp
    note ih = Suc.IH[OF i_le_q]

    have suc_ic_le_qc: (Suc i) * c ≤ ?q * c
      by (rule mult_le_mono1[OF Suc_i_le_q])
    have block_fits: (Suc i) * c ≤ length w
      using suc_ic_le_qc div_mod_eq by linarith
    have i_lt_q: i < ?q using Suc_i_le_q by simp
    have ic_lt_qc: i * c < ?q * c
      using i_lt_q c_pos by simp
    have ic_lt: i * c < length w
      using ic_lt_qc div_mod_eq by linarith

    have pack_in_Sigma: pack (take c (drop (i * c) w)) ∈ Sigma_tm M
      by (rule pack_contract_in_Sigma[OF ic_lt])

    have step:
      (mid_encoder_config_gen (k_tm M) M pack c w i,
        mid_encoder_config_gen (k_tm M) M pack c w (Suc i)) ∈ ?R ^^ c
      by (rule mid_encoder_step_relpow
          [where pack = pack,
```

```
    OF vM c_pos w_alpha block_fits pack_in_Sigma bl_ne_le k2])

    from ih step
    have (mid_encoder_config_gen (k_tm M) M pack c w 0,
          mid_encoder_config_gen (k_tm M) M pack c w (Suc i)) ∈ ?R ^^ (c * i
+ c)
      by (auto simp: relpow_add)
    moreover have c * i + c = c * Suc i by simp
    ultimately show ?case by simp
  qed
qed
```

11.5 Helper (1c)(r=0): clean-termination tail step

When *length w mod c = 0*, a single *wrap_buf_empty_end_delta_gen* step takes the post-final-cycle waypoint to *post_encoder_config_gen*; boundary-insensitive (the empty-end step writes nothing).

```
lemma mid_to_post_empty:
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and r_zero: length w mod c = 0
  shows (mid_encoder_config_gen (k_tm M) M pack c w (length w div c),
         post_encoder_config_gen (k_tm M) M pack c w)
        ∈ mttm_step (wrap_delta M pack c Σu)
proof −
  let ?q = length w div c
  let ?src = mid_encoder_config_gen (k_tm M) M pack c w ?q

  have qc_eq: ?q * c = length w
    using div_mult_mod_eq[of length w c] r_zero by simp
  have len_wenc: length (wrap_enc pack c w) = ?q
    using length_wrap_enc[OF c_pos, of pack w] r_zero by simp

  have bl_in: bl_tm M ∈ Γ_tm M
    by (rule valid_mttm_blank_in_Gamma[OF vM])
  have le_in: le_tm M ∈ Γ_tm M
    by (rule valid_mttm_LE_in_Gamma[OF vM])

  have src_state: mt_state ?src = W_Buf []
    by (simp add: mid_encoder_config_gen_def)
  have src_pos:
    mt_pos ?src
      = (λt. case t of 0 ⇒ 1 + ?q * c
                     | Suc k ⇒ (if k = 0 then 1 + ?q else 0))
    by (simp add: mid_encoder_config_gen_def fun_eq_iff)
  have src_tape:
    mt_tape ?src
      = (λt n. if t < k_tm M
               then (case t of
```

```
        0 ⇒ (if n = 0 then Enc (le_tm M)
              else if n ≤ length w then Raw (w ! (n − 1))
              else Enc (bl_tm M))
      | Suc k ⇒ (if n = 0 then Enc (le_tm M)
              else if k = 0 ∧ n ≤ ?q
                then Enc (wrap_enc pack c w ! (n − 1))
              else Enc (bl_tm M)))
    else Enc (bl_tm M))
  by (simp add: mid_encoder_config_gen_def fun_eq_iff)

have src_eq: ?src = Config_M (W_Buf []) (mt_tape ?src) (mt_pos ?src)
  using src_state src_pos src_tape
  by (cases ?src) simp

have read_W_User:
  mt_tape ?src 0 (mt_pos ?src 0) = Enc (bl_tm M)
  by (simp add: src_tape src_pos qc_eq)

have rd_range:
  (λk. mt_tape (mid_encoder_config_gen (k_tm M) M pack c w ?q) k
        (mt_pos (mid_encoder_config_gen (k_tm M) M pack c w ?q) k))
    ∈ UNIV → Raw ' Σu ∪ Enc ' Γ_tm M
  using bl_in le_in qc_eq
  by (auto simp: mid_encoder_config_gen_def split: nat.split)

have tuple_in:
  (W_Buf [],
    λk. mt_tape ?src k (mt_pos ?src k),
    W_Reset,
    λk. mt_tape ?src k (mt_pos ?src k),
    λ_. dir.N)
   ∈ wrap_delta M pack c Σu
proof −
  have (W_Buf [],
        λk. mt_tape ?src k (mt_pos ?src k),
        W_Reset,
        λk. mt_tape ?src k (mt_pos ?src k),
        λ_. dir.N)
       ∈ wrap_buf_empty_end_delta_gen (k_tm M) M Σu
    unfolding wrap_buf_empty_end_delta_gen_def
    using read_W_User rd_range
    by (auto simp: src_tape)
  thus ?thesis by (simp add: wrap_delta_def)
qed

have step_holds:
  (Config_M (W_Buf []) (mt_tape ?src) (mt_pos ?src),
    Config_M W_Reset
      (λk. (mt_tape ?src k)((mt_pos ?src k) :=
```

```
                        mt_tape ?src k (mt_pos ?src k)))
      (λk. go_dir dir.N (mt_pos ?src k)))
   ∈ mttm_step (wrap_delta M pack c Σu)
  by (rule mttm_step.step
       [where ts = mt_tape ?src and n = mt_pos ?src
            and a = λk. mt_tape ?src k (mt_pos ?src k)
            and dir = λ_. dir.N,
        OF tuple_in])

 have target_eq:
  Config_M W_Reset
    (λk. (mt_tape ?src k)((mt_pos ?src k) :=
                    mt_tape ?src k (mt_pos ?src k)))
    (λk. go_dir dir.N (mt_pos ?src k))
  = post_encoder_config_gen (k_tm M) M pack c w
  unfolding post_encoder_config_gen_def
  using qc_eq len_wenc
  by (auto simp: src_pos src_tape fun_eq_iff
        split: nat.split)

 from step_holds target_eq src_eq
 show ?thesis by simp
qed
```

11.6 Helper (1c)(r≠0): partial-block tail step

When *length w mod c ≠ 0*, *length w mod c* extends plus one *wrap_buf_nonempty_end_delta_gen* close pack the partial block onto physical tape *1* and reach *post_encoder_config_gen*. Needs *2 ≤ k_tm M* for the partial-block write to physical tape *1*.

```
lemma mid_to_post_partial:
 assumes vM: valid_mttm M
    and c_pos: 0 < c
    and w_alpha: set w ⊆ Σu
    and r_pos: length w mod c ≠ 0
    and pack_tail_in_Sigma:
      pack (drop ((length w div c) * c) w) ∈ Sigma_tm M
    and bl_ne_le: bl_tm M ≠ le_tm M
    and k2: 2 ≤ k_tm M
 shows (mid_encoder_config_gen (k_tm M) M pack c w (length w div c),
        post_encoder_config_gen (k_tm M) M pack c w)
       ∈ (mttm_step (wrap_delta M pack c Σu))*
proof −
 let ?q = length w div c
 let ?r = length w mod c
 let ?R = mttm_step (wrap_delta M pack c Σu)
 let ?tail = drop (?q * c) w
 let ?src = mid_buf_extending_config_gen (k_tm M) M pack c w ?q ?r
```

```
  have div_mod_eq: ?q * c + ?r = length w
    by (rule div_mult_mod_eq)
  have r_lt: ?r < c using c_pos by simp
  have tail_len: length ?tail = ?r
  proof −
    have length ?tail = length w − ?q * c by simp
    also have ... = ?r using div_mod_eq by linarith
    finally show ?thesis .
  qed
  have take_tail_eq: take ?r ?tail = ?tail
    using tail_len by simp
  have buf_take_eq: take ?r (drop (?q * c) w) = ?tail
    using take_tail_eq by simp
  have set_tail: set ?tail ⊆ Σu
    using w_alpha by (meson set_drop_subset subset_trans)
  have pack_tail_eq: pack (take ?r (drop (?q * c) w)) = pack ?tail
    using buf_take_eq by simp

  have len_wenc: length (wrap_enc pack c w) = ?q + 1
    using length_wrap_enc[OF c_pos, of pack w] r_pos by simp
  have wenc_q_eq: wrap_enc pack c w ! ?q = pack ?tail
    using wrap_enc_nth_partial[OF c_pos r_pos] .

  have bl_in: bl_tm M ∈ Γ_tm M
    by (rule valid_mttm_blank_in_Gamma[OF vM])
  have le_in: le_tm M ∈ Γ_tm M
    by (rule valid_mttm_LE_in_Gamma[OF vM])

  have base_eq:
    mid_buf_extending_config_gen (k_tm M) M pack c w ?q 0
      = mid_encoder_config_gen (k_tm M) M pack c w ?q
                    by (simp add: mid_encoder_config_gen_def
mid_buf_extending_config_gen_def
          fun_eq_iff split: nat.split)

  have iter:
    j ≤ ?r ⟹
     (mid_buf_extending_config_gen (k_tm M) M pack c w ?q 0,
      mid_buf_extending_config_gen (k_tm M) M pack c w ?q j) ∈ ?R*
    for j
  proof (induction j)
    case 0
    show ?case by simp
  next
    case (Suc j)
    from Suc.prems have j_le: j ≤ ?r by simp
    note ih = Suc.IH[OF j_le]
    from Suc.prems r_lt have Suc_j_lt: Suc j < c by linarith
    from Suc.prems r_lt have j_lt: j < ?r by linarith
```

```
  have block_in: ?q * c + j < length w
    using j_lt div_mod_eq by linarith
  have step:
    (mid_buf_extending_config_gen (k_tm M) M pack c w ?q j,
      mid_buf_extending_config_gen (k_tm M) M pack c w ?q (Suc j)) ∈ ?R
    by (rule mid_buf_extend_step
          [OF vM c_pos w_alpha Suc_j_lt block_in])
  show ?case using ih step by (rule rtrancl_into_rtrancl)
qed

have extends_done:
  (mid_buf_extending_config_gen (k_tm M) M pack c w ?q 0,
    mid_buf_extending_config_gen (k_tm M) M pack c w ?q ?r) ∈ ?R*
  using iter[of ?r] by simp

have src_state: mt_state ?src = W_Buf ?tail
  using buf_take_eq by (simp add: mid_buf_extending_config_gen_def)
have src_pos:
  mt_pos ?src
    = (λt. case t of 0 ⇒ 1 + ?q * c + ?r
                   | Suc k ⇒ (if k = 0 then 1 + ?q else 0))
  by (simp add: mid_buf_extending_config_gen_def fun_eq_iff)
have src_tape:
  mt_tape ?src
    = (λt n. if t < k_tm M
          then (case t of
            0 ⇒ (if n = 0 then Enc (le_tm M)
                  else if n ≤ length w then Raw (w ! (n − 1))
                  else Enc (bl_tm M))
          | Suc k ⇒ (if n = 0 then Enc (le_tm M)
                  else if k = 0 ∧ n ≤ ?q
                    then Enc (wrap_enc pack c w ! (n − 1))
                  else Enc (bl_tm M)))
          else Enc (bl_tm M))
  by (simp add: mid_buf_extending_config_gen_def fun_eq_iff)

have src_eq: ?src = Config_M (W_Buf ?tail) (mt_tape ?src) (mt_pos ?src)
  using src_state src_pos src_tape
  by (cases ?src) simp

have read_W_User:
  mt_tape ?src 0 (mt_pos ?src 0) = Enc (bl_tm M)
  using div_mod_eq by (simp add: src_tape src_pos)
have read_W_M_0:
  mt_tape ?src (Suc 0) (mt_pos ?src (Suc 0)) = Enc (bl_tm M)
  by (simp add: src_tape src_pos)

have rd_range:
  (λk. mt_tape (mid_buf_extending_config_gen (k_tm M) M pack c w ?q ?r) k
```

```
        (mt_pos (mid_buf_extending_config_gen (k_tm M) M pack c w ?q ?r)
k))
     ∈ UNIV → Raw ' Σu ∪ Enc ' Γ_tm M
   using bl_in le_in div_mod_eq
   by (auto simp: mid_buf_extending_config_gen_def split: nat.split)

  have sym_W_M_0_ne_LE:
   mt_tape ?src (Suc 0) (mt_pos ?src (Suc 0)) ≠ Enc (le_tm M)
   using read_W_M_0 bl_ne_le by simp

  have tail_bl:
   ∀j≥k_tm M. mt_tape ?src j (mt_pos ?src j) = Enc (bl_tm M)
   by (simp add: src_tape)

  have tuple_in:
   (W_Buf ?tail,
    λk. mt_tape ?src k (mt_pos ?src k),
    W_Reset,
    (λt. case t of
          Suc k ⇒ if k = 0 then Enc (pack ?tail)
                   else mt_tape ?src (Suc k) (mt_pos ?src (Suc k))
        | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0)),
    λt. case t of Suc k ⇒ if k = 0 then dir.R else dir.N
              | 0 ⇒ dir.N)
   ∈ wrap_delta M pack c Σu
  proof −
   have tail_nonempty: ?tail ≠ []
    using tail_len r_pos by (cases ?tail) auto
   have (W_Buf ?tail,
         λk. mt_tape ?src k (mt_pos ?src k),
         W_Reset,
         (λt. case t of
               Suc k ⇒ if k = 0 then Enc (pack ?tail)
                        else mt_tape ?src (Suc k) (mt_pos ?src (Suc k))
             | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0)),
         λt. case t of Suc k ⇒ if k = 0 then dir.R else dir.N
                   | 0 ⇒ dir.N)
        ∈ wrap_buf_nonempty_end_delta_gen (k_tm M) M pack c Σu
    unfolding wrap_buf_nonempty_end_delta_gen_def
    using tail_nonempty tail_len r_lt set_tail read_W_User
          sym_W_M_0_ne_LE pack_tail_in_Sigma rd_range tail_bl
    by auto
   thus ?thesis by (simp add: wrap_delta_def)
  qed

  have step_holds:
   (Config_M (W_Buf ?tail) (mt_tape ?src) (mt_pos ?src),
    Config_M W_Reset
     (λk. (mt_tape ?src k)
```

```
            ((mt_pos ?src k) :=
               (case k of
                  Suc k' ⇒ if k' = 0 then Enc (pack ?tail)
                            else mt_tape ?src (Suc k') (mt_pos ?src (Suc k'))
                | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0))))
        (λk. go_dir
            ((λt. case t of Suc k' ⇒ if k' = 0 then dir.R else dir.N
                          | 0 ⇒ dir.N) k)
            (mt_pos ?src k)))
    ∈ mttm_step (wrap_delta M pack c Σu)
   by (rule mttm_step.step
        [where ts = mt_tape ?src and n = mt_pos ?src
           and a = λt. case t of
                         Suc k ⇒ if k = 0 then Enc (pack ?tail)
                                  else mt_tape ?src (Suc k) (mt_pos ?src (Suc k))
                       | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0)
           and dir = λt. case t of Suc k' ⇒ if k' = 0 then dir.R else dir.N
                                 | 0 ⇒ dir.N,
         OF tuple_in])

  have target_eq:
    Config_M W_Reset
      (λk. (mt_tape ?src k)
           ((mt_pos ?src k) :=
              (case k of
                 Suc k' ⇒ if k' = 0 then Enc (pack ?tail)
                           else mt_tape ?src (Suc k') (mt_pos ?src (Suc k'))
               | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0))))
      (λk. go_dir
           ((λt. case t of Suc k' ⇒ if k' = 0 then dir.R else dir.N
                         | 0 ⇒ dir.N) k)
           (mt_pos ?src k))
    = post_encoder_config_gen (k_tm M) M pack c w
   unfolding post_encoder_config_gen_def
   using div_mod_eq len_wenc wenc_q_eq r_pos valid_mttm_k_pos[OF vM] k2
   by (auto simp: src_pos src_tape fun_eq_iff
         split: nat.split if_splits)

  have final_step:
    (?src, post_encoder_config_gen (k_tm M) M pack c w) ∈ ?R
    using step_holds target_eq src_eq by simp

  from extends_done final_step
  have (mid_buf_extending_config_gen (k_tm M) M pack c w ?q 0,
        post_encoder_config_gen (k_tm M) M pack c w) ∈ ?R*
    by (rule rtrancl_into_rtrancl)
  with base_eq show ?thesis by simp
qed
```

L4: relpow form of *mid_to_post_partial* the partial tail takes exactly

Suc (*length w mod c*) wrap-steps.

lemma *mid_to_post_partial_relpow*:
  assumes *vM*: *valid_mttm M*
    and *c_pos*: *0 < c*
    and *w_alpha*: *set w ⊆ Σu*
    and *r_pos*: *length w mod c ≠ 0*
    and *pack_tail_in_Sigma*:
      pack (*drop* ((*length w div c*) * *c*) *w*) ∈ *Sigma_tm M*
    and *bl_ne_le*: *bl_tm M ≠ le_tm M*
    and *k2*: *2 ≤ k_tm M*
  shows (*mid_encoder_config_gen* (*k_tm M*) *M pack c w* (*length w div c*),
      post_encoder_config_gen (*k_tm M*) *M pack c w*)
    ∈ (*mttm_step* (*wrap_delta M pack c Σu*)) ^^ *Suc* (*length w mod c*)
proof −
  let *?q* = *length w div c*
  let *?r* = *length w mod c*
  let *?R* = *mttm_step* (*wrap_delta M pack c Σu*)
  let *?tail* = *drop* (*?q* * *c*) *w*
  let *?src* = *mid_buf_extending_config_gen* (*k_tm M*) *M pack c w ?q ?r*

  have *div_mod_eq*: *?q* * *c* + *?r* = *length w*
    by (*rule div_mult_mod_eq*)
  have *r_lt*: *?r < c* **using** *c_pos* **by** *simp*
  have *tail_len*: *length ?tail* = *?r*
  proof −
    have *length ?tail* = *length w* − *?q* * *c* **by** *simp*
    also have ... = *?r* **using** *div_mod_eq* **by** *linarith*
    finally show *?thesis* **.**
  qed
  have *take_tail_eq*: *take ?r ?tail* = *?tail*
    using *tail_len* **by** *simp*
  have *buf_take_eq*: *take ?r* (*drop* (*?q* * *c*) *w*) = *?tail*
    using *take_tail_eq* **by** *simp*
  have *set_tail*: *set ?tail* ⊆ *Σu*
    using *w_alpha* **by** (*meson set_drop_subset subset_trans*)
  have *pack_tail_eq*: *pack* (*take ?r* (*drop* (*?q* * *c*) *w*)) = *pack ?tail*
    using *buf_take_eq* **by** *simp*

  have *len_wenc*: *length* (*wrap_enc pack c w*) = *?q* + *1*
    using *length_wrap_enc*[*OF c_pos*, *of pack w*] *r_pos* **by** *simp*
  have *wenc_q_eq*: *wrap_enc pack c w* ! *?q* = *pack ?tail*
    using *wrap_enc_nth_partial*[*OF c_pos r_pos*] **.**

  have *bl_in*: *bl_tm M* ∈ Γ*_tm M*
    by (*rule valid_mttm_blank_in_Gamma*[*OF vM*])
  have *le_in*: *le_tm M* ∈ Γ*_tm M*
    by (*rule valid_mttm_LE_in_Gamma*[*OF vM*])

  have *base_eq*:

```
  mid_buf_extending_config_gen (k_tm M) M pack c w ?q 0
    = mid_encoder_config_gen (k_tm M) M pack c w ?q
                    by (simp add: mid_encoder_config_gen_def
mid_buf_extending_config_gen_def
          fun_eq_iff split: nat.split)

 have iter:
  j ≤ ?r ⟹
   (mid_buf_extending_config_gen (k_tm M) M pack c w ?q 0,
    mid_buf_extending_config_gen (k_tm M) M pack c w ?q j) ∈ ?R ^^ j
  for j
 proof (induction j)
  case 0
  show ?case by simp
 next
  case (Suc j)
  from Suc.prems have j_le: j ≤ ?r by simp
  note ih = Suc.IH[OF j_le]
  from Suc.prems r_lt have Suc_j_lt: Suc j < c by linarith
  from Suc.prems r_lt have j_lt: j < ?r by linarith
  have block_in: ?q * c + j < length w
   using j_lt div_mod_eq by linarith
  have step:
   (mid_buf_extending_config_gen (k_tm M) M pack c w ?q j,
    mid_buf_extending_config_gen (k_tm M) M pack c w ?q (Suc j)) ∈ ?R
   by (rule mid_buf_extend_step
       [OF vM c_pos w_alpha Suc_j_lt block_in])
  from ih step show ?case by (rule relpow_Suc_I)
 qed

 have extends_done:
  (mid_buf_extending_config_gen (k_tm M) M pack c w ?q 0,
   mid_buf_extending_config_gen (k_tm M) M pack c w ?q ?r) ∈ ?R ^^ ?r
  using iter[of ?r] by simp

 have src_state: mt_state ?src = W_Buf ?tail
  using buf_take_eq by (simp add: mid_buf_extending_config_gen_def)
 have src_pos:
  mt_pos ?src
   = (λt. case t of 0 ⇒ 1 + ?q * c + ?r
                  | Suc k ⇒ (if k = 0 then 1 + ?q else 0))
  by (simp add: mid_buf_extending_config_gen_def fun_eq_iff)
 have src_tape:
  mt_tape ?src
   = (λt n. if t < k_tm M
         then (case t of
           0 ⇒ (if n = 0 then Enc (le_tm M)
                 else if n ≤ length w then Raw (w ! (n − 1))
                 else Enc (bl_tm M))
```

```
                | Suc k ⇒ (if n = 0 then Enc (le_tm M)
                          else if k = 0 ∧ n ≤ ?q
                            then Enc (wrap_enc pack c w ! (n − 1))
                          else Enc (bl_tm M)))
              else Enc (bl_tm M))
    by (simp add: mid_buf_extending_config_gen_def fun_eq_iff)

  have src_eq: ?src = Config_M (W_Buf ?tail) (mt_tape ?src) (mt_pos ?src)
    using src_state src_pos src_tape
    by (cases ?src) simp

  have read_W_User:
    mt_tape ?src 0 (mt_pos ?src 0) = Enc (bl_tm M)
    using div_mod_eq by (simp add: src_tape src_pos)
  have read_W_M_0:
    mt_tape ?src (Suc 0) (mt_pos ?src (Suc 0)) = Enc (bl_tm M)
    by (simp add: src_tape src_pos)

  have rd_range:
    (λk. mt_tape (mid_buf_extending_config_gen (k_tm M) M pack c w ?q ?r) k
          (mt_pos (mid_buf_extending_config_gen (k_tm M) M pack c w ?q ?r)
k))
      ∈ UNIV → Raw ‘ Σu ∪ Enc ‘ Γ_tm M
    using bl_in le_in div_mod_eq
    by (auto simp: mid_buf_extending_config_gen_def split: nat.split)

  have sym_W_M_0_ne_LE:
    mt_tape ?src (Suc 0) (mt_pos ?src (Suc 0)) ≠ Enc (le_tm M)
    using read_W_M_0 bl_ne_le by simp

  have tail_bl:
    ∀j≥k_tm M. mt_tape ?src j (mt_pos ?src j) = Enc (bl_tm M)
    by (simp add: src_tape)

  have tuple_in:
    (W_Buf ?tail,
      λk. mt_tape ?src k (mt_pos ?src k),
      W_Reset,
      (λt. case t of
            Suc k ⇒ if k = 0 then Enc (pack ?tail)
                    else mt_tape ?src (Suc k) (mt_pos ?src (Suc k))
          | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0)),
      λt. case t of Suc k ⇒ if k = 0 then dir.R else dir.N
                  | 0 ⇒ dir.N)
    ∈ wrap_delta M pack c Σu
  proof −
    have tail_nonempty: ?tail ≠ []
      using tail_len r_pos by (cases ?tail) auto
    have (W_Buf ?tail,
```

```
      λk. mt_tape ?src k (mt_pos ?src k),
      W_Reset,
      (λt. case t of
            Suc k ⇒ if k = 0 then Enc (pack ?tail)
                     else mt_tape ?src (Suc k) (mt_pos ?src (Suc k))
          | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0)),
      λt. case t of Suc k ⇒ if k = 0 then dir.R else dir.N
                  | 0 ⇒ dir.N)
    ∈ wrap_buf_nonempty_end_delta_gen (k_tm M) M pack c Σu
  unfolding wrap_buf_nonempty_end_delta_gen_def
  using tail_nonempty tail_len r_lt set_tail read_W_User
      sym_W_M_0_ne_LE pack_tail_in_Sigma rd_range tail_bl
  by auto
 thus ?thesis by (simp add: wrap_delta_def)
qed

have step_holds:
 (Config_M (W_Buf ?tail) (mt_tape ?src) (mt_pos ?src),
   Config_M W_Reset
    (λk. (mt_tape ?src k)
         ((mt_pos ?src k) :=
           (case k of
             Suc k' ⇒ if k' = 0 then Enc (pack ?tail)
                      else mt_tape ?src (Suc k') (mt_pos ?src (Suc k'))
           | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0))))
    (λk. go_dir
         ((λt. case t of Suc k' ⇒ if k' = 0 then dir.R else dir.N
                       | 0 ⇒ dir.N) k)
         (mt_pos ?src k)))
 ∈ mttm_step (wrap_delta M pack c Σu)
 by (rule mttm_step.step
      [where ts = mt_tape ?src and n = mt_pos ?src
        and a = λt. case t of
                     Suc k ⇒ if k = 0 then Enc (pack ?tail)
                              else mt_tape ?src (Suc k) (mt_pos ?src (Suc k))
                   | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0)
        and dir = λt. case t of Suc k' ⇒ if k' = 0 then dir.R else dir.N
                              | 0 ⇒ dir.N,
       OF tuple_in])

have target_eq:
 Config_M W_Reset
   (λk. (mt_tape ?src k)
        ((mt_pos ?src k) :=
          (case k of
            Suc k' ⇒ if k' = 0 then Enc (pack ?tail)
                     else mt_tape ?src (Suc k') (mt_pos ?src (Suc k'))
          | 0 ⇒ mt_tape ?src 0 (mt_pos ?src 0))))
   (λk. go_dir
```

```
          ((λt. case t of Suc k' ⇒ if k' = 0 then dir.R else dir.N
                        | 0 ⇒ dir.N) k)
          (mt_pos ?src k))
    = post_encoder_config_gen (k_tm M) M pack c w
   unfolding post_encoder_config_gen_def
   using div_mod_eq len_wenc wenc_q_eq r_pos valid_mttm_k_pos[OF vM] k2
   by (auto simp: src_pos src_tape fun_eq_iff
         split: nat.split if_splits)

  have final_step:
   (?src, post_encoder_config_gen (k_tm M) M pack c w) ∈ ?R
   using step_holds target_eq src_eq by simp

  from extends_done final_step
  have (mid_buf_extending_config_gen (k_tm M) M pack c w ?q 0,
       post_encoder_config_gen (k_tm M) M pack c w) ∈ ?R ^^ Suc ?r
   by (rule relpow_Suc_I)
  with base_eq show ?thesis by simp
qed
```

11.7 Helper (1c): tail dispatch

Tail step from the post-final-full-cycle waypoint to *post_encoder_config_gen*, dispatching on *length w mod c*.

