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

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*⟩

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*⟩

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 ⇒ ('c :: enum ⇒ 'a) where
  LE_block le = (λ_. 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 ⇒ ('c :: enum ⇒ 'a) where
  bl_block bl = (λ_. 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 ⇒ 'a list ⇒ (('c :: enum) ⇒ 'a) list where
  encode_input bl w =
    (let c = card (UNIV :: 'c set) in
      map (λi. (λx. 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 ⇒ ('c :: enum ⇒ 'a) ⇒ 'a list where
  ae_decode_block bl f =
    takeWhile (λa. a ≠ bl) (map f (enum_class.enum :: 'c list))

definition ae_decode_input :: 'a ⇒ (('c :: enum) ⇒ 'a) list ⇒ '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* ::
  $'a \Rightarrow (nat \Rightarrow (('c :: enum \Rightarrow 'a)$
    $\times ('c \Rightarrow 'a)$
    $\times ('c \Rightarrow 'a)))$ **where**
  init_buffer le = $(\lambda\_.$ (*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 = $(\lambda\_.$ *AE_Home*)

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

definition *init_stage* ::
  $'a \Rightarrow ('a, 'c :: enum)$ *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* ::
  $'a \Rightarrow ('c :: enum \Rightarrow 'a) \Rightarrow bool$ **where**
  is_pure_block bl f = $(\forall x.\ f\ x \neq bl)$

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* ::
  $'a \Rightarrow ('c :: enum \Rightarrow 'a) \Rightarrow bool$ **where**
  is_padded_block bl f =
    $(\exists k.\ k \geq 1 \wedge k < length\ (enum\_class.enum :: 'c\ list)$
      $\wedge\ (\forall x.\ c\_idx\ x < k \longrightarrow f\ x \neq bl)$
      $\wedge\ (\forall x.\ c\_idx\ x \geq k \longrightarrow f\ x = bl))$

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* ∨ *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* ::
 $('q,\ 'a)\ mttm$
  $\Rightarrow (('q \times ('a,\ 'c :: enum)\ ae\_stage)$
   $\times (nat \Rightarrow ('c \Rightarrow 'a))$
   $\times ('q \times ('a,\ 'c)\ ae\_stage)$
   $\times (nat \Rightarrow ('c \Rightarrow 'a))$
   $\times (nat \Rightarrow dir))\ set$ **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.
   $q \in Q\_tm\ M \wedge q \neq t\_tm\ M \wedge q \neq r\_tm\ M$
   $\wedge\ (a\ 0 = LE\_block\ (le\_tm\ M)$

```
      ∨ 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* = (λ*k*. *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 ⇒ 'a ⇒ ('c :: enum) bp ⇒ dir ⇒ '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'*)) ∈ *m_step_buffered M*
  ⟨*proof*⟩

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 $n \leq c =$ *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⟩
```

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
  ⟨proof⟩
```

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*⟩

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 $n\_i \leq c$ 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 $n2 - n1 \geq 1$ steps require a transition from *s1*. The disjunction on *s1* forces either $n1 = c$ (contradicting $n1 < n2 \leq c$) 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⟩
```

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* $\geq$ *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* $\Rightarrow$ *'a*) $\Rightarrow$ *nat*
$\Rightarrow$ (*'c* :: *enum*) *bp*
$\Rightarrow$ ((*'c* $\Rightarrow$ *'a*) $\times$ (*'c* $\Rightarrow$ *'a*) $\times$ (*'c* $\Rightarrow$ *'a*))
$\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
ae_window_invariant tM nM bp blocks p_start $\longleftrightarrow$
p_start $\geq$ *1*
$\wedge$ *nM* = *p_start* + *bp_linear bp*
$\wedge$ ($\forall$ *i*. *i* < *3* $*$ *card* (*UNIV* :: *'c set*)
$\longrightarrow$ *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$\leftrightarrow$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* $\mapsto$ *0*, *AE_Home* $\mapsto$ *1* + *c_idx ofs*, *AE_Right* $\mapsto$ *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* $\Rightarrow$ *'a*) $\Rightarrow$ *nat*
$\Rightarrow$ (*'c* :: *enum*) *bp*
$\Rightarrow$ ((*'c* $\Rightarrow$ *'a*) $\times$ (*'c* $\Rightarrow$ *'a*) $\times$ (*'c* $\Rightarrow$ *'a*))
$\Rightarrow$ *'a* $\Rightarrow$ *bool* **where**
ae_window_invariant_le1 tM nM bp blocks le $\longleftrightarrow$
fst blocks = *LE_block le*
$\wedge$ *tM 0* = *le*
$\wedge$ (*case fst bp of*
AE_Left $\Rightarrow$ *snd bp* = *c_last* $\wedge$ *nM* = *0*
| *AE_Home* $\Rightarrow$ *nM* = *Suc* (*c_idx* (*snd bp*))
| *AE_Right* $\Rightarrow$ *nM* = *Suc* (*card* (*UNIV* :: *'c set*) + *c_idx* (*snd bp*)))
$\wedge$ ($\forall$ *i*. *i* < *2* $*$ *card* (*UNIV* :: *'c set*)

$\longrightarrow$ *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* $\mapsto$ *0* (M sits at LE regardless of *snd bp*, since home is all-LE) and *AE_Right* $\mapsto$ *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* = (λ*k*. (*fst* (*buf k*),
if k < *k_tm M then fst* (*snd* (*buf k*)) *else a k*,
a k))
∧ ((*q*, *buf_full*, λ*k*. (*AE_Home*, *ofs k*)),
(*q'*, *bufC*, *end_pos*)) ∈ *m_steps_buffered M*
∧ *ofs'* = (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 bufC k else init_buffer* (*le_tm M*) *k*)
∧ *dest'* = (λ*k*. *if k* < *k_tm M then fst* (*end_pos k*) *else init_dest k*)
∧ *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)}

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 ⟷ *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* ≠ *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* ∈ {*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* ≥ *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* ≠ *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*, ...) 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'* = (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*))
    ∧ *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)}

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*))
    ∧ *stage'* = (*if q* ∈ {*t_tm M*, *r_tm M*}
                then init_stage (*le_tm M*)
                else (*ofs*, *buf*, *init_dest*, *SS1*))
    ∧ *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*))
    ∧ *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 ⇒ (*('c* :: *enum)* ⇒ *'a*) *set* **where**
gamma_block Γ = {*f*. *range f* ⊆ Γ}

lemma *gamma_block_mono*: *A* ⊆ *B* ⟹ *gamma_block A* ⊆ *gamma_block B*
⟨*proof*⟩

lemma *finite_gamma_block*:
fixes *A* :: *'a set*
assumes *finA*: *finite A*
shows *finite* (*gamma_block A* :: (*'c* :: *enum* ⇒ *'a*) *set*)
⟨*proof*⟩

lemma *LE_block_in_gamma_block*:
le ∈ Γ ⟹ *LE_block le* ∈ *gamma_block* Γ
⟨*proof*⟩

lemma *bl_block_in_gamma_block*:
bl ∈ Γ ⟹ *bl_block bl* ∈ *gamma_block* Γ
⟨*proof*⟩

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* ⇒ *'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*)

$\wedge\ (\forall j{\geq}K.\ dst\ j = init\_dest\ j))$

lemma *ae_valid_stage_init*:
assumes $le \in \Gamma$
shows *ae_valid_stage* Γ *le* K (*init_stage le*)
⟨*proof*⟩

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*: $le \in \Gamma$
and *ib_def*: *ib* = *init_buffer le*
shows $\forall k.$ *fst* (*ib k*) ∈ *gamma_block* Γ
$\wedge$ *fst* (*snd* (*ib k*)) ∈ *gamma_block* Γ
$\wedge$ *snd* (*snd* (*ib k*)) ∈ *gamma_block* Γ
⟨*proof*⟩

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 $\forall kk$ :: *nat*.
fst (*init_buffer* (*le_tm M*) *kk*
:: ($'c$ :: *enum* ⇒ $'a$) × ($'c$ ⇒ $'a$) × ($'c$ ⇒ $'a$))
∈ *gamma_block* (Γ_*tm M*)
$\wedge$ *fst* (*snd* (*init_buffer* (*le_tm M*) *kk*
:: ($'c$ ⇒ $'a$) × ($'c$ ⇒ $'a$) × ($'c$ ⇒ $'a$)))
∈ *gamma_block* (Γ_*tm M*)
$\wedge$ *snd* (*snd* (*init_buffer* (*le_tm M*) *kk*
:: ($'c$ ⇒ $'a$) × ($'c$ ⇒ $'a$) × ($'c$ ⇒ $'a$)))
∈ *gamma_block* (Γ_*tm M*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *ae_ss6_action_LE* [*simp*]:
  ae_ss6_action le (*LE_block le*) *buf ds* = (*LE_block le*, *dir.R*)
⟨*proof*⟩

lemma *ae_ss7_action_LE* [*simp*]:
  ae_ss7_action le (*LE_block le*) *buf ds* = (*LE_block le*, *dir.N*)
⟨*proof*⟩

lemma *ae_ss8_action_LE* [*simp*]:
  ae_ss8_action le (*LE_block le*) *buf ds* = (*LE_block le*, *dir.N*)
⟨*proof*⟩

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*⟩

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*⟩

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*: $\forall k.\ fst\ (blocks\ k) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ fst\ (snd\ (blocks\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ snd\ (snd\ (blocks\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
shows $\forall k.\ fst\ (blocks'\ k) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ fst\ (snd\ (blocks'\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ snd\ (snd\ (blocks'\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
⟨*proof*⟩