```
lemma mid_to_post:
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and w_alpha: set w ⊆ Σu
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
      and pack_tail_in_Sigma:
        length w mod c ≠ 0 ⟹
         pack (drop ((length w div c) * c) w) ∈ Sigma_tm M
  shows (mid_encoder_config_gen (k_tm M) M pack c w (length w div c),
         post_encoder_config_gen (k_tm M) M pack c w)
        ∈ (mttm_step (wrap_delta M pack c Σu))*
proof (cases length w mod c = 0)
  case True
  from mid_to_post_empty[OF vM c_pos True, of pack Σu]
  show ?thesis by (rule r_into_rtrancl)
next
  case False
  show ?thesis
   by (rule mid_to_post_partial
        [where pack = pack,
         OF vM c_pos w_alpha False
           pack_tail_in_Sigma[OF False] bl_ne_le k2])
qed
```

11.8 Encoder-phase reachability and step count

The full encoder phase reaches *post_encoder_config_gen*.

lemma *encoder_phase_terminates*:
  assumes *vM*: *valid_mttm M*
    and *c_pos*: *0 < c*
    and *w_alpha*: *set w ⊆ Σu*
    and *bl_ne_le*: *bl_tm M ≠ le_tm M*
    and *k2*: *2 ≤ k_tm M*
    and *pack_contract_in_Sigma*:
      *⋀i. i * c < length w ⟹*
        *pack (take c (drop (i * c) w)) ∈ Sigma_tm M*
  shows *(init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),*
      post_encoder_config_gen (k_tm M) M pack c w)
     *∈ (mttm_step (wrap_delta M pack c Σu))**
proof −
  let *?q = length w div c*
  let *?r = length w mod c*
  let *?R = mttm_step (wrap_delta M pack c Σu)*

  have *div_mod_eq*: *?q * c + ?r = length w*
    by *(rule div_mult_mod_eq)*
  have *r_lt*: *?r < c* **using** *c_pos* **by** *simp*

  have *init_step*:
    (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
      mid_encoder_config_gen (k_tm M) M pack c w 0) ∈ ?R
    by *(rule init_to_mid_zero[OF vM c_pos])*

  have *iter*:
    i ≤ ?q ⟹
     (mid_encoder_config_gen (k_tm M) M pack c w 0,
      *mid_encoder_config_gen (k_tm M) M pack c w i) ∈ ?R**
    for *i*
  proof *(induction i)*
    case *0*
    show *?case* **by** *simp*
  next
    case *(Suc i)*
    from *Suc.prems* **have** *i_le_q*: *i ≤ ?q* **by** *simp*
    from *Suc.prems* **have** *Suc_i_le_q*: *Suc i ≤ ?q* **by** *simp*
    note *ih = Suc.IH[OF i_le_q]*

    have *suc_ic_le_qc*: *(Suc i) * c ≤ ?q * c*
      by *(rule mult_le_mono1[OF Suc_i_le_q])*
    have *block_full*: *(Suc i) * c ≤ length w*
      using *suc_ic_le_qc div_mod_eq* **by** *linarith*
    have *i_lt_q*: *i < ?q* **using** *Suc_i_le_q* **by** *simp*
    have *ic_lt_qc*: *i * c < ?q * c*

```
    using i_lt_q c_pos by simp
  have ic_lt: i * c < length w
    using ic_lt_qc div_mod_eq by linarith

  have pack_in_Sigma: pack (take c (drop (i * c) w)) ∈ Sigma_tm M
    by (rule pack_contract_in_Sigma[OF ic_lt])

  have step:
    (mid_encoder_config_gen (k_tm M) M pack c w i,
      mid_encoder_config_gen (k_tm M) M pack c w (Suc i)) ∈ ?R*
    by (rule mid_encoder_step
          [where pack = pack,
           OF vM c_pos w_alpha block_full pack_in_Sigma bl_ne_le k2])
  show ?case using ih step by (rule rtrancl_trans)
qed

have outer_done:
  (mid_encoder_config_gen (k_tm M) M pack c w 0,
    mid_encoder_config_gen (k_tm M) M pack c w ?q) ∈ ?R*
  using iter[of ?q] by simp

have qc_lt: ?r ≠ 0 ⟹ ?q * c < length w
  using div_mod_eq by linarith
have take_drop_eq:
  ?r ≠ 0 ⟹ take c (drop (?q * c) w) = drop (?q * c) w
proof −
  assume ?r ≠ 0
  have length (drop (?q * c) w) = length w − ?q * c
    by simp
  also have ... = ?r using div_mod_eq by linarith
  finally have length (drop (?q * c) w) = ?r .
  moreover have ?r < c using r_lt .
  ultimately show take c (drop (?q * c) w) = drop (?q * c) w
    by simp
qed
have pack_tail_in_Sigma:
  ?r ≠ 0 ⟹ pack (drop (?q * c) w) ∈ Sigma_tm M
proof −
  assume r_pos: ?r ≠ 0
  from pack_contract_in_Sigma[OF qc_lt[OF r_pos]]
  have pack (take c (drop (?q * c) w)) ∈ Sigma_tm M .
  with take_drop_eq[OF r_pos] show ?thesis by simp
qed

have tail_step:
  (mid_encoder_config_gen (k_tm M) M pack c w ?q,
    post_encoder_config_gen (k_tm M) M pack c w) ∈ ?R*
  by (rule mid_to_post
        [where pack = pack,
```

OF vM c_pos w_alpha bl_ne_le k2 pack_tail_in_Sigma])

from *outer_done tail_step*
have (*mid_encoder_config_gen* (*k_tm M*) *M pack c w 0*,
post_encoder_config_gen (*k_tm M*) *M pack c w*) ∈ *?R**
by (*rule rtrancl_trans*)
with *init_step*
show *?thesis*
by (*rule converse_rtrancl_into_rtrancl*)
qed

L5: relpow form the encoder phase takes exactly *length w* + *2* wrap-steps.

lemma *init_to_post_encoder_relpow*:
assumes *vM*: *valid_mttm M*
and *c_pos*: *0* < *c*
and *w_alpha*: *set w* ⊆ Σ*u*
and *bl_ne_le*: *bl_tm M* ≠ *le_tm M*
and *k2*: *2* ≤ *k_tm M*
and *pack_contract_in_Sigma*:
⋀*i*. *i* * *c* < *length w* ⟹
pack (*take c* (*drop* (*i* * *c*) *w*)) ∈ *Sigma_tm M*
shows (*init_config_mttm* (*encoding_wrap M pack c* Σ*u*) (*map Raw w*),
post_encoder_config_gen (*k_tm M*) *M pack c w*)
∈ (*mttm_step* (*wrap_delta M pack c* Σ*u*)) ^^ (*length w* + *2*)
proof −
let *?q* = *length w div c*
let *?r* = *length w mod c*
let *?R* = *mttm_step* (*wrap_delta M pack c* Σ*u*)

have *div_mod_eq*: *?q* * *c* + *?r* = *length w*
by (*rule div_mult_mod_eq*)
have *r_lt*: *?r* < *c* **using** *c_pos* **by** *simp*

have *init_step*:
(*init_config_mttm* (*encoding_wrap M pack c* Σ*u*) (*map Raw w*),
mid_encoder_config_gen (*k_tm M*) *M pack c w 0*) ∈ *?R*
by (*rule init_to_mid_zero*[*OF vM c_pos*])

have *iter_step*:
(*mid_encoder_config_gen* (*k_tm M*) *M pack c w 0*,
mid_encoder_config_gen (*k_tm M*) *M pack c w ?q*)
∈ *?R* ^^ (*c* * *?q*)
by (*rule mid_encoder_iter_relpow*
[**where** *pack* = *pack*,
OF vM c_pos w_alpha bl_ne_le k2 pack_contract_in_Sigma order_refl])

from *relpow_Suc_I2*[*OF init_step iter_step*]
have *init_to_q*:

```
    (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
      mid_encoder_config_gen (k_tm M) M pack c w ?q) ∈ ?R ^^ Suc (c * ?q) .

  show ?thesis
  proof (cases ?r = 0)
    case True
    have tail_step:
      (mid_encoder_config_gen (k_tm M) M pack c w ?q,
        post_encoder_config_gen (k_tm M) M pack c w) ∈ ?R
      by (rule mid_to_post_empty[OF vM c_pos True])
    from relpow_Suc_I[OF init_to_q tail_step]
    have chain:
      (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
        post_encoder_config_gen (k_tm M) M pack c w) ∈ ?R ^^ Suc (Suc (c *
?q)) .
    have len_eq: Suc (Suc (c * ?q)) = length w + 2
      using True div_mod_eq by (simp add: mult.commute)
    from chain len_eq show ?thesis by simp
  next
    case False
    have pack_tail_in_Sigma:
      pack (drop (?q * c) w) ∈ Sigma_tm M
    proof −
      have qc_lt: ?q * c < length w
        using False div_mod_eq by linarith
      have pack (take c (drop (?q * c) w))
            = pack (drop (?q * c) w)
      proof −
        have len_drop_eq: length (drop (?q * c) w) = ?r
        proof −
          have length (drop (?q * c) w) = length w − ?q * c by simp
          also have ... = ?r using div_mod_eq by linarith
          finally show ?thesis .
        qed
        from r_lt len_drop_eq have take c (drop (?q * c) w) = drop (?q * c) w
          by simp
        thus ?thesis by simp
      qed
      with pack_contract_in_Sigma[OF qc_lt]
      show ?thesis by simp
    qed
    have tail_step:
      (mid_encoder_config_gen (k_tm M) M pack c w ?q,
        post_encoder_config_gen (k_tm M) M pack c w) ∈ ?R ^^ Suc ?r
      by (rule mid_to_post_partial_relpow
            [where pack = pack,
             OF vM c_pos w_alpha False pack_tail_in_Sigma bl_ne_le k2])
    from init_to_q tail_step
    have chain:
```

```
    (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
      post_encoder_config_gen (k_tm M) M pack c w)
      ∈ ?R ^^ (Suc (c * ?q) + Suc ?r)
    unfolding relpow_add by blast
  have len_eq: Suc (c * ?q) + Suc ?r = length w + 2
    using div_mod_eq by (simp add: mult.commute)
  from chain len_eq show ?thesis by simp
 qed
qed

end
theory Wrap_Forcing
 imports Wrap_Encoder
begin
```

12 Faithful (k-tape) encoding wrap: forcing machinery

The forward language inclusion needs that any *accepting* wrap-trace factors canonically through the encoder: the encoder / combined-reset / dispatch transitions are *forced* (functional in (*state*, *read−vector*) regardless of M's nondeterminism), and only the *wrap_run_delta* portion follows M's δ.

The state-routing graph of *wrap_delta* is *W_Init* → *W_Buf* → *W_Reset* → *W_Disp* → *W_Run*; the families are the boundary-parameterised encoder families (*k_tm M* via the *_gen* families), the combined reset, and the transposed run. The combined reset's *le*-guards make a few routing / determinism closes need (*auto split*: *if_splits*).

12.1 State-routing exclusions and forward determinism

Target-side exclusion for *W_Run* entry: if a wrap-tuple's target state is *W_Run q′* and its source is non-*W_Run*, only *wrap_disp_delta_gen* fires, forcing source *W_Disp* and target *W_Run* (*s_tm M*). Every other non-*W_Run*-source family has a non-*W_Run* target.

```
lemma wrap_delta_W_Run_target_from_W_Disp:
 fixes M :: ('q, 'b) mttm
   and sym :: nat ⇒ ('a, 'b) wrap_alphabet
   and a' :: nat ⇒ ('a, 'b) wrap_alphabet
   and dir :: nat ⇒ dir
 assumes tuple_in: (q, sym, W_Run q', a', dir) ∈ wrap_delta M pack c Σu
     and non_run_src: ∀ q_M. q ≠ W_Run q_M
   shows q = W_Disp ∧ q' = s_tm M
       ∧ (q, sym, W_Run q', a', dir) ∈ wrap_disp_delta_gen (k_tm M) M Σu
 using tuple_in non_run_src
 unfolding wrap_delta_def
         wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
```

```
        wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
        wrap_buf_nonempty_end_delta_gen_def
        wrap_rewind_loop_delta_gen_def
        wrap_plant_done_delta_def
        wrap_disp_delta_gen_def wrap_run_delta_def
  by auto
```

Unconditional forward determinism in the non-*W_Run* portion of *wrap_delta*. The *W_Reset* branch needs *split*: *if_splits* for the combined reset's *le*-guards (cf. *wrap_det*).

```
lemma wrap_delta_non_W_Run_det:
  assumes t1: (q, a, p₁, b₁, d₁) ∈ wrap_delta M pack c Σu
      and t2: (q, a, p₂, b₂, d₂) ∈ wrap_delta M pack c Σu
      and non_run: ∀ q_M. q ≠ W_Run q_M
    shows (p₁, b₁, d₁) = (p₂, b₂, d₂)
proof (cases q)
  case W_Init
  with t1 t2 show ?thesis
    unfolding wrap_delta_def
          wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
          wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
          wrap_buf_nonempty_end_delta_gen_def
          wrap_rewind_loop_delta_gen_def
          wrap_plant_done_delta_def
          wrap_disp_delta_gen_def wrap_run_delta_def
    by auto
next
  case (W_Buf ws)
  with t1 t2 show ?thesis
    unfolding wrap_delta_def
          wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
          wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
          wrap_buf_nonempty_end_delta_gen_def
          wrap_rewind_loop_delta_gen_def
          wrap_plant_done_delta_def
          wrap_disp_delta_gen_def wrap_run_delta_def
    by auto
next
  case W_Reset
  with t1 t2 show ?thesis
    unfolding wrap_delta_def
          wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
          wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
          wrap_buf_nonempty_end_delta_gen_def
          wrap_rewind_loop_delta_gen_def
          wrap_plant_done_delta_def
          wrap_disp_delta_gen_def wrap_run_delta_def
    by (auto split: if_splits)
next
```

```
  case W_Disp
  with t1 t2 show ?thesis
    unfolding wrap_delta_def
              wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
              wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
              wrap_buf_nonempty_end_delta_gen_def
              wrap_rewind_loop_delta_gen_def
              wrap_plant_done_delta_def
              wrap_disp_delta_gen_def wrap_run_delta_def
    by auto
next
  case (W_Run q_M)
  with non_run show ?thesis by blast
next
  case W_Rej
  with t1 show ?thesis
    unfolding wrap_delta_def
              wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
              wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
              wrap_buf_nonempty_end_delta_gen_def
              wrap_rewind_loop_delta_gen_def
              wrap_plant_done_delta_def
              wrap_disp_delta_gen_def wrap_run_delta_def
    by auto
qed
```

Once the wrap-state has the form *W_Run q*, every successor along the wrap rtrancl also has that form. By *wrap_delta_W_Run_source_only_run*, any step out of *W_Run q* comes from *wrap_run_delta*, whose target is structurally *W_Run q′*.

```
lemma wrap_W_Run_persistent:
  assumes steps: (C, C') ∈ (mttm_step (wrap_delta M pack c Σu))*
      and src:   mt_state C = W_Run q
    shows ∃ q'. mt_state C' = W_Run q'
  using steps
proof induction
  case base
  show ?case using src by blast
next
  case (step y z)
  from step.IH obtain qy where qy: mt_state y = W_Run qy by blast
  obtain y_q ts n where y_decomp: y = Config_M y_q ts n
    by (cases y)
  with qy have y_eq: y = Config_M (W_Run qy) ts n by simp
  from step.hyps(2) y_eq
  obtain qs' a' dir' where
      tuple_in: (W_Run qy, λk. ts k (n k), qs', a', dir')
                ∈ wrap_delta M pack c Σu
    and z_eq: z = Config_M qs' (λk. (ts k)(n k := a' k))
```

```
                                (λk. go_dir (dir' k) (n k))
    by (auto elim: mttm_step.cases)
  from wrap_delta_W_Run_source_only_run[OF tuple_in]
  have (W_Run qy, λk. ts k (n k), qs', a', dir')
         ∈ wrap_run_delta M Σu .
  hence ∃ q'. qs' = W_Run q'
    unfolding wrap_run_delta_def by auto
  thus ?case using z_eq by auto
qed
```

Symmetric routing exclusion at the *W_Disp* source: only *wrap_disp_delta_gen* sources from *W_Disp*.

```
lemma wrap_delta_W_Disp_source_only_disp:
  fixes M :: ('q, 'b) mttm
    and inp :: nat ⇒ ('a, 'b) wrap_alphabet
    and qs' :: ('q, 'a, 'b) wrap_state
    and a' :: nat ⇒ ('a, 'b) wrap_alphabet
    and dir :: nat ⇒ dir
  assumes (W_Disp, inp, qs', a', dir) ∈ wrap_delta M pack c Σu
  shows (W_Disp, inp, qs', a', dir) ∈ wrap_disp_delta_gen (k_tm M) M Σu
  using assms
  unfolding wrap_delta_def
            wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
            wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
            wrap_buf_nonempty_end_delta_gen_def
            wrap_rewind_loop_delta_gen_def
            wrap_plant_done_delta_def
            wrap_run_delta_def
  by auto
```

Any wrap-step from a *W_Disp* config lands in *W_Run* (*s_tm M*).

```
lemma wrap_step_W_Disp_to_W_Run:
  fixes M :: ('q, 'b) mttm
  assumes step: (C, C') ∈ mttm_step (wrap_delta M pack c Σu)
      and src:  mt_state C = W_Disp
    shows mt_state C' = W_Run (s_tm M)
proof −
  obtain c_q ts n where c_decomp: C = Config_M c_q ts n
    by (cases C)
  with src have c_eq: C = Config_M W_Disp ts n by simp
  from step c_eq obtain qs' a' dir' where
      tuple_in: (W_Disp, λk. ts k (n k), qs', a', dir')
                  ∈ wrap_delta M pack c Σu
    and dst_eq: C' = Config_M qs' (λk. (ts k)(n k := a' k))
                                  (λk. go_dir (dir' k) (n k))
    by (auto elim: mttm_step.cases)
  from wrap_delta_W_Disp_source_only_disp[OF tuple_in]
  have (W_Disp, λk. ts k (n k), qs', a', dir')
         ∈ wrap_disp_delta_gen (k_tm M) M Σu .
```

hence $qs' = W_Run\ (s_tm\ M)$
unfolding *wrap_disp_delta_gen_def* **by** *auto*
with *dst_eq* **show** *?thesis* **by** *simp*
qed

Any non-empty wrap-trace from a *W_Disp* config ends in *W_Run*: one step into *W_Run* then *wrap_W_Run_persistent*.

lemma *wrap_W_Disp_then_W_Run*:
fixes $M :: ('q, 'b)\ mttm$
assumes *path*: $(C, C') \in (mttm_step\ (wrap_delta\ M\ pack\ c\ \Sigma u))\ ^^\ n$
and *src*: $mt_state\ C = W_Disp$
and *n_pos*: $0 < n$
shows $\exists q.\ mt_state\ C' = W_Run\ q$
proof −
from *n_pos* **obtain** m **where** *n_eq*: $n = Suc\ m$
by (*cases n*) *auto*
from *path n_eq* **have** *path_Suc*:
$(C, C') \in (mttm_step\ (wrap_delta\ M\ pack\ c\ \Sigma u))\ ^^\ Suc\ m$
by *simp*
from *path_Suc* **obtain** Y **where**
first_step: $(C, Y) \in mttm_step\ (wrap_delta\ M\ pack\ c\ \Sigma u)$
and *rest*: $(Y, C') \in (mttm_step\ (wrap_delta\ M\ pack\ c\ \Sigma u))\ ^^\ m$
by (*rule relpow_Suc_E2*)
from *wrap_step_W_Disp_to_W_Run*[*OF first_step src*]
have *Y_run*: $mt_state\ Y = W_Run\ (s_tm\ M)$ **.**
from *rest* **have** *rest_rt*: $(Y, C') \in (mttm_step\ (wrap_delta\ M\ pack\ c\ \Sigma u))^*$
by (*rule relpow_imp_rtrancl*)
from *wrap_W_Run_persistent*[*OF rest_rt Y_run*]
show *?thesis* **.**
qed

Structural decomposition of an accepting wrap-trace at the first entry into the *W_Run* phase. Purely structural no forward determinism or pack contracts.

lemma *wrap_first_W_Run_decomp*:
fixes $M :: ('q, 'b)\ mttm$
assumes *trace*: $(init_config_mttm\ (encoding_wrap\ M\ pack\ c\ \Sigma u)\ (map\ Raw\ w),$
$C_acc)$
$\in (mttm_step\ (wrap_delta\ M\ pack\ c\ \Sigma u))^*$
and *accept*: $\exists q.\ mt_state\ C_acc = W_Run\ q$
shows $\exists C_pre\ C_post.$
$(init_config_mttm\ (encoding_wrap\ M\ pack\ c\ \Sigma u)\ (map\ Raw\ w),\ C_pre)$
$\in (mttm_step\ (wrap_delta\ M\ pack\ c\ \Sigma u))^*$
$\wedge\ mt_state\ C_pre = W_Disp$
$\wedge\ (C_pre, C_post) \in mttm_step\ (wrap_delta\ M\ pack\ c\ \Sigma u)$
$\wedge\ mt_state\ C_post = W_Run\ (s_tm\ M)$
$\wedge\ (C_post, C_acc) \in (mttm_step\ (wrap_delta\ M\ pack\ c\ \Sigma u))^*$
proof −
let *?init_W* = $init_config_mttm\ (encoding_wrap\ M\ pack\ c\ \Sigma u)\ (map\ Raw\ w)$

```
let ?R = mttm_step (wrap_delta M pack c Σu)
have init_state: mt_state ?init_W = W_Init by simp
from trace obtain n where pow: (?init_W, C_acc) ∈ ?R ^^ n
  using rtrancl_imp_relpow by metis
from pow accept
show ?thesis
proof (induction n arbitrary: C_acc)
  case 0
  from 0.prems(1) have C_eq: C_acc = ?init_W by simp
  from 0.prems(2) C_eq init_state show ?case by auto
next
  case (Suc m)
  from Suc.prems(1) obtain y where
      prefix: (?init_W, y) ∈ ?R ^^ m
    and last_step: (y, C_acc) ∈ ?R
    by (auto elim: relpow_Suc_E)
  show ?case
  proof (cases ∃ q. mt_state y = W_Run q)
    case True
    from Suc.IH[OF prefix True] obtain C_pre C_post where
        dec: (?init_W, C_pre) ∈ ?R*
             mt_state C_pre = W_Disp
             (C_pre, C_post) ∈ ?R
             mt_state C_post = W_Run (s_tm M)
             (C_post, y) ∈ ?R*
      by blast
    from dec(5) last_step have (C_post, C_acc) ∈ ?R*
      by (rule rtrancl_into_rtrancl)
    with dec(1-4) show ?thesis by blast
  next
    case False
    obtain y_q y_ts y_n where y_eq: y = Config_M y_q y_ts y_n
      by (cases y)
    from Suc.prems(2) obtain q_acc where
        acc_state: mt_state C_acc = W_Run q_acc by blast
    obtain Cq Cts Cn where C_eq: C_acc = Config_M Cq Cts Cn
      by (cases C_acc)
    with acc_state have C_q_eq: Cq = W_Run q_acc by simp
    from last_step y_eq C_eq obtain a' dir' where
        tuple_in: (y_q, λk. y_ts k (y_n k), Cq, a', dir')
                  ∈ wrap_delta M pack c Σu
      by (auto elim: mttm_step.cases)
    with C_q_eq have tuple_W_Run:
        (y_q, λk. y_ts k (y_n k), W_Run q_acc, a', dir')
          ∈ wrap_delta M pack c Σu
      by simp
    have non_run_y: ∀ q_M. y_q ≠ W_Run q_M
    proof (intro allI notI)
      fix q_M assume y_q = W_Run q_M
```

```
      hence mt_state y = W_Run q_M using y_eq by simp
      with False show False by blast
    qed
        from wrap_delta_W_Run_target_from_W_Disp[OF tuple_W_Run
non_run_y]
    have y_W_Disp: y_q = W_Disp
     and q_acc_s: q_acc = s_tm M
      by auto
    have y_state: mt_state y = W_Disp using y_eq y_W_Disp by simp
    have C_state: mt_state C_acc = W_Run (s_tm M)
      using acc_state q_acc_s by simp
    from prefix have prefix_rt: (?init_W, y) ∈ ?R*
      by (rule relpow_imp_rtrancl)
    have refl_C: (C_acc, C_acc) ∈ ?R* by simp
    from prefix_rt y_state last_step C_state refl_C show ?thesis by blast
  qed
 qed
qed
```

Single-step forward determinism from a non-*W_Run* source: lifts *wrap_delta_non_W_Run_det* to *mttm_step* target-config uniqueness.

```
lemma wrap_step_unique_non_run:
 assumes step1: (C, C1) ∈ mttm_step (wrap_delta M pack c Σu)
    and step2: (C, C2) ∈ mttm_step (wrap_delta M pack c Σu)
    and non_run: ∀ q_M. mt_state C ≠ W_Run q_M
  shows C1 = C2
proof −
 obtain Cq Cts Cn where C_eq: C = Config_M Cq Cts Cn by (cases C)
 from C_eq non_run have non_run_q: ∀ q_M. Cq ≠ W_Run q_M by simp
 from step1 C_eq obtain qs1' a1' dir1' where
    tuple1: (Cq, λk. Cts k (Cn k), qs1', a1', dir1') ∈ wrap_delta M pack c Σu
  and C1_eq: C1 = Config_M qs1'
              (λk. (Cts k)(Cn k := a1' k))
              (λk. go_dir (dir1' k) (Cn k))
  by (auto elim: mttm_step.cases)
 from step2 C_eq obtain qs2' a2' dir2' where
    tuple2: (Cq, λk. Cts k (Cn k), qs2', a2', dir2') ∈ wrap_delta M pack c Σu
  and C2_eq: C2 = Config_M qs2'
              (λk. (Cts k)(Cn k := a2' k))
              (λk. go_dir (dir2' k) (Cn k))
  by (auto elim: mttm_step.cases)
 from wrap_delta_non_W_Run_det[OF tuple1 tuple2 non_run_q]
 have tuple_eq: (qs1', a1', dir1') = (qs2', a2', dir2') .
 show ?thesis using C1_eq C2_eq tuple_eq by simp
qed
```

Iterated forward determinism: two same-length wrap-traces from a common source to non-*W_Run* endpoints coincide. Intermediates are automatically non-*W_Run* by *wrap_W_Run_persistent*.

```
lemma wrap_path_det_non_run:
  fixes M :: ('q, 'b) mttm
  assumes path1: (C0, C1) ∈ (mttm_step (wrap_delta M pack c Σu)) ^^ n
      and path2: (C0, C2) ∈ (mttm_step (wrap_delta M pack c Σu)) ^^ n
      and non_run_C1: ∀ q_M. mt_state C1 ≠ W_Run q_M
      and non_run_C2: ∀ q_M. mt_state C2 ≠ W_Run q_M
    shows C1 = C2
  using path1 path2 non_run_C1 non_run_C2
proof (induction n arbitrary: C1 C2)
  case 0
  thus ?case by auto
next
  case (Suc m)
  from Suc.prems(1) obtain Y1 where
      pre1: (C0, Y1) ∈ (mttm_step (wrap_delta M pack c Σu)) ^^ m
    and last1: (Y1, C1) ∈ mttm_step (wrap_delta M pack c Σu)
    by (auto elim: relpow_Suc_E)
  from Suc.prems(2) obtain Y2 where
      pre2: (C0, Y2) ∈ (mttm_step (wrap_delta M pack c Σu)) ^^ m
    and last2: (Y2, C2) ∈ mttm_step (wrap_delta M pack c Σu)
    by (auto elim: relpow_Suc_E)
  have non_run_Y1: ∀ q_M. mt_state Y1 ≠ W_Run q_M
  proof (intro allI notI)
    fix q_M assume Y1_run: mt_state Y1 = W_Run q_M
    have rt1: (Y1, C1) ∈ (mttm_step (wrap_delta M pack c Σu))*
      using last1 by (rule r_into_rtrancl)
    from wrap_W_Run_persistent[OF rt1 Y1_run]
    obtain q' where C1_run: mt_state C1 = W_Run q' by blast
    from C1_run Suc.prems(3) show False by blast
  qed
  have non_run_Y2: ∀ q_M. mt_state Y2 ≠ W_Run q_M
  proof (intro allI notI)
    fix q_M assume Y2_run: mt_state Y2 = W_Run q_M
    have rt2: (Y2, C2) ∈ (mttm_step (wrap_delta M pack c Σu))*
      using last2 by (rule r_into_rtrancl)
    from wrap_W_Run_persistent[OF rt2 Y2_run]
    obtain q' where C2_run: mt_state C2 = W_Run q' by blast
    from C2_run Suc.prems(4) show False by blast
  qed
  from Suc.IH[OF pre1 pre2 non_run_Y1 non_run_Y2]
  have Y_eq: Y1 = Y2 .
  have step_a: (Y1, C1) ∈ mttm_step (wrap_delta M pack c Σu) using last1 .
  have step_b: (Y1, C2) ∈ mttm_step (wrap_delta M pack c Σu) using last2
Y_eq by simp
  show ?case
    by (rule wrap_step_unique_non_run[OF step_a step_b non_run_Y1])
qed
```

12.2 Pack-contract forcing

Wrap-tuple family identification at full-buffer close: a *W_Buf* source with *length ws + 1 = c* reading *Raw a* on the user-tape fires only *wrap_buf_close_delta_gen*.

lemma *wrap_delta_full_buf_in_close*:
fixes *M* :: ('*q*, '*b*) *mttm*
and *inp* :: *nat* ⇒ ('*a*, '*b*) *wrap_alphabet*
and *a'* :: *nat* ⇒ ('*a*, '*b*) *wrap_alphabet*
and *dir'* :: *nat* ⇒ *dir*
assumes *tuple_in*: (*W_Buf ws*, *inp*, *qs'*, *a'*, *dir'*) ∈ *wrap_delta M pack c* Σ*u*
and *ws_full*: *length ws* + *1* = *c*
and *inp_W_User_Raw*: *inp 0* = *Raw a*
shows (*W_Buf ws*, *inp*, *qs'*, *a'*, *dir'*) ∈ *wrap_buf_close_delta_gen* (*k_tm M*) *M pack c* Σ*u*
using *tuple_in ws_full inp_W_User_Raw*
unfolding *wrap_delta_def*
wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
wrap_buf_empty_end_delta_gen_def
wrap_buf_nonempty_end_delta_gen_def
wrap_rewind_loop_delta_gen_def
wrap_plant_done_delta_def
wrap_disp_delta_gen_def wrap_run_delta_def
by *auto*

Family identification at the partial-block end-of-input point: a *W_Buf* source with non-empty partial buffer reading *Enc* (*bl_tm M*) fires only *wrap_buf_nonempty_end_delta_gen*.

lemma *wrap_delta_partial_buf_in_nonempty_end*:
fixes *M* :: ('*q*, '*b*) *mttm*
and *inp* :: *nat* ⇒ ('*a*, '*b*) *wrap_alphabet*
and *a'* :: *nat* ⇒ ('*a*, '*b*) *wrap_alphabet*
and *dir'* :: *nat* ⇒ *dir*
assumes *tuple_in*: (*W_Buf ws*, *inp*, *qs'*, *a'*, *dir'*) ∈ *wrap_delta M pack c* Σ*u*
and *ws_partial*: *length ws* < *c*
and *ws_nonempty*: *ws* ≠ []
and *inp_W_User_bl*: *inp 0* = *Enc* (*bl_tm M*)
shows (*W_Buf ws*, *inp*, *qs'*, *a'*, *dir'*) ∈ *wrap_buf_nonempty_end_delta_gen* (*k_tm M*) *M pack c* Σ*u*
using *tuple_in ws_partial ws_nonempty inp_W_User_bl*
unfolding *wrap_delta_def*
wrap_init_delta_gen_def wrap_buf_extend_delta_gen_def
wrap_buf_close_delta_gen_def wrap_buf_empty_end_delta_gen_def
wrap_rewind_loop_delta_gen_def
wrap_plant_done_delta_def
wrap_disp_delta_gen_def wrap_run_delta_def
by *auto*

Forward forcing of the close-step at a full block boundary. At

mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1) with *Suc i ∗ c ≤ length w*, any firing wrap-step forces both the pack contract *pack (take c (drop (i ∗ c) w)) ∈ Sigma_tm M* and the successor identity *mid_encoder_config_gen (k_tm M) M pack c w (Suc i)*. The canonical step is *mid_buf_close_step* (needs *2 ≤ k_tm M*).

```
lemma wrap_close_step_forces_pack:
  fixes M :: ('q, 'b) mttm
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and w_alpha: set w ⊆ Σu
      and block_full: (Suc i) ∗ c ≤ length w
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
      and step: (mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1),
C')
                ∈ mttm_step (wrap_delta M pack c Σu)
    shows pack (take c (drop (i ∗ c) w)) ∈ Sigma_tm M
        ∧ C' = mid_encoder_config_gen (k_tm M) M pack c w (Suc i)
proof −
  define cm where cm = c − 1
  let ?src = mid_buf_extending_config_gen (k_tm M) M pack c w i cm
  let ?ws = take cm (drop (i ∗ c) w)
  let ?aa = w ! (i ∗ c + cm)
  let ?packed = pack (take c (drop (i ∗ c) w))
  let ?inp = λk. mt_tape ?src k (mt_pos ?src k)

  from c_pos have c_succ: Suc cm = c unfolding cm_def by simp
  from block_full have full_block_form: i ∗ c + c ≤ length w by simp
  from full_block_form c_pos have a_idx: i ∗ c + cm < length w
    unfolding cm_def by linarith
  have ws_lt_drop: cm < length (drop (i ∗ c) w) using a_idx by simp
  have len_ws: length ?ws = cm using ws_lt_drop by simp
  have len_ws_plus_one: length ?ws + 1 = c
    using len_ws c_succ by simp
  have take_app: ?ws @ [?aa] = take c (drop (i ∗ c) w)
  proof −
    have take c (drop (i ∗ c) w) = take (Suc cm) (drop (i ∗ c) w)
      using c_succ by simp
    also have ... = take cm (drop (i ∗ c) w) @ [drop (i ∗ c) w ! cm]
      using ws_lt_drop by (simp add: take_Suc_conv_app_nth)
    also have drop (i ∗ c) w ! cm = w ! (i ∗ c + cm)
      using a_idx by simp
    finally show ?thesis by simp
  qed

  have k_pos: 0 < k_tm M using k2 by simp
  have src_state: mt_state ?src = W_Buf ?ws
    by (simp add: mid_buf_extending_config_gen_def)
```

```
  have src_pos: mt_pos ?src
      = (λt. case t of 0 ⇒ 1 + i * c + cm
                     | Suc k ⇒ (if k = 0 then 1 + i else 0))
    by (simp add: mid_buf_extending_config_gen_def fun_eq_iff)
  have src_tape: mt_tape ?src
      = (λt n. if t < k_tm M
             then (case t of
               0 ⇒ (if n = 0 then Enc (le_tm M)
                     else if n ≤ length w then Raw (w ! (n − 1))
                     else Enc (bl_tm M))
             | Suc k ⇒ (if n = 0 then Enc (le_tm M)
                     else if k = 0 ∧ n ≤ i
                       then Enc (wrap_enc pack c w ! (n − 1))
                     else Enc (bl_tm M)))
             else Enc (bl_tm M))
    by (simp add: mid_buf_extending_config_gen_def fun_eq_iff)
  have read_W_User: ?inp 0 = Raw ?aa
    using a_idx k_pos by (simp add: src_tape src_pos)

  have src_decomp: ?src = Config_M (W_Buf ?ws) (mt_tape ?src) (mt_pos ?src)
    using src_state src_pos src_tape by (cases ?src) simp

  have step_src: (?src, C') ∈ mttm_step (wrap_delta M pack c Σu)
    using step unfolding cm_def by simp
  from step_src src_decomp obtain qs' a' dir' where
      C'_step: C' = Config_M qs'
                  (λk. (mt_tape ?src k)(mt_pos ?src k := a' k))
                  (λk. go_dir (dir' k) (mt_pos ?src k))
    and tuple_in: (W_Buf ?ws, ?inp, qs', a', dir') ∈ wrap_delta M pack c Σu
    by (auto elim: mttm_step.cases)

      from wrap_delta_full_buf_in_close[OF tuple_in len_ws_plus_one
read_W_User]
  have tuple_close:
    (W_Buf ?ws, ?inp, qs', a', dir') ∈ wrap_buf_close_delta_gen (k_tm M) M
pack c Σu .

  from tuple_close obtain ws_x a_x inp_x where
      close_eq: (W_Buf ?ws, ?inp, qs', a', dir')
              = (W_Buf ws_x, inp_x, W_Buf [],
                 (λt. case t of Suc k ⇒ if k = 0 then Enc (pack (ws_x @ [a_x]))
                                        else inp_x (Suc k)
                              | 0 ⇒ inp_x 0),
                 λt. case t of 0 ⇒ dir.R
                             | Suc k ⇒ if k = 0 then dir.R else dir.N)
    and close_inp_W_User: inp_x 0 = Raw a_x
    and close_pack: pack (ws_x @ [a_x]) ∈ Sigma_tm M
    unfolding wrap_buf_close_delta_gen_def by blast
```

```
  have ws_eq: ?ws = ws_x using close_eq by simp
  have inp_eq: ?inp = inp_x using close_eq by simp
  have a_eq: ?aa = a_x
  proof −
    from inp_eq have inp_at_User: ?inp 0 = inp_x 0
      by (rule fun_cong)
    from read_W_User inp_at_User close_inp_W_User
    have Raw ?aa = Raw a_x by simp
    thus ?thesis by simp
  qed
  have pack_in: ?packed ∈ Sigma_tm M
    using close_pack ws_eq a_eq take_app by simp

  have canonical:
    (mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1),
      mid_encoder_config_gen (k_tm M) M pack c w (Suc i))
       ∈ mttm_step (wrap_delta M pack c Σu)
    by (rule mid_buf_close_step[where pack = pack,
            OF vM c_pos w_alpha block_full pack_in bl_ne_le k2])
  have non_run_src:
    ∀ q_M. mt_state (mid_buf_extending_config_gen (k_tm M) M pack c w i (c −
1)) ≠ W_Run q_M
    by (simp add: mid_buf_extending_config_gen_def)
  from wrap_step_unique_non_run[OF step canonical non_run_src]
  have C'_eq_final: C' = mid_encoder_config_gen (k_tm M) M pack c w (Suc
i) .

  from pack_in C'_eq_final show ?thesis by simp
qed
```

Forward forcing of the partial close-step at end-of-input. At *mid_buf_extending_config_gen* (*k_tm M*) *M pack c w i* (*length w* − *i* ∗ *c*) with *i* the partial block index, any firing wrap-step forces the pack contract and the successor identity *post_encoder_config_gen* (*k_tm M*) *M pack c w*. The successor is computed explicitly (no single-step canonical lemma to the partial close) and needs *2* ≤ *k_tm M* for the tape-*1* write to land in range.

```
lemma wrap_partial_close_step_forces_pack:
  fixes M :: ('q, 'b) mttm
    and i :: nat
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and w_alpha: set w ⊆ Σu
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
      and i_lt: i ∗ c < length w
      and i_partial: length w < (Suc i) ∗ c
      and i_div: i = length w div c
       and step: (mid_buf_extending_config_gen (k_tm M) M pack c w i (length
w − i ∗ c), C')
```

```
                ∈ mttm_step (wrap_delta M pack c Σu)
    shows pack (take c (drop (i * c) w)) ∈ Sigma_tm M
        ∧ C' = post_encoder_config_gen (k_tm M) M pack c w
proof −
  define kp where kp = length w − i * c
  let ?src = mid_buf_extending_config_gen (k_tm M) M pack c w i kp
  let ?tail = drop (i * c) w
  let ?inp = λk. mt_tape ?src k (mt_pos ?src k)

  from i_lt have kp_pos: 0 < kp unfolding kp_def by simp
  from i_partial have len_lt: length w < c + i * c by simp
  from i_lt len_lt have kp_lt_c: kp < c
    unfolding kp_def by linarith
  have ic_plus_kp: i * c + kp = length w
    using i_lt unfolding kp_def by simp
  from c_pos i_lt have r_lt: length w mod c < c by simp
  have r_eq: length w mod c = kp
  proof −
    have length w = (length w div c) * c + length w mod c
      by (simp add: div_mult_mod_eq)
    with i_div have length w = i * c + length w mod c by simp
    thus length w mod c = kp unfolding kp_def by linarith
  qed
  have r_pos: length w mod c ≠ 0
    using r_eq kp_pos by simp
  have tail_len: length ?tail = kp
    using ic_plus_kp by simp
  have tail_eq_take: ?tail = take kp ?tail
    using tail_len by simp
  have take_kp: take kp ?tail = ?tail
    using tail_len by simp
  have take_c_eq: take c ?tail = ?tail
    using tail_len kp_lt_c by simp
  have pack_eq: pack (take c ?tail) = pack ?tail
    using take_c_eq by simp

  have set_tail: set ?tail ⊆ Σu
    using w_alpha by (meson set_drop_subset subset_trans)
  have len_wenc: length (wrap_enc pack c w) = i + 1
    using length_wrap_enc[OF c_pos, of pack w] r_pos i_div by simp
  have wenc_q_eq: wrap_enc pack c w ! i = pack ?tail
    using wrap_enc_nth_partial[OF c_pos r_pos] i_div by simp
  have bl_in: bl_tm M ∈ Γ_tm M
    by (rule valid_mttm_blank_in_Gamma[OF vM])
  have le_in: le_tm M ∈ Γ_tm M
    by (rule valid_mttm_LE_in_Gamma[OF vM])
  have k_pos: 0 < k_tm M using k2 by simp

  have buf_take_kp: take kp (drop (i * c) w) = ?tail
```

```
  using take_kp by simp
 have src_state: mt_state ?src = W_Buf ?tail
  using buf_take_kp by (simp add: mid_buf_extending_config_gen_def)
 have src_pos: mt_pos ?src
    = (λt. case t of 0 ⇒ 1 + i * c + kp
                   | Suc k ⇒ (if k = 0 then 1 + i else 0))
  by (simp add: mid_buf_extending_config_gen_def fun_eq_iff)
 have src_tape: mt_tape ?src
    = (λt n. if t < k_tm M
          then (case t of
            0 ⇒ (if n = 0 then Enc (le_tm M)
                  else if n ≤ length w then Raw (w ! (n − 1))
                  else Enc (bl_tm M))
          | Suc k ⇒ (if n = 0 then Enc (le_tm M)
                  else if k = 0 ∧ n ≤ i
                    then Enc (wrap_enc pack c w ! (n − 1))
                  else Enc (bl_tm M)))
          else Enc (bl_tm M))
  by (simp add: mid_buf_extending_config_gen_def fun_eq_iff)
 have read_W_User: ?inp 0 = Enc (bl_tm M)
  using ic_plus_kp k_pos by (simp add: src_tape src_pos)

 have src_decomp: ?src = Config_M (W_Buf ?tail) (mt_tape ?src) (mt_pos ?src)
  using src_state src_pos src_tape by (cases ?src) simp

 have step_src: (?src, C') ∈ mttm_step (wrap_delta M pack c Σu)
  using step unfolding kp_def by simp
 from step_src src_decomp obtain qs' a' dir' where
    C'_step: C' = Config_M qs'
               (λk. (mt_tape ?src k)(mt_pos ?src k := a' k))
               (λk. go_dir (dir' k) (mt_pos ?src k))
  and tuple_in: (W_Buf ?tail, ?inp, qs', a', dir') ∈ wrap_delta M pack c Σu
  by (auto elim: mttm_step.cases)

 have tail_nonempty: ?tail ≠ []
  using tail_len kp_pos by (cases ?tail) auto
 have tail_lt_c: length ?tail < c
  using tail_len kp_lt_c by simp
 from wrap_delta_partial_buf_in_nonempty_end
     [OF tuple_in tail_lt_c tail_nonempty read_W_User]
 have tuple_neEnd:
  (W_Buf ?tail, ?inp, qs', a', dir') ∈ wrap_buf_nonempty_end_delta_gen (k_tm
M) M pack c Σu .

 from tuple_neEnd obtain ws_x inp_x where
    neEnd_eq: (W_Buf ?tail, ?inp, qs', a', dir')
            = (W_Buf ws_x, inp_x, W_Reset,
              (λt. case t of Suc k ⇒ if k = 0 then Enc (pack ws_x)
                                    else inp_x (Suc k)
```

```
                                | 0 ⇒ inp_x 0),
                        λt. case t of Suc k ⇒ if k = 0 then dir.R else dir.N
                                | 0 ⇒ dir.N)
  and neEnd_pack: pack ws_x ∈ Sigma_tm M
  unfolding wrap_buf_nonempty_end_delta_gen_def by blast
have ws_eq: ?tail = ws_x using neEnd_eq by simp
have inp_eq: ?inp = inp_x using neEnd_eq by simp
have qs'_eq: qs' = W_Reset using neEnd_eq by simp
have a'_eq:
  a' = (λt. case t of Suc k ⇒ if k = 0 then Enc (pack ws_x)
                                      else inp_x (Suc k)
                        | 0 ⇒ inp_x 0)
  using neEnd_eq by simp
have dir'_eq:
  dir' = (λt. case t of Suc k ⇒ if k = 0 then dir.R else dir.N
                        | 0 ⇒ dir.N)
  using neEnd_eq by simp
have pack_tail_in: pack ?tail ∈ Sigma_tm M
  using neEnd_pack ws_eq by simp
have pack_in: pack (take c (drop (i * c) w)) ∈ Sigma_tm M
  using pack_tail_in pack_eq by simp

have a'_concrete:
  a' = (λt. case t of 0 ⇒ ?inp 0
                    | Suc k ⇒ if k = 0 then Enc (pack ?tail)
                                      else ?inp (Suc k))
  using a'_eq ws_eq inp_eq by (auto simp: fun_eq_iff split: nat.split)
have C'_concrete:
  C' = Config_M W_Reset
        (λk. (mt_tape ?src k)
             ((mt_pos ?src k) :=
               (case k of
                  0 ⇒ ?inp 0
                | Suc k' ⇒ if k' = 0 then Enc (pack ?tail)
                                     else ?inp (Suc k'))))
        (λk. go_dir
             ((λt. case t of 0 ⇒ dir.N
                           | Suc k' ⇒ if k' = 0 then dir.R else dir.N) k)
             (mt_pos ?src k))
  using C'_step qs'_eq a'_concrete dir'_eq
  by (auto simp: fun_eq_iff split: nat.split)

have target_eq:
  Config_M W_Reset
    (λk. (mt_tape ?src k)
         ((mt_pos ?src k) :=
           (case k of
              0 ⇒ ?inp 0
            | Suc k' ⇒ if k' = 0 then Enc (pack ?tail)
```

```
                    else ?inp (Suc k'))))
      (λk. go_dir
           ((λt. case t of 0 ⇒ dir.N
                         | Suc k' ⇒ if k' = 0 then dir.R else dir.N) k)
           (mt_pos ?src k))
    = post_encoder_config_gen (k_tm M) M pack c w
   unfolding post_encoder_config_gen_def
   using ic_plus_kp len_wenc wenc_q_eq r_pos k2 valid_mttm_k_pos[OF vM]
   by (auto simp: src_pos src_tape fun_eq_iff
         split: nat.split if_splits)

  from C'_concrete target_eq pack_in
  show ?thesis by simp
qed
end
theory Wrap_Canonical
  imports Wrap_Forcing
begin
```

13 Faithful (k-tape) encoding wrap: canonical factor

The pack contracts and the canonical-factor lemma for the faithful wrap: any *accepting* wrap-trace reaches *post_plant_dispatch_config* and the per-block pack contracts hold in *Sigma_tm M*-form. These are the forward-direction obligations for *wrap_delta*, built over the forcing layer *Multi-tape_Alphabet_Enlargement.Wrap_Forcing*.

13.1 Canonical chain construction

Canonical sub-chain to *mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1)*: given pack contracts for blocks $j < i$, the wrap-trace extends through the encoder phase up to (but not including) block i's close-step. Carries $2 \le$ *k_tm M*.

```
lemma init_to_mid_buf_extending:
  fixes M :: ('q, 'b) mttm
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and w_alpha: set w ⊆ Σu
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
      and i_full: (Suc i) * c ≤ length w
      and pack_ih: ⋀j. j < i ⟹ pack (take c (drop (j * c) w)) ∈ Sigma_tm M
    shows (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
           mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1))
             ∈ (mttm_step (wrap_delta M pack c Σu))*
```

```
proof −
  let ?R = mttm_step (wrap_delta M pack c Σu)

  have init_step:
    (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
      mid_encoder_config_gen (k_tm M) M pack c w 0) ∈ ?R
    by (rule init_to_mid_zero[OF vM c_pos])

  have iter:
    j ≤ i ⟹
     (mid_encoder_config_gen (k_tm M) M pack c w 0,
      mid_encoder_config_gen (k_tm M) M pack c w j) ∈ ?R* for j
  proof (induction j)
    case 0 show ?case by simp
  next
    case (Suc j)
    from Suc.prems have j_le_i: j ≤ i by simp
    from Suc.prems have Suc_j_le_i: Suc j ≤ i by simp
    from Suc.IH[OF j_le_i]
    have ih: (mid_encoder_config_gen (k_tm M) M pack c w 0,
              mid_encoder_config_gen (k_tm M) M pack c w j) ∈ ?R* .
    have block_full: (Suc j) * c ≤ length w
    proof −
      from Suc_j_le_i have (Suc j) * c ≤ i * c
        by (rule mult_le_mono1)
      also have ... ≤ (Suc i) * c by simp
      also have ... ≤ length w using i_full .
      finally show ?thesis .
    qed
    from Suc_j_le_i have j_lt_i: j < i by simp
    from pack_ih[OF j_lt_i]
    have pack_block_j: pack (take c (drop (j * c) w)) ∈ Sigma_tm M .
    have step:
      (mid_encoder_config_gen (k_tm M) M pack c w j,
        mid_encoder_config_gen (k_tm M) M pack c w (Suc j)) ∈ ?R*
      by (rule mid_encoder_step[where pack = pack,
            OF vM c_pos w_alpha block_full pack_block_j bl_ne_le k2])
    from ih step show ?case by (rule rtrancl_trans)
  qed

  have to_mid_i: (mid_encoder_config_gen (k_tm M) M pack c w 0,
                  mid_encoder_config_gen (k_tm M) M pack c w i) ∈ ?R*
    using iter[of i] by simp

  have boundary:
    mid_encoder_config_gen (k_tm M) M pack c w i
      = mid_buf_extending_config_gen (k_tm M) M pack c w i 0
                    by (simp add: mid_encoder_config_gen_def
mid_buf_extending_config_gen_def
```

```
        fun_eq_iff split: nat.split)

have extend_iter:
  k < c ⟹
   (mid_buf_extending_config_gen (k_tm M) M pack c w i 0,
    mid_buf_extending_config_gen (k_tm M) M pack c w i k) ∈ ?R* for k
proof (induction k)
  case 0 show ?case by simp
next
  case (Suc k)
  from Suc.prems have Suc_k_lt_c: Suc k < c by simp
  from Suc_k_lt_c have k_lt_c: k < c by simp
  from Suc.IH[OF k_lt_c]
  have ih_k: (mid_buf_extending_config_gen (k_tm M) M pack c w i 0,
              mid_buf_extending_config_gen (k_tm M) M pack c w i k) ∈ ?R* .
  have ic_k_lt: i * c + k < length w
  proof −
    have i * c + k < i * c + c using k_lt_c by simp
    also have ... = (Suc i) * c by simp
    also have ... ≤ length w using i_full .
    finally show ?thesis .
  qed
  have step:
    (mid_buf_extending_config_gen (k_tm M) M pack c w i k,
      mid_buf_extending_config_gen (k_tm M) M pack c w i (Suc k)) ∈ ?R
   by (rule mid_buf_extend_step[OF vM c_pos w_alpha Suc_k_lt_c ic_k_lt])
  from ih_k step show ?case
    by (rule rtrancl_into_rtrancl)
qed

have to_extending:
  (mid_buf_extending_config_gen (k_tm M) M pack c w i 0,
    mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1)) ∈ ?R*
  using extend_iter[of c − 1] c_pos by simp

from init_step have init_rt:
  (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
    mid_encoder_config_gen (k_tm M) M pack c w 0) ∈ ?R*
  by (rule r_into_rtrancl)
from init_rt to_mid_i
have init_to_mid_i:
  (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
    mid_encoder_config_gen (k_tm M) M pack c w i) ∈ ?R*
  by (rule rtrancl_trans)
from to_extending boundary
have mid_to_ext:
  (mid_encoder_config_gen (k_tm M) M pack c w i,
    mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1)) ∈ ?R*
  by simp
```

```
  from init_to_mid_i mid_to_ext
  show ?thesis by (rule rtrancl_trans)
qed
```

Canonical sub-chain to the partial-block end-of-input config. Given pack contracts for every full block $j < i$, the trace extends through the encoder up to the partial block's last extend, stopping right before the partial close.

```
lemma init_to_mid_buf_partial:
  fixes M :: ('q, 'b) mttm
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and w_alpha: set w ⊆ Σu
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
      and i_lt: i * c < length w
      and i_partial: length w < (Suc i) * c
      and pack_ih: ⋀j. j < i ⟹ pack (take c (drop (j * c) w)) ∈ Sigma_tm M
    shows (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
           mid_buf_extending_config_gen (k_tm M) M pack c w i (length w − i
* c))
            ∈ (mttm_step (wrap_delta M pack c Σu))*
proof −
  let ?R = mttm_step (wrap_delta M pack c Σu)
  define kp where kp = length w − i * c

  from i_lt have kp_pos: 0 < kp unfolding kp_def by simp
  from i_partial have len_lt: length w < c + i * c by simp
  from i_lt len_lt have kp_lt_c: kp < c
    unfolding kp_def by linarith
  have ic_plus_kp: i * c + kp = length w
    using i_lt unfolding kp_def by simp

  have init_step:
    (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
      mid_encoder_config_gen (k_tm M) M pack c w 0) ∈ ?R
    by (rule init_to_mid_zero[OF vM c_pos])

  have iter:
    j ≤ i ⟹
     (mid_encoder_config_gen (k_tm M) M pack c w 0,
      mid_encoder_config_gen (k_tm M) M pack c w j) ∈ ?R* for j
  proof (induction j)
    case 0 show ?case by simp
  next
    case (Suc j)
    from Suc.prems have j_le_i: j ≤ i by simp
    from Suc.prems have Suc_j_le_i: Suc j ≤ i by simp
    from Suc.IH[OF j_le_i]
    have ih: (mid_encoder_config_gen (k_tm M) M pack c w 0,
```

```
        mid_encoder_config_gen (k_tm M) M pack c w j) ∈ ?R* .
  have block_full: (Suc j) * c ≤ length w
  proof −
    from Suc_j_le_i have (Suc j) * c ≤ i * c
      by (rule mult_le_mono1)
    also have ... ≤ length w using i_lt by simp
    finally show ?thesis .
  qed
  from Suc_j_le_i have j_lt_i: j < i by simp
  from pack_ih[OF j_lt_i]
  have pack_block_j: pack (take c (drop (j * c) w)) ∈ Sigma_tm M .
  have step:
    (mid_encoder_config_gen (k_tm M) M pack c w j,
      mid_encoder_config_gen (k_tm M) M pack c w (Suc j)) ∈ ?R*
    by (rule mid_encoder_step[where pack = pack,
          OF vM c_pos w_alpha block_full pack_block_j bl_ne_le k2])
  from ih step show ?case by (rule rtrancl_trans)
qed

have to_mid_i: (mid_encoder_config_gen (k_tm M) M pack c w 0,
                mid_encoder_config_gen (k_tm M) M pack c w i) ∈ ?R*
  using iter[of i] by simp

have boundary:
  mid_encoder_config_gen (k_tm M) M pack c w i
    = mid_buf_extending_config_gen (k_tm M) M pack c w i 0
                    by (simp add: mid_encoder_config_gen_def
mid_buf_extending_config_gen_def
              fun_eq_iff split: nat.split)

have extend_iter:
  k ≤ kp ⟹
   (mid_buf_extending_config_gen (k_tm M) M pack c w i 0,
    mid_buf_extending_config_gen (k_tm M) M pack c w i k) ∈ ?R* for k
proof (induction k)
  case 0 show ?case by simp
next
  case (Suc k)
  from Suc.prems have Suc_k_le_kp: Suc k ≤ kp by simp
  from Suc_k_le_kp have k_le_kp: k ≤ kp by simp
  from Suc.IH[OF k_le_kp]
  have ih_k: (mid_buf_extending_config_gen (k_tm M) M pack c w i 0,
              mid_buf_extending_config_gen (k_tm M) M pack c w i k) ∈ ?R* .
  from Suc_k_le_kp kp_lt_c have Suc_k_lt_c: Suc k < c by simp
  from Suc_k_le_kp ic_plus_kp have ic_k_lt: i * c + k < length w
    by linarith
  have step:
    (mid_buf_extending_config_gen (k_tm M) M pack c w i k,
      mid_buf_extending_config_gen (k_tm M) M pack c w i (Suc k)) ∈ ?R
```

```
    by (rule mid_buf_extend_step[OF vM c_pos w_alpha Suc_k_lt_c ic_k_lt])
  from ih_k step show ?case
    by (rule rtrancl_into_rtrancl)
qed

have to_extending:
  (mid_buf_extending_config_gen (k_tm M) M pack c w i 0,
    mid_buf_extending_config_gen (k_tm M) M pack c w i kp) ∈ ?R*
  using extend_iter[of kp] by simp

from init_step have init_rt:
  (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
    mid_encoder_config_gen (k_tm M) M pack c w 0) ∈ ?R*
  by (rule r_into_rtrancl)
from init_rt to_mid_i
have init_to_mid_i:
  (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
    mid_encoder_config_gen (k_tm M) M pack c w i) ∈ ?R*
  by (rule rtrancl_trans)
from to_extending boundary
have mid_to_ext:
  (mid_encoder_config_gen (k_tm M) M pack c w i,
    mid_buf_extending_config_gen (k_tm M) M pack c w i kp) ∈ ?R*
  by simp
from init_to_mid_i mid_to_ext
have (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
      mid_buf_extending_config_gen (k_tm M) M pack c w i kp) ∈ ?R*
  by (rule rtrancl_trans)
thus ?thesis unfolding kp_def .
qed
```

13.2 Pack contracts by strong induction

Pack contracts for full blocks via strong induction: given an accepting wrap-trace, every full block i satisfies $pack\ (take\ c\ (drop\ (i * c)\ w)) \in Sigma_tm\ M$.

```
lemma wrap_full_block_pack_contract:
  fixes M :: ('q, 'b) mttm
    and i :: nat
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and bl_ne_le: bl_tm M ≠ le_tm M
      and w_alpha: set w ⊆ Σu
      and k2: 2 ≤ k_tm M
      and trace: (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
C_acc)
                 ∈ (mttm_step (wrap_delta M pack c Σu))*
      and accept: ∃q. mt_state C_acc = W_Run q
      and i_full: (Suc i) * c ≤ length w
```

```
    shows pack (take c (drop (i * c) w)) ∈ Sigma_tm M
proof −
  let ?R = mttm_step (wrap_delta M pack c Σu)
  let ?init = init_config_mttm (encoding_wrap M pack c Σu) (map Raw w)

  from wrap_first_W_Run_decomp[OF trace accept]
  obtain C_pre C_post where
      pre_chain: (?init, C_pre) ∈ ?R*
    and pre_state: mt_state C_pre = W_Disp
    and pre_step: (C_pre, C_post) ∈ ?R
    and post_state: mt_state C_post = W_Run (s_tm M)
    and post_chain: (C_post, C_acc) ∈ ?R*
    by blast

  have C_pre_non_run: ∀ q. mt_state C_pre ≠ W_Run q
    by (simp add: pre_state)

  from pre_chain obtain T_pre where pre_pow:
    (?init, C_pre) ∈ ?R ^^ T_pre
    by (auto dest: rtrancl_imp_relpow)

  from i_full
  show ?thesis
  proof (induction i rule: less_induct)
    case (less i)
    note i_full_curr = less.prems

    have pack_ih: ⋀j. j < i ⟹ pack (take c (drop (j * c) w)) ∈ Sigma_tm M
    proof −
      fix j assume j_lt: j < i
      have j_full: (Suc j) * c ≤ length w
      proof −
        from j_lt have Suc j ≤ i by simp
        hence (Suc j) * c ≤ i * c by (rule mult_le_mono1)
        also have ... ≤ (Suc i) * c by simp
        also have ... ≤ length w using i_full_curr .
        finally show ?thesis .
      qed
      from less.IH[OF j_lt j_full]
      show pack (take c (drop (j * c) w)) ∈ Sigma_tm M .
    qed

    have chain_i:
      (?init, mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1)) ∈
?R*
      by (rule init_to_mid_buf_extending
            [where pack = pack,
             OF vM c_pos w_alpha bl_ne_le k2 i_full_curr pack_ih])
    from chain_i obtain N_i where chain_pow:
```

```
    (?init, mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1)) ∈
?R ^^ N_i
    by (auto dest: rtrancl_imp_relpow)

  have end_state:
    mt_state (mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1))
      = W_Buf (take (c − 1) (drop (i ∗ c) w))
    by (simp add: mid_buf_extending_config_gen_def)
  have end_non_run:
   ∀ q. mt_state (mid_buf_extending_config_gen (k_tm M) M pack c w i (c −
1)) ≠ W_Run q
    using end_state by auto

  have T_pre_ge: N_i ≤ T_pre
  proof (rule ccontr)
    assume ¬ N_i ≤ T_pre
    hence T_lt: T_pre < N_i by simp
    have add_eq: T_pre + (N_i − T_pre) = N_i
      using T_lt by simp
    have rel_add:
     (?R ^^ T_pre) O (?R ^^ (N_i − T_pre)) = ?R ^^ N_i
     using add_eq by (metis relpow_add)
    from chain_pow rel_add
     have (?init, mid_buf_extending_config_gen (k_tm M) M pack c w i (c −
1))
          ∈ (?R ^^ T_pre) O (?R ^^ (N_i − T_pre))
     by simp
    then obtain Z where
       Z_path: (?init, Z) ∈ ?R ^^ T_pre
     and Z_to_end: (Z, mid_buf_extending_config_gen (k_tm M) M pack c w
i (c − 1))
                   ∈ ?R ^^ (N_i − T_pre)
     by auto

    have Z_to_end_rt:
     (Z, mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1)) ∈ ?R*
     using Z_to_end by (rule relpow_imp_rtrancl)
    have Z_non_run: ∀ q. mt_state Z ≠ W_Run q
    proof (intro allI notI)
     fix q assume Z_run: mt_state Z = W_Run q
     from wrap_W_Run_persistent[OF Z_to_end_rt Z_run]
     obtain q′ where
       mt_state (mid_buf_extending_config_gen (k_tm M) M pack c w i (c −
1)) = W_Run q′
       by blast
     with end_non_run show False by blast
    qed

          from wrap_path_det_non_run[OF Z_path pre_pow Z_non_run
```

```
C_pre_non_run]
    have Z_eq: Z = C_pre .
    have Z_state: mt_state Z = W_Disp using Z_eq pre_state by simp
    have N_i_gt: 0 < N_i - T_pre using T_lt by simp
    from wrap_W_Disp_then_W_Run[OF Z_to_end Z_state N_i_gt]
    obtain q' where
      mt_state (mid_buf_extending_config_gen (k_tm M) M pack c w i (c -
1)) = W_Run q'
      by blast
    with end_non_run show False by blast
  qed

  from T_pre_ge obtain T_suffix where T_split: T_pre = N_i + T_suffix
    by (metis le_iff_add)
  have rel_add_pre:
    (?R ^^ N_i) O (?R ^^ T_suffix) = ?R ^^ T_pre
    using T_split by (metis relpow_add)
  from pre_pow rel_add_pre
  have (?init, C_pre) ∈ (?R ^^ N_i) O (?R ^^ T_suffix)
    by simp
  then obtain Y_N where
    init_to_Y: (?init, Y_N) ∈ ?R ^^ N_i
    and Y_to_pre: (Y_N, C_pre) ∈ ?R ^^ T_suffix
    by auto

  have Y_to_pre_rt: (Y_N, C_pre) ∈ ?R*
    using Y_to_pre by (rule relpow_imp_rtrancl)
  have Y_N_non_run: ∀ q. mt_state Y_N ≠ W_Run q
  proof (intro allI notI)
    fix q assume Y_N_run: mt_state Y_N = W_Run q
    from wrap_W_Run_persistent[OF Y_to_pre_rt Y_N_run]
    obtain q' where mt_state C_pre = W_Run q' by blast
    with C_pre_non_run show False by blast
  qed
    from wrap_path_det_non_run[OF init_to_Y chain_pow Y_N_non_run
end_non_run]
  have Y_N_eq: Y_N = mid_buf_extending_config_gen (k_tm M) M pack c
w i (c - 1) .

  have T_suffix_pos: 0 < T_suffix
  proof (rule ccontr)
    assume ¬ 0 < T_suffix
    hence T_suff_0: T_suffix = 0 by simp
    from Y_to_pre T_suff_0 have Y_eq_pre: Y_N = C_pre by simp
    from Y_eq_pre Y_N_eq pre_state
    have mt_state (mid_buf_extending_config_gen (k_tm M) M pack c w i (c
- 1)) = W_Disp
      by simp
    with end_state show False by simp
```

```
    qed

    from T_suffix_pos obtain T' where T_suff_eq: T_suffix = Suc T'
      by (cases T_suffix) auto
    from Y_to_pre T_suff_eq
    have Y_to_pre_Suc: (Y_N, C_pre) ∈ ?R ^^ Suc T' by simp
    from Y_to_pre_Suc obtain Z_next where
        Y_to_Z: (Y_N, Z_next) ∈ ?R
      and Z_to_pre: (Z_next, C_pre) ∈ ?R ^^ T'
      by (rule relpow_Suc_E2)

    have step_close:
      (mid_buf_extending_config_gen (k_tm M) M pack c w i (c − 1), Z_next)
∈ ?R
      using Y_to_Z Y_N_eq by simp
    from wrap_close_step_forces_pack
        [OF vM c_pos w_alpha i_full_curr bl_ne_le k2 step_close]
    have pack (take c (drop (i * c) w)) ∈ Sigma_tm M
        ∧ Z_next = mid_encoder_config_gen (k_tm M) M pack c w (Suc i) .
    thus ?case by simp
  qed
qed
```

Pack contract for the partial block, by single-shot argument.