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*: $((q, blocks, pos), (q', blocks', pos')) \in m\_steps\_buffered\ M$
and *pre*: $\forall k.\ fst\ (blocks\ k) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ fst\ (snd\ (blocks\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ snd\ (snd\ (blocks\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
shows $\forall k.\ fst\ (blocks'\ k) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ fst\ (snd\ (blocks'\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ snd\ (snd\ (blocks'\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
⟨*proof*⟩

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*: $(s, a, s', a', d) \in ae\_delta\_ss1\_ss2\ M$
and *a_gamma*: $\forall k.\ a\ k \in gamma\_block\ (\Gamma\_tm\ M)$
and *src_valid*: $ae\_valid\_stage\ (\Gamma\_tm\ M)\ (le\_tm\ M)\ (k\_tm\ M)\ (snd\ s)$
shows $ae\_valid\_stage\ (\Gamma\_tm\ M)\ (le\_tm\ M)\ (k\_tm\ M)\ (snd\ s')$
⟨*proof*⟩

lemma *ae_delta_ss2_ss3_dest_valid*:
assumes *vM*: *valid_mttm M*
and *rel*: $(s, a, s', a', d) \in ae\_delta\_ss2\_ss3\ M$
and *a_gamma*: $\forall k.\ a\ k \in gamma\_block\ (\Gamma\_tm\ M)$
and *src_valid*: $ae\_valid\_stage\ (\Gamma\_tm\ M)\ (le\_tm\ M)\ (k\_tm\ M)\ (snd\ s)$
shows $ae\_valid\_stage\ (\Gamma\_tm\ M)\ (le\_tm\ M)\ (k\_tm\ M)\ (snd\ s')$
⟨*proof*⟩

lemma *ae_delta_ss3_ss4_dest_valid*:
  assumes *rel*: $(s,\ a,\ s',\ a',\ d) \in$ *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*⟩

lemma *ae_delta_ss4_ss5_dest_valid*:
  assumes *vM*: *valid_mttm M*
    and *rel*: $(s,\ a,\ s',\ a',\ d) \in$ *ae_delta_ss4_ss5 M*
    and *a_gamma*: $\forall\, k.\ a\ k \in$ *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*⟩

lemma *ae_delta_ss5_ss6_dest_valid*:
  assumes *rel*: $(s,\ a,\ s',\ a',\ d) \in$ *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*⟩

lemma *ae_delta_ss6_ss7_dest_valid*:
  assumes *rel*: $(s,\ a,\ s',\ a',\ d) \in$ *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*⟩

lemma *ae_delta_ss7_ss8_dest_valid*:
  assumes *rel*: $(s,\ a,\ s',\ a',\ d) \in$ *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*⟩

lemma *ae_delta_ss8_ss1_dest_valid*:
  assumes *vM*: *valid_mttm M*
    and *rel*: $(s,\ a,\ s',\ a',\ d) \in$ *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*⟩

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*: $(s,\ a,\ s',\ a',\ d) \in$ *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*⟩

lemma *ae_delta_val_fwd_to_padded_dest_valid*:

assumes *rel*: $(s,\ a,\ s',\ a',\ d) \in$ *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*⟩

lemma *ae_delta_val_fwd_reject_dest_valid*:
assumes *vM*: *valid_mttm M*
and *rel*: $(s,\ a,\ s',\ a',\ d) \in$ *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*⟩

lemma *ae_delta_val_fwd_to_ret_dest_valid*:
assumes *rel*: $(s,\ a,\ s',\ a',\ d) \in$ *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*⟩

lemma *ae_delta_val_pad_to_ret_dest_valid*:
assumes *rel*: $(s,\ a,\ s',\ a',\ d) \in$ *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*⟩

lemma *ae_delta_val_pad_reject_dest_valid*:
assumes *vM*: *valid_mttm M*
and *rel*: $(s,\ a,\ s',\ a',\ d) \in$ *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*⟩

lemma *ae_delta_val_ret_step_dest_valid*:
assumes *rel*: $(s,\ a,\ s',\ a',\ d) \in$ *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*⟩

lemma *ae_delta_val_ret_to_sim_dest_valid*:
assumes *rel*: $(s,\ a,\ s',\ a',\ d) \in$ *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*⟩

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* (Γ_*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* ::
 $('q, 'a)\ mttm$
 $\Rightarrow (('q \times ('a, 'c :: enum)\ ae\_stage)$
 $\times (nat \Rightarrow ('c \Rightarrow 'a))$
 $\times ('q \times ('a, 'c)\ ae\_stage)$
 $\times (nat \Rightarrow ('c \Rightarrow 'a))$
 $\times (nat \Rightarrow dir))\ set$ **where**
 alphabet_enlarge_delta $M =$
 $(ae\_delta\_val\_fwd\_advance\ M$
 $\cup\ ae\_delta\_val\_fwd\_to\_padded\ M$
 $\cup\ ae\_delta\_val\_fwd\_reject\ M$
 $\cup\ ae\_delta\_val\_fwd\_to\_ret\ M$
 $\cup\ ae\_delta\_val\_pad\_to\_ret\ M$
 $\cup\ ae\_delta\_val\_pad\_reject\ M$
 $\cup\ ae\_delta\_val\_ret\_step\ M$
 $\cup\ ae\_delta\_val\_ret\_to\_sim\ M$
 $\cup\ ae\_delta\_ss1\_ss2\ M$
 $\cup\ ae\_delta\_ss2\_ss3\ M$
 $\cup\ ae\_delta\_ss3\_ss4\ M$
 $\cup\ ae\_delta\_ss4\_ss5\ M$
 $\cup\ ae\_delta\_ss5\_ss6\ M$
 $\cup\ ae\_delta\_ss6\_ss7\ M$
 $\cup\ ae\_delta\_ss7\_ss8\ M$
 $\cup\ ae\_delta\_ss8\_ss1\ M)$
 $\cap\ \{(s, a, s', a', d).$
 $(\forall k.\ a\ k \in gamma\_block\ (\Gamma\_tm\ M))$
 $\wedge\ (\forall k.\ a'\ k \in gamma\_block\ (\Gamma\_tm\ M))\}$
 $\cap\ \{(s, a, s', a', d).$
 $\forall k.\ a'\ k = LE\_block\ (le\_tm\ M) \longrightarrow a\ k = LE\_block\ (le\_tm\ M)\}$
 $\cap\ \{(s, a, s', a', d).$
 $ae\_valid\_stage\ (\Gamma\_tm\ M)\ (le\_tm\ M)\ (k\_tm\ M)\ (snd\ s)$
 $\wedge\ ae\_valid\_stage\ (\Gamma\_tm\ M)\ (le\_tm\ M)\ (k\_tm\ M)\ (snd\ s')\}$

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 $(s, a, s1, a1, d1) \in$ *ae_delta_val_fwd_advance M*
and $(s, a, s2, a2, d2) \in$ *ae_delta_val_fwd_advance M*
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

lemma *ae_delta_val_fwd_to_padded_functional*:
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_padded M*
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

lemma *ae_delta_val_fwd_reject_functional*:
assumes $(s, a, s1, a1, d1) \in$ *ae_delta_val_fwd_reject M*
and $(s, a, s2, a2, d2) \in$ *ae_delta_val_fwd_reject M*
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

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 $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

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 $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

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 $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

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 $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

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 $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

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 $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

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 $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

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 $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

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 consumes 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*: $(s, a, s1, a1, d1) \in ae\_delta\_ss4\_ss5\ M$
and *h2*: $(s, a, s2, a2, d2) \in ae\_delta\_ss4\_ss5\ M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

lemma *ae_delta_ss5_ss6_functional*:
assumes $(s, a, s1, a1, d1) \in ae\_delta\_ss5\_ss6\ M$
and $(s, a, s2, a2, d2) \in ae\_delta\_ss5\_ss6\ M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

lemma *ae_delta_ss6_ss7_functional*:
assumes $(s, a, s1, a1, d1) \in ae\_delta\_ss6\_ss7\ M$
and $(s, a, s2, a2, d2) \in ae\_delta\_ss6\_ss7\ M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

lemma *ae_delta_ss7_ss8_functional*:
assumes $(s, a, s1, a1, d1) \in ae\_delta\_ss7\_ss8\ M$
and $(s, a, s2, a2, d2) \in ae\_delta\_ss7\_ss8\ M$

shows *s1* = *s2* ∧ *a1* = *a2* ∧ *d1* = *d2*
⟨*proof*⟩

lemma *ae_delta_ss8_ss1_functional*:
assumes (*s*, *a*, *s1*, *a1*, *d1*) ∈ *ae_delta_ss8_ss1 M*
and (*s*, *a*, *s2*, *a2*, *d2*) ∈ *ae_delta_ss8_ss1 M*
shows *s1* = *s2* ∧ *a1* = *a2* ∧ *d1* = *d2*
⟨*proof*⟩

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*) ∈ *ae_delta_val_pad_to_ret M*
and (*s*, *a*, *s2*, *a2*, *d2*) ∈ *ae_delta_val_pad_reject M*
shows *False*
⟨*proof*⟩

lemma *ae_delta_val_ret_step_ret_to_sim_disjoint*:
assumes (*s*, *a*, *s1*, *a1*, *d1*) ∈ *ae_delta_val_ret_step M*
and (*s*, *a*, *s2*, *a2*, *d2*) ∈ *ae_delta_val_ret_to_sim M*
shows *False*
⟨*proof*⟩