```
lemma wrap_partial_block_pack_contract:
  fixes M :: ('q, 'b) mttm
    and i :: nat
  assumes vM: valid_mttm M
      and c_pos: 0 < c
      and bl_ne_le: bl_tm M ≠ le_tm M
      and w_alpha: set w ⊆ Σu
      and k2: 2 ≤ k_tm M
      and trace: (init_config_mttm (encoding_wrap M pack c Σu) (map Raw w),
C_acc)
                 ∈ (mttm_step (wrap_delta M pack c Σu))*
      and accept: ∃ q. mt_state C_acc = W_Run q
      and i_lt: i * c < length w
      and i_partial: length w < (Suc i) * c
      and i_div: i = length w div c
      and pack_ih: ⋀j. j < i ⟹ pack (take c (drop (j * c) w)) ∈ Sigma_tm M
    shows pack (take c (drop (i * c) w)) ∈ Sigma_tm M
proof −
  let ?R = mttm_step (wrap_delta M pack c Σu)
  let ?init = init_config_mttm (encoding_wrap M pack c Σu) (map Raw w)
  define kp where kp = length w − i * c

  from wrap_first_W_Run_decomp[OF trace accept]
  obtain C_pre C_post where
      pre_chain: (?init, C_pre) ∈ ?R*
```

```
    and pre_state: mt_state C_pre = W_Disp
    and pre_step: (C_pre, C_post) ∈ ?R
    and post_state: mt_state C_post = W_Run (s_tm M)
    and post_chain: (C_post, C_acc) ∈ ?R*
    by blast

  have C_pre_non_run: ∀ q. mt_state C_pre ≠ W_Run q
    by (simp add: pre_state)

  from pre_chain obtain T_pre where pre_pow:
    (?init, C_pre) ∈ ?R ^^ T_pre
    by (auto dest: rtrancl_imp_relpow)

  have chain_i:
    (?init, mid_buf_extending_config_gen (k_tm M) M pack c w i kp) ∈ ?R*
    using init_to_mid_buf_partial
          [where pack = pack,
           OF vM c_pos w_alpha bl_ne_le k2 i_lt i_partial pack_ih]
    unfolding kp_def .
  from chain_i obtain N_i where chain_pow:
    (?init, mid_buf_extending_config_gen (k_tm M) M pack c w i kp) ∈ ?R ^^
N_i
    by (auto dest: rtrancl_imp_relpow)

  have end_state:
    mt_state (mid_buf_extending_config_gen (k_tm M) M pack c w i kp)
      = W_Buf (take kp (drop (i * c) w))
    by (simp add: mid_buf_extending_config_gen_def)
  have end_non_run:
    ∀ q. mt_state (mid_buf_extending_config_gen (k_tm M) M pack c w i kp) ≠
W_Run q
    using end_state by auto

  have T_pre_ge: N_i ≤ T_pre
  proof (rule ccontr)
    assume ¬ N_i ≤ T_pre
    hence T_lt: T_pre < N_i by simp
    have add_eq: T_pre + (N_i - T_pre) = N_i
      using T_lt by simp
    have rel_add:
      (?R ^^ T_pre) O (?R ^^ (N_i - T_pre)) = ?R ^^ N_i
      using add_eq by (metis relpow_add)
    from chain_pow rel_add
    have (?init, mid_buf_extending_config_gen (k_tm M) M pack c w i kp)
          ∈ (?R ^^ T_pre) O (?R ^^ (N_i - T_pre))
      by simp
    then obtain Z where
       Z_path: (?init, Z) ∈ ?R ^^ T_pre
      and Z_to_end: (Z, mid_buf_extending_config_gen (k_tm M) M pack c w i
```

```
kp)
                    ∈ ?R ^^ (N_i - T_pre)
    by auto

  have Z_to_end_rt:
    (Z, mid_buf_extending_config_gen (k_tm M) M pack c w i kp) ∈ ?R*
    using Z_to_end by (rule relpow_imp_rtrancl)
  have Z_non_run: ∀ q. mt_state Z ≠ W_Run q
  proof (intro allI notI)
    fix q assume Z_run: mt_state Z = W_Run q
    from wrap_W_Run_persistent[OF Z_to_end_rt Z_run]
    obtain q' where
      mt_state (mid_buf_extending_config_gen (k_tm M) M pack c w i kp) =
W_Run q'
      by blast
    with end_non_run show False by blast
  qed

        from wrap_path_det_non_run[OF Z_path pre_pow Z_non_run
C_pre_non_run]
  have Z_eq: Z = C_pre .
  have Z_state: mt_state Z = W_Disp using Z_eq pre_state by simp
  have N_i_gt: 0 < N_i - T_pre using T_lt by simp
  from wrap_W_Disp_then_W_Run[OF Z_to_end Z_state N_i_gt]
  obtain q' where
    mt_state (mid_buf_extending_config_gen (k_tm M) M pack c w i kp) =
W_Run q'
    by blast
  with end_non_run show False by blast
 qed

 from T_pre_ge obtain T_suffix where T_split: T_pre = N_i + T_suffix
   by (metis le_iff_add)
 have rel_add_pre:
   (?R ^^ N_i) O (?R ^^ T_suffix) = ?R ^^ T_pre
   using T_split by (metis relpow_add)
 from pre_pow rel_add_pre
 have (?init, C_pre) ∈ (?R ^^ N_i) O (?R ^^ T_suffix)
   by simp
 then obtain Y_N where
    init_to_Y: (?init, Y_N) ∈ ?R ^^ N_i
   and Y_to_pre: (Y_N, C_pre) ∈ ?R ^^ T_suffix
   by auto

 have Y_to_pre_rt: (Y_N, C_pre) ∈ ?R*
   using Y_to_pre by (rule relpow_imp_rtrancl)
 have Y_N_non_run: ∀ q. mt_state Y_N ≠ W_Run q
 proof (intro allI notI)
   fix q assume Y_N_run: mt_state Y_N = W_Run q
```

```
    from wrap_W_Run_persistent[OF Y_to_pre_rt Y_N_run]
    obtain q' where mt_state C_pre = W_Run q' by blast
    with C_pre_non_run show False by blast
  qed
    from wrap_path_det_non_run[OF init_to_Y chain_pow Y_N_non_run
end_non_run]
  have Y_N_eq: Y_N = mid_buf_extending_config_gen (k_tm M) M pack c w
i kp .

  have T_suffix_pos: 0 < T_suffix
  proof (rule ccontr)
    assume ¬ 0 < T_suffix
    hence T_suff_0: T_suffix = 0 by simp
    from Y_to_pre T_suff_0 have Y_eq_pre: Y_N = C_pre by simp
    from Y_eq_pre Y_N_eq pre_state
     have mt_state (mid_buf_extending_config_gen (k_tm M) M pack c w i kp)
= W_Disp
      by simp
    with end_state show False by simp
  qed

  from T_suffix_pos obtain T' where T_suff_eq: T_suffix = Suc T'
    by (cases T_suffix) auto
  from Y_to_pre T_suff_eq
  have Y_to_pre_Suc: (Y_N, C_pre) ∈ ?R ^^ Suc T' by simp
  from Y_to_pre_Suc obtain Z_next where
      Y_to_Z: (Y_N, Z_next) ∈ ?R
    and Z_to_pre: (Z_next, C_pre) ∈ ?R ^^ T'
    by (rule relpow_Suc_E2)

  have step_close:
    (mid_buf_extending_config_gen (k_tm M) M pack c w i kp, Z_next) ∈ ?R
    using Y_to_Z Y_N_eq by simp
  have step_close_kp:
    (mid_buf_extending_config_gen (k_tm M) M pack c w i (length w − i * c),
Z_next) ∈ ?R
    using step_close unfolding kp_def .
  from wrap_partial_close_step_forces_pack
       [OF vM c_pos w_alpha bl_ne_le k2 i_lt i_partial i_div step_close_kp]
  have pack (take c (drop (i * c) w)) ∈ Sigma_tm M
       ∧ Z_next = post_encoder_config_gen (k_tm M) M pack c w .
  thus ?thesis by simp
qed
```

13.3 The canonical-factor lemma

Any accepting wrap-trace reaches the floated *post_plant_dispatch_config* and the per-block pack contracts hold. The canonical chain to the *W_Disp* config *mid_plant_disp_config* goes through the plant reset

(*plant_rewind_loop_relpow* + *plant_done_step*), and the dispatch step is *plant_disp_step*.

lemma *wrap_accept_canonical_factor*:
  fixes M :: ('*q*, '*b*) *mttm*
  assumes *vM*: *valid_mttm M*
    and *c_pos*: *0* < *c*
    and *bl_ne_le*: *bl_tm M* ≠ *le_tm M*
    and *w_alpha*: *set w* ⊆ Σ*u*
    and *k2*: *2* ≤ *k_tm M*
    and *trace*: (*init_config_mttm* (*encoding_wrap M pack c* Σ*u*) (*map Raw w*),
          C_acc)
           ∈ (*mttm_step* (*wrap_delta M pack c* Σ*u*))*
    and *accept_state*: ∃ *q*. *mt_state C_acc* = *W_Run q*
  shows (*post_plant_dispatch_config M pack c w*, *C_acc*)
        ∈ (*mttm_step* (*wrap_delta M pack c* Σ*u*))*
    ∧ (∀ *i*. *i* * *c* < *length w*
        ⟶ *pack* (*take c* (*drop* (*i* * *c*) *w*)) ∈ *Sigma_tm M*)
proof −
  let *?R* = *mttm_step* (*wrap_delta M pack c* Σ*u*)
  let *?init* = *init_config_mttm* (*encoding_wrap M pack c* Σ*u*) (*map Raw w*)

  obtain *Q* Σ*i* Γ*set bl le* δ*M sM tM rM kM* **where** *M_eq*:
    M = *MTTM Q* Σ*i* Γ*set bl le* δ*M sM tM rM kM*
    by (*cases M*)
  have *Sigma_sub_Gamma*: *Sigma_tm M* ⊆ Γ*_tm M* **using** *vM M_eq* **by** *auto*
  have *le_not_in_Sigma*: *le_tm M* ∉ *Sigma_tm M* **using** *vM M_eq* **by** *auto*

  — Pack contracts: full blocks via *wrap_full_block_pack_contract*, the partial block via *wrap_partial_block_pack_contract*.
  have *pack_in_Sigma*:
    ⋀*i*. *i* * *c* < *length w* ⟹ *pack* (*take c* (*drop* (*i* * *c*) *w*)) ∈ *Sigma_tm M*
  proof −
    fix *i* :: *nat*
    assume *i_lt*: *i* * *c* < *length w*
    show *pack* (*take c* (*drop* (*i* * *c*) *w*)) ∈ *Sigma_tm M*
    proof (*cases* (*Suc i*) * *c* ≤ *length w*)
      case *True*
      show *?thesis*
        by (*rule wrap_full_block_pack_contract*
            [**where** *pack* = *pack*,
             OF vM c_pos bl_ne_le w_alpha k2 trace accept_state True])
    next
      case *False*
      hence *i_partial*: *length w* < (*Suc i*) * *c* **by** *simp*
      have *i_div*: *i* = *length w div c*
      proof −
        have *lo*: *i* ≤ *length w div c*
        proof −
          from *i_lt* **have** *i* * *c* ≤ *length w* **by** *simp*

```
      hence (i * c) div c ≤ length w div c by (rule div_le_mono)
      with c_pos show i ≤ length w div c by simp
    qed
    have hi: length w div c ≤ i
    proof −
      from i_partial c_pos have length w div c < Suc i
        by (simp add: div_less_iff_less_mult mult.commute)
      thus length w div c ≤ i by simp
    qed
    from lo hi show ?thesis by simp
  qed
  have pack_ih:
    ⋀j. j < i ⟹ pack (take c (drop (j * c) w)) ∈ Sigma_tm M
  proof −
    fix j assume j_lt: j < i
    have j_full: (Suc j) * c ≤ length w
    proof −
      from j_lt have Suc j ≤ i by simp
      hence (Suc j) * c ≤ i * c by (rule mult_le_mono1)
      also have ... ≤ length w using i_lt by simp
      finally show ?thesis .
    qed
    show pack (take c (drop (j * c) w)) ∈ Sigma_tm M
      by (rule wrap_full_block_pack_contract
            [where pack = pack,
             OF vM c_pos bl_ne_le w_alpha k2 trace accept_state j_full])
  qed
  show ?thesis
    by (rule wrap_partial_block_pack_contract
          [where pack = pack,
           OF vM c_pos bl_ne_le w_alpha k2 trace accept_state
             i_lt i_partial i_div pack_ih])
 qed
qed

have pack_in_Gamma:
  ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ∈ Γ_tm M
proof −
  fix i :: nat assume i_lt: i * c < length w
  from pack_in_Sigma[OF i_lt] Sigma_sub_Gamma
  show pack (take c (drop (i * c) w)) ∈ Γ_tm M by auto
qed
have pack_ne_le:
  ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ≠ le_tm M
proof −
  fix i :: nat assume i_lt: i * c < length w
  from pack_in_Sigma[OF i_lt] le_not_in_Sigma
  show pack (take c (drop (i * c) w)) ≠ le_tm M by auto
qed
```

— Canonical chain to *mid_plant_disp_config*: encoder, plant storage rewind (input head frozen), plant-done.
have *enc_step*:
(*?init*, *post_encoder_config_gen* (*k_tm M*) *M pack c w*) ∈ *?R**
by (*rule encoder_phase_terminates*
[**where** *pack* = *pack*,
OF vM c_pos w_alpha bl_ne_le k2 pack_in_Sigma])
have *loop_relpow*:
(*post_encoder_config_gen* (*k_tm M*) *M pack c w*,
mid_reset_combined_config_gen (*k_tm M*) *M pack c w* (*length w* + *1*) *0*)
∈ *?R* ^^ (*Suc* (*length* (*wrap_enc pack c w*)))
apply (*rule plant_rewind_loop_relpow*[*OF vM c_pos w_alpha bl_ne_le k2*])
apply (*fact pack_in_Gamma*)
apply (*fact pack_ne_le*)
done
have *loop_step*:
(*post_encoder_config_gen* (*k_tm M*) *M pack c w*,
mid_reset_combined_config_gen (*k_tm M*) *M pack c w* (*length w* + *1*) *0*) ∈ *?R**
using *loop_relpow* **by** (*rule relpow_imp_rtrancl*)
have *done_step*:
(*mid_reset_combined_config_gen* (*k_tm M*) *M pack c w* (*length w* + *1*) *0*,
mid_plant_disp_config M pack c w) ∈ *?R*
by (*rule plant_done_step*[*OF vM k2*])
have *done_step_rt*:
(*mid_reset_combined_config_gen* (*k_tm M*) *M pack c w* (*length w* + *1*) *0*,
mid_plant_disp_config M pack c w) ∈ *?R**
using *done_step* **by** (*rule r_into_rtrancl*)
from *enc_step loop_step* **have** *init_to_loop*:
(*?init*, *mid_reset_combined_config_gen* (*k_tm M*) *M pack c w* (*length w* + *1*) *0*) ∈ *?R**
by (*rule rtrancl_trans*)
from *init_to_loop done_step_rt* **have** *canon_to_disp*:
(*?init*, *mid_plant_disp_config M pack c w*) ∈ *?R**
by (*rule rtrancl_trans*)

have *canon_disp_step*:
(*mid_plant_disp_config M pack c w*, *post_plant_dispatch_config M pack c w*) ∈ *?R*
by (*rule plant_disp_step*[*OF vM k2*])

from *wrap_first_W_Run_decomp*[*OF trace accept_state*]
obtain *C_pre C_post* **where**
pre_chain: (*?init*, *C_pre*) ∈ *?R**
and *pre_state*: *mt_state C_pre* = *W_Disp*
and *pre_step*: (*C_pre*, *C_post*) ∈ *?R*
and *post_state*: *mt_state C_post* = *W_Run* (*s_tm M*)
and *post_chain*: (*C_post*, *C_acc*) ∈ *?R**

by *blast*

have *C_pre_non_run*: $\forall\, q.$ *mt_state C_pre* $\neq$ *W_Run q*
by (*simp add*: *pre_state*)
have *disp_state*:
mt_state (*mid_plant_disp_config M pack c w*) = *W_Disp*
by (*simp add*: *mid_plant_disp_config_def*)
have *disp_non_run*:
$\forall\, q.$ *mt_state* (*mid_plant_disp_config M pack c w*) $\neq$ *W_Run q*
by (*simp add*: *disp_state*)

from *pre_chain* **obtain** *T_pre* **where** *pre_pow*:
(*?init*, *C_pre*) $\in$ *?R* $^{\frown\frown}$ *T_pre*
by (*auto dest*: *rtrancl_imp_relpow*)
from *canon_to_disp* **obtain** *N* **where** *canon_pow*:
(*?init*, *mid_plant_disp_config M pack c w*) $\in$ *?R* $^{\frown\frown}$ *N*
by (*auto dest*: *rtrancl_imp_relpow*)

have *T_pre_ge*: *N* $\leq$ *T_pre*
proof (*rule ccontr*)
assume $\neg$ *N* $\leq$ *T_pre*
hence *T_lt*: *T_pre* < *N* **by** *simp*
have *add_eq*: *T_pre* + (*N* − *T_pre*) = *N* **using** *T_lt* **by** *simp*
have *rel_add*:
(*?R* $^{\frown\frown}$ *T_pre*) *O* (*?R* $^{\frown\frown}$ (*N* − *T_pre*)) = *?R* $^{\frown\frown}$ *N*
using *add_eq* **by** (*metis relpow_add*)
from *canon_pow rel_add*
have (*?init*, *mid_plant_disp_config M pack c w*)
$\in$ (*?R* $^{\frown\frown}$ *T_pre*) *O* (*?R* $^{\frown\frown}$ (*N* − *T_pre*))
by *simp*
then obtain *Z* **where**
Z_path: (*?init*, *Z*) $\in$ *?R* $^{\frown\frown}$ *T_pre*
and *Z_to_disp*: (*Z*, *mid_plant_disp_config M pack c w*) $\in$ *?R* $^{\frown\frown}$ (*N* − *T_pre*)
by *auto*
have *Z_to_disp_rt*: (*Z*, *mid_plant_disp_config M pack c w*) $\in$ *?R**
using *Z_to_disp* **by** (*rule relpow_imp_rtrancl*)
have *Z_non_run*: $\forall\, q.$ *mt_state Z* $\neq$ *W_Run q*
proof (*intro allI notI*)
fix *q* **assume** *Z_run*: *mt_state Z* = *W_Run q*
from *wrap_W_Run_persistent*[*OF Z_to_disp_rt Z_run*]
obtain *q′* **where**
mt_state (*mid_plant_disp_config M pack c w*) = *W_Run q′* **by** *blast*
with *disp_non_run* **show** *False* **by** *blast*
qed
from *wrap_path_det_non_run*[*OF Z_path pre_pow Z_non_run C_pre_non_run*]
have *Z_eq*: *Z* = *C_pre* **.**
have *Z_state*: *mt_state Z* = *W_Disp* **using** *Z_eq pre_state* **by** *simp*

```
  have N_gt: 0 < N − T_pre using T_lt by simp
  from wrap_W_Disp_then_W_Run[OF Z_to_disp Z_state N_gt]
  obtain q' where
    mt_state (mid_plant_disp_config M pack c w) = W_Run q' by blast
  with disp_non_run show False by blast
 qed

 have T_pre_le: T_pre ≤ N
 proof (rule ccontr)
  assume ¬ T_pre ≤ N
  hence N_lt: N < T_pre by simp
  have add_eq: N + (T_pre − N) = T_pre using N_lt by simp
  have rel_add:
   (?R ^^ N) O (?R ^^ (T_pre − N)) = ?R ^^ T_pre
   using add_eq by (metis relpow_add)
  from pre_pow rel_add
  have (?init, C_pre) ∈ (?R ^^ N) O (?R ^^ (T_pre − N))
   by simp
  then obtain Y where
     Y_path: (?init, Y) ∈ ?R ^^ N
   and Y_to_pre: (Y, C_pre) ∈ ?R ^^ (T_pre − N)
   by auto
  have Y_to_pre_rt: (Y, C_pre) ∈ ?R*
   using Y_to_pre by (rule relpow_imp_rtrancl)
  have Y_non_run: ∀ q. mt_state Y ≠ W_Run q
  proof (intro allI notI)
   fix q assume Y_run: mt_state Y = W_Run q
   from wrap_W_Run_persistent[OF Y_to_pre_rt Y_run]
   obtain q' where mt_state C_pre = W_Run q' by blast
   with C_pre_non_run show False by blast
  qed
     from wrap_path_det_non_run[OF Y_path canon_pow Y_non_run
disp_non_run]
  have Y_eq: Y = mid_plant_disp_config M pack c w .
  have Y_state: mt_state Y = W_Disp using Y_eq disp_state by simp
  have T_gt: 0 < T_pre − N using N_lt by simp
  from wrap_W_Disp_then_W_Run[OF Y_to_pre Y_state T_gt]
  obtain q' where mt_state C_pre = W_Run q' by blast
  with C_pre_non_run show False by blast
 qed

 from T_pre_ge T_pre_le have T_eq_N: T_pre = N by simp

 from pre_pow T_eq_N have pre_pow_N: (?init, C_pre) ∈ ?R ^^ N by simp
  from wrap_path_det_non_run[OF pre_pow_N canon_pow C_pre_non_run
disp_non_run]
 have C_pre_eq: C_pre = mid_plant_disp_config M pack c w .

 have actual_disp_step:
```

```
    (mid_plant_disp_config M pack c w, C_post) ∈ ?R
    using pre_step C_pre_eq by simp
  have C_post_eq:
    C_post = post_plant_dispatch_config M pack c w
    by (rule wrap_step_unique_non_run
          [OF actual_disp_step canon_disp_step disp_non_run])

  from post_chain C_post_eq have suffix:
    (post_plant_dispatch_config M pack c w, C_acc) ∈ ?R*
    by simp

  from suffix pack_in_Sigma show ?thesis by blast
qed
end
theory Wrap_Language
  imports Wrap_Canonical
begin
```

14 Faithful (k-tape) plant-*le* wrap: language preservation

The user-facing language of the faithful plant-*le* wrap equals the encoded-input language of M: *Lang_user_wrap* (*encoding_wrap M pack c* Σ*u*) = {*w*. *set w* ⊆ Σ*u* ∧ *wrap_enc pack c w* ∈ *Lang_mttm M*}. As with the transpose wrap this is mutual inclusion of accepting languages (two one-way trace arguments; not a bisimulation, since *M* is nondeterministic).

The *reverse* inclusion (M-acceptance of the encoded input gives wrap-acceptance) differs from the transpose at the reset→run seam. Where the transpose rewinds all the way to the true origin and hands the run the genuine *lift_M_config*, the plant wrap stops at the *floated* configuration *post_plant_dispatch_config* — no input rewind, a fresh *le* planted at the reused input head. The origin-float bridge reconciles the two: the M-run lifts (via *run_steps_forward_relpow*) to a run from the *τ*-lift, and *shift_reach_state_forward* — instantiated at the wrap machine itself — re-bases that run onto the floated config the plant reset actually reaches, preserving the accept state.

14.1 Reverse inclusion: chaining the phases through the floated origin

```
lemma wrap_language_reverse:
  assumes vM: valid_mttm M
      and fin_Sigmau: finite Σu
      and c_pos: 0 < c
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
```

```
    shows {w. set w ⊆ Σu ∧ wrap_enc pack c w ∈ Lang_mttm M}
          ⊆ Lang_user_wrap (encoding_wrap M pack c Σu)
proof
  fix w
  assume w ∈ {w. set w ⊆ Σu ∧ wrap_enc pack c w ∈ Lang_mttm M}
  hence w_alpha: set w ⊆ Σu
    and enc_lang: wrap_enc pack c w ∈ Lang_mttm M
    by auto

  from enc_lang have enc_in_Sigma: set (wrap_enc pack c w) ⊆ Sigma_tm M
    by (simp add: Lang_mttm_def)
  from enc_lang obtain w_ts cf_pos where M_trace:
    (init_config_mttm M (wrap_enc pack c w),
      Config_M (t_tm M) w_ts cf_pos)
     ∈ (mttm_step (delta_tm M))*
    by (auto simp: Lang_mttm_def)

  let ?W = encoding_wrap M pack c Σu
  let ?init_M = init_config_mttm M (wrap_enc pack c w)
  let ?cf_M = Config_M (t_tm M) w_ts cf_pos
  let ?R_W = mttm_step (wrap_delta M pack c Σu)
  let ?d = λp::nat. if p = 0 then length w + 1 else 0

  obtain Q Σi Γset bl le δM sM tM rM kM where M_eq:
    M = MTTM Q Σi Γset bl le δM sM tM rM kM
    by (cases M)
  have Sigma_sub_Γ: Sigma_tm M ⊆ Γ_tm M using vM M_eq by auto
  have le_not_in_Sigma: le_tm M ∉ Sigma_tm M using vM M_eq by auto

  — Pack contracts from the Sigma containment (as in the transpose reverse).
  have pack_in_Sigma:
    ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ∈ Sigma_tm M
  proof −
    fix i :: nat
    assume i_c_lt: i * c < length w
    have pack (take c (drop (i * c) w)) ∈ set (wrap_enc pack c w)
      by (rule pack_take_in_set_wrap_enc[OF c_pos i_c_lt])
    thus pack (take c (drop (i * c) w)) ∈ Sigma_tm M
      using enc_in_Sigma by auto
  qed
  have pack_in:
    ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ∈ Γ_tm M
  proof −
    fix i :: nat
    assume i_c_lt: i * c < length w
    from pack_in_Sigma[OF i_c_lt] Sigma_sub_Γ
    show pack (take c (drop (i * c) w)) ∈ Γ_tm M by auto
  qed
  have pack_ne_le:
```

```
  ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ≠ le_tm M
 proof −
  fix i :: nat
  assume i_c_lt: i * c < length w
  from pack_in_Sigma[OF i_c_lt] le_not_in_Sigma
  show pack (take c (drop (i * c) w)) ≠ le_tm M by auto
 qed

 — Phase 1: encoder.
 have enc_step:
  (init_config_mttm ?W (map Raw w),
    post_encoder_config_gen (k_tm M) M pack c w) ∈ ?R_W*
  by (rule encoder_phase_terminates
      [where pack = pack, OF vM c_pos w_alpha bl_ne_le k2 pack_in_Sigma])
 — Phase 2+3: storage rewind, plant-le, dispatch — reaching the floated engine
config. The input tape is never walked.
 have pr_step:
  (post_encoder_config_gen (k_tm M) M pack c w,
    post_plant_dispatch_config M pack c w) ∈ ?R_W*
  by (rule plant_reset_dispatch_rtrancl
          [where pack = pack, OF vM c_pos w_alpha bl_ne_le k2 pack_in
pack_ne_le])