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*, . . . , *SS8*} \ {*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*) ∈ *alphabet_enlarge_delta M*
and *snd* (*snd* (*snd* (*snd s*))) = *SS1*
shows (*s*, *a*, *s'*, *a'*, *d*) ∈ *ae_delta_ss1_ss2 M*
⟨*proof*⟩

lemma *ae_delta_ss2_only*:
fixes *M* :: ('*q*, '*a*) *mttm*
assumes (*s*, *a*, *s'*, *a'*, *d*) ∈ *alphabet_enlarge_delta M*
and *snd* (*snd* (*snd* (*snd s*))) = *SS2*
shows (*s*, *a*, *s'*, *a'*, *d*) ∈ *ae_delta_ss2_ss3 M*
⟨*proof*⟩

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 $(s, a, s', a', d) \in ae\_delta\_ss3\_ss4\ M$
⟨*proof*⟩

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 $(s, a, s', a', d) \in ae\_delta\_ss4\_ss5\ M$
⟨*proof*⟩

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 $(s, a, s', a', d) \in ae\_delta\_ss5\_ss6\ M$
⟨*proof*⟩

lemma *ae_delta_ss6_only*:
fixes $M :: ('q, 'a)\ mttm$
assumes $(s, a, s', a', d) \in alphabet\_enlarge\_delta\ M$
and $snd\ (snd\ (snd\ (snd\ s))) = SS6$
shows $(s, a, s', a', d) \in ae\_delta\_ss6\_ss7\ M$
⟨*proof*⟩

lemma *ae_delta_ss7_only*:
fixes $M :: ('q, 'a)\ mttm$
assumes $(s, a, s', a', d) \in alphabet\_enlarge\_delta\ M$
and $snd\ (snd\ (snd\ (snd\ s))) = SS7$
shows $(s, a, s', a', d) \in ae\_delta\_ss7\_ss8\ M$
⟨*proof*⟩

lemma *ae_delta_ss8_only*:
fixes $M :: ('q, 'a)\ mttm$
assumes $(s, a, s', a', d) \in alphabet\_enlarge\_delta\ M$
and $snd\ (snd\ (snd\ (snd\ s))) = SS8$
shows $(s, a, s', a', d) \in ae\_delta\_ss8\_ss1\ M$
⟨*proof*⟩

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 Σ*_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 $w ::$ ('$c \Rightarrow$ '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))
         (λi 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⟩
```

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))
  ⟨proof⟩
```

lemma *t_tm_alphabet_enlarge*:
fixes $M :: ('q, 'a)\ mttm$
shows $t\_tm\ (alphabet\_enlarge\ M$
$:: ('q \times ('a, ('c :: enum))\ ae\_stage,$
$'c \Rightarrow 'a)\ mttm)$
$= (t\_tm\ M,\ init\_stage\ (le\_tm\ M))$
⟨*proof*⟩

lemma *r_tm_alphabet_enlarge*:
fixes $M :: ('q, 'a)\ mttm$
shows $r\_tm\ (alphabet\_enlarge\ M$
$:: ('q \times ('a, ('c :: enum))\ ae\_stage,$
$'c \Rightarrow 'a)\ mttm)$
$= (r\_tm\ M,\ init\_stage\ (le\_tm\ M))$
⟨*proof*⟩

lemma *bl_tm_alphabet_enlarge*:
fixes $M :: ('q, 'a)\ mttm$
shows $bl\_tm\ (alphabet\_enlarge\ M$
$:: ('q \times ('a, ('c :: enum))\ ae\_stage,$
$'c \Rightarrow 'a)\ mttm)$
$= bl\_block\ (bl\_tm\ M)$
⟨*proof*⟩

lemma *le_tm_alphabet_enlarge*:
fixes $M :: ('q, 'a)\ mttm$
shows $le\_tm\ (alphabet\_enlarge\ M$
$:: ('q \times ('a, ('c :: enum))\ ae\_stage,$
$'c \Rightarrow 'a)\ mttm)$
$= LE\_block\ (le\_tm\ M)$
⟨*proof*⟩

lemma *delta_tm_alphabet_enlarge*:
fixes $M :: ('q, 'a)\ mttm$
shows $delta\_tm\ (alphabet\_enlarge\ M$
$:: ('q \times ('a, ('c :: enum))\ ae\_stage,$
$'c \Rightarrow 'a)\ mttm)$
$= alphabet\_enlarge\_delta\ M$
⟨*proof*⟩

lemma *Sigma_tm_alphabet_enlarge*:
fixes $M :: ('q, 'a)\ mttm$
shows $Sigma\_tm\ (alphabet\_enlarge\ M$
$:: ('q \times ('a, ('c :: enum))\ ae\_stage,$
$'c \Rightarrow 'a)\ mttm)$
$= gamma\_block\ (Sigma\_tm\ M \cup \{bl\_tm\ M\})$
$- \{bl\_block\ (bl\_tm\ M),\ LE\_block\ (le\_tm\ M)\}$
⟨*proof*⟩

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*)
  ⟨*proof*⟩

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'. Iteration to $\frown\!\frown$ 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*⟩

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*⟩

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*: $(c', c'') \in$ *mttm_step* (*alphabet_enlarge_delta M*) ^^ *n*
    shows *ae_buffer_in_gamma_block M c''* ∧ *ae_tape_in_gamma_block M c''*
⟨*proof*⟩

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* ∈ {*SS2*, …, *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*)))

$\wedge$ *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*
    $\Rightarrow$ ('c :: *enum* $\Rightarrow$ 'a,
        'q $\times$ ('a, 'c) *ae_stage*) *mt_config*
    $\Rightarrow$ *bool* **where**
  ae_inv_ss1 M cM' $\longleftrightarrow$
    (*case mt_state cM' of* (*qM'*, _, _, _, *idx*) $\Rightarrow$
      idx = *SS1* $\wedge$ *qM'* $\in$ *Q_tm M*)

definition *ae_inv_ss2* ::
  ('q, 'a) *mttm*
    $\Rightarrow$ ('c :: *enum* $\Rightarrow$ 'a,
        'q $\times$ ('a, 'c) *ae_stage*) *mt_config*
    $\Rightarrow$ *bool* **where**
  ae_inv_ss2 M cM' $\longleftrightarrow$
    (*case mt_state cM' of* (*qM'*, _, _, _, *idx*) $\Rightarrow$
      idx = *SS2* $\wedge$ *qM'* $\in$ *Q_tm M*)

definition *ae_inv_ss3* ::
  ('q, 'a) *mttm*
    $\Rightarrow$ ('c :: *enum* $\Rightarrow$ 'a,
        'q $\times$ ('a, 'c) *ae_stage*) *mt_config*
    $\Rightarrow$ *bool* **where**
  ae_inv_ss3 M cM' $\longleftrightarrow$
    (*case mt_state cM' of* (*qM'*, _, _, _, *idx*) $\Rightarrow$
      idx = *SS3* $\wedge$ *qM'* $\in$ *Q_tm M*)

definition *ae_inv_ss4* ::
  ('q, 'a) *mttm*
    $\Rightarrow$ ('c :: *enum* $\Rightarrow$ 'a,
        'q $\times$ ('a, 'c) *ae_stage*) *mt_config*
    $\Rightarrow$ *bool* **where**
  ae_inv_ss4 M cM' $\longleftrightarrow$
    (*case mt_state cM' of* (*qM'*, _, _, _, *idx*) $\Rightarrow$
      idx = *SS4* $\wedge$ *qM'* $\in$ *Q_tm M*)

definition *ae_inv_ss5* ::
  ('q, 'a) *mttm*
    $\Rightarrow$ ('c :: *enum* $\Rightarrow$ '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 sideband 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* ∈ {*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* ≥ *1*; the substrate's δLE forbids *LE_block* at any position ≥ *1*, and the c-step compute (via *ae_m_steps_buffered_correct*) preserves this.

2. SS6 (*dest* ≠ *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* ≥ *2*) the antecedent l = *LE_block* is vacuously false (l is loaded from block *s*−*1* ≥ *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* ≠ *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 substep actions' direction rules (*ae_ss5_action*'s "move L when *dest* ≠ *AE_Left*", etc.) deliver the head to block 0 at exactly the right substeps 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)
```

$= LE\_block\ (le\_tm\ M)))$
$\wedge\ (idx = SS8 \longrightarrow$
$(\forall k{<}k\_tm\ M.\ dest\ k = AE\_Left$
$\longrightarrow fst\ (buf\ k) = LE\_block\ (le\_tm\ M)$
$\longrightarrow mt\_tape\ cM'\ k\ (mt\_pos\ cM'\ k)$
$= LE\_block\ (le\_tm\ M))))$

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* ($r \neq LE\_block$ makes the antecedent vacuous); a-write branches close trivially via $a' = a$; the l-write branch (SS6 with *dest* $\neq$ *AE_Left*; SS8 with *dest* = *AE_Left*) closes via the second conjunct (the position-link's $l = LE\_block \longrightarrow ts\ (n) = LE\_block$ 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* $\Rightarrow$ 'a,
'q $\times$ ('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*⟩

lemma *ae_position_link_discharges_ss7*:
fixes M :: ('q, 'a) *mttm*
and c' :: ('c :: *enum* $\Rightarrow$ 'a,
'q $\times$ ('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*⟩

lemma *ae_position_link_discharges_ss8*:
fixes M :: ('q, 'a) *mttm*
and c' :: ('c :: *enum* $\Rightarrow$ 'a,
'q $\times$ ('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*⟩

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 $(s, a, s', a', d) \in$ *ae_delta_ss4_ss5 M*
shows *snd* (*snd* (*snd* (*snd s*))) = *SS4*
⟨*proof*⟩

lemma *ae_delta_ss5_ss6_pre_idx*:
assumes $(s, a, s', a', d) \in$ *ae_delta_ss5_ss6 M*
shows *snd* (*snd* (*snd* (*snd s*))) = *SS5*
⟨*proof*⟩

lemma *ae_delta_ss7_ss8_pre_idx*:
assumes $(s, a, s', a', d) \in$ *ae_delta_ss7_ss8 M*
shows *snd* (*snd* (*snd* (*snd s*))) = *SS7*
⟨*proof*⟩

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*⟩

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*⟩

lemma *ae_pos_link_aux_void_ss7_ss8*:
fixes *M* :: ('*q*, '*a*) *mttm*
and *c_pre c_post* :: ('*c* :: *enum* ⇒ '*a*,

$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
assumes *pre_neq*: $snd\ (snd\ (snd\ (snd\ (mt\_state\ c\_pre)))) \neq SS7$
and *step*: $(c\_pre, c\_post) \in mttm\_step\ (ae\_delta\_ss7\_ss8\ M)$
shows $ae\_position\_link\ M\ c\_post$
⟨*proof*⟩

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 $c'\ c'' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
assumes *pre*: $ae\_position\_link\ M\ c'$
and *step*: $(c', c'') \in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
and *aux_ss4_ss5*:
$(c', c'') \in mttm\_step\ (ae\_delta\_ss4\_ss5\ M)$
$\Longrightarrow ae\_position\_link\ M\ c''$
and *aux_ss5_ss6*:
$(c', c'') \in mttm\_step\ (ae\_delta\_ss5\_ss6\ M)$
$\Longrightarrow ae\_position\_link\ M\ c''$
and *aux_ss7_ss8*:
$(c', c'') \in mttm\_step\ (ae\_delta\_ss7\_ss8\ M)$
$\Longrightarrow ae\_position\_link\ M\ c''$
shows $ae\_position\_link\ M\ c''$
⟨*proof*⟩

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*⟩

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⟩

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⟩
```

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⟩
```

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*⟩

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*⟩

Step-existence at SS8: end of the write-back phase. The action selects l, r, or a per stage-kind × 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 $\exists\, c''.\ (c',\ c'') \in$ *mttm_step* (*ae_delta_ss8_ss1 M*)
$\wedge\ (c',\ c'') \in$ *mttm_step* (*alphabet_enlarge_delta M*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *s_tm_in_Q_tm*:
assumes *valid_mttm M*
shows *s_tm M* $\in$ *Q_tm M*
⟨*proof*⟩

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* ⇒ *'a*) *list*)
= (*length u* + *card* (*UNIV* :: *'c set*) − *1*) *div card* (*UNIV* :: *'c set*)
⟨*proof*⟩

lemma *encode_input_nth*:
fixes *u* :: *'a list*
and *x* :: *'c* :: *enum*
assumes *i* < *length* (*encode_input bl u* :: (*'c* ⇒ *'a*) *list*)
shows ((*encode_input bl u* :: (*'c* ⇒ *'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*⟩

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*⟩

lemma *c_idx_lt_card*: *c_idx* (*x* :: *'c* :: *enum*) < *card* (*UNIV* :: *'c set*)
⟨*proof*⟩

lemma *card_eq_length_enum*:

card (*UNIV* :: *'c* :: *enum set*) = *length* (*enum_class.enum* :: *'c list*)
⟨*proof*⟩

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*⟩

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* ⇒ *'a*
shows *is_pure_block bl f* ⟹ ¬ *is_padded_block bl f*
⟨*proof*⟩

lemma *not_is_pure_bl_block*:
¬ *is_pure_block* (*bl* :: *'a*) (*bl_block bl* :: *'c* :: *enum* ⇒ *'a*)
⟨*proof*⟩

lemma *not_is_padded_bl_block*:
¬ *is_padded_block* (*bl* :: *'a*) (*bl_block bl* :: *'c* :: *enum* ⇒ *'a*)
⟨*proof*⟩

lemma *is_pure_LE_block*:
(*le* :: *'a*) ≠ *bl* ⟹
is_pure_block bl (*LE_block le* :: *'c* :: *enum* ⇒ *'a*)
⟨*proof*⟩

lemma *not_is_padded_LE_block*:
(*le* :: *'a*) ≠ *bl* ⟹
¬ *is_padded_block bl* (*LE_block le* :: *'c* :: *enum* ⇒ *'a*)
⟨*proof*⟩

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*⟩

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 $a :: nat \Rightarrow 'c :: enum \Rightarrow 'a$
  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_padded M*
  shows *False*
⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

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*⟩

lemma *ae_delta_val_fwd_to_padded_reject_disjoint*:
  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_reject M*
  shows *False*
  ⟨*proof*⟩

lemma *ae_delta_val_fwd_to_ret_reject_disjoint*:
  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_reject M*
  shows *False*
  ⟨*proof*⟩

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*: $le\_tm\ M \neq bl\_tm\ M$
and *h1*: $(s, a, s1, a1, d1) \in alphabet\_enlarge\_delta\ M$
and *h2*: $(s, a, s2, a2, d2) \in alphabet\_enlarge\_delta\ M$
and *idx*: $snd\ (snd\ (snd\ (snd\ s))) = VFwd$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

lemma *ae_delta_VRet_functional*:
fixes $M :: ('q, 'a)$ *mttm*
assumes *h1*: $(s, a, s1, a1, d1) \in alphabet\_enlarge\_delta\ M$
and *h2*: $(s, a, s2, a2, d2) \in alphabet\_enlarge\_delta\ M$
and *idx*: $snd\ (snd\ (snd\ (snd\ s))) = VRet$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

lemma *ae_delta_VFwdPad_functional*:
fixes $M :: ('q, 'a)$ *mttm*
assumes *h1*: $(s, a, s1, a1, d1) \in alphabet\_enlarge\_delta\ M$
and *h2*: $(s, a, s2, a2, d2) \in alphabet\_enlarge\_delta\ M$
and *idx*: $snd\ (snd\ (snd\ (snd\ s))) = VFwdPad$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

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*: $le\_tm\ M \neq bl\_tm\ M$
and *h1*: $(s, a, s1, a1, d1) \in alphabet\_enlarge\_delta\ M$
and *h2*: $(s, a, s2, a2, d2) \in alphabet\_enlarge\_delta\ M$
shows $s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
⟨*proof*⟩

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*⟩

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*
  ⟨*proof*⟩

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*⟩

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*⟩

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*
  ⟨*proof*⟩

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*⟩

lemma *c_succ_idx_none*:
  fixes $x :: 'c :: enum$
  assumes $c\_succ\ x = None$
  shows $Suc\ (c\_idx\ x) = card\ (UNIV :: 'c\ set)$
⟨*proof*⟩

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*⟩

lemma *c_pred_idx_none*:
  fixes $x :: 'c :: enum$
  assumes $c\_pred\ x = None$
  shows $c\_idx\ x = 0$
  ⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bp_linear_AE_Home* [*simp*]:
fixes *off* :: *'c* :: *enum*
shows *bp_linear* (*AE_Home*, *off*) = *card* (*UNIV* :: *'c set*) + *c_idx off*
⟨*proof*⟩

lemma *bp_linear_AE_Right* [*simp*]:
fixes *off* :: *'c* :: *enum*
shows *bp_linear* (*AE_Right*, *off*)
= *2* ∗ *card* (*UNIV* :: *'c set*) + *c_idx off*
⟨*proof*⟩

lemma *bp_linear_lt_3c*:
fixes *p* :: (*'c* :: *enum*) *bp*
shows *bp_linear p* < *3* ∗ *card* (*UNIV* :: *'c set*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *bp_advance_N_linear*:
bp_advance p dir.N = *Some p* ∧ *bp_linear p* = *bp_linear p*
⟨*proof*⟩

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*⟩

lemma *c_idx_last*:
shows *c_idx* (*c_last* :: *'c* :: *enum*) = *card* (*UNIV* :: *'c set*) − *1*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

Single-step *bp_linear* preservation for *bp_advance_le*: the $c \leq$ *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*⟩

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*⟩

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*⟩

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, offset *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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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 (⇒): case-split on *fst bp*, compute *bp_linear* from the case, conclude the algebraic relation.

Backward direction (⇐): 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*
          $AE\_Left \Rightarrow snd\ bp = c\_last \wedge nM = 0$
        $\mid AE\_Home \Rightarrow nM = Suc\ (c\_idx\ (snd\ bp))$
        $\mid AE\_Right \Rightarrow nM = Suc\ (card\ (UNIV :: 'c\ set)$
                              $+\ c\_idx\ (snd\ bp)))$
      $\longleftrightarrow Suc\ (bp\_linear\ bp) = nM + card\ (UNIV :: 'c\ set)$
⟨*proof*⟩

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 $tM :: nat \Rightarrow 'a$
    and $nM :: nat$
    and $bp\ bp' :: ('c :: enum)\ bp$
    and $blocks :: ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$
    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*: $tM\ nM = le \Longrightarrow a' = le \wedge d \in \{dir.N,\ dir.R\}$
and *a_not_le*: $tM\ nM \neq le \Longrightarrow a' \neq le$
and *no_le_win*: $\forall i.\ 0 < i \wedge i \leq 2 * card\ (UNIV :: 'c\ set) \longrightarrow tM\ i \neq le$
shows *ae_window_invariant_le1*
$(tM(nM := a'))\ (go\_dir\ d\ nM)$
$bp'\ (write\_bp\ blocks\ bp\ a')\ le$
⟨*proof*⟩

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 :: $nat \Rightarrow 'a$
and nM :: *nat*
and *bp* bp' :: $('c :: enum)\ bp$
and *blocks* :: $('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$
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*: $tM\ nM = le \Longrightarrow a' = le \wedge d \in \{dir.N,\ dir.R\}$
and *a_not_le*: $tM\ nM \neq le \Longrightarrow a' \neq le$
and *no_le_win*: $\forall i.\ 0 < i \wedge i \leq card\ (UNIV :: 'c\ set) \longrightarrow tM\ i \neq le$