 — Phase 4: the run, spliced through the origin float.
 have val_init_M: valid_config_mttm M ?init_M
  by (rule valid_init_config_mttm[OF vM enc_in_Sigma])
 from M_trace obtain n where M_trace_n:
  (?init_M, ?cf_M) ∈ (mttm_step (delta_tm M)) ^^ n
  by (auto simp: rtrancl_power)
 have run_relpow:
  (lift_M_config M ?init_M, lift_M_config M ?cf_M) ∈ ?R_W ^^ n
  by (rule run_steps_forward_relpow[OF vM k2 val_init_M M_trace_n])

 have wf_wrap: valid_mttm ?W
  by (rule wrap_wf[OF vM bl_ne_le[THEN not_sym] fin_Sigmau c_pos k2])
 have shift:
  shift_rel ?d (lift_M_config M ?init_M)
     (post_plant_dispatch_config M pack c w)
  by (rule post_plant_dispatch_shift_lift[OF k2])
 have wrap_delta_eq: delta_tm ?W = wrap_delta M pack c Σu
  by simp

 — le0: only physical 0 floats, and the τ-lift holds le there — the M-init left
endmarker at cell 0 of the active storage tape Suc 0 (in range since 2 ≤ k_tm M).
 have suc0_lt: Suc 0 < k_tm M using k2 by simp
 have le0:
  ∀k. ?d k ≠ 0 ⟶ mt_tape (lift_M_config M ?init_M) k 0 = le_tm ?W
 proof (intro allI impI)
  fix k :: nat
```

```
    assume ?d k ≠ 0
    hence k0: k = 0 by (auto split: if_splits)
    have mt_tape (lift_M_config M ?init_M) 0 0
          = Enc (mt_tape ?init_M (wrap_tau 0) 0)
      using k2 by (simp add: lift_M_config_def)
    also have ... = Enc (mt_tape ?init_M (Suc 0) 0)
      by (simp add: wrap_tau_def)
    also have mt_tape ?init_M (Suc 0) 0 = le_tm M
      using suc0_lt by (simp add: M_eq)
    finally show mt_tape (lift_M_config M ?init_M) k 0 = le_tm ?W
      using k0 by simp
  qed

  have acc_lift: mt_state (lift_M_config M ?cf_M) = W_Run (t_tm M)
    by (simp add: lift_M_config_def)
  have run_relpow':
    (lift_M_config M ?init_M, lift_M_config M ?cf_M)
      ∈ (mttm_step (delta_tm ?W)) ^^ n
    using run_relpow wrap_delta_eq by simp
  from shift_reach_state_forward[OF wf_wrap le0 shift run_relpow' acc_lift]
  obtain cb where cb_run:
      (post_plant_dispatch_config M pack c w, cb)
        ∈ (mttm_step (delta_tm ?W)) ^^ n
    and cb_state: mt_state cb = W_Run (t_tm M)
    by blast
  have cb_run_R: (post_plant_dispatch_config M pack c w, cb) ∈ ?R_W ^^ n
    using cb_run wrap_delta_eq by simp
  have cb_run_rtrancl: (post_plant_dispatch_config M pack c w, cb) ∈ ?R_W*
    by (rule relpow_imp_rtrancl[OF cb_run_R])

  — Compose the four phases into one accepting wrap-trace.
  from enc_step pr_step have er_step:
    (init_config_mttm ?W (map Raw w),
      post_plant_dispatch_config M pack c w) ∈ ?R_W*
    by (rule rtrancl_trans)
  from er_step cb_run_rtrancl have full_trace:
    (init_config_mttm ?W (map Raw w), cb) ∈ ?R_W*
    by (rule rtrancl_trans)

  — cb is in the wrap accept state t_tm (encoding_wrap M pack c Σu) = W_Run
(t_tm M).
  obtain cb_ts cb_n where cb_eq:
    cb = Config_M (t_tm ?W) cb_ts cb_n
  proof −
    have t_tm ?W = W_Run (t_tm M) by simp
    hence cb = Config_M (t_tm ?W) (mt_tape cb) (mt_pos cb)
      using cb_state by (cases cb) simp
    thus thesis by (rule that)
  qed
```

```
  from full_trace cb_eq have accept_trace:
    (init_config_mttm ?W (map Raw w),
      Config_M (t_tm ?W) cb_ts cb_n) ∈ ?R_W*
    by simp

  have set_Raw_w: set (map Raw w) ⊆ Sigma_tm ?W
    using w_alpha by auto

  show w ∈ Lang_user_wrap ?W
    unfolding Lang_user_wrap_def Lang_mttm_def
    using set_Raw_w accept_trace wrap_delta_eq by auto
qed
```

14.2 Forward inclusion: factoring through the floated origin

The forward inclusion: wrap-B acceptance of *map Raw w* gives M-acceptance of *wrap_enc pack c w*. The accepting wrap-trace factors via *wrap_accept_canonical_factor* (reaching the *floated post_plant_dispatch_config*), then the origin float runs in reverse: *shift_reach_state_reverse* re-bases the run-phase tail from the floated config to the τ-lift, and *run_steps_reverse* projects it to an accepting M-trace on *wrap_enc pack c w*.

```
lemma wrap_language_forward:
  fixes M :: ('q, 'b) mttm
  assumes vM: valid_mttm M
      and fin_Sigmau: finite Σu
      and c_pos: 0 < c
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
  shows Lang_user_wrap (encoding_wrap M pack c Σu)
        ⊆ {w. set w ⊆ Σu ∧ wrap_enc pack c w ∈ Lang_mttm M}
proof
  fix w
  assume mem: w ∈ Lang_user_wrap (encoding_wrap M pack c Σu)
  hence raw_w_lang:
    map Raw w ∈ Lang_mttm (encoding_wrap M pack c Σu)
    unfolding Lang_user_wrap_def by simp

  have w_alpha: set w ⊆ Σu
  proof
    fix x assume x_in: x ∈ set w
    hence raw_x_in: Raw x ∈ set (map Raw w) by simp
    from raw_w_lang have map_raw_in:
      set (map Raw w) ⊆ Raw ` Σu
      unfolding Lang_mttm_def by auto
    from raw_x_in map_raw_in have Raw x ∈ Raw ` Σu by blast
    thus x ∈ Σu by auto
  qed
```

let *?W = encoding_wrap M pack c Σu*
let *?R_W = mttm_step (wrap_delta M pack c Σu)*
let *?init_M = init_config_mttm M (wrap_enc pack c w)*
let *?d = λp::nat. if p = 0 then length w + 1 else 0*

from *raw_w_lang* **obtain** *w_ts cf_pos* **where** *wrap_trace_raw*:
(init_config_mttm ?W (map Raw w),
Config$_M$ (t_tm ?W) w_ts cf_pos)
∈ (mttm_step (delta_tm ?W))*
unfolding *Lang_mttm_def* **by** *auto*

let *?C_acc = Config$_M$ (W_Run (t_tm M)) w_ts cf_pos*

have *wrap_trace*:
(init_config_mttm ?W (map Raw w), ?C_acc) ∈ ?R_W*
using *wrap_trace_raw* **by** *simp*
have *accept_state*: *∃ q. mt_state ?C_acc = W_Run q* **by** *simp*

from *wrap_accept_canonical_factor*
[OF vM c_pos bl_ne_le w_alpha k2 wrap_trace accept_state]
have *suffix*:
(post_plant_dispatch_config M pack c w, ?C_acc) ∈ ?R_W*
and *pack_in_Sigma*:
*⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ∈ Sigma_tm M*
by *auto*

have *enc_in_Sigma*: *set (wrap_enc pack c w) ⊆ Sigma_tm M*
proof
fix *x*
assume *x ∈ set (wrap_enc pack c w)*
then obtain *j* **where** *j_lt*: *j < length (wrap_enc pack c w)*
and *x_eq*: *x = wrap_enc pack c w ! j*
by (*auto simp*: *in_set_conv_nth*)
from *wrap_enc_in_Sigma*[*OF c_pos pack_in_Sigma j_lt*]
show *x ∈ Sigma_tm M* **using** *x_eq* **by** *simp*
qed
have *val_init_M*: *valid_config_mttm M ?init_M*
by (*rule valid_init_config_mttm*[*OF vM enc_in_Sigma*])

— Reverse origin float: the accepting run from the floated config lifts to a run from the τ-lift, reaching the same accept state.
have *wf_wrap*: *valid_mttm ?W*
by (*rule wrap_wf*[*OF vM bl_ne_le*[*THEN not_sym*] *fin_Sigmau c_pos k2*])
have *shift*:
shift_rel ?d (lift_M_config M ?init_M)
(post_plant_dispatch_config M pack c w)
by (*rule post_plant_dispatch_shift_lift*[*OF k2*])
have *wrap_delta_eq*: *delta_tm ?W = wrap_delta M pack c Σu*

```
  by simp

obtain Q Σi Γset bl le δM sM tM rM kM where M_eq:
  M = MTTM Q Σi Γset bl le δM sM tM rM kM
  by (cases M)
have suc0_lt: Suc 0 < k_tm M using k2 by simp
have le0:
  ∀k. ?d k ≠ 0 ⟶ mt_tape (lift_M_config M ?init_M) k 0 = le_tm ?W
proof (intro allI impI)
  fix k :: nat
  assume ?d k ≠ 0
  hence k0: k = 0 by (auto split: if_splits)
  have mt_tape (lift_M_config M ?init_M) 0 0
        = Enc (mt_tape ?init_M (wrap_tau 0) 0)
    using k2 by (simp add: lift_M_config_def)
  also have ... = Enc (mt_tape ?init_M (Suc 0) 0)
    by (simp add: wrap_tau_def)
  also have mt_tape ?init_M (Suc 0) 0 = le_tm M
    using suc0_lt by (simp add: M_eq)
  finally show mt_tape (lift_M_config M ?init_M) k 0 = le_tm ?W
    using k0 by simp
qed

from suffix obtain n where suffix_n:
  (post_plant_dispatch_config M pack c w, ?C_acc) ∈ ?R_W ^^ n
  by (auto simp: rtrancl_power)
have suffix_n':
  (post_plant_dispatch_config M pack c w, ?C_acc)
    ∈ (mttm_step (delta_tm ?W)) ^^ n
  using suffix_n wrap_delta_eq by simp
have acc_state: mt_state ?C_acc = W_Run (t_tm M) by simp
from shift_reach_state_reverse[OF wf_wrap le0 shift suffix_n' acc_state]
obtain ca where ca_run:
    (lift_M_config M ?init_M, ca) ∈ (mttm_step (delta_tm ?W)) ^^ n
  and ca_state: mt_state ca = W_Run (t_tm M)
  by blast
have ca_run_rt: (lift_M_config M ?init_M, ca) ∈ ?R_W*
proof −
  have (lift_M_config M ?init_M, ca) ∈ ?R_W ^^ n
    using ca_run wrap_delta_eq by simp
  thus ?thesis by (rule relpow_imp_rtrancl)
qed

from run_steps_reverse[OF vM k2 val_init_M ca_run_rt]
obtain cM_acc where
    ca_lift: ca = lift_M_config M cM_acc
  and M_trace_acc: (?init_M, cM_acc) ∈ (mttm_step (delta_tm M))*
  by blast
```

```
  obtain cM_q cM_ts cM_n where cM_decomp:
    cM_acc = Config_M cM_q cM_ts cM_n
    by (cases cM_acc)
  with ca_lift have lift_state_eq:
    mt_state ca = W_Run cM_q
    by (simp add: lift_M_config_def)
  with ca_state have cM_q_eq: cM_q = t_tm M by simp

  have enc_lang: wrap_enc pack c w ∈ Lang_mttm M
    unfolding Lang_mttm_def
    using enc_in_Sigma M_trace_acc cM_decomp cM_q_eq
    by auto

  show w ∈ {w. set w ⊆ Σu ∧ wrap_enc pack c w ∈ Lang_mttm M}
    using w_alpha enc_lang by simp
qed
```

14.3 Language equality headline

The user-facing language of the faithful k-tape plant-le wrap equals the encoded-input language of M — both inclusions, spliced through the origin float in opposite directions.

```
lemma wrap_language:
  assumes valid_mttm M
      and finite Σu
      and 0 < c
      and bl_tm M ≠ le_tm M
      and 2 ≤ k_tm M
  shows Lang_user_wrap (encoding_wrap M pack c Σu)
        = {w. set w ⊆ Σu ∧ wrap_enc pack c w ∈ Lang_mttm M}
  using wrap_language_forward[OF assms] wrap_language_reverse[OF assms]
  by blast

end
theory Wrap_Time
  imports Wrap_Language
begin
```

15 Faithful (k-tape) plant-le wrap: time bound

The wrap's user-facing time on w decomposes into the encoder phase (*length w + 2* steps, *init_to_post_encoder_relpow*), the storage rewind + plant-*le* + dispatch (*Suc* (*length* (*wrap_enc pack c w*)) + *2* steps, *plant_reset_dispatch_relpow*, reaching the floated engine configuration *post_plant_dispatch_config*), and the run phase, which the origin float carries onto the floated configuration one-for-one (*shift_reach_state_forward*, step count preserved). The total is *length w* + *length* (*wrap_enc pack c w*)

$+\ 5\ +\ t$.

The linear-in-*length w* part has coefficient *1*: unlike the transpose wrap's combined reset (cost $2 * \mathit{length}\ w$), wrapper B *never walks back over the raw input* — the plant-*le* reset rewinds only the storage tape ($\mathit{length}\ (\mathit{wrap_enc}\ \mathit{pack}\ c\ w)$ packed cells, at most $\mathit{length}\ w$), then plants a fresh *le* at the input origin. This is the $(1 + \varepsilon)\cdot n$ setup the linear-time speed-up [6, Theorem 12.4] needs; the transpose's $\sim 2\cdot n$ rewind is what the super-linear [6, Theorem 12.3] tolerates but 12.4 does not.

The per-input precondition $\mathit{set}\ (\mathit{wrap_enc}\ \mathit{pack}\ c\ w) \subseteq \mathit{Sigma_tm}\ M$ feeds the encoder pack-contracts and *valid_init_config_mttm*.

```
lemma wrap_time:
  assumes vM: valid_mttm M
      and fin_Sigmau: finite Σu
      and c_pos: 0 < c
      and bl_ne_le: bl_tm M ≠ le_tm M
      and k2: 2 ≤ k_tm M
      and w_alpha: set w ⊆ Σu
      and enc_in_Sigma: set (wrap_enc pack c w) ⊆ Sigma_tm M
      and accept_M: accepts_in_time_mttm M (wrap_enc pack c w) t
  shows accepts_in_time_user_wrap (encoding_wrap M pack c Σu) w
          (length w + length (wrap_enc pack c w) + 5 + t)
proof −
  let ?R_W = mttm_step (wrap_delta M pack c Σu)
  let ?W = encoding_wrap M pack c Σu
  let ?init_M = init_config_mttm M (wrap_enc pack c w)
  let ?d = λp::nat. if p = 0 then length w + 1 else 0

  — Unpack M's accepting trace at exact length n.
  from accept_M obtain n cM_n where
      n_le: n ≤ t
      and M_trace: (?init_M, cM_n) ∈ (mttm_step (delta_tm M)) ^^ n
      and cM_n_accept: mt_state cM_n = t_tm M
    unfolding accepts_in_time_mttm_def by auto

  — Substrate facts and pack contracts (as in the reverse inclusion).
  obtain Q Σi Γset bl le δM sM tM rM kM where M_eq:
    M = MTTM Q Σi Γset bl le δM sM tM rM kM
    by (cases M)
  have Sigma_sub_Gamma: Sigma_tm M ⊆ Γ_tm M using vM M_eq by auto
  have le_not_in_Sigma: le_tm M ∉ Sigma_tm M using vM M_eq by auto

  have pack_in_Sigma:
    ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ∈ Sigma_tm M
  proof −
    fix i :: nat
    assume i_c_lt: i * c < length w
    have pack (take c (drop (i * c) w)) ∈ set (wrap_enc pack c w)
      by (rule pack_take_in_set_wrap_enc[OF c_pos i_c_lt])
```

```
    thus pack (take c (drop (i * c) w)) ∈ Sigma_tm M
      using enc_in_Sigma by auto
  qed
  have pack_in:
    ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ∈ Γ_tm M
  proof −
    fix i :: nat
    assume i_c_lt: i * c < length w
    from pack_in_Sigma[OF i_c_lt] Sigma_sub_Gamma
    show pack (take c (drop (i * c) w)) ∈ Γ_tm M by auto
  qed
  have pack_ne_le:
    ⋀i. i * c < length w ⟹ pack (take c (drop (i * c) w)) ≠ le_tm M
  proof −
    fix i :: nat
    assume i_c_lt: i * c < length w
    from pack_in_Sigma[OF i_c_lt] le_not_in_Sigma
    show pack (take c (drop (i * c) w)) ≠ le_tm M by auto
  qed

  — Phase 1: encoder — kept at exact relpow length length w + 2.
  have enc_chain:
    (init_config_mttm ?W (map Raw w),
      post_encoder_config_gen (k_tm M) M pack c w) ∈ ?R_W ^^ (length w +
2)
    by (rule init_to_post_encoder_relpow
        [where pack = pack, OF vM c_pos w_alpha bl_ne_le k2 pack_in_Sigma])

  — Phase 2+3: storage rewind, plant-le, dispatch — reaching the floated engine
config in Suc (length (wrap_enc pack c w)) + 2 steps. The input tape is never
walked (contrast the transpose's length w + 3).
  have rd_chain:
    (post_encoder_config_gen (k_tm M) M pack c w,
      post_plant_dispatch_config M pack c w)
      ∈ ?R_W ^^ (Suc (length (wrap_enc pack c w)) + 2)
    by (rule plant_reset_dispatch_relpow
          [where pack = pack, OF vM c_pos w_alpha bl_ne_le k2 pack_in
pack_ne_le])

  — Phase 4: the run, spliced through the origin float — step count preserved.
  have val_init_M: valid_config_mttm M ?init_M
    by (rule valid_init_config_mttm[OF vM enc_in_Sigma])
  have run_relpow:
    (lift_M_config M ?init_M, lift_M_config M cM_n) ∈ ?R_W ^^ n
    by (rule run_steps_forward_relpow[OF vM k2 val_init_M M_trace])
  have wf_wrap: valid_mttm ?W
    by (rule wrap_wf[OF vM bl_ne_le[THEN not_sym] fin_Sigmau c_pos k2])
  have shift:
    shift_rel ?d (lift_M_config M ?init_M)
```

```
      (post_plant_dispatch_config M pack c w)
  by (rule post_plant_dispatch_shift_lift[OF k2])
 have wrap_delta_eq: delta_tm ?W = wrap_delta M pack c Σu
  by simp

 — le0: only physical 0 floats, and the τ-lift holds le there — the M-init left
endmarker at cell 0 of the active storage tape Suc 0 (in range since 2 ≤ k_tm M).
 have suc0_lt: Suc 0 < k_tm M using k2 by simp
 have le0:
  ∀k. ?d k ≠ 0 ⟶ mt_tape (lift_M_config M ?init_M) k 0 = le_tm ?W
 proof (intro allI impI)
  fix k :: nat
  assume ?d k ≠ 0
  hence k0: k = 0 by (auto split: if_splits)
  have mt_tape (lift_M_config M ?init_M) 0 0
        = Enc (mt_tape ?init_M (wrap_tau 0) 0)
    using k2 by (simp add: lift_M_config_def)
  also have ... = Enc (mt_tape ?init_M (Suc 0) 0)
    by (simp add: wrap_tau_def)
  also have mt_tape ?init_M (Suc 0) 0 = le_tm M
    using suc0_lt by (simp add: M_eq)
  finally show mt_tape (lift_M_config M ?init_M) k 0 = le_tm ?W
    using k0 by simp
 qed

 have acc_lift: mt_state (lift_M_config M cM_n) = W_Run (t_tm M)
  using cM_n_accept by (cases cM_n) (simp add: lift_M_config_def)
 have run_relpow':
  (lift_M_config M ?init_M, lift_M_config M cM_n)
    ∈ (mttm_step (delta_tm ?W)) ^^ n
  using run_relpow wrap_delta_eq by simp
 from shift_reach_state_forward[OF wf_wrap le0 shift run_relpow' acc_lift]
 obtain cb where cb_run:
    (post_plant_dispatch_config M pack c w, cb)
      ∈ (mttm_step (delta_tm ?W)) ^^ n
  and cb_state: mt_state cb = W_Run (t_tm M)
  by blast
 have run_chain: (post_plant_dispatch_config M pack c w, cb) ∈ ?R_W ^^ n
  using cb_run wrap_delta_eq by simp

 let ?N_total = (length w + 2) + (Suc (length (wrap_enc pack c w)) + 2) + n

 — Compose phases via ?R^(?m + ?n) = ?R^?m O ?R^?n.
 from enc_chain rd_chain
 have er_chain:
  (init_config_mttm ?W (map Raw w),
    post_plant_dispatch_config M pack c w)
    ∈ ?R_W ^^ ((length w + 2) + (Suc (length (wrap_enc pack c w)) + 2))
  unfolding relpow_add by blast
```

```
  from er_chain run_chain
  have full_chain:
    (init_config_mttm ?W (map Raw w), cb) ∈ ?R_W ^^ ?N_total
    unfolding relpow_add by blast

  have total_bound:
    ?N_total ≤ length w + length (wrap_enc pack c w) + 5 + t
    using n_le by linarith

  — The reached config cb is in the wrap accept state.
  have wrap_accept: t_tm ?W = W_Run (t_tm M) by simp
  have cb_accept: mt_state cb = t_tm ?W
    using cb_state wrap_accept by simp

  have delta_eq: delta_tm ?W = wrap_delta M pack c Σu by simp
  have fact_b:
    (init_config_mttm ?W (map Raw w), cb)
      ∈ mttm_step (delta_tm ?W) ^^ ?N_total
    using full_chain delta_eq by simp

  have ex_witness:
    ∃m cw. m ≤ length w + length (wrap_enc pack c w) + 5 + t
        ∧ (init_config_mttm ?W (map Raw w), cw)
            ∈ mttm_step (delta_tm ?W) ^^ m
        ∧ mt_state cw = t_tm ?W
    using total_bound fact_b cb_accept by blast

  show accepts_in_time_user_wrap ?W w
        (length w + length (wrap_enc pack c w) + 5 + t)
    unfolding accepts_in_time_user_wrap_def accepts_in_time_mttm_def
    using ex_witness by simp
qed

end
theory Wrap_Speedup
  imports AlphabetEnlargement_Pack Wrap_Time
begin
```

16 Faithful (k-tape) classical linear-speedup headlines — plant-*le* wrap

The faithful k-tape statements of the classical linear-speedup theorems, over the plant-*le* wrap *encoding_wrap* at tape count *k_tm M*. These are the textbook statements of Hopcroft–Ullman [6, Theorem 12.4] and its nondeterministic corollary, genuinely k-tape-to-k-tape (matching the textbook $k > 1$).

The development has two layers: a Form-1 raw composition (*lin-*

ear_speedup_form_1_raw_nae / *_dae_B*) that glues the alphabet-enlargement engine to the plant-*le* wrap's correctness lemmas (*wrap_wf*, *wrap_language*, *wrap_time*, *wrap_det*), then the textbook corollaries (*linear_speedup_HU_12_4_nae* = the nondeterministic corollary [6, p. 291], *_dae_B* = Theorem 12.4 proper) as arithmetic specialisations.

The wrap correctness lemmas carry the side-condition $2 \le k_tm\ M$, which holds on the source via $k_tm\ (alphabet_enlarge\ M) = k_tm\ M$. The Form-1 time bound has a coefficient-*1* linear term *length w* (the encoder pass): the plant-*le* reset rewinds only the storage tape (at most $(length\ w + card\ UNIV - 1)\ div\ card\ UNIV$ cells) and never the raw input, so unlike the transpose's coefficient-*2* bound this meets the $(1+\varepsilon)n$ form HU 12.4 requires.

16.1 Linear speedup theorem (Form 1, raw composition) — faithful

The faithful Form-1 corollary: the plant-*le*-wrap-over-AE machine M'' satisfies the three det-free conjuncts of Form 1 — wf, language-equality, and an explicit time bound whose linear-in-$|w|$ coefficient is *1* (the $(1+\varepsilon)n$ payoff — no input rewind).

```
lemma linear_speedup_form_1_raw_nae:
  fixes M :: ('q, 'a) mttm
    and T :: nat ⇒ nat
  assumes wf:        well_formed_mttm M
      and c_pos:     0 < (card (UNIV :: ('c :: enum) set))
      and k2:        2 ≤ k_tm M
  shows valid_mttm
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
    and Lang_user_wrap
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
        = Lang_mttm M
    and ∀ w. set w ⊆ Sigma_tm M
            ⟶ accepts_in_time_mttm M w (T (length w))
            ⟶ accepts_in_time_user_wrap
                (encoding_wrap
                  (alphabet_enlarge M
                    :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
```

```
                    (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
                    (card (UNIV :: 'c set))
                    (Sigma_tm M))
                  w
                  (length w
                   + (length w + card (UNIV :: 'c set) − 1)
                         div card (UNIV :: 'c set)
                   + 5
                   + 2 ∗ ((length w + card (UNIV :: 'c set) − 1)
                         div card (UNIV :: 'c set))
                   + 8 ∗ ((T (length w) + card (UNIV :: 'c set) − 1)
                         div card (UNIV :: 'c set))
                   + 4)
proof −
  from wf have vM:       valid_mttm M
         and le_neq_bl: le_tm M ≠ bl_tm M
    by auto

  let ?M' = alphabet_enlarge M
            :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm
  let ?c = card (UNIV :: 'c set)
  let ?pack = ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a)
  let ?M'' = encoding_wrap ?M' ?pack ?c (Sigma_tm M)

  — AE preserves validity.
  have vM': valid_mttm ?M'
    by (rule alphabet_enlarge_wf[OF vM])

  — Sigma_tm M is finite (Sigma ⊆ Gamma, Gamma finite).
  have finSigma: finite (Sigma_tm M)
    using valid_mttm_Sigma_sub_Gamma[OF vM]
          valid_mttm_finite_Gamma[OF vM]
          rev_finite_subset by blast

  — bl ?M' ≠ le ?M' (from the AE substrate projections).
  have bl_ne_le': bl_tm ?M' ≠ le_tm ?M'
    unfolding bl_tm_alphabet_enlarge le_tm_alphabet_enlarge
    using le_neq_bl by (simp add: bl_block_def LE_block_def fun_eq_iff)

  — AE preserves the tape count, so 2 ≤ k_tm ?M'.
  have k_eq: k_tm ?M' = k_tm M
    by (cases M) (simp add: alphabet_enlarge_def)
  have k2': 2 ≤ k_tm ?M' using k2 k_eq by simp

  — The faithful wrap preserves validity. Since the clause-2 relaxation wrap_wf
needs only valid_mttm and le ≠ bl of its input (not LE-uniqueness), both of which
AE preserves.
  have le_neq_bl': le_tm ?M' ≠ bl_tm ?M' using bl_ne_le' by simp
  have wf_wrap: valid_mttm ?M''
```

by (*rule wrap_wf*[*OF vM' le_neq_bl' finSigma c_pos k2'*])

— Glue: *wrap_enc* = *encode_input* (under the *ae_pack* instantiation).
have *glue*: ⋀*w*. *wrap_enc ?pack ?c w* = *encode_input* (*bl_tm M*) *w*
proof −
fix *w*
show *wrap_enc ?pack ?c w* = *encode_input* (*bl_tm M*) *w*
by (*rule wrap_enc_eq_encode_input*[*OF c_pos*])
qed

— For any input word w with set w ⊆ *Sigma_tm* M, the encoded form lives in *Sigma_tm* ?M'.
have *enc_in_Sigma_M'*:
⋀*w*. *set w* ⊆ *Sigma_tm M*
⟹ *set* (*wrap_enc ?pack ?c w*) ⊆ *Sigma_tm ?M'*
proof −
fix *w* :: '*a list* **assume** *w_sub*: *set w* ⊆ *Sigma_tm M*
have *in_gamma*:
set (*encode_input* (*bl_tm M*) *w* :: ('*c* ⇒ '*a*) *list*)
⊆ *gamma_block* (*Sigma_tm M* ∪ {*bl_tm M*})
by (*rule encode_input_in_gamma_block*[*OF w_sub*])
have *no_bl*:
bl_block (*bl_tm M*) ∉ *set* (*encode_input* (*bl_tm M*) *w* :: ('*c* ⇒ '*a*) *list*)
by (*rule encode_input_no_bl_block*[*OF vM w_sub*])
have *no_LE*:
LE_block (*le_tm M*) ∉ *set* (*encode_input* (*bl_tm M*) *w* :: ('*c* ⇒ '*a*) *list*)
by (*rule encode_input_no_LE_block*[*OF vM w_sub*])
from *in_gamma no_bl no_LE*
have *set* (*encode_input* (*bl_tm M*) *w* :: ('*c* ⇒ '*a*) *list*)
⊆ *gamma_block* (*Sigma_tm M* ∪ {*bl_tm M*})
− {*bl_block* (*bl_tm M*), *LE_block* (*le_tm M*)}
by *auto*
hence *set* (*encode_input* (*bl_tm M*) *w* :: ('*c* ⇒ '*a*) *list*)
⊆ *Sigma_tm ?M'*
unfolding *Sigma_tm_alphabet_enlarge* **by** *simp*
thus *set* (*wrap_enc ?pack ?c w*) ⊆ *Sigma_tm ?M'*
using *glue* **by** *simp*
qed

— Language equivalence (faithful wrap).
have *lang_eq*: *Lang_user_wrap ?M''* = *Lang_mttm M*
proof −
have *wrap_lang*:
Lang_user_wrap ?M''
= {*w*. *set w* ⊆ *Sigma_tm M*
∧ *wrap_enc ?pack ?c w* ∈ *Lang_mttm ?M'*}
using *wrap_language*[*OF vM' finSigma c_pos bl_ne_le' k2'*] .
have *ae_lang*:
∀ *w*. *set w* ⊆ *Sigma_tm M*

```
        ⟶ (encode_input (bl_tm M) w ∈ Lang_mttm ?M')
           = (w ∈ Lang_mttm M)
    using alphabet_enlarge_language[OF wf]
    by simp
  have lang_subset: Lang_mttm M ⊆ {w. set w ⊆ Sigma_tm M}
    unfolding Lang_mttm_def by auto
  show ?thesis
  proof
    show Lang_user_wrap ?M'' ⊆ Lang_mttm M
      using wrap_lang glue ae_lang by auto
  next
    show Lang_mttm M ⊆ Lang_user_wrap ?M''
    proof
      fix w assume wL: w ∈ Lang_mttm M
      with lang_subset have w_sub: set w ⊆ Sigma_tm M by auto
      from wL ae_lang w_sub
      have encode_input (bl_tm M) w ∈ Lang_mttm ?M' by simp
      hence wrap_enc ?pack ?c w ∈ Lang_mttm ?M' using glue by simp
      with w_sub wrap_lang show w ∈ Lang_user_wrap ?M'' by auto
    qed
  qed
qed
```

— Time bound: compose *wrap_time* + *alphabet_enlarge_time* + glue. The faithful wrap charges the encoder pass plus the storage rewind (length of *wrap_enc*, i.e. $\lceil |w|/c \rceil$) — no input rewind, so the linear term is *length w* with coefficient *1*.
— The AE structural constants are literal: $\alpha = 2$, $f = 4$ (from the explicit AE time bound *alphabet_enlarge_time_explicit*). They are pinned to named locals so the arithmetic body below reads unchanged; the headline states them as literals.

```
  obtain α f :: nat where alpha2: α = (2::nat) and f4: f = (4::nat)
    by blast
  have ae_time:
    ∀ w. set w ⊆ Sigma_tm M
          ⟶ accepts_in_time_mttm M w (T (length w))
          ⟶ accepts_in_time_mttm ?M'
              (encode_input (bl_tm M) w)
              (α * ((length w + ?c − 1) div ?c)
               + 8 * ((T (length w)) + ?c − 1) div ?c)
               + f)
    unfolding alpha2 f4 using alphabet_enlarge_time_explicit[OF wf, of T] by
blast

  have time_bound:
    ∀ w. set w ⊆ Sigma_tm M
          ⟶ accepts_in_time_mttm M w (T (length w))
          ⟶ accepts_in_time_user_wrap ?M'' w
              (length w
               + (length w + ?c − 1) div ?c
               + 5
```

```
                + α * ((length w + ?c − 1) div ?c)
                + 8 * ((T (length w) + ?c − 1) div ?c)
                + f)
  proof (intro allI impI)
    fix w :: 'a list
    assume w_sub: set w ⊆ Sigma_tm M
      and accept_M: accepts_in_time_mttm M w (T (length w))
    let ?t_ae = α * ((length w + ?c − 1) div ?c)
              + 8 * ((T (length w) + ?c − 1) div ?c) + f
    from ae_time w_sub accept_M
    have ae_step:
      accepts_in_time_mttm ?M' (encode_input (bl_tm M) w) ?t_ae
      by blast
    have wrap_input: accepts_in_time_mttm ?M' (wrap_enc ?pack ?c w) ?t_ae
      using ae_step glue by simp
    have enc_in: set (wrap_enc ?pack ?c w) ⊆ Sigma_tm ?M'
      using enc_in_Sigma_M'[OF w_sub] .

    — The plant wrap's storage-rewind term is exactly the encoded-tape length,
which is the ceiling ⌈|w|/c⌉ (via the encode_input length lemma and the wrap_enc
= encode_input glue).
    have len_enc: length (wrap_enc ?pack ?c w) = (length w + ?c − 1) div ?c
    proof −
      have length (wrap_enc ?pack ?c w)
            = length (encode_input (bl_tm M) w :: ('c ⇒ 'a) list)
        using glue by simp
      also have ... = (length w + ?c − 1) div ?c
        by (rule length_encode_input)
      finally show ?thesis .
    qed

    have wrap_step:
      accepts_in_time_user_wrap ?M'' w
         (length w + length (wrap_enc ?pack ?c w) + 5 + ?t_ae)
      using wrap_time[OF vM' finSigma c_pos bl_ne_le' k2'
                         w_sub enc_in wrap_input] .
    — Rewrite the storage-rewind term to the ceiling form and re-bracket; argo
substitutes through the uninterpreted predicate.
    have teq:
      length w + length (wrap_enc ?pack ?c w) + 5 + ?t_ae
        = length w + (length w + ?c − 1) div ?c + 5
          + α * ((length w + ?c − 1) div ?c)
          + 8 * ((T (length w) + ?c − 1) div ?c) + f
      using len_enc by simp
    show accepts_in_time_user_wrap ?M'' w
          (length w
           + (length w + ?c − 1) div ?c
           + 5
           + α * ((length w + ?c − 1) div ?c)
```

```
        + 8 * ((T (length w) + ?c - 1) div ?c)
        + f)
    using wrap_step teq by argo
qed

show valid_mttm ?M'' by (rule wf_wrap)
show Lang_user_wrap ?M'' = Lang_mttm M by (rule lang_eq)
show ∀ w. set w ⊆ Sigma_tm M
        ⟶ accepts_in_time_mttm M w (T (length w))
        ⟶ accepts_in_time_user_wrap ?M'' w
            (length w
             + (length w + ?c - 1) div ?c
             + 5
             + 2 * ((length w + ?c - 1) div ?c)
             + 8 * ((T (length w) + ?c - 1) div ?c)
             + 4)
  by (rule time_bound[unfolded alpha2 f4])
qed
```

Determinism-preserving faithful Form-1 raw composition (DAE). The det-free base *linear_speedup_form_1_raw_nae* supplies the validity, language, and time conjuncts; the extra *det_mttm* output conjunct is the one piece that consumes *det_mttm M* (via *wrap_det*).

```
lemma linear_speedup_form_1_raw_dae:
  fixes M :: ('q, 'a) mttm
    and T :: nat ⇒ nat
  assumes wf:       well_formed_mttm M
      and det:      det_mttm M
      and c_pos:    0 < (card (UNIV :: ('c :: enum) set))
      and k2:       2 ≤ k_tm M
  shows valid_mttm
        (encoding_wrap
          (alphabet_enlarge M
            :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
          (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
          (card (UNIV :: 'c set))
          (Sigma_tm M))
    and det_mttm
        (encoding_wrap
          (alphabet_enlarge M
            :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
          (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
          (card (UNIV :: 'c set))
          (Sigma_tm M))
    and Lang_user_wrap
        (encoding_wrap
          (alphabet_enlarge M
            :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
          (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
```

```
          (card (UNIV :: 'c set))
          (Sigma_tm M))
       = Lang_mttm M
  and ∀w. set w ⊆ Sigma_tm M
        ⟶ accepts_in_time_mttm M w (T (length w))
        ⟶ accepts_in_time_user_wrap
            (encoding_wrap
              (alphabet_enlarge M
                :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
              (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
              (card (UNIV :: 'c set))
              (Sigma_tm M))
            w
            (length w
             + (length w + card (UNIV :: 'c set) − 1)
                   div card (UNIV :: 'c set)
             + 5
             + 2 * ((length w + card (UNIV :: 'c set) − 1)
                   div card (UNIV :: 'c set))
             + 8 * ((T (length w) + card (UNIV :: 'c set) − 1)
                   div card (UNIV :: 'c set))
             + 4)
proof −
  from wf have vM: valid_mttm M
        and le_neq_bl: le_tm M ≠ bl_tm M
    by auto
  let ?M' = alphabet_enlarge M
          :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm
  let ?c = card (UNIV :: 'c set)
  let ?pack = ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a)
  let ?M'' = encoding_wrap ?M' ?pack ?c (Sigma_tm M)

  — Validity, language, and time come from the det-free base (now with literal AE
constants 2, 8, 4).
  have v: valid_mttm ?M''
    and l: Lang_user_wrap ?M'' = Lang_mttm M
    and t: ∀w. set w ⊆ Sigma_tm M
             ⟶ accepts_in_time_mttm M w (T (length w))
             ⟶ accepts_in_time_user_wrap ?M'' w
                 (length w
                  + (length w + ?c − 1) div ?c
                  + 5
                  + 2 * ((length w + ?c − 1) div ?c)
                  + 8 * ((T (length w) + ?c − 1) div ?c)
                  + 4)
    by (rule linear_speedup_form_1_raw_nae[OF wf c_pos k2])+

  — Determinism is the one extra conjunct; it needs det_mttm M.
  have vM': valid_mttm ?M' by (rule alphabet_enlarge_wf[OF vM])
```

```
  have detM': det_mttm ?M'
    unfolding det_mttm_def delta_tm_alphabet_enlarge
    using alphabet_enlarge_delta_functional[OF vM det le_neq_bl] by blast
  have finSigma: finite (Sigma_tm M)
    using valid_mttm_Sigma_sub_Gamma[OF vM]
          valid_mttm_finite_Gamma[OF vM]
          rev_finite_subset by blast
  have det_wrap: det_mttm ?M''
    by (rule wrap_det[OF vM' detM' finSigma c_pos])

  show valid_mttm ?M'' by (rule v)
  show det_mttm ?M'' by (rule det_wrap)
  show Lang_user_wrap ?M'' = Lang_mttm M by (rule l)
  show ∀ w. set w ⊆ Sigma_tm M
             ⟶ accepts_in_time_mttm M w (T (length w))
             ⟶ accepts_in_time_user_wrap ?M'' w
                 (length w
                  + (length w + ?c − 1) div ?c
                  + 5
                  + 2 ∗ ((length w + ?c − 1) div ?c)
                  + 8 ∗ ((T (length w) + ?c − 1) div ?c)
                  + 4)
    by (rule t)
qed
```

16.2 Textbook linear-speedup theorem (HU 12.4) — faithful k-tape

The textbook linear-speedup theorem, Form 1 [6, Theorem 12.4]: the linear-time-bound case $T(n) \leq d_0 \cdot n + b$.

HU 12.4 is the linear-T corollary of the raw composition *linear_speedup_form_1_raw_dae*: where the raw form carries the full ceiling-divisional bound $|w| + \lceil |w|/c \rceil + 5 + 2\lceil |w|/c \rceil + 8\lceil T(|w|)/c \rceil + 4$ (the AE structural constants are now the literals $\alpha = 2, f = 4$), HU 12.4 absorbs the *T_linear* hypothesis $T(n) \leq d_0 \cdot n + b$ and the ceiling-division slack into the *explicit* textbook constant $K = 28 + 8\ d_0 + 8\ b$ under a positive-nat q (the textbook real-valued ε taken as $1/q$). The conclusion shape collapses to $|w| + |w|$ *div* $q + (28 + 8\ d_0 + 8\ b)$, gated by a single *c_large* side condition $q * (3 + 8 * d_0) \leq c$ on the cardinality of the grouping-factor type $'c$.

Statement is parametric in $'c$ (the AE grouping-factor type) and in q. Consumers instantiate $'c$ at use sites via a *HOL.Library.Numeral_Type* of cardinality large enough to satisfy the *c_large* side condition on the time-bound conjunct. With α now literal the constant K is a closed expression in d_0, b (no *obtains*), and the wf, det, and language-equality conjuncts hold unconditionally (they do not depend on q); only the time-bound conjunct is gated by *c_large*.

Note on packaging: the *c_large* side condition for HU 12.4 takes the form $q * (3 + 8 * d_0) \leq c$. It is α-free (the simulation constant is the literal *2*), but *d_0*-dependent, and is retained INSIDE the statement as an implication on the time-bound conjunct rather than externalised as an outer *assumes*, so the four convention / eventual corollaries thread it uniformly. HU 12.3 below packages its corresponding side condition as an outer *assumes* *c_large*: $16 * q \leq c$, its constant $16 * q$ being *d_0*-free.

```
theorem linear_speedup_HU_12_4_nae:
  fixes M :: ('q, 'a) mttm
    and T :: nat ⇒ nat
    and d_0 b q :: nat
  assumes wf:        well_formed_mttm M
      and T_linear:  ∀ n. T n ≤ d_0 * n + b
      and q_pos:     0 < q
      and c_pos:     0 < (card (UNIV :: ('c :: enum) set))
      and k2:        2 ≤ k_tm M
  shows valid_mttm
         (encoding_wrap
           (alphabet_enlarge M
             :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
           (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
           (card (UNIV :: 'c set))
           (Sigma_tm M))
    and Lang_user_wrap
         (encoding_wrap
           (alphabet_enlarge M
             :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
           (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
           (card (UNIV :: 'c set))
           (Sigma_tm M))
        = Lang_mttm M
    and q * (3 + 8 * d_0) ≤ card (UNIV :: 'c set)
        ⟶ (∀ w. set w ⊆ Sigma_tm M
                ⟶ accepts_in_time_mttm M w (T (length w))
                ⟶ accepts_in_time_user_wrap
                     (encoding_wrap
                       (alphabet_enlarge M
                         :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
                       (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
                       (card (UNIV :: 'c set))
                       (Sigma_tm M))
                     w
                     (length w + length w div q + (28 + 8 * d_0 + 8 * b)))
proof −
  let ?M' = alphabet_enlarge M
            :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm
  let ?c = card (UNIV :: 'c set)
  let ?pack = ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a)
```

let *?M''* = *encoding_wrap ?M' ?pack ?c* (*Sigma_tm M*)

— Materialise form-1's structural conjuncts. The AE constants are literal (α = *2*, f = *4*); pinned to named locals so the specialisation arithmetic below reads unchanged.

obtain α f :: *nat* **where** *alpha2*: α = (*2*::*nat*) **and** *f4*: f = (*4*::*nat*)
by *blast*
have *wf_M''*: *valid_mttm ?M''*
and *lang_M''*: *Lang_user_wrap ?M''* = *Lang_mttm M*
and *time_F1*: ∀ *w*. *set w* ⊆ *Sigma_tm M*
⟶ *accepts_in_time_mttm M w* (*T* (*length w*))
⟶ *accepts_in_time_user_wrap ?M'' w*
(*length w*
+ (*length w* + *?c* − *1*) *div ?c*
+ *5*
+ α ∗ ((*length w* + *?c* − *1*) *div ?c*)
+ *8* ∗ ((*T* (*length w*) + *?c* − *1*) *div ?c*)
+ *f*)
unfolding *alpha2 f4*
by (*rule linear_speedup_form_1_raw_nae*[*OF wf c_pos k2*])+

— HU 12.4 specialisation constant: pick K big enough to absorb the ceiling-division slack plus the *T_linear* additive b plus form-1's structural constants. Under $c \geq q * (1 + \alpha + 8 * d_0)$ the form-1 bound simplifies to $|w| + |w|$ *div* $q + K$.

define K :: *nat* **where**
K = *22* + α + *8* ∗ *d_0* + *8* ∗ b + f

— Monotonicity of *accepts_in_time_user_wrap* in its time argument (immediate from the $n \leq t$ bound inside *accepts_in_time_mttm*).

have *user_mono*:
⋀*w* t_1 t_2. *accepts_in_time_user_wrap ?M'' w* t_1 ⟹ $t_1 \leq t_2$
⟹ *accepts_in_time_user_wrap ?M'' w* t_2
unfolding *accepts_in_time_user_wrap_def accepts_in_time_mttm_def*
by (*meson le_trans*)

have *time_HU_12_4*:
q ∗ (*1* + α + *8* ∗ *d_0*) ≤ *?c*
⟶ (∀ *w*. *set w* ⊆ *Sigma_tm M*
⟶ *accepts_in_time_mttm M w* (*T* (*length w*))
⟶ *accepts_in_time_user_wrap ?M'' w*
(*length w* + *length w div q* + *K*))
proof
assume *c_large*: *q* ∗ (*1* + α + *8* ∗ *d_0*) ≤ *?c*
show ∀ *w*. *set w* ⊆ *Sigma_tm M*
⟶ *accepts_in_time_mttm M w* (*T* (*length w*))
⟶ *accepts_in_time_user_wrap ?M'' w*
(*length w* + *length w div q* + *K*)
proof
fix *w* :: *'a list*

```
show set w ⊆ Sigma_tm M
      ⟶ accepts_in_time_mttm M w (T (length w))
      ⟶ accepts_in_time_user_wrap ?M'' w
          (length w + length w div q + K)
proof
  assume w_sub: set w ⊆ Sigma_tm M
  show accepts_in_time_mttm M w (T (length w))
        ⟶ accepts_in_time_user_wrap ?M'' w
            (length w + length w div q + K)
  proof
    assume accepts_M: accepts_in_time_mttm M w (T (length w))

    let ?n = length w
    let ?B1 = ?n + (?n + ?c − 1) div ?c + 5
              + α ∗ ((?n + ?c − 1) div ?c)
              + 8 ∗ ((T ?n + ?c − 1) div ?c) + f

    — Form-1's bound applies to w.
    have form_1_accepts: accepts_in_time_user_wrap ?M'' w ?B1
      using time_F1 w_sub accepts_M by blast

    — Ceiling-via-floor: (x + c − 1) div c ≤ x div c + 1.
    have ceil_le: (x + ?c − 1) div ?c ≤ x div ?c + 1 for x :: nat
    proof −
      have a: x + ?c − 1 ≤ x + ?c by simp
      have b: (x + ?c) div ?c = x div ?c + 1
        using c_pos by (simp add: div_add_self2)
      from a have (x + ?c − 1) div ?c ≤ (x + ?c) div ?c
        by (rule div_le_mono)
      with b show ?thesis by simp
    qed

    — Two consequences of ceil_le.
    have base_ceil_le: (?n + ?c − 1) div ?c ≤ ?n div ?c + 1
      by (rule ceil_le)
    have alpha_ceil_le:
      α ∗ ((?n + ?c − 1) div ?c) ≤ α ∗ (?n div ?c) + α
    proof −
      have α ∗ ((?n + ?c − 1) div ?c) ≤ α ∗ (?n div ?c + 1)
        using base_ceil_le by (rule mult_le_mono2)
      also have ... = α ∗ (?n div ?c) + α by simp
      finally show ?thesis .
    qed

    — T_linear hypothesis specialised at length w.
    have T_n_bound: T ?n ≤ d_0 ∗ ?n + b
      using T_linear by blast

    — Strict overshoot: n < c ∗ (n div c + 1). From Euclidean n = c ∗ (n div
```

```
c) + n mod c plus n mod c < c.
    have n_lt_c_succ: ?n < ?c * (?n div ?c + 1)
    proof −
      have e: ?c * (?n div ?c) + ?n mod ?c = ?n
        by (rule mult_div_mod_eq)
      have m: ?n mod ?c < ?c using c_pos by simp
      have d: ?c * (?n div ?c + 1) = ?c * (?n div ?c) + ?c
        by (simp add: distrib_left)
      from e m d show ?thesis by linarith
    qed

  — Scalar overshoot: d_0 * n ≤ c * (d_0 * (n div c + 1)). Avoids reasoning
about (d_0 * n) div c directly.
    have d0n_le_dc: d_0 * ?n ≤ ?c * (d_0 * (?n div ?c + 1))
    proof −
      have a: ?n ≤ ?c * (?n div ?c + 1)
        using n_lt_c_succ by simp
      have b: d_0 * ?n ≤ d_0 * (?c * (?n div ?c + 1))
        using a by (rule mult_le_mono2)
      have c: d_0 * (?c * (?n div ?c + 1)) = ?c * (d_0 * (?n div ?c + 1))
        by (metis mult.commute mult.assoc)
      show ?thesis using b unfolding c .
    qed

  — The clean T-ceiling bound: (T n + c − 1) div c ≤ d_0 * (n div c +
1) + (b + c − 1) div c. Derived from T_n_bound + d0n_le_dc + div_le_mono
+ div_mult_self4.
    have T_to_c_div:
      (T ?n + ?c − 1) div ?c
        ≤ d_0 * (?n div ?c + 1) + (b + ?c − 1) div ?c
    proof −
      have h1: T ?n + ?c − 1 ≤ d_0 * ?n + b + ?c − 1
        using T_n_bound by simp
      have h2: d_0 * ?n + b + ?c − 1
              ≤ ?c * (d_0 * (?n div ?c + 1)) + (b + ?c − 1)
        using d0n_le_dc by simp
      have h3: T ?n + ?c − 1
              ≤ ?c * (d_0 * (?n div ?c + 1)) + (b + ?c − 1)
        using h1 h2 by linarith
      have h4: (T ?n + ?c − 1) div ?c
              ≤ (?c * (d_0 * (?n div ?c + 1)) + (b + ?c − 1)) div ?c
        using h3 by (rule div_le_mono)
      have c_nz: ?c ≠ 0 using c_pos by simp
      have h5: (?c * (d_0 * (?n div ?c + 1)) + (b + ?c − 1)) div ?c
              = d_0 * (?n div ?c + 1) + (b + ?c − 1) div ?c
        using c_nz by (rule div_mult_self4)
      from h4 h5 show ?thesis by simp
    qed
```

— Collapse the $(b + c - 1)$ *div* c residue into $b + 1$.
have *b_div_le_b*: b *div* *?c* $\leq b$
by (*rule div_le_dividend*)
have *ceil_b_simple*: $(b + ?c - 1)$ *div* *?c* $\leq b + 1$
proof −
have *a*: $(b + ?c - 1)$ *div* *?c* $\leq b$ *div* *?c* $+ 1$ **by** (*rule ceil_le*)
from *a b_div_le_b* **show** *?thesis* **by** *linarith*
qed

— Full T-ceiling bound via *T_to_c_div* + *ceil_b_simple*, multiplied by 8. Needs an explicit $d_0 * (k+1) = d_0 * k + d_0$ expansion since *linarith* doesn't multiply by variables.
have *eight_T_full*:
$8 * ((T\ ?n + ?c - 1)$ *div* *?c*)
$\leq 8 * d_0 * (?n$ *div* *?c*$) + 8 * d_0 + 8 * b + 8$
proof −
have *d0_succ*: $d_0 * (?n$ *div* *?c* $+ 1) = d_0 * (?n$ *div* *?c*$) + d_0$
by *simp*
have *step*: $(T\ ?n + ?c - 1)$ *div* *?c*
$\leq d_0 * (?n$ *div* *?c*$) + d_0 + b + 1$
using *T_to_c_div d0_succ ceil_b_simple* **by** *linarith*
have *eight_d0*: $8 * (d_0 * (?n$ *div* *?c*$)) = 8 * d_0 * (?n$ *div* *?c*$)$
by (*simp add*: *mult.assoc*)
from *step eight_d0* **show** *?thesis* **by** *linarith*
qed

— Compact form: factor the $(1 + \alpha + 8\ d_0)$ coefficient via *distrib_right*.
have *B1_compact*:
?B1 $\leq ?n + (1 + \alpha + 8 * d_0) * (?n$ *div* *?c*$)$
$+ (14 + \alpha + 8 * d_0 + 8 * b + f)$
proof −
have *combine*:
$(1 + \alpha + 8 * d_0) * (?n$ *div* *?c*$)$
$= ?n$ *div* *?c* $+ \alpha * (?n$ *div* *?c*$) + 8 * d_0 * (?n$ *div* *?c*$)$
by (*simp add*: *algebra_simps*)
show *?thesis*
using *base_ceil_le alpha_ceil_le eight_T_full combine*
by *linarith*
qed

— *c_large* absorption: from $q * (1 + \alpha + 8\ d_0) \leq c$ derive $(1 + \alpha + 8\ d_0) * (n$ *div* $c) \leq n$ *div* q. Uses *mult_div_mod_eq* for $c * (n$ *div* $c) \leq n$ and *less_eq_div_iff_mult_less_eq* for the final step.
have *c_absorb*: $(1 + \alpha + 8 * d_0) * (?n$ *div* *?c*$) \leq ?n$ *div* q
proof −
let *?X* $= (1 + \alpha + 8 * d_0) * (?n$ *div* *?c*$)$
have *eu*: *?c* $* (?n$ *div* *?c*$) + ?n$ *mod* *?c* $= ?n$
by (*rule mult_div_mod_eq*)
have *c_n*: *?c* $* (?n$ *div* *?c*$) \leq ?n$ **using** *eu* **by** *simp*

```
      have h2: q * (1 + α + 8 * d_0) * (?n div ?c) ≤ ?c * (?n div ?c)
        using c_large by (rule mult_le_mono1)
      have assoc: q * (1 + α + 8 * d_0) * (?n div ?c) = q * ?X
        by (simp only: mult.assoc)
      have q_X_le_n: q * ?X ≤ ?n
        using h2 assoc c_n by linarith
      have comm: q * ?X = ?X * q by (metis mult.commute)
      have X_q_le_n: ?X * q ≤ ?n
        using q_X_le_n comm by linarith
      show ?X ≤ ?n div q
        using X_q_le_n q_pos
        by (simp add: less_eq_div_iff_mult_less_eq)
    qed

    — Final composition: B1_compact + c_absorb + K_def via linarith (everything linear in the atomic (1 + α + 8 d_0) * (n div c) term and its absorption).
    have B1_le_target: ?B1 ≤ ?n + ?n div q + K
      using B1_compact c_absorb K_def by linarith

    — Conclude via monotonicity of accepts_in_time_user_wrap in time.
    show accepts_in_time_user_wrap ?M'' w (?n + ?n div q + K)
      using user_mono[OF form_1_accepts B1_le_target] .
  qed
 qed
qed
qed
```

— Discharge the three literal conjuncts. The side condition and the additive constant are stated as literals; the pinning equations $\alpha = 2$, $f = 4$ reconcile them with the α-parametric *time_HU_12_4* ($q(1 + \alpha + 8\ d_0) = q(3 + 8\ d_0)$ and $K = 22 + \alpha + 8\ d_0 + 8\ b + f = 28 + 8\ d_0 + 8\ b$).

```
show valid_mttm ?M'' by (rule wf_M'')
show Lang_user_wrap ?M'' = Lang_mttm M by (rule lang_M'')
show q * (3 + 8 * d_0) ≤ ?c
    ⟶ (∀ w. set w ⊆ Sigma_tm M
            ⟶ accepts_in_time_mttm M w (T (length w))
            ⟶ accepts_in_time_user_wrap ?M'' w
                (length w + length w div q + (28 + 8 * d_0 + 8 * b)))
proof −
  have s0: (3::nat) + 8 * d_0 = 1 + α + 8 * d_0 using alpha2 by simp
  have kk: (28::nat) + 8 * d_0 + 8 * b = K using K_def alpha2 f4 by simp
  show ?thesis unfolding s0 kk by (rule time_HU_12_4)
qed
qed
```

Deterministic specialisation of *linear_speedup_HU_12_4_nae* (Theorem 12.4 proper): adds determinism preservation (via *wrap_det*) on top of the nondeterministic linear-time-case corollary.

theorem *linear_speedup_HU_12_4_dae*:

```
  fixes M :: ('q, 'a) mttm
    and T :: nat ⇒ nat
    and d_0 b q :: nat
  assumes wf:        well_formed_mttm M
      and det:       det_mttm M
      and T_linear:  ∀ n. T n ≤ d_0 * n + b
      and q_pos:     0 < q
      and c_pos:     0 < (card (UNIV :: ('c :: enum) set))
      and k2:        2 ≤ k_tm M
  shows valid_mttm
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
    and det_mttm
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
    and Lang_user_wrap
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
        = Lang_mttm M
    and q * (3 + 8 * d_0) ≤ card (UNIV :: 'c set)
        ⟶ (∀ w. set w ⊆ Sigma_tm M
                ⟶ accepts_in_time_mttm M w (T (length w))
                ⟶ accepts_in_time_user_wrap
                    (encoding_wrap
                      (alphabet_enlarge M
                        :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
                      (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
                      (card (UNIV :: 'c set))
                      (Sigma_tm M))
                    w
                    (length w + length w div q + (28 + 8 * d_0 + 8 * b)))
proof −
  from wf have vM: valid_mttm M
           and le_neq_bl: le_tm M ≠ bl_tm M
    by auto
  let ?M' = alphabet_enlarge M
            :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm
```

```
  let ?c = card (UNIV :: 'c set)
  let ?pack = ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a)
  let ?M'' = encoding_wrap ?M' ?pack ?c (Sigma_tm M)

  have v: valid_mttm ?M''
    and l: Lang_user_wrap ?M'' = Lang_mttm M
    and tm: q * (3 + 8 * d_0) ≤ ?c
          ⟶ (∀ w. set w ⊆ Sigma_tm M
                ⟶ accepts_in_time_mttm M w (T (length w))
                ⟶ accepts_in_time_user_wrap ?M'' w
                    (length w + length w div q + (28 + 8 * d_0 + 8 * b)))
    by (rule linear_speedup_HU_12_4_nae[OF wf T_linear q_pos c_pos k2])+

  have vM': valid_mttm ?M' by (rule alphabet_enlarge_wf[OF vM])
  have detM': det_mttm ?M'
    unfolding det_mttm_def delta_tm_alphabet_enlarge
    using alphabet_enlarge_delta_functional[OF vM det le_neq_bl] by blast
  have finSigma: finite (Sigma_tm M)
    using valid_mttm_Sigma_sub_Gamma[OF vM]
          valid_mttm_finite_Gamma[OF vM]
          rev_finite_subset by blast
  have det_wrap: det_mttm ?M''
    by (rule wrap_det[OF vM' detM' finSigma c_pos])

  show valid_mttm ?M'' by (rule v)
  show det_mttm ?M'' by (rule det_wrap)
  show Lang_user_wrap ?M'' = Lang_mttm M by (rule l)
  show q * (3 + 8 * d_0) ≤ ?c
        ⟶ (∀ w. set w ⊆ Sigma_tm M
              ⟶ accepts_in_time_mttm M w (T (length w))
              ⟶ accepts_in_time_user_wrap ?M'' w
                  (length w + length w div q + (28 + 8 * d_0 + 8 * b)))
    by (rule tm)
qed
```

16.3 Textbook linear-speedup theorem (HU 12.3) faithful k-tape

The textbook linear-speedup theorem, Form 1 [6, Theorem 12.3], faithful at *k_tm M* tapes (matching HU's $k > 1$ via $2 \le k_tm\ M$): the super-linear-T case ($lim\ T(n)/n = \infty$ as the growth hypothesis $\forall d.\ \exists N.\ \forall n \ge N.\ d * n \le T\ n$). This is HU's nondeterministic corollary (a) no *det_mttm M* hypothesis, no determinism asserted of the result. The growth hypothesis absorbs the linear-in-$|w|$ overhead into $T(|w|)/(2q)$, so the Form-1 bound's coefficient-*1* linear term (the encoder pass *length w* plus the storage-tape rewind, and — unlike the transpose's coefficient-*2* *2 * length w* — no input rewind) drops out and the textbook bound *T (length w) div q + K* follows. Reuses the faithful Form-1 raw base *linear_speedup_form_1_raw_nae*, so

the plant wrap carries both HU 12.3 (this, general super-linear T) and HU 12.4 (the tight $(1+\varepsilon)n$ linear-T case above).