shows *ae_window_invariant_le0*
$(tM(nM := a'))\ (go\_dir\ d\ nM)$
$bp'\ (write\_bp\ blocks\ bp\ a')\ le$
⟨*proof*⟩

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*⟩

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*)
  ⟨*proof*⟩

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*⟩

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

lemma *ae_decode_input_append*:
  ae_decode_input bl (*xs* @ *ys*)
    $=$ *ae_decode_input bl xs* @ *ae_decode_input bl ys*
  ⟨*proof*⟩

lemma *length_ae_decode_input_pure*:
  fixes *xs* :: ($'c :: enum \Rightarrow 'a$) *list*
  assumes $\forall s <$ *length xs. is_pure_block bl* (*xs* ! *s*)
  shows *length* (*ae_decode_input bl xs*) $=$ *length xs* $*$ *card* (*UNIV* $:: 'c$ *set*)
  ⟨*proof*⟩

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 $(m * c + c - 1)\ div\ c = m$
⟨*proof*⟩

lemma *ceil_div_2c_minus_1*:
fixes $c :: nat$
assumes $0 < c$
shows $(c + c - 1)\ div\ c = 1$
⟨*proof*⟩

lemma *ceil_div_k_plus_c_minus_1*:
fixes $k\ c :: nat$
assumes $0 < c\ 0 < k\ k \leq c$
shows $(k + c - 1)\ div\ c = 1$
⟨*proof*⟩

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 $f :: 'c :: enum \Rightarrow 'a$
shows $encode\_input\ bl\ (map\ f\ (enum\_class.enum :: 'c\ list))$
$= ([f] :: ('c \Rightarrow 'a)\ list)$
⟨*proof*⟩

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 $f :: 'c :: enum \Rightarrow 'a$
assumes *m_pos*: $0 < m$
and *m_le*: $m \leq length\ (enum\_class.enum :: 'c\ list)$
and *pad*: $\forall x.\ c\_idx\ x \geq m \longrightarrow f\ x = bl$
shows $encode\_input\ bl\ (map\ f\ (take\ m\ (enum\_class.enum :: 'c\ list)))$
$= ([f] :: ('c \Rightarrow 'a)\ list)$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

Encoder image lies inside *gamma_block* (Σ_*M* ∪ {*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*⟩

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* ∉ *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*⟩

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* ∉ *Σ_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*⟩

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*⟩

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 $x :: 'c :: enum \Rightarrow 'a$
  assumes *is_canonical_block bl x*
  shows *encode_input bl* (*ae_decode_block bl x*)
        $= ([x] :: ('c \Rightarrow 'a)\ list)$
⟨*proof*⟩

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 $w :: ('c :: enum \Rightarrow 'a)\ list$
  assumes *wf*: *ae_input_well_formed bl w*
  shows *encode_input bl* (*ae_decode_input bl w*) $= w$
⟨*proof*⟩

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* $\cup$ $\{bl\_M\}$), 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 $c :: 'c :: enum \Rightarrow 'a$
    and *Sigma* $:: 'a\ set$
    and $bl :: 'a$
  assumes $c \in$ *gamma_block* (*Sigma* $\cup$ $\{bl\}$)
  shows *set* (*ae_decode_block bl c*) $\subseteq$ *Sigma*
⟨*proof*⟩

The "in encoder image $\longrightarrow$ 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* $u \subseteq$ *Sigma_tm M* suffices) and is cited by *ae_validation_post_state_canonical* for going from "given $u \in$ *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*⟩

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*⟩

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 $a' :: nat \Rightarrow ('c \Rightarrow 'a)$
and $d :: nat \Rightarrow dir$
assumes *rel*: $(s, (\lambda k.\ ts\ k\ (n\ k)), s', a', d)$
$\in alphabet\_enlarge\_delta\ M$
shows $(Config_M\ s\ ts\ n,$
$Config_M\ s'\ (\lambda k.\ (ts\ k)(n\ k := a'\ k))$
$(\lambda k.\ go\_dir\ (d\ k)\ (n\ k)))$
$\in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
⟨*proof*⟩

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*:
$(\forall j{\geq}K.\ init\_offset\ j = init\_offset\ j)$
$\wedge\ (\forall j{\geq}K.\ init\_buffer\ le\ j = init\_buffer\ le\ j)$
$\wedge\ (\forall j{\geq}K.\ init\_dest\ j = init\_dest\ j)$
⟨*proof*⟩

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 $ts :: nat \Rightarrow nat \Rightarrow ('c :: enum \Rightarrow 'a)$
and $n :: nat \Rightarrow nat$
and $q :: 'q$
and $ofs :: nat \Rightarrow 'c$
and $buf :: nat \Rightarrow ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$
and $dest :: nat \Rightarrow ae\_dest$
assumes *q_in*: $q \in Q\_tm\ M$
and *q_neq_t*: $q \neq t\_tm\ M$
and *q_neq_r*: $q \neq r\_tm\ M$
and *read*: $ts\ (0 :: nat)\ (n\ 0) = LE\_block\ (le\_tm\ M)$
$\vee\ is\_pure\_block\ (bl\_tm\ M)\ (ts\ 0\ (n\ 0))$
and *bt*: $\forall j{\geq}k\_tm\ M.\ \forall i.\ ts\ j\ i = bl\_block\ (bl\_tm\ M)$
and *stage_tail*: $(\forall j{\geq}k\_tm\ M.\ ofs\ j = init\_offset\ j)$
$\wedge\ (\forall j{\geq}k\_tm\ M.\ buf\ j = init\_buffer\ (le\_tm\ M)\ j)$
$\wedge\ (\forall j{\geq}k\_tm\ M.\ dest\ j = init\_dest\ j)$
and *gamma*: $\forall k.\ ts\ k\ (n\ k) \in gamma\_block\ (\Gamma\_tm\ M)$
and *buf_gamma*: $\forall k.\ fst\ (buf\ k) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ fst\ (snd\ (buf\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ snd\ (snd\ (buf\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
shows $(Config_M\ (q, ofs, buf, dest, VFwd)\ ts\ n,$
$Config_M\ (q, ofs, buf, dest, VFwd)\ ts$
$(\lambda k.\ go\_dir\ (if\ k = 0\ then\ dir.R\ else\ dir.N)\ (n\ k)))$
$\in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
⟨*proof*⟩

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 $ts :: nat \Rightarrow nat \Rightarrow ('c :: enum \Rightarrow 'a)$
    and $n :: nat \Rightarrow nat$
    and $q :: 'q$
    and $ofs :: nat \Rightarrow 'c$
    and $buf :: nat \Rightarrow ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$
    and $dest :: nat \Rightarrow ae\_dest$
  assumes *q_in*: $q \in Q\_tm\ M$
      and *q_neq_t*: $q \neq t\_tm\ M$
      and *q_neq_r*: $q \neq r\_tm\ M$
      and *read*: $is\_padded\_block\ (bl\_tm\ M)\ (ts\ (0 :: nat)\ (n\ 0))$
      and *bt*: $\forall j{\geq}k\_tm\ M.\ \forall i.\ ts\ j\ i = bl\_block\ (bl\_tm\ M)$
      and *stage_tail*: $(\forall j{\geq}k\_tm\ M.\ ofs\ j = init\_offset\ j)$
                  $\wedge\ (\forall j{\geq}k\_tm\ M.\ buf\ j = init\_buffer\ (le\_tm\ M)\ j)$
                  $\wedge\ (\forall j{\geq}k\_tm\ M.\ dest\ j = init\_dest\ j)$
      and *gamma*: $\forall k.\ ts\ k\ (n\ k) \in gamma\_block\ (\Gamma\_tm\ M)$
      and *buf_gamma*: $\forall k.\ fst\ (buf\ k) \in gamma\_block\ (\Gamma\_tm\ M)$
                  $\wedge\ fst\ (snd\ (buf\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
                  $\wedge\ snd\ (snd\ (buf\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
  shows $(Config_M\ (q,\ ofs,\ buf,\ dest,\ VFwd)\ ts\ n,$
          $Config_M\ (q,\ ofs,\ buf,\ dest,\ VFwdPad)\ ts$
            $(\lambda k.\ go\_dir\ (if\ k = 0\ then\ dir.R\ else\ dir.N)\ (n\ k)))$
        $\in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
⟨*proof*⟩

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 $ts :: nat \Rightarrow nat \Rightarrow ('c :: enum \Rightarrow 'a)$
    and $n :: nat \Rightarrow nat$
    and $q :: 'q$
    and $ofs :: nat \Rightarrow 'c$
    and $buf :: nat \Rightarrow ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$
    and $dest :: nat \Rightarrow ae\_dest$
  assumes *q_in*: $q \in Q\_tm\ M$
      and *q_neq_t*: $q \neq t\_tm\ M$
      and *q_neq_r*: $q \neq r\_tm\ M$
      and *read*: $ts\ (0 :: nat)\ (n\ 0) = bl\_block\ (bl\_tm\ M)$
      and *bt*: $\forall j{\geq}k\_tm\ M.\ \forall i.\ ts\ j\ i = bl\_block\ (bl\_tm\ M)$
      and *stage_tail*: $(\forall j{\geq}k\_tm\ M.\ ofs\ j = init\_offset\ j)$
                  $\wedge\ (\forall j{\geq}k\_tm\ M.\ buf\ j = init\_buffer\ (le\_tm\ M)\ j)$
                  $\wedge\ (\forall j{\geq}k\_tm\ M.\ dest\ j = init\_dest\ j)$
      and *gamma*: $\forall k.\ ts\ k\ (n\ k) \in gamma\_block\ (\Gamma\_tm\ M)$
      and *buf_gamma*: $\forall k.\ fst\ (buf\ k) \in gamma\_block\ (\Gamma\_tm\ M)$
                  $\wedge\ fst\ (snd\ (buf\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$

$\wedge$ *snd* (*snd* (*buf k*)) $\in$ *gamma_block* (Γ*_tm M*)
shows (*Config*$_M$ (*q*, *ofs*, *buf*, *dest*, *VFwd*) *ts n*,
Config$_M$ (*q*, *ofs*, *buf*, *dest*, *VRet*) *ts n*)
$\in$ *mttm_step* (*alphabet_enlarge_delta M*)
⟨*proof*⟩

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* $\in$ *Q_tm M*
and *read*: *ts* (*0* :: *nat*) (*n 0*) = *bl_block* (*bl_tm M*)
and *bt*: $\forall$ *j*≥*k_tm M*. $\forall$ *i*. *ts j i* = *bl_block* (*bl_tm M*)
and *stage_tail*: ($\forall$ *j*≥*k_tm M*. *ofs j* = *init_offset j*)
$\wedge$ ($\forall$ *j*≥*k_tm M*. *buf j* = *init_buffer* (*le_tm M*) *j*)
$\wedge$ ($\forall$ *j*≥*k_tm M*. *dest j* = *init_dest j*)
and *gamma*: $\forall$ *k*. *ts k* (*n k*) $\in$ *gamma_block* (Γ*_tm M*)
and *buf_gamma*: $\forall$ *k*. *fst* (*buf k*) $\in$ *gamma_block* (Γ*_tm M*)
$\wedge$ *fst* (*snd* (*buf k*)) $\in$ *gamma_block* (Γ*_tm M*)
$\wedge$ *snd* (*snd* (*buf k*)) $\in$ *gamma_block* (Γ*_tm M*)
shows (*Config*$_M$ (*q*, *ofs*, *buf*, *dest*, *VFwdPad*) *ts n*,
Config$_M$ (*q*, *ofs*, *buf*, *dest*, *VRet*) *ts n*)
$\in$ *mttm_step* (*alphabet_enlarge_delta M*)
⟨*proof*⟩

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* $\in$ *Q_tm M*
and *read*: *ts* (*0* :: *nat*) (*n 0*) $\neq$ *LE_block* (*le_tm M*)
and *bt*: $\forall$ *j*≥*k_tm M*. $\forall$ *i*. *ts j i* = *bl_block* (*bl_tm M*)
and *stage_tail*: ($\forall$ *j*≥*k_tm M*. *ofs j* = *init_offset j*)
$\wedge$ ($\forall$ *j*≥*k_tm M*. *buf j* = *init_buffer* (*le_tm M*) *j*)
$\wedge$ ($\forall$ *j*≥*k_tm M*. *dest j* = *init_dest j*)
and *gamma*: $\forall$ *k*. *ts k* (*n k*) $\in$ *gamma_block* (Γ*_tm M*)
and *buf_gamma*: $\forall$ *k*. *fst* (*buf k*) $\in$ *gamma_block* (Γ*_tm M*)

$\wedge$ *fst* (*snd* (*buf k*)) $\in$ *gamma_block* (Γ*_tm M*)
$\wedge$ *snd* (*snd* (*buf k*)) $\in$ *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*)))
$\in$ *mttm_step* (*alphabet_enlarge_delta M*)
⟨*proof*⟩

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* $\in$ *Q_tm M*
and *read*: *ts* (*0* :: *nat*) (*n 0*) = *LE_block* (*le_tm M*)
and *bt*: $\forall$ *j*≥*k_tm M*. $\forall$ *i*. *ts j i* = *bl_block* (*bl_tm M*)
and *stage_tail*: ($\forall$ *j*≥*k_tm M*. *ofs j* = *init_offset j*)
$\wedge$ ($\forall$ *j*≥*k_tm M*. *buf j* = *init_buffer* (*le_tm M*) *j*)
$\wedge$ ($\forall$ *j*≥*k_tm M*. *dest j* = *init_dest j*)
and *gamma*: $\forall$ *k*. *ts k* (*n k*) $\in$ *gamma_block* (Γ*_tm M*)
and *buf_gamma*: $\forall$ *k*. *fst* (*buf k*) $\in$ *gamma_block* (Γ*_tm M*)
$\wedge$ *fst* (*snd* (*buf k*)) $\in$ *gamma_block* (Γ*_tm M*)
$\wedge$ *snd* (*snd* (*buf k*)) $\in$ *gamma_block* (Γ*_tm M*)
shows (*Config*$_M$ (*q*, *ofs*, *buf*, *dest*, *VRet*) *ts n*,
Config$_M$ (*q*, *ofs*, *buf*, *dest*, *SS1*) *ts n*)
$\in$ *mttm_step* (*alphabet_enlarge_delta M*)
⟨*proof*⟩

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* $\in$ *Q_tm M*
and *q_neq_t*: *q* $\neq$ *t_tm M*
and *q_neq_r*: *q* $\neq$ *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*⟩

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*⟩

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*))
⟨*proof*⟩

lemma *ae_init_config_pos*:
mt_pos (*ae_init_config M w*) *k* = *0*
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

Blank-tail of the initial configuration: every cell of every inactive tape (index ≥ *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 ∀ *j*≥*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*)
⟨*proof*⟩

Gamma-block membership of every tape cell of the initial configuration. Uniformly discharges the ∀ *k*. *ts k* (*n k*) ∈ *gamma_block* (Γ*_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*⟩

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*...*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*))
(λ*i* :: *nat*. *if i* = *0 then Suc k else 0*))
∈ *mttm_step* (*alphabet_enlarge_delta M*) ^^ *Suc k*
⟨*proof*⟩

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*...*p* is hoisted into the goal as a ⟶-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: $q \in Q\_tm\ M$
and $tape\_le$: $ts\ (0 :: nat)\ 0 = LE\_block\ (le\_tm\ M)$
and $gamma$: $\forall k\ i.\ ts\ k\ i \in gamma\_block\ (\Gamma\_tm\ M)$
and buf_gamma: $\forall k.\ fst\ (buf\ k) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ fst\ (snd\ (buf\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ snd\ (snd\ (buf\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
and bt: $\forall k{\geq}k\_tm\ M.\ \forall i.\ ts\ k\ i = bl\_block\ (bl\_tm\ M)$
and $stage\_tail$: $(\forall j{\geq}k\_tm\ M.\ ofs\ j = init\_offset\ j)$
$\wedge\ (\forall j{\geq}k\_tm\ M.\ buf\ j = init\_buffer\ (le\_tm\ M)\ j)$
$\wedge\ (\forall j{\geq}k\_tm\ M.\ dest\ j = init\_dest\ j)$
shows $(\forall i.\ 1 \leq i \wedge i \leq p \longrightarrow ts\ 0\ i \neq LE\_block\ (le\_tm\ M))$
$\longrightarrow (Config_M\ (q,\ ofs,\ buf,\ dest,\ VRet)\ ts$
$(\lambda i :: nat.\ if\ i = 0\ then\ p\ else\ 0),$
$Config_M\ (q,\ ofs,\ buf,\ dest,\ SS1)\ ts\ (\lambda\_.\ 0))$
$\in mttm\_step\ (alphabet\_enlarge\_delta\ M)\ \widehat{\ }\ Suc\ p$
⟨*proof*⟩

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 $ts :: nat \Rightarrow nat \Rightarrow ('c :: enum \Rightarrow 'a)$
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 $p :: nat$
assumes q_in: $q \in Q\_tm\ M$
and $gamma$: $\forall k\ i.\ ts\ k\ i \in gamma\_block\ (\Gamma\_tm\ M)$
and buf_gamma: $\forall k.\ fst\ (buf\ k) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ fst\ (snd\ (buf\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
$\wedge\ snd\ (snd\ (buf\ k)) \in gamma\_block\ (\Gamma\_tm\ M)$
and bt: $\forall k{\geq}k\_tm\ M.\ \forall i.\ ts\ k\ i = bl\_block\ (bl\_tm\ M)$
and $stage\_tail$: $(\forall j{\geq}k\_tm\ M.\ ofs\ j = init\_offset\ j)$
$\wedge\ (\forall j{\geq}k\_tm\ M.\ buf\ j = init\_buffer\ (le\_tm\ M)\ j)$
$\wedge\ (\forall j{\geq}k\_tm\ M.\ dest\ j = init\_dest\ j)$
and non_le: $\forall i.\ 1 \leq i \wedge i \leq p \longrightarrow ts\ 0\ i \neq LE\_block\ (le\_tm\ M)$
shows $m \leq p$
$\longrightarrow (Config_M\ (q,\ ofs,\ buf,\ dest,\ VRet)\ ts$
$(\lambda i :: nat.\ if\ i = 0\ then\ p\ else\ 0),$
$Config_M\ (q,\ ofs,\ buf,\ dest,\ VRet)\ ts$
$(\lambda i :: nat.\ if\ i = 0\ then\ p - m\ else\ 0))$
$\in mttm\_step\ (alphabet\_enlarge\_delta\ M)\ \widehat{\ }\ m$
⟨*proof*⟩

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* $\subseteq$ *Sigma_tm M*
  shows $\forall k{<}k\_tm\ M.$ *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* $\Rightarrow$ *'a*) *list*)) *k*)
⟨*proof*⟩

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 $w :: ('c :: enum \Rightarrow 'a)\ list$

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*⟩

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*))))
∈ {*VFwd*, *VFwdPad*, *VRet*}
⟨*proof*⟩

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 :: $('c :: enum \Rightarrow 'a)$ *list*
  assumes *vM*: *valid_mttm M*