```
theorem linear_speedup_HU_12_3_nae:
  fixes M :: ('q, 'a) mttm
    and T :: nat ⇒ nat
    and q :: nat
  assumes wf:        well_formed_mttm M
      and growth:    ∀ d. ∃ N. ∀ n. N ≤ n ⟶ d * n ≤ T n
      and q_pos:     0 < q
      and c_large:   16 * q ≤ card (UNIV :: ('c :: enum) set)
      and k2:        2 ≤ k_tm M
  obtains K N0 :: nat
  where valid_mttm
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
    and Lang_user_wrap
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
        = Lang_mttm M
    and ∀ w. set w ⊆ Sigma_tm M
             ⟶ N0 ≤ length w
             ⟶ accepts_in_time_mttm M w (T (length w))
             ⟶ accepts_in_time_user_wrap
                  (encoding_wrap
                    (alphabet_enlarge M
                      :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
                    (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
                    (card (UNIV :: 'c set))
                    (Sigma_tm M))
                  w
                  (T (length w) div q + K)
proof −
  let ?M' = alphabet_enlarge M
            :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm
  let ?c = card (UNIV :: 'c set)
  let ?pack = ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a)
  let ?M'' = encoding_wrap ?M' ?pack ?c (Sigma_tm M)

  have c_pos: 0 < ?c
    using c_large q_pos by linarith
```

— The AE constants are literal ($\alpha = 2$, $f = 4$); pinned to named locals so the superlinear absorption below reads unchanged.

```
  obtain α f :: nat where alpha2: α = (2::nat) and f4: f = (4::nat)
    by blast
  have wf_M'':   valid_mttm ?M''
    and lang_M'': Lang_user_wrap ?M'' = Lang_mttm M
    and time_F1:  ∀ w. set w ⊆ Sigma_tm M
                  ⟶ accepts_in_time_mttm M w (T (length w))
                  ⟶ accepts_in_time_user_wrap ?M'' w
                      (length w
                       + (length w + ?c − 1) div ?c
                       + 5
                       + α ∗ ((length w + ?c − 1) div ?c)
                       + 8 ∗ ((T (length w) + ?c − 1) div ?c)
                       + f)
    unfolding alpha2 f4
    by (rule linear_speedup_form_1_raw_nae[OF wf c_pos k2])+

  have user_mono:
    ⋀w t1 t2. accepts_in_time_user_wrap ?M'' w t1 ⟹ t1 ≤ t2
              ⟹ accepts_in_time_user_wrap ?M'' w t2
    unfolding accepts_in_time_user_wrap_def accepts_in_time_mttm_def
    by (meson le_trans)

  define d :: nat where d = 2 ∗ q ∗ (2 + α)
  obtain N :: nat where N_growth: ∀ n. N ≤ n ⟶ d ∗ n ≤ T n
    using growth by blast

  define K :: nat where K = 14 + α + f
  define N0 :: nat where N0 = N

  have time_HU_12_3:
    ∀ w. set w ⊆ Sigma_tm M
          ⟶ N0 ≤ length w
          ⟶ accepts_in_time_mttm M w (T (length w))
          ⟶ accepts_in_time_user_wrap ?M'' w
              (T (length w) div q + K)
  proof (intro allI impI)
    fix w :: 'a list
    assume w_sub: set w ⊆ Sigma_tm M
      and n_ge_N0: N0 ≤ length w
      and accepts_M: accepts_in_time_mttm M w (T (length w))

    let ?n = length w
    let ?B1 = ?n + (?n + ?c − 1) div ?c + 5
              + α ∗ ((?n + ?c − 1) div ?c)
              + 8 ∗ ((T ?n + ?c − 1) div ?c) + f

    — Form-1's bound applies to w.
```

have *form_1_accepts*: *accepts_in_time_user_wrap ?M'' w ?B1*
using *time_F1 w_sub accepts_M* **by** *blast*

— Ceiling-via-floor: $(x + c - 1)\ div\ c \leq x\ div\ c + 1$.
have *ceil_le*: $(x + ?c - 1)\ div\ ?c \leq x\ div\ ?c + 1$ **for** $x :: nat$
proof −
have *a*: $x + ?c - 1 \leq x + ?c$ **by** *simp*
have *b*: $(x + ?c)\ div\ ?c = x\ div\ ?c + 1$
using *c_pos* **by** (*simp add*: *div_add_self2*)
from *a* **have** $(x + ?c - 1)\ div\ ?c \leq (x + ?c)\ div\ ?c$
by (*rule div_le_mono*)
with *b* **show** *?thesis* **by** *simp*
qed

— Consequences of *ceil_le*, mirroring HU 12.4.
have *base_ceil_le*: $(?n + ?c - 1)\ div\ ?c \leq ?n\ div\ ?c + 1$
by (*rule ceil_le*)
have *alpha_ceil_le*:
$\alpha * ((?n + ?c - 1)\ div\ ?c) \leq \alpha * (?n\ div\ ?c) + \alpha$
proof −
have $\alpha * ((?n + ?c - 1)\ div\ ?c) \leq \alpha * (?n\ div\ ?c + 1)$
using *base_ceil_le* **by** (*rule mult_le_mono2*)
also have $\ldots = \alpha * (?n\ div\ ?c) + \alpha$ **by** *simp*
finally show *?thesis* **.**
qed
have *eight_T_ceil_le*:
$8 * ((T\ ?n + ?c - 1)\ div\ ?c) \leq 8 * (T\ ?n\ div\ ?c) + 8$
proof −
have $8 * ((T\ ?n + ?c - 1)\ div\ ?c) \leq 8 * (T\ ?n\ div\ ?c + 1)$
using *ceil_le*[*of T ?n*] **by** (*rule mult_le_mono2*)
also have $\ldots = 8 * (T\ ?n\ div\ ?c) + 8$ **by** *simp*
finally show *?thesis* **.**
qed

— *c_large* absorption: from $16 * q \leq c$ derive $8 * (T\ n\ div\ c) \leq T\ n\ div\ (2 * q)$.
have *eight_T_to_2q*: $8 * (T\ ?n\ div\ ?c) \leq T\ ?n\ div\ (2 * q)$
proof −
let *?Y* = *T ?n div ?c*
let *?Z* = $8 * ?Y$
have *c_n*: $?c * ?Y \leq T\ ?n$
using *mult_div_mod_eq*[*of ?c T ?n*] **by** *linarith*
have *c16*: $16 * q * ?Y \leq ?c * ?Y$
using *c_large* **by** (*rule mult_le_mono1*)
have *step1*: $16 * q * ?Y \leq T\ ?n$ **using** *c16 c_n* **by** *linarith*
have *assoc*: $16 * q * ?Y = ?Z * (2 * q)$ **by** *simp*
have *step2*: $?Z * (2 * q) \leq T\ ?n$ **using** *step1 assoc* **by** *linarith*
have *q2_pos*: $0 < 2 * q$ **using** *q_pos* **by** *linarith*
show *?thesis* **using** *step2 q2_pos*

```
    by (simp add: less_eq_div_iff_mult_less_eq)
  qed

  — Growth at length w: from N0 ≤ ?n and N0 = N.
  have growth_at_n: d * ?n ≤ T ?n
  proof −
    have N ≤ ?n using n_ge_N0 unfolding N0_def by simp
    then show ?thesis using N_growth by blast
  qed

  — Convert to the T n div (2 * q) form.
  have linear_to_2q: (2 + α) * ?n ≤ T ?n div (2 * q)
  proof −
    have d_assoc: d * ?n = ((2 + α) * ?n) * (2 * q)
      unfolding d_def by (simp add: algebra_simps)
    have step1: ((2 + α) * ?n) * (2 * q) ≤ T ?n
      using growth_at_n d_assoc by linarith
    have q2_pos: 0 < 2 * q using q_pos by linarith
    show ?thesis using step1 q2_pos
      by (simp add: less_eq_div_iff_mult_less_eq)
  qed

  — Universal nat-fact: 2 * (T n div (2 * q)) ≤ T n div q.
  have double_div: 2 * (T ?n div (2 * q)) ≤ T ?n div q
  proof −
    let ?D = T ?n div (2 * q)
    have q2_n: (2 * q) * ?D ≤ T ?n
      using mult_div_mod_eq[of 2 * q T ?n] by linarith
    have rebracket: (2 * q) * ?D = (2 * ?D) * q
      by (simp add: ac_simps)
    have step: (2 * ?D) * q ≤ T ?n
      using q2_n rebracket by linarith
    show ?thesis using step q_pos
      by (simp add: less_eq_div_iff_mult_less_eq)
  qed

  — Arithmetic discharge: ?B1 ≤ T ?n div q + K. The plant wrap's leading
linear term is n plus the storage-rewind ceiling (n + c − 1) div c (coefficient 1,
no input rewind); the ceiling is bounded by n + 1 via ceil_le, so the same growth
absorption as the transpose's 2 · n applies.
  have B1_le_target: ?B1 ≤ T ?n div q + K
  proof −
    have n_div_n: ?n div ?c ≤ ?n by (rule div_le_dividend)
    have ceil_n_bound: (?n + ?c − 1) div ?c ≤ ?n + 1
      using base_ceil_le n_div_n by linarith
    have alpha_n_div: α * (?n div ?c) ≤ α * ?n
      using n_div_n by (rule mult_le_mono2)
    have b_n: α * ((?n + ?c − 1) div ?c) ≤ α * ?n + α
      using alpha_ceil_le alpha_n_div by linarith
```

```
    have eight_T_full:
      8 * ((T ?n + ?c - 1) div ?c) ≤ T ?n div (2 * q) + 8
      using eight_T_ceil_le eight_T_to_2q by linarith
    have lin_split: ?n + ?n + α * ?n ≤ T ?n div (2 * q)
    proof -
      have eq: ?n + ?n + α * ?n = (2 + α) * ?n
        by (simp add: algebra_simps)
      show ?thesis unfolding eq by (rule linear_to_2q)
    qed
    show ?thesis
      using b_n eight_T_full lin_split double_div K_def ceil_n_bound
      by linarith
  qed

  show accepts_in_time_user_wrap ?M'' w (T ?n div q + K)
    using user_mono[OF form_1_accepts B1_le_target] .
 qed

 show ?thesis
 proof (rule that)
  show valid_mttm ?M'' by (rule wf_M'')
  show Lang_user_wrap ?M'' = Lang_mttm M by (rule lang_M'')
  show ∀ w. set w ⊆ Sigma_tm M
            ⟶ N0 ≤ length w
            ⟶ accepts_in_time_mttm M w (T (length w))
            ⟶ accepts_in_time_user_wrap ?M'' w
                (T (length w) div q + K)
    by (rule time_HU_12_3)
 qed
qed
```

Deterministic specialisation of *linear_speedup_HU_12_3_nae* (Theorem 12.3 proper, faithful k-tape): adds determinism preservation on top of the nondeterministic super-linear-case corollary.

```
theorem linear_speedup_HU_12_3_dae:
 fixes M :: ('q, 'a) mttm
   and T :: nat ⇒ nat
   and q :: nat
 assumes wf:        well_formed_mttm M
     and det:       det_mttm M
     and growth:    ∀ d. ∃ N. ∀ n. N ≤ n ⟶ d * n ≤ T n
     and q_pos:     0 < q
     and c_large:   16 * q ≤ card (UNIV :: ('c :: enum) set)
     and k2:        2 ≤ k_tm M
 obtains K N0 :: nat
 where valid_mttm
        (encoding_wrap
          (alphabet_enlarge M
            :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
```

```
      (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
      (card (UNIV :: 'c set))
      (Sigma_tm M))
  and det_mttm
    (encoding_wrap
      (alphabet_enlarge M
        :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
      (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
      (card (UNIV :: 'c set))
      (Sigma_tm M))
  and Lang_user_wrap
    (encoding_wrap
      (alphabet_enlarge M
        :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
      (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
      (card (UNIV :: 'c set))
      (Sigma_tm M))
    = Lang_mttm M
  and ∀ w. set w ⊆ Sigma_tm M
        ⟶ N0 ≤ length w
        ⟶ accepts_in_time_mttm M w (T (length w))
        ⟶ accepts_in_time_user_wrap
            (encoding_wrap
              (alphabet_enlarge M
                :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
              (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
              (card (UNIV :: 'c set))
              (Sigma_tm M))
            w
            (T (length w) div q + K)
proof −
  from wf have vM: valid_mttm M
       and le_neq_bl: le_tm M ≠ bl_tm M
    by auto
  let ?M' = alphabet_enlarge M
        :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm
  let ?c = card (UNIV :: 'c set)
  let ?pack = ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a)
  let ?M'' = encoding_wrap ?M' ?pack ?c (Sigma_tm M)

  have c_pos: 0 < ?c
  proof −
    have 0 < 16 * q using q_pos by simp
    thus ?thesis using c_large by linarith
  qed

  obtain K N0 :: nat where
    v: valid_mttm ?M''
    and l: Lang_user_wrap ?M'' = Lang_mttm M
```

```
  and tm: ∀ w. set w ⊆ Sigma_tm M
            ⟶ N0 ≤ length w
            ⟶ accepts_in_time_mttm M w (T (length w))
            ⟶ accepts_in_time_user_wrap ?M'' w
                (T (length w) div q + K)
  by (rule linear_speedup_HU_12_3_nae[OF wf growth q_pos c_large k2])

 have vM': valid_mttm ?M' by (rule alphabet_enlarge_wf[OF vM])
 have detM': det_mttm ?M'
  unfolding det_mttm_def delta_tm_alphabet_enlarge
  using alphabet_enlarge_delta_functional[OF vM det le_neq_bl] by blast
 have finSigma: finite (Sigma_tm M)
  using valid_mttm_Sigma_sub_Gamma[OF vM]
       valid_mttm_finite_Gamma[OF vM]
       rev_finite_subset by blast
 have det_wrap: det_mttm ?M''
  by (rule wrap_det[OF vM' detM' finSigma c_pos])

 show ?thesis
 proof (rule that)
  show valid_mttm ?M'' by (rule v)
  show det_mttm ?M'' by (rule det_wrap)
  show Lang_user_wrap ?M'' = Lang_mttm M by (rule l)
  show ∀ w. set w ⊆ Sigma_tm M
            ⟶ N0 ≤ length w
            ⟶ accepts_in_time_mttm M w (T (length w))
            ⟶ accepts_in_time_user_wrap ?M'' w
                (T (length w) div q + K)
    by (rule tm)
 qed
qed

end
theory Wrap_Convention
 imports Wrap_Speedup Multitape_TM_Substrate.Multitape_Time_Convention
begin
```

17 The linear-speedup headlines in Hopcroft–Ullman's time-complexity convention

The *Multitape_Alphabet_Enlargement.Wrap_Speedup* headlines *linear_speedup_HU_12_3 / linear_speedup_HU_12_4* prove a *raw* bound: a per-run additive constant K (and for 12.3 a bare length threshold *N0*), gated on the input run itself completing within T. Hopcroft and Ullman state 12.3 / 12.4 in a *time-complexity convention* [6, p. 291]: a machine "runs in time $cT(n)$" means it accepts every word of its language within $max(n + 1,\ ceil(cT(n)))$ steps, the $n + 1$ floor being the cost of reading

the input. On our substrate the floor is $n + 2$ (the left endmarker is read before the first input symbol); this is exactly the *time_bounded_conv* predicate of *Multitape_TM_Substrate.Multitape_Time_Convention*.

This theory restates the two headlines in that convention. The wrapped machine is a genuine *mttm*, and its user-facing language / timing (*Lang_user_wrap* / *accepts_in_time_user_wrap*) are the *Lang_mttm* / *accepts_in_time_mttm* of the wrap read through *Raw*, so *time_bounded_conv* applied to the wrap *is* the user-level convention statement (*Lang_mttm_encoding_wrap_ex_Raw* below is the bridge). The two cases diverge:

- **12.4 (linear T).** The raw bound $|w| + |w|$ *div* $q + K$ already holds for *all* lengths, so the convention statement is the max-floored restatement, provided M runs in time T on its language. The additive K cannot be folded into a smaller $eps = 1 / q$ on all inputs: the clean form *time_bounded_conv* M $(\lambda n.\ n + n\ div\ q)$ is provably *unattainable* in general (the counterexample *clean_convention_unattainable* in *Multitape_TM_Substrate.Multitape_Time_Convention* — a valid machine can accept its language yet exceed the floor $n + 2$ on its shortest input, as the wrap's own setup phases do at $n = 0$). The obstruction is the finite-prefix *band*: an outer *finite_patch* cleanup scans proportionally to its cutoff, so its overhead never drops below *eps*, and shaving the rewind only narrows the band. It is *not* the origin rewind of the faithful k-tape construction (a distinct obstruction). So K is kept explicit.

- **12.3 (superlinear T).** Here superlinear growth dominates any linear-in-cutoff cost, so the *finite_patch* small-input cleanup (*time_cleanup* on the table $\lambda w.\ w \in$ *Lang_mttm* M) and the additive constant both wash out into a slightly larger multiplicative constant — a clean *ceil*$(c\ T(n))$ convention bound with no residual constant.

  The raw $+ K$ theorems remain the explicit-constant corollaries (nothing is lost).

17.1 Bridge: an accepted word of the wrap is *Raw*-encoded

Every word of *Lang_mttm* of an *encoding_wrap* is *map Raw w* for a user word w, because the wrap's input alphabet is *Raw* ' Σu. This lets a *time_bounded_conv* goal over the wrap's *Lang_mttm* be discharged through the user-level *accepts_in_time_user_wrap* bound the raw headlines supply, and conversely.

lemma *Lang_mttm_encoding_wrap_ex_Raw*:
  assumes $v \in$ *Lang_mttm* (*encoding_wrap M pack c* Σu)

shows $\exists w.\ v = map\ Raw\ w \wedge w \in Lang_user_wrap\ (encoding_wrap\ M\ pack\ c\ \Sigma u)$
proof −
let *?W* = *encoding_wrap M pack c* Σ*u*
have *sv*: *set v* ⊆ *Raw* ' Σ*u*
using *assms* **unfolding** *Lang_mttm_def* **by** *simp*
have ∀ *x*∈*set v*. ∃ *y*. *x* = *Raw y* **using** *sv* **by** *auto*
then obtain *w* **where** *vw*: *v* = *map Raw w* **by** (*metis ex_map_conv*)
have *map Raw w* ∈ *Lang_mttm ?W* **using** *assms vw* **by** *simp*
hence *w* ∈ *Lang_user_wrap ?W* **unfolding** *Lang_user_wrap_def* **by** *simp*
thus *?thesis* **using** *vw* **by** *blast*
qed

17.2 Arithmetic core of the superlinear absorption

The one inequality that makes 12.3 clean where 12.4 is not: a constant C is absorbed by dropping the speedup denominator from $q + 1$ to q, provided the numerator a is at least $q\ (q + 1)\ C$. In use $a = T\ n$, and the superlinear growth of T makes the premise hold for all large n — including $C = K + 2\ N1 + 4$ where *N1* is the finite-control cutoff, because the numerator grows superlinearly in n while C grows only linearly in *N1*.

lemma *div_absorb_step*:
fixes *a q C* :: *nat*
assumes *q_pos*: 0 < *q* **and** *big*: *q* ∗ (*q* + 1) ∗ *C* ≤ *a*
shows *a div* (*q* + 1) + *C* ≤ *a div q*
proof −
have *qne*: *q* ≠ 0 **using** *q_pos* **by** *simp*
have *q1ne*: *q* + 1 ≠ 0 **by** *simp*
have *qC*: *q* ∗ *C* ≤ *a div* (*q* + 1)
proof −
have (*q* + 1) ∗ (*q* ∗ *C*) ≤ *a* **using** *big* **by** (*simp add*: *algebra_simps*)
hence ((*q* + 1) ∗ (*q* ∗ *C*)) *div* (*q* + 1) ≤ *a div* (*q* + 1) **by** (*rule div_le_mono*)
moreover have ((*q* + 1) ∗ (*q* ∗ *C*)) *div* (*q* + 1) = *q* ∗ *C*
by (*metis nonzero_mult_div_cancel_left q1ne*)
ultimately show *?thesis* **by** *simp*
qed
have *q* ∗ (*a div* (*q* + 1) + *C*) = *q* ∗ (*a div* (*q* + 1)) + *q* ∗ *C*
by (*simp add*: *algebra_simps*)
also have ... ≤ *q* ∗ (*a div* (*q* + 1)) + *a div* (*q* + 1) **using** *qC* **by** *simp*
also have ... = (*q* + 1) ∗ (*a div* (*q* + 1)) **by** (*simp add*: *algebra_simps*)
also have ... ≤ *a* **by** (*metis mult.commute div_times_less_eq_dividend*)
finally have *le_a*: *q* ∗ (*a div* (*q* + 1) + *C*) ≤ *a* **.**
hence (*q* ∗ (*a div* (*q* + 1) + *C*)) *div q* ≤ *a div q* **by** (*rule div_le_mono*)
moreover have (*q* ∗ (*a div* (*q* + 1) + *C*)) *div q* = *a div* (*q* + 1) + *C*
by (*metis nonzero_mult_div_cancel_left qne*)
ultimately show *?thesis* **by** *simp*
qed

17.3 HU 12.4 in the convention (linear T)

The nondeterministic linear-T headline, restated as a *time_bounded_conv* bound. The extra hypothesis over the raw *linear_speedup_HU_12_4_nae* is *Mtime*: M accepts every word of its language within T — i.e. $L(M)$ *is* a T-time language, the textbook premise. The bound $\lambda n.\ n + n\ div\ q + K$ is the raw $+\ K$ constant read in the $max(n + 2, ...)$ convention.

```
theorem linear_speedup_HU_12_4_nae_conv:
  fixes M :: ('q, 'a) mttm
    and T :: nat ⇒ nat
    and d_0 b q :: nat
  assumes wf:        well_formed_mttm M
      and Mtime:     ⋀w. w ∈ Lang_mttm M ⟹ accepts_in_time_mttm M w (T
(length w))
      and T_linear:  ∀n. T n ≤ d_0 * n + b
      and q_pos:     0 < q
      and c_pos:     0 < (card (UNIV :: ('c :: enum) set))
      and k2:        2 ≤ k_tm M
  shows valid_mttm
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
    and Lang_user_wrap
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
         = Lang_mttm M
    and q * (3 + 8 * d_0) ≤ card (UNIV :: 'c set)
        ⟶ time_bounded_conv
            (encoding_wrap
              (alphabet_enlarge M
                :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
              (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
              (card (UNIV :: 'c set))
              (Sigma_tm M))
            (λn. n + n div q + (28 + 8 * d_0 + 8 * b))
proof −
  let ?W = encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M)
```

```
  obtain α K :: nat where alpha2: α = (2::nat)
    and Kdef: K = 28 + 8 * d_0 + 8 * b by blast
  have V: valid_mttm ?W
    and L: Lang_user_wrap ?W = Lang_mttm M
    and Tb': q * (3 + 8 * d_0) ≤ card (UNIV :: 'c set)
        ⟶ (∀ w. set w ⊆ Sigma_tm M
                ⟶ accepts_in_time_mttm M w (T (length w))
                ⟶ accepts_in_time_user_wrap ?W w
                    (length w + length w div q + (28 + 8 * d_0 + 8 * b)))
    by (rule linear_speedup_HU_12_4_nae[OF wf T_linear q_pos c_pos k2])+
  have e: (1::nat) + α + 8 * d_0 = 3 + 8 * d_0 using alpha2 by simp
  have Tb: q * (1 + α + 8 * d_0) ≤ card (UNIV :: 'c set)
        ⟶ (∀ w. set w ⊆ Sigma_tm M
                ⟶ accepts_in_time_mttm M w (T (length w))
                ⟶ accepts_in_time_user_wrap ?W w
                    (length w + length w div q + K))
    unfolding e Kdef by (rule Tb')
  have conv: q * (1 + α + 8 * d_0) ≤ card (UNIV :: 'c set)
          ⟶ time_bounded_conv ?W (λn. n + n div q + K)
  proof
    assume cL: q * (1 + α + 8 * d_0) ≤ card (UNIV :: 'c set)
    show time_bounded_conv ?W (λn. n + n div q + K)
      unfolding time_bounded_conv_def
    proof
      fix v assume vW: v ∈ Lang_mttm ?W
     then obtain w where vmap: v = map Raw w and wU: w ∈ Lang_user_wrap
?W
        using Lang_mttm_encoding_wrap_ex_Raw by blast
      from wU L have wM: w ∈ Lang_mttm M by simp
      hence wSg: set w ⊆ Sigma_tm M unfolding Lang_mttm_def by simp
      from wM have accM: accepts_in_time_mttm M w (T (length w)) by (rule
Mtime)
      have accepts_in_time_user_wrap ?W w (length w + length w div q + K)
        using Tb cL wSg accM by blast
      hence accepts_in_time_mttm ?W v (length w + length w div q + K)
        using vmap unfolding accepts_in_time_user_wrap_def by simp
      hence accepts_in_time_mttm ?W v (length v + length v div q + K)
        using vmap by simp
      thus accepts_in_time_mttm ?W v
            (max (length v + 2) ((λn. n + n div q + K) (length v)))
        by (auto elim: accepts_in_time_mttm_mono)
    qed
  qed
  show valid_mttm ?W by (rule V)
  show Lang_user_wrap ?W = Lang_mttm M by (rule L)
  show q * (3 + 8 * d_0) ≤ card (UNIV :: 'c set)
       ⟶ time_bounded_conv ?W (λn. n + n div q + (28 + 8 * d_0 + 8 * b))
  proof −
    have s0: (3::nat) + 8 * d_0 = 1 + α + 8 * d_0 using alpha2 by simp
```

show *?thesis* **unfolding** *s0 Kdef*[*symmetric*] **by** (*rule conv*)
qed
qed

The nondeterministic linear-T headline in the *eventual*, constant-free form: past an explicit input-length threshold the wrap runs in $n + n\ div\ q$ exactly — the additive K of *linear_speedup_HU_12_4_nae_conv* is gone, at the cost of a hypothesis $q{\cdot}(q{+}1){\cdot}K \le length\ v$ instead of the convention floor. This is the honest $(1{+}\varepsilon){\cdot}n$ shape (with $\varepsilon = 1/q$): clean coefficient, no residual constant, but only for long enough inputs — the small-input band is not covered (it cannot be, without the non-effective finite-exceptions table; see the counterexample *clean_convention_unattainable*). Proof: run *linear_speedup_HU_12_4_nae* one denominator tighter (at $q{+}1$), then absorb K into the extra *div*-slack via *div_absorb_step* once $q{\cdot}(q{+}1){\cdot}K \le length\ v$ — the same inequality that makes the superlinear 12.3 constant-free, here read in the linear regime as an explicit threshold rather than an absorbed constant. With K now the literal $28 + 8\ d_0 + 8\ b$, the crossover threshold is the closed formula $q{\cdot}(q{+}1){\cdot}(28 + 8\ d_0 + 8\ b)$ — quadratic in q (i.e. $O(1/\varepsilon^2)$), and linear in the input machine's time-bound constants d_0, b.

theorem *linear_speedup_HU_12_4_nae_eventual*:
fixes M :: $('q, 'a)$ *mttm*
and T :: $nat \Rightarrow nat$
and $d_0\ b\ q$ :: *nat*
assumes *wf*: *well_formed_mttm M*
and *Mtime*: $\bigwedge w.\ w \in Lang_mttm\ M \Longrightarrow accepts_in_time_mttm\ M\ w\ (T\ (length\ w))$
and *T_linear*: $\forall n.\ T\ n \le d_0 * n + b$
and *q_pos*: $0 < q$
and *c_pos*: $0 < (card\ (UNIV :: ('c :: enum)\ set))$
and *k2*: $2 \le k_tm\ M$
shows *valid_mttm*
(*encoding_wrap*
(*alphabet_enlarge M*
:: $('q \times ('a, 'c)\ ae_stage, 'c \Rightarrow 'a)\ mttm$)
(*ae_pack* (*bl_tm M*) :: $'a\ list \Rightarrow ('c \Rightarrow 'a)$)
(*card* (*UNIV* :: $'c\ set$))
(*Sigma_tm M*))
and *Lang_user_wrap*
(*encoding_wrap*
(*alphabet_enlarge M*
:: $('q \times ('a, 'c)\ ae_stage, 'c \Rightarrow 'a)\ mttm$)
(*ae_pack* (*bl_tm M*) :: $'a\ list \Rightarrow ('c \Rightarrow 'a)$)
(*card* (*UNIV* :: $'c\ set$))
(*Sigma_tm M*))
= *Lang_mttm M*
and $(q + 1) * (3 + 8 * d_0) \le card\ (UNIV :: 'c\ set)$

```
      ⟶ (∀ v ∈ Lang_mttm
            (encoding_wrap
              (alphabet_enlarge M
                 :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
              (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
              (card (UNIV :: 'c set))
              (Sigma_tm M)).
          q * (q + 1) * (28 + 8 * d_0 + 8 * b) ≤ length v
          ⟶ accepts_in_time_mttm
              (encoding_wrap
                (alphabet_enlarge M
                   :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
                (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
                (card (UNIV :: 'c set))
                (Sigma_tm M))
              v
              (length v + length v div q))
proof −
  let ?W = encoding_wrap
          (alphabet_enlarge M
             :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
          (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
          (card (UNIV :: 'c set))
          (Sigma_tm M)
  have q1_pos: 0 < q + 1 by simp
  obtain α K :: nat where alpha2: α = (2::nat)
    and Kdef: K = 28 + 8 * d_0 + 8 * b by blast
  have V: valid_mttm ?W
    and L: Lang_user_wrap ?W = Lang_mttm M
    and Tb': (q + 1) * (3 + 8 * d_0) ≤ card (UNIV :: 'c set)
        ⟶ (∀ w. set w ⊆ Sigma_tm M
              ⟶ accepts_in_time_mttm M w (T (length w))
              ⟶ accepts_in_time_user_wrap ?W w
                 (length w + length w div (q + 1) + (28 + 8 * d_0 + 8 * b)))
    by (rule linear_speedup_HU_12_4_nae[where q = q + 1, OF wf T_linear
q1_pos c_pos k2])+
  have e: (1::nat) + α + 8 * d_0 = 3 + 8 * d_0 using alpha2 by simp
  have Tb: (q + 1) * (1 + α + 8 * d_0) ≤ card (UNIV :: 'c set)
        ⟶ (∀ w. set w ⊆ Sigma_tm M
              ⟶ accepts_in_time_mttm M w (T (length w))
              ⟶ accepts_in_time_user_wrap ?W w
                   (length w + length w div (q + 1) + K))
    unfolding e Kdef by (rule Tb')
  have ev: (q + 1) * (1 + α + 8 * d_0) ≤ card (UNIV :: 'c set)
          ⟶ (∀ v ∈ Lang_mttm ?W. q * (q + 1) * K ≤ length v
              ⟶ accepts_in_time_mttm ?W v (length v + length v div q))
  proof
    assume cL: (q + 1) * (1 + α + 8 * d_0) ≤ card (UNIV :: 'c set)
    show ∀ v ∈ Lang_mttm ?W. q * (q + 1) * K ≤ length v
```

```
        ⟶ accepts_in_time_mttm ?W v (length v + length v div q)
   proof
    fix v assume vW: v ∈ Lang_mttm ?W
    show q * (q + 1) * K ≤ length v
          ⟶ accepts_in_time_mttm ?W v (length v + length v div q)
    proof
     assume lv: q * (q + 1) * K ≤ length v
     from vW obtain w where vmap: v = map Raw w
      and wU: w ∈ Lang_user_wrap ?W
      using Lang_mttm_encoding_wrap_ex_Raw by blast
     from wU L have wM: w ∈ Lang_mttm M by simp
     hence wSg: set w ⊆ Sigma_tm M unfolding Lang_mttm_def by simp
    from wM have accM: accepts_in_time_mttm M w (T (length w)) by (rule
Mtime)
     have lenwv: length w = length v using vmap by simp
     have raw: accepts_in_time_user_wrap ?W w
                (length w + length w div (q + 1) + K)
      using Tb cL wSg accM by blast
     have absorb: length w div (q + 1) + K ≤ length w div q
      by (rule div_absorb_step[OF q_pos]) (use lv lenwv in simp)
     hence le: length w + length w div (q + 1) + K
                ≤ length w + length w div q by simp
     have accepts_in_time_mttm ?W (map Raw w)
           (length w + length w div (q + 1) + K)
      using raw unfolding accepts_in_time_user_wrap_def by simp
     hence accepts_in_time_mttm ?W (map Raw w) (length w + length w div
q)
      using le by (auto elim: accepts_in_time_mttm_mono)
     thus accepts_in_time_mttm ?W v (length v + length v div q)
      by (simp only: vmap length_map)
    qed
   qed
  qed
  show valid_mttm ?W by (rule V)
  show Lang_user_wrap ?W = Lang_mttm M by (rule L)
  show (q + 1) * (3 + 8 * d_0) ≤ card (UNIV :: 'c set)
        ⟶ (∀ v ∈ Lang_mttm ?W. q * (q + 1) * (28 + 8 * d_0 + 8 * b) ≤
length v
              ⟶ accepts_in_time_mttm ?W v (length v + length v div q))
  proof −
   have s0: (3::nat) + 8 * d_0 = 1 + α + 8 * d_0 using alpha2 by simp
   show ?thesis unfolding s0 Kdef[symmetric] by (rule ev)
  qed
qed
```

The deterministic specialisation: adds the *det* hypothesis on M and the determinism-preservation conjunct on the wrap, on top of *linear_speedup_HU_12_4_nae_conv*.

```
theorem linear_speedup_HU_12_4_dae_conv:
```

```
  fixes M :: ('q, 'a) mttm
    and T :: nat ⇒ nat
    and d_0 b q :: nat
  assumes wf:        well_formed_mttm M
      and det:       det_mttm M
      and Mtime:     ⋀w. w ∈ Lang_mttm M ⟹ accepts_in_time_mttm M w (T
(length w))
      and T_linear:  ∀ n. T n ≤ d_0 * n + b
      and q_pos:     0 < q
      and c_pos:     0 < (card (UNIV :: ('c :: enum) set))
      and k2:        2 ≤ k_tm M
  shows valid_mttm
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
    and det_mttm
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
    and Lang_user_wrap
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
        = Lang_mttm M
    and q * (3 + 8 * d_0) ≤ card (UNIV :: 'c set)
        ⟶ time_bounded_conv
            (encoding_wrap
              (alphabet_enlarge M
                :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
              (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
              (card (UNIV :: 'c set))
              (Sigma_tm M))
            (λn. n + n div q + (28 + 8 * d_0 + 8 * b))
proof −
  let ?W = encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M)
```

```
 obtain α K :: nat where alpha2: α = (2::nat)
  and Kdef: K = 28 + 8 * d_0 + 8 * b by blast
 have V: valid_mttm ?W
  and D: det_mttm ?W
  and L: Lang_user_wrap ?W = Lang_mttm M
  and Tb': q * (3 + 8 * d_0) ≤ card (UNIV :: 'c set)
       ⟶ (∀ w. set w ⊆ Sigma_tm M
              ⟶ accepts_in_time_mttm M w (T (length w))
              ⟶ accepts_in_time_user_wrap ?W w
                   (length w + length w div q + (28 + 8 * d_0 + 8 * b)))
 by (rule linear_speedup_HU_12_4_dae[OF wf det T_linear q_pos c_pos k2])+
 have e: (1::nat) + α + 8 * d_0 = 3 + 8 * d_0 using alpha2 by simp
 have Tb: q * (1 + α + 8 * d_0) ≤ card (UNIV :: 'c set)
       ⟶ (∀ w. set w ⊆ Sigma_tm M
              ⟶ accepts_in_time_mttm M w (T (length w))
              ⟶ accepts_in_time_user_wrap ?W w
                   (length w + length w div q + K))
  unfolding e Kdef by (rule Tb')
 have conv: q * (1 + α + 8 * d_0) ≤ card (UNIV :: 'c set)
          ⟶ time_bounded_conv ?W (λn. n + n div q + K)
 proof
  assume cL: q * (1 + α + 8 * d_0) ≤ card (UNIV :: 'c set)
  show time_bounded_conv ?W (λn. n + n div q + K)
   unfolding time_bounded_conv_def
  proof
   fix v assume vW: v ∈ Lang_mttm ?W
   then obtain w where vmap: v = map Raw w and wU: w ∈ Lang_user_wrap
?W
     using Lang_mttm_encoding_wrap_ex_Raw by blast
    from wU L have wM: w ∈ Lang_mttm M by simp
    hence wSg: set w ⊆ Sigma_tm M unfolding Lang_mttm_def by simp
    from wM have accM: accepts_in_time_mttm M w (T (length w)) by (rule
Mtime)
    have accepts_in_time_user_wrap ?W w (length w + length w div q + K)
     using Tb cL wSg accM by blast
    hence accepts_in_time_mttm ?W v (length w + length w div q + K)
     using vmap unfolding accepts_in_time_user_wrap_def by simp
    hence accepts_in_time_mttm ?W v (length v + length v div q + K)
     using vmap by simp
    thus accepts_in_time_mttm ?W v
          (max (length v + 2) ((λn. n + n div q + K) (length v)))
     by (auto elim: accepts_in_time_mttm_mono)
  qed
 qed
 show valid_mttm ?W by (rule V)
 show det_mttm ?W by (rule D)
 show Lang_user_wrap ?W = Lang_mttm M by (rule L)
 show q * (3 + 8 * d_0) ≤ card (UNIV :: 'c set)
      ⟶ time_bounded_conv ?W (λn. n + n div q + (28 + 8 * d_0 + 8 * b))
```

```
  proof −
    have s0: (3::nat) + 8 * d_0 = 1 + α + 8 * d_0 using alpha2 by simp
    show ?thesis unfolding s0 Kdef[symmetric] by (rule conv)
  qed
qed
```

The determinism-preserving eventual form: *linear_speedup_HU_12_4_nae_eventual* with the *det* hypothesis and the determinism-preservation conjunct, over *linear_speedup_HU_12_4_dae*.

```
theorem linear_speedup_HU_12_4_dae_eventual:
  fixes M :: ('q, 'a) mttm
    and T :: nat ⇒ nat
    and d_0 b q :: nat
  assumes wf:        well_formed_mttm M
      and det:       det_mttm M
      and Mtime:     ⋀w. w ∈ Lang_mttm M ⟹ accepts_in_time_mttm M w (T
(length w))
      and T_linear:  ∀ n. T n ≤ d_0 * n + b
      and q_pos:     0 < q
      and c_pos:     0 < (card (UNIV :: ('c :: enum) set))
      and k2:        2 ≤ k_tm M
  shows valid_mttm
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
    and det_mttm
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
    and Lang_user_wrap
          (encoding_wrap
            (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
            (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
            (card (UNIV :: 'c set))
            (Sigma_tm M))
        = Lang_mttm M
    and (q + 1) * (3 + 8 * d_0) ≤ card (UNIV :: 'c set)
        ⟶ (∀ v ∈ Lang_mttm
                  (encoding_wrap
                    (alphabet_enlarge M
                      :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
                    (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
```

```
                (card (UNIV :: 'c set))
                (Sigma_tm M)).
            q * (q + 1) * (28 + 8 * d_0 + 8 * b) ≤ length v
            ⟶ accepts_in_time_mttm
                (encoding_wrap
                  (alphabet_enlarge M
                    :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
                  (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
                  (card (UNIV :: 'c set))
                  (Sigma_tm M))
                v
                (length v + length v div q))
proof −
  let ?W = encoding_wrap
           (alphabet_enlarge M
             :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
           (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
           (card (UNIV :: 'c set))
           (Sigma_tm M)
  have q1_pos: 0 < q + 1 by simp
  obtain α K :: nat where alpha2: α = (2::nat)
    and Kdef: K = 28 + 8 * d_0 + 8 * b by blast
  have V: valid_mttm ?W
    and D: det_mttm ?W
    and L: Lang_user_wrap ?W = Lang_mttm M
    and Tb': (q + 1) * (3 + 8 * d_0) ≤ card (UNIV :: 'c set)
        ⟶ (∀ w. set w ⊆ Sigma_tm M
                ⟶ accepts_in_time_mttm M w (T (length w))
                ⟶ accepts_in_time_user_wrap ?W w
                  (length w + length w div (q + 1) + (28 + 8 * d_0 + 8 * b)))
   by (rule linear_speedup_HU_12_4_dae[where q = q + 1, OF wf det T_linear
q1_pos c_pos k2])+
  have e: (1::nat) + α + 8 * d_0 = 3 + 8 * d_0 using alpha2 by simp
  have Tb: (q + 1) * (1 + α + 8 * d_0) ≤ card (UNIV :: 'c set)
        ⟶ (∀ w. set w ⊆ Sigma_tm M
                ⟶ accepts_in_time_mttm M w (T (length w))
                ⟶ accepts_in_time_user_wrap ?W w
                    (length w + length w div (q + 1) + K))
    unfolding e Kdef by (rule Tb')
  have ev: (q + 1) * (1 + α + 8 * d_0) ≤ card (UNIV :: 'c set)
          ⟶ (∀ v ∈ Lang_mttm ?W. q * (q + 1) * K ≤ length v
                ⟶ accepts_in_time_mttm ?W v (length v + length v div q))
  proof
    assume cL: (q + 1) * (1 + α + 8 * d_0) ≤ card (UNIV :: 'c set)
    show ∀ v ∈ Lang_mttm ?W. q * (q + 1) * K ≤ length v
         ⟶ accepts_in_time_mttm ?W v (length v + length v div q)
    proof
      fix v assume vW: v ∈ Lang_mttm ?W
      show q * (q + 1) * K ≤ length v
```

```
        ⟶ accepts_in_time_mttm ?W v (length v + length v div q)
    proof
      assume lv: q * (q + 1) * K ≤ length v
      from vW obtain w where vmap: v = map Raw w
        and wU: w ∈ Lang_user_wrap ?W
        using Lang_mttm_encoding_wrap_ex_Raw by blast
      from wU L have wM: w ∈ Lang_mttm M by simp
      hence wSg: set w ⊆ Sigma_tm M unfolding Lang_mttm_def by simp
     from wM have accM: accepts_in_time_mttm M w (T (length w)) by (rule
Mtime)
      have lenwv: length w = length v using vmap by simp
      have raw: accepts_in_time_user_wrap ?W w
                (length w + length w div (q + 1) + K)
        using Tb cL wSg accM by blast
      have absorb: length w div (q + 1) + K ≤ length w div q
        by (rule div_absorb_step[OF q_pos]) (use lv lenwv in simp)
      hence le: length w + length w div (q + 1) + K
                ≤ length w + length w div q by simp
      have accepts_in_time_mttm ?W (map Raw w)
            (length w + length w div (q + 1) + K)
        using raw unfolding accepts_in_time_user_wrap_def by simp
      hence accepts_in_time_mttm ?W (map Raw w) (length w + length w div
q)
        using le by (auto elim: accepts_in_time_mttm_mono)
      thus accepts_in_time_mttm ?W v (length v + length v div q)
        by (simp only: vmap length_map)
    qed
   qed
  qed
  show valid_mttm ?W by (rule V)
  show det_mttm ?W by (rule D)
  show Lang_user_wrap ?W = Lang_mttm M by (rule L)
  show (q + 1) * (3 + 8 * d_0) ≤ card (UNIV :: 'c set)
       ⟶ (∀ v ∈ Lang_mttm ?W. q * (q + 1) * (28 + 8 * d_0 + 8 * b) ≤
length v
             ⟶ accepts_in_time_mttm ?W v (length v + length v div q))
  proof −
    have s0: (3::nat) + 8 * d_0 = 1 + α + 8 * d_0 using alpha2 by simp
    show ?thesis unfolding s0 Kdef[symmetric] by (rule ev)
  qed
qed
```

17.4 HU 12.3 in the convention (superlinear T): the clean bound

The nondeterministic superlinear-T headline, restated as a *constant-free time_bounded_conv* bound $\lambda n.\ T\ n\ \mathrm{div}\ q$. Unlike 12.4, no additive constant survives: the raw simulation constant K *and* the finite-control overhead $2\ N1 + 4$ are both absorbed into a slightly smaller speedup denominator (the

raw construction is run at $q + 1$, the headline states q), because superlinear T makes *div_absorb_step*'s premise $q\ (q + 1)\ C \leq T\ n$ hold for all long n with $C = K + 2\ N1 + 4$.

The machine is *finite_patch* of the wrap under the table $u \in Lang_mttm$ of the wrap — the wrap with the finite-control small-input cleanup planted on top (short inputs decided by table lookup in $n + 2$, long inputs run the wrap after the O(1) rewind). The extra hypothesis over the raw *linear_speedup_HU_12_3_nae* is again *Mtime*, and the cardinality side condition tightens from $16\ q$ to $16\ (q + 1)$ (the faster inner run).

```
theorem linear_speedup_HU_12_3_nae_conv:
  fixes M :: ('q, 'a) mttm
    and T :: nat ⇒ nat
    and q :: nat
  assumes wf:      well_formed_mttm M
      and Mtime:   ⋀w. w ∈ Lang_mttm M ⟹ accepts_in_time_mttm M w (T
(length w))
      and growth:  ∀ d. ∃ N. ∀ n. N ≤ n ⟶ d * n ≤ T n
      and q_pos:   0 < q
      and c_large: 16 * (q + 1) ≤ card (UNIV :: ('c :: enum) set)
      and k2:      2 ≤ k_tm M
  obtains W' :: ((('q × ('a, 'c::enum) ae_stage, 'a, 'c ⇒ 'a) wrap_state,
                  ('a, 'c ⇒ 'a) wrap_alphabet) fp_state,
                 ('a, 'c ⇒ 'a) wrap_alphabet) mttm
  where valid_mttm W'
    and Lang_user_wrap W' = Lang_mttm M
    and time_bounded_conv W' (λn. T n div q)
proof −
  let ?W = encoding_wrap
             (alphabet_enlarge M
                :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
             (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
             (card (UNIV :: 'c set))
             (Sigma_tm M)
  have q1pos: 0 < q + 1 by simp
  obtain K N0 where
    Vraw: valid_mttm ?W and
    Lraw: Lang_user_wrap ?W = Lang_mttm M and
    Traw: ∀ w. set w ⊆ Sigma_tm M
              ⟶ N0 ≤ length w
              ⟶ accepts_in_time_mttm M w (T (length w))
              ⟶ accepts_in_time_user_wrap ?W w (T (length w) div (q + 1) +
K)
    by (rule linear_speedup_HU_12_3_nae[OF wf growth q1pos c_large k2])
  have blle: bl_tm ?W ≠ le_tm ?W
  proof −
    have le_neq_bl: le_tm M ≠ bl_tm M using wf by auto
    have bl_tm (alphabet_enlarge M :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
         ≠ le_tm (alphabet_enlarge M :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
```

```
    unfolding bl_tm_alphabet_enlarge le_tm_alphabet_enlarge
    using le_neq_bl by (simp add: bl_block_def LE_block_def fun_eq_iff)
  thus ?thesis by simp
qed
from growth obtain Nd where
  growthN: ∀ n. Nd ≤ n ⟶ 3 * q * (q + 1) * n ≤ T n by blast
define N1 where N1 = max N0 (max Nd (K + 4))
have N1_N0: N0 ≤ N1 and N1_Nd: Nd ≤ N1 and N1_K: K + 4 ≤ N1
  unfolding N1_def by auto
let ?W' = finite_patch ?W (λu. u ∈ Lang_mttm ?W) N1
have lang': Lang_mttm ?W' = Lang_mttm ?W
proof −
  have Lang_mttm ?W' = {u. set u ⊆ Sigma_tm ?W ∧ u ∈ Lang_mttm ?W}
    using finite_patch_language[OF Vraw blle] by simp
  also have ... = Lang_mttm ?W unfolding Lang_mttm_def by auto
  finally show ?thesis .
qed
have valid': valid_mttm ?W' by (rule finite_patch_valid[OF Vraw])
have Luser': Lang_user_wrap ?W' = Lang_mttm M
  using lang' Lraw unfolding Lang_user_wrap_def by simp
have conv': time_bounded_conv ?W' (λn. T n div q)
  unfolding time_bounded_conv_def
proof
  fix v assume vW': v ∈ Lang_mttm ?W'
  hence vW: v ∈ Lang_mttm ?W using lang' by simp
  hence vSg: set v ⊆ Sigma_tm ?W unfolding Lang_mttm_def by simp
  show accepts_in_time_mttm ?W' v (max (length v + 2) ((λn. T n div q)
(length v)))
  proof (cases length v ≤ N1)
    case True
    have accepts_in_time_mttm ?W' v (length v + 2)
      using Vraw vSg True vW by (rule fp_short_time)
    thus ?thesis by (auto elim: accepts_in_time_mttm_mono)
  next
    case False
    hence N1lt: N1 < length v by simp
    obtain w where vmap: v = map Raw w and wU: w ∈ Lang_user_wrap ?W
      using vW Lang_mttm_encoding_wrap_ex_Raw by blast
    from wU Lraw have wM: w ∈ Lang_mttm M by simp
    hence wSgM: set w ⊆ Sigma_tm M unfolding Lang_mttm_def by simp
    from wM have accM: accepts_in_time_mttm M w (T (length w)) by (rule
Mtime)
    have lenwv: length w = length v using vmap by simp
    have N0w: N0 ≤ length w using N1lt lenwv N1_N0 by simp
    have accepts_in_time_user_wrap ?W w (T (length w) div (q + 1) + K)
      using Traw wSgM N0w accM by blast
    hence accW: accepts_in_time_mttm ?W v (T (length v) div (q + 1) + K)
      by (simp only: accepts_in_time_user_wrap_def vmap[symmetric] lenwv)
    have accepts_in_time_mttm ?W' v ((T (length v) div (q + 1) + K) + (2
```

```
* N1 + 4))
      using Vraw vSg N1lt accW by (rule fp_long_time)
    moreover have (T (length v) div (q + 1) + K) + (2 * N1 + 4) ≤ T (length
v) div q
    proof −
      have Ndlv: Nd ≤ length v using N1_Nd N1lt by simp
      have TN: 3 * q * (q + 1) * length v ≤ T (length v)
        using growthN Ndlv by blast
      have K + 2 * N1 + 4 ≤ 3 * length v
      proof −
        have K + 2 * N1 + 4 ≤ 3 * N1 using N1_K by linarith
        also have ... ≤ 3 * length v using N1lt by linarith
        finally show ?thesis .
      qed
      hence q * (q + 1) * (K + 2 * N1 + 4) ≤ q * (q + 1) * (3 * length v)
        by (rule mult_le_mono2)
      also have q * (q + 1) * (3 * length v) = 3 * q * (q + 1) * length v
        by (simp add: algebra_simps)
      also note TN
      finally have big: q * (q + 1) * (K + 2 * N1 + 4) ≤ T (length v) .
      have T (length v) div (q + 1) + (K + 2 * N1 + 4) ≤ T (length v) div q
        using q_pos big by (rule div_absorb_step)
      thus ?thesis by simp
    qed
    ultimately have accepts_in_time_mttm ?W' v (T (length v) div q)
      by (auto elim: accepts_in_time_mttm_mono)
    thus ?thesis by (auto elim: accepts_in_time_mttm_mono)
  qed
 qed
 show thesis
 proof (rule that[of ?W'])
  show valid_mttm ?W' by (rule valid')
  show Lang_user_wrap ?W' = Lang_mttm M by (rule Luser')
  show time_bounded_conv ?W' (λn. T n div q) by (rule conv')
 qed
qed
```

The deterministic specialisation of *linear_speedup_HU_12_3_nae_conv*: adds the *det* hypothesis on M and the determinism-preservation conjunct on the cleaned machine (via *finite_patch_det*).

```
theorem linear_speedup_HU_12_3_dae_conv:
 fixes M :: ('q, 'a) mttm
  and T :: nat ⇒ nat
  and q :: nat
 assumes wf:      well_formed_mttm M
    and det:     det_mttm M
    and Mtime:   ⋀w. w ∈ Lang_mttm M ⟹ accepts_in_time_mttm M w (T
(length w))
```

```
    and growth: ∀ d. ∃ N. ∀ n. N ≤ n ⟶ d * n ≤ T n
    and q_pos:  0 < q
    and c_large: 16 * (q + 1) ≤ card (UNIV :: ('c :: enum) set)
    and k2:     2 ≤ k_tm M
  obtains W' :: ((('q × ('a, 'c::enum) ae_stage, 'a, 'c ⇒ 'a) wrap_state,
                  ('a, 'c ⇒ 'a) wrap_alphabet) fp_state,
                 ('a, 'c ⇒ 'a) wrap_alphabet) mttm
  where valid_mttm W'
    and det_mttm W'
    and Lang_user_wrap W' = Lang_mttm M
    and time_bounded_conv W' (λn. T n div q)
proof −
  let ?W = encoding_wrap
           (alphabet_enlarge M
              :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
           (ae_pack (bl_tm M) :: 'a list ⇒ ('c ⇒ 'a))
           (card (UNIV :: 'c set))
           (Sigma_tm M)
  have q1pos: 0 < q + 1 by simp
  obtain K N0 where
    Vraw: valid_mttm ?W and
    Draw: det_mttm ?W and
    Lraw: Lang_user_wrap ?W = Lang_mttm M and
    Traw: ∀ w. set w ⊆ Sigma_tm M
              ⟶ N0 ≤ length w
              ⟶ accepts_in_time_mttm M w (T (length w))
              ⟶ accepts_in_time_user_wrap ?W w (T (length w) div (q + 1) +
K)
    by (rule linear_speedup_HU_12_3_dae[OF wf det growth q1pos c_large k2])
  have blle: bl_tm ?W ≠ le_tm ?W
  proof −
    have le_neq_bl: le_tm M ≠ bl_tm M using wf by auto
    have bl_tm (alphabet_enlarge M :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
          ≠ le_tm (alphabet_enlarge M :: ('q × ('a, 'c) ae_stage, 'c ⇒ 'a) mttm)
      unfolding bl_tm_alphabet_enlarge le_tm_alphabet_enlarge
      using le_neq_bl by (simp add: bl_block_def LE_block_def fun_eq_iff)
    thus ?thesis by simp
  qed
  from growth obtain Nd where
    growthN: ∀ n. Nd ≤ n ⟶ 3 * q * (q + 1) * n ≤ T n by blast
  define N1 where N1 = max N0 (max Nd (K + 4))
  have N1_N0: N0 ≤ N1 and N1_Nd: Nd ≤ N1 and N1_K: K + 4 ≤ N1
    unfolding N1_def by auto
  let ?W' = finite_patch ?W (λu. u ∈ Lang_mttm ?W) N1
  have lang': Lang_mttm ?W' = Lang_mttm ?W
  proof −
    have Lang_mttm ?W' = {u. set u ⊆ Sigma_tm ?W ∧ u ∈ Lang_mttm ?W}
      using finite_patch_language[OF Vraw blle] by simp
    also have ... = Lang_mttm ?W unfolding Lang_mttm_def by auto
```

```
    finally show ?thesis .
  qed
  have valid': valid_mttm ?W' by (rule finite_patch_valid[OF Vraw])
  have det': det_mttm ?W' by (rule finite_patch_det[OF Vraw blle Draw])
  have Luser': Lang_user_wrap ?W' = Lang_mttm M
    using lang' Lraw unfolding Lang_user_wrap_def by simp
  have conv': time_bounded_conv ?W' (λn. T n div q)
    unfolding time_bounded_conv_def
  proof
    fix v assume vW': v ∈ Lang_mttm ?W'
    hence vW: v ∈ Lang_mttm ?W using lang' by simp
    hence vSg: set v ⊆ Sigma_tm ?W unfolding Lang_mttm_def by simp
     show accepts_in_time_mttm ?W' v (max (length v + 2) ((λn. T n div q)
(length v)))
    proof (cases length v ≤ N1)
      case True
      have accepts_in_time_mttm ?W' v (length v + 2)
        using Vraw vSg True vW by (rule fp_short_time)
      thus ?thesis by (auto elim: accepts_in_time_mttm_mono)
    next
      case False
      hence N1lt: N1 < length v by simp
      obtain w where vmap: v = map Raw w and wU: w ∈ Lang_user_wrap ?W
        using vW Lang_mttm_encoding_wrap_ex_Raw by blast
      from wU Lraw have wM: w ∈ Lang_mttm M by simp
      hence wSgM: set w ⊆ Sigma_tm M unfolding Lang_mttm_def by simp
      from wM have accM: accepts_in_time_mttm M w (T (length w)) by (rule
Mtime)
      have lenwv: length w = length v using vmap by simp
      have N0w: N0 ≤ length w using N1lt lenwv N1_N0 by simp
      have accepts_in_time_user_wrap ?W w (T (length w) div (q + 1) + K)
        using Traw wSgM N0w accM by blast
      hence accW: accepts_in_time_mttm ?W v (T (length v) div (q + 1) + K)
        by (simp only: accepts_in_time_user_wrap_def vmap[symmetric] lenwv)
       have accepts_in_time_mttm ?W' v ((T (length v) div (q + 1) + K) + (2
* N1 + 4))
        using Vraw vSg N1lt accW by (rule fp_long_time)
     moreover have (T (length v) div (q + 1) + K) + (2 * N1 + 4) ≤ T (length
v) div q
      proof −
        have Ndlv: Nd ≤ length v using N1_Nd N1lt by simp
        have TN: 3 * q * (q + 1) * length v ≤ T (length v)
          using growthN Ndlv by blast
        have K + 2 * N1 + 4 ≤ 3 * length v
        proof −
          have K + 2 * N1 + 4 ≤ 3 * N1 using N1_K by linarith
          also have ... ≤ 3 * length v using N1lt by linarith
          finally show ?thesis .
        qed
```

```
      hence q * (q + 1) * (K + 2 * N1 + 4) ≤ q * (q + 1) * (3 * length v)
        by (rule mult_le_mono2)
      also have q * (q + 1) * (3 * length v) = 3 * q * (q + 1) * length v
        by (simp add: algebra_simps)
      also note TN
      finally have big: q * (q + 1) * (K + 2 * N1 + 4) ≤ T (length v) .
      have T (length v) div (q + 1) + (K + 2 * N1 + 4) ≤ T (length v) div q
        using q_pos big by (rule div_absorb_step)
      thus ?thesis by simp
    qed
    ultimately have accepts_in_time_mttm ?W' v (T (length v) div q)
      by (auto elim: accepts_in_time_mttm_mono)
    thus ?thesis by (auto elim: accepts_in_time_mttm_mono)
  qed
 qed
 show thesis
 proof (rule that[of ?W'])
  show valid_mttm ?W' by (rule valid')
  show det_mttm ?W' by (rule det')
  show Lang_user_wrap ?W' = Lang_mttm M by (rule Luser')
  show time_bounded_conv ?W' (λn. T n div q) by (rule conv')
 qed
qed

end
```

References

[1] F. J. Balbach. The Cook-Levin theorem. *Archive of Formal Proofs*, January 2023. https://isa-afp.org/entries/Cook_Levin.html, Formal proof development.

[2] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity I*, volume 11 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.

[3] C. Dalvit and R. Thiemann. A verified translation of multitape Turing machines into singletape Turing machines. *Archive of Formal Proofs*, November 2022. https://isa-afp.org/entries/Multitape_To_Singletape_TM.html, Formal proof development.

[4] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the AMS*, 117:285–306, 1965.

[5] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison-Wesley, 1969.

[6] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[7] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[8] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 1–66. Elsevier, 1990.

[9] J. Xu, X. Zhang, C. Urban, S. J. C. Joosten, and F. Regensburger. Universal Turing machine. *Archive of Formal Proofs*, February 2019. https://isa-afp.org/entries/Universal_Turing_Machine.html, Formal proof development.