      and *w_sub*: *set w* $\subseteq$ *gamma_block* (*Sigma_tm M* $\cup$ $\{$*bl_tm M*$\}$)
      and *wf*: *ae_input_well_formed* (*bl_tm M*) *w*
      and *s_in_Q*: *s_tm M* $\in$ *Q_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*
  shows $\forall i{<}2 * length\ w + 4.\ \forall d.$
          (*ae_init_config 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*$\}$
⟨*proof*⟩

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* $\subseteq$ *Sigma_tm M*
      and *le_neq_bl*: *le_tm M* $\neq$ *bl_tm M*
  obtains n :: *nat* **and** c' **where**
      (*ae_init_config M* (*encode_input* (*bl_tm M*) *u*), c')
          $\in$ *mttm_step* (*alphabet_enlarge_delta M*) $^\frown\!\!^\frown$ *n*
    and *ae_simulates M*
          (*init_config_mttm M u*)
          (c' :: ($'c :: enum \Rightarrow 'a$,
                $'q \times ('a, 'c)$ *ae_stage*) *mt_config*)
    and *ae_buffer_in_gamma_block M* c'
    and $\forall kk{<}k\_tm\ M.$ *mt_tape* c' *kk 0* = *LE_block* (*le_tm M*)
    and $\forall i{<}n.\ \forall d :: ('c :: enum \Rightarrow 'a,$
                    $'q \times ('a, 'c)$ *ae_stage*) *mt_config*.
          (*ae_init_config M* (*encode_input* (*bl_tm M*) *u*), *d*)
             $\in$ *mttm_step* (*alphabet_enlarge_delta M*) $^\frown\!\!^\frown$ *i*
           $\longrightarrow$ *snd* (*snd* (*snd* (*snd* (*mt_state d*))))
               $\in$ $\{$*VFwd*, *VFwdPad*, *VRet*$\}$
⟨*proof*⟩

Noncanonical inputs reject. The *bl_block bl_M* $\notin$ *set w* hypothesis matches *M*'s input alphabet Σ' (Σ' = *gamma_block* (Σ*_M* $\cup$ $\{$*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* (Σ*_M* $\cup$ $\{$*bl_M*$\}$) plus *le_M* $\notin$ Σ*_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) ⇒ 'a) list
  assumes vM: valid_mttm M
      and w_sub: set w ⊆ gamma_block (Sigma_tm M ∪ {bl_tm M})
      and bl_notin: bl_block (bl_tm M) ∉ set w
      and w_bad: ¬ ae_input_well_formed (bl_tm M) w
      and s_neq_t: s_tm M ≠ t_tm M
      and s_neq_r: s_tm M ≠ r_tm M
  obtains n :: nat and c' where
      n ≤ length w + 2
    and (ae_init_config M w, c')
          ∈ mttm_step (alphabet_enlarge_delta M) ^^ n
    and case mt_state c' of (qM', _, _, _, _) ⇒ qM' = r_tm M
⟨proof⟩

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⟩

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→SSn+*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* ∈ *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*.

$(s,\ a,\ s',\ a',\ d) \in \delta$
$\Longrightarrow \exists\, q\ ofs\ buf\ dest.$
$s' = (q,\ ofs,\ buf,\ dest,\ tgt)$
$\wedge\ q \in Q\_tm\ M$
shows *case mt_state c'' of* $(qM',\ \_,\ \_,\ \_,\ idx) \Rightarrow$
$idx = tgt \wedge qM' \in Q\_tm\ M$
⟨*proof*⟩

lemma *ae_step_ss1_ss2_invariant*:
fixes $M :: ('q,\ 'a)\ mttm$
and $c'\ c'' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a,\ 'c)\ ae\_stage)\ mt\_config$
assumes *valid_mttm M*
and *ae_inv_ss1 M c'*
and $(c',\ c'') \in mttm\_step\ (ae\_delta\_ss1\_ss2\ M)$
shows *ae_inv_ss2 M c''*
⟨*proof*⟩

lemma *ae_step_ss2_ss3_invariant*:
fixes $M :: ('q,\ 'a)\ mttm$
and $c'\ c'' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a,\ 'c)\ ae\_stage)\ mt\_config$
assumes *valid_mttm M*
and *ae_inv_ss2 M c'*
and $(c',\ c'') \in mttm\_step\ (ae\_delta\_ss2\_ss3\ M)$
shows *ae_inv_ss3 M c''*
⟨*proof*⟩

lemma *ae_step_ss3_ss4_invariant*:
fixes $M :: ('q,\ 'a)\ mttm$
and $c'\ c'' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a,\ 'c)\ ae\_stage)\ mt\_config$
assumes *valid_mttm M*
and *ae_inv_ss3 M c'*
and $(c',\ c'') \in mttm\_step\ (ae\_delta\_ss3\_ss4\ M)$
shows *ae_inv_ss4 M c''*
⟨*proof*⟩

lemma *ae_step_ss4_ss5_invariant*:
fixes $M :: ('q,\ 'a)\ mttm$
and $c'\ c'' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a,\ 'c)\ ae\_stage)\ mt\_config$
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*⟩

lemma *ae_step_ss5_ss6_invariant*:

fixes $M :: ('q, 'a)\ mttm$
and $c'\ c'' :: ('c :: enum \Rightarrow 'a, 'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
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*⟩

lemma *ae_step_ss6_ss7_invariant*:
fixes $M :: ('q, 'a)\ mttm$
and $c'\ c'' :: ('c :: enum \Rightarrow 'a, 'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
assumes *valid_mttm M*
and *ae_inv_ss6 M c'*
and $(c', c'') \in mttm\_step\ (ae\_delta\_ss6\_ss7\ M)$
shows *ae_inv_ss7 M c''*
⟨*proof*⟩

lemma *ae_step_ss7_ss8_invariant*:
fixes $M :: ('q, 'a)\ mttm$
and $c'\ c'' :: ('c :: enum \Rightarrow 'a, 'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
assumes *valid_mttm M*
and *ae_inv_ss7 M c'*
and $(c', c'') \in mttm\_step\ (ae\_delta\_ss7\_ss8\ M)$
shows *ae_inv_ss8 M c''*
⟨*proof*⟩

lemma *ae_step_ss8_ss1_invariant*:
fixes $M :: ('q, 'a)\ mttm$
and $c'\ c'' :: ('c :: enum \Rightarrow 'a, 'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
assumes *valid_mttm M*
and *ae_inv_ss8 M c'*
and *non_halt*: *case mt_state c' of* $(qM', \_, \_, \_, \_) \Rightarrow$
$qM' \neq t\_tm\ M \wedge qM' \neq r\_tm\ M$
and *step*: $(c', c'') \in mttm\_step\ (ae\_delta\_ss8\_ss1\ M)$
shows *ae_inv_ss1 M c''*
⟨*proof*⟩

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 $c'\ c'' :: ('c :: enum \Rightarrow 'a,$
                $'q \times ('a, 'c)$ *ae_stage*$)$ *mt_config*
  assumes *inv*: *ae_inv_ss8* $M\ c'$
      and *halt*: *case mt_state* c' *of* $(qM', \_, \_, \_, \_) \Rightarrow$
                $qM' \in \{t\_tm\ M, r\_tm\ M\}$
      and *step*: $(c', c'') \in$ *mttm_step* (*ae_delta_ss8_ss1* M)
    shows (*case mt_state* c'' *of* $(qM', \_, \_, \_, idx) \Rightarrow$
            $qM' \in \{t\_tm\ M, r\_tm\ M\} \wedge idx = VFwd$)
⟨*proof*⟩

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 $c\ c' :: ('c :: enum \Rightarrow 'a,$
                $'q \times ('a, 'c)$ *ae_stage*$)$ *mt_config*
  assumes *step*: $(c, c') \in$ *mttm_step* (*alphabet_enlarge_delta* M)
      and *ss1*: *snd* (*snd* (*snd* (*snd* (*mt_state* c)))) = *SS1*
    shows $(c, c') \in$ *mttm_step* (*ae_delta_ss1_ss2* M)
⟨*proof*⟩

lemma *mttm_step_ae_delta_ss2_only*:
  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* (*alphabet_enlarge_delta* M)
      and *ss2*: *snd* (*snd* (*snd* (*snd* (*mt_state* c)))) = *SS2*
    shows $(c, c') \in$ *mttm_step* (*ae_delta_ss2_ss3* M)
⟨*proof*⟩

lemma *mttm_step_ae_delta_ss3_only*:
  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* (*alphabet_enlarge_delta* M)
      and *ss3*: *snd* (*snd* (*snd* (*snd* (*mt_state* c)))) = *SS3*
    shows $(c, c') \in$ *mttm_step* (*ae_delta_ss3_ss4* M)
⟨*proof*⟩

lemma *mttm_step_ae_delta_ss4_only*:

fixes $M :: ('q, 'a)\ mttm$
and $c\ c' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
assumes *step*: $(c, c') \in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
and *ss4*: $snd\ (snd\ (snd\ (snd\ (mt\_state\ c)))) = SS4$
shows $(c, c') \in mttm\_step\ (ae\_delta\_ss4\_ss5\ M)$
⟨*proof*⟩

lemma *mttm_step_ae_delta_ss5_only*:
fixes $M :: ('q, 'a)\ mttm$
and $c\ c' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
assumes *step*: $(c, c') \in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
and *ss5*: $snd\ (snd\ (snd\ (snd\ (mt\_state\ c)))) = SS5$
shows $(c, c') \in mttm\_step\ (ae\_delta\_ss5\_ss6\ M)$
⟨*proof*⟩

lemma *mttm_step_ae_delta_ss6_only*:
fixes $M :: ('q, 'a)\ mttm$
and $c\ c' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
assumes *step*: $(c, c') \in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
and *ss6*: $snd\ (snd\ (snd\ (snd\ (mt\_state\ c)))) = SS6$
shows $(c, c') \in mttm\_step\ (ae\_delta\_ss6\_ss7\ M)$
⟨*proof*⟩

lemma *mttm_step_ae_delta_ss7_only*:
fixes $M :: ('q, 'a)\ mttm$
and $c\ c' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
assumes *step*: $(c, c') \in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
and *ss7*: $snd\ (snd\ (snd\ (snd\ (mt\_state\ c)))) = SS7$
shows $(c, c') \in mttm\_step\ (ae\_delta\_ss7\_ss8\ M)$
⟨*proof*⟩

lemma *mttm_step_ae_delta_ss8_only*:
fixes $M :: ('q, 'a)\ mttm$
and $c\ c' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
assumes *step*: $(c, c') \in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
and *ss8*: $snd\ (snd\ (snd\ (snd\ (mt\_state\ c)))) = SS8$
shows $(c, c') \in mttm\_step\ (ae\_delta\_ss8\_ss1\ M)$
⟨*proof*⟩

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* ⇒ *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* ≥ *2*. Uniform *p_start* = (*pos* − *2*) ∗ *c* + *1* simplifies to *1* on *pos* ∈ {*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 $\exists buf'\ end\_pos.$
$((qM, buf\_full, \lambda k.\ (AE\_Home, ofs\ k)),$
$(mt\_state\ cM\_n, buf', end\_pos))$
$\in (m\_step\_buffered\ M)\ ^{\frown\frown}\ n$
$\wedge\ (\forall k{<}k\_tm\ M.\ ae\_window\_invariant\_general$

$$
\begin{aligned}
&\quad (mt\_tape\ cM\_n\ k)\ (mt\_pos\ cM\_n\ k)\\
&\quad (end\_pos\ k)\ (buf'\ k)\ (pos\ k)\ (le\_tm\ M))\\
&\wedge (\forall k.\ (pos\ k = 0\\
&\qquad \longrightarrow (\forall i.\ i < card\ (UNIV :: 'c\ set)\\
&\qquad\quad \longrightarrow mt\_tape\ cM\_n\ k\ (Suc\ i) \neq le\_tm\ M))\\
&\quad \wedge (pos\ k = 1\\
&\qquad \longrightarrow (\forall i.\ i < 2 * card\ (UNIV :: 'c\ set)\\
&\qquad\quad \longrightarrow mt\_tape\ cM\_n\ k\ (Suc\ i) \neq le\_tm\ M))\\
&\quad \wedge (pos\ k \geq 2\\
&\qquad \longrightarrow (\forall i.\ i < 3 * card\ (UNIV :: 'c\ set)\\
&\qquad\quad \longrightarrow mt\_tape\ cM\_n\ k\\
&\qquad\qquad ((pos\ k - 2) * card\ (UNIV :: 'c\ set)\\
&\qquad\qquad\quad + 1 + i)\\
&\qquad\qquad \neq le\_tm\ M)))\\
&\wedge (\forall k{<}k\_tm\ M.\ pos\ k = 0\\
&\qquad \longrightarrow fst\ (buf'\ k) \neq LE\_block\ (le\_tm\ M))\\
&\wedge (\forall k.\ card\ (UNIV :: 'c\ set) \leq bp\_linear\ (end\_pos\ k) + n\\
&\qquad \wedge bp\_linear\ (end\_pos\ k)\\
&\qquad\quad < 2 * card\ (UNIV :: 'c\ set) + n)\\
&\wedge (\forall k{\geq}k\_tm\ M.\ fst\ (snd\ (buf'\ k)) = bl\_block\ (bl\_tm\ M)\\
&\qquad \wedge end\_pos\ k = (AE\_Home,\ ofs\ k))
\end{aligned}
$$

⟨*proof*⟩

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 $tsM :: nat \Rightarrow nat \Rightarrow 'a$
    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 $kM :: nat$
    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*: $(Config_M\ qM\ tsM\ nM,\ cM\_k)$
$\in mttm\_step\ (delta\_tm\ M)\ \widehat{\ }\widehat{\ }\ kM$

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*:
$\forall\, k{<}k\_tm\ M.\ pos\ k = 0 \longrightarrow fst\ (buf\_full\ k) \neq LE\_block\ (le\_tm\ M)$

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 $\forall\, k.\ (pos\ k = 0$
$\longrightarrow (\forall\, i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow mt\_tape\ cM\_k\ k\ (Suc\ i) \neq le\_tm\ M))$
$\wedge\ (pos\ k = 1$
$\longrightarrow (\forall\, i.\ i < 2 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt\_tape\ cM\_k\ k\ (Suc\ i) \neq le\_tm\ M))$
$\wedge\ (pos\ k \geq 2$
$\longrightarrow (\forall\, i.\ i < 3 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt\_tape\ cM\_k\ k$
$((pos\ k - 2) * card\ (UNIV :: 'c\ set)$
$+ 1 + i)$
$\neq le\_tm\ M))$

and $\forall\, k{<}k\_tm\ M.\ pos\ k = 0$
$\longrightarrow fst\ (buf'\ k) \neq LE\_block\ (le\_tm\ M)$

⟨*proof*⟩

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*) ∈ *delta_tm*⁀⁀*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*: $((q, \lambda k.\ (fst\ (buf\ k),$
        if $k < k\_tm\ M$ *then fst* $(snd\ (buf\ k))$ *else ts k* $(n\ k)$,
        ts k $(n\ k))$,
      $\lambda k.\ (AE\_Home, ofs\ k))$,
    $(q\_out, buf', end\_pos)) \in m\_steps\_buffered\ M$
  obtains c'' **where**

$(c', c'') \in mttm\_step\ (ae\_delta\_ss4\_ss5\ M)$
and $(c', c'') \in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
and $mt\_state\ c'' = (q\_out,$
$\lambda k.\ if\ k < k\_tm\ M\ then\ snd\ (end\_pos\ k)\ else\ init\_offset\ k,$
$\lambda k.\ if\ k < k\_tm\ M\ then\ buf'\ k\ else\ init\_buffer\ (le\_tm\ M)\ k,$
$\lambda k.\ if\ k < k\_tm\ M\ then\ fst\ (end\_pos\ k)\ else\ init\_dest\ k,\ SS5)$
and $mt\_tape\ c'' = ts$
⟨*proof*⟩

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 $c' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
and $tsM :: nat \Rightarrow nat \Rightarrow 'a$
and $nM :: nat \Rightarrow nat$
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*:
$(Config_M\ (fst\ (mt\_state\ c'))\ tsM\ nM,\ cM\_k)$
$\in mttm\_step\ (delta\_tm\ M)\ ^{\frown\frown}\ kM$
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*:
$case\ mt\_state\ c'\ of\ (\_,\ \_,\ buf,\ \_,\ \_) \Rightarrow$
$\forall k{<}k\_tm\ M.\ pos\ k = 0 \longrightarrow fst\ (buf\ k) \neq LE\_block\ (le\_tm\ M)$
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*⟩

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 (paral-

lel 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 ≥ 1*, home ≠ *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 ≥ 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*,
           (λ*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*),
            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*⟩

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*⟩

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 $c' :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
assumes *le_anchor*: $\forall k{<}k\_tm\ M.\ mt\_tape\ c'\ k\ 0 = LE\_block\ (le\_tm\ M)$
and *no_le_per_tape*:
$\forall k.\ (mt\_pos\ c'\ k \geq 2$
$\longrightarrow (\forall i.\ i < 3 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt\_tape\ cM\ k$
$((mt\_pos\ c'\ k - 2)$
$* card\ (UNIV :: 'c\ set) + 1 + i)$
$\neq le\_tm\ M))$
$\wedge\ (mt\_pos\ c'\ k = 1$
$\longrightarrow (\forall i.\ i < 2 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt\_tape\ cM\ k\ (Suc\ i) \neq le\_tm\ M))$
$\wedge\ (mt\_pos\ c'\ k = 0$
$\longrightarrow (\forall i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow mt\_tape\ cM\ k\ (Suc\ i) \neq le\_tm\ M))$
and *tape_corr*:
$\forall k{<}k\_tm\ M.\ ae\_tape\_correspondence\ (le\_tm\ M)$
$(mt\_tape\ cM\ k)\ (mt\_tape\ c'\ k)$
shows $\forall k{<}k\_tm\ M.\ (mt\_pos\ c'\ k = 0$
$\longrightarrow mt\_tape\ c'\ k\ (mt\_pos\ c'\ k) = LE\_block\ (le\_tm\ M))$
$\wedge\ (mt\_pos\ c'\ k \geq 1$
$\longrightarrow mt\_tape\ c'\ k\ (mt\_pos\ c'\ k) \neq LE\_block\ (le\_tm\ M))$
⟨*proof*⟩

Auxiliary: unpack the *ae_simulates* invariant at SS1 entry into the struc-

tural 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 equality, the M↔M' tape and position correspondences, the gamma-block side-band, 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 $c' :: ('c :: enum \Rightarrow 'a,$
            $'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
  assumes *sim*: $ae\_simulates\ M\ cM\ c'$
      and *qM_in_Q*: $mt\_state\ cM \in Q\_tm\ M$
      and *q_neq_t*: $mt\_state\ cM \neq t\_tm\ M$
      and *q_neq_r*: $mt\_state\ cM \neq r\_tm\ M$
  obtains $qM'\ ofs\ buf\ dest$ **where**
      $mt\_state\ c' = (qM', ofs, buf, dest, SS1)$
    and $mt\_state\ cM = qM'$
    and $qM' \in Q\_tm\ M$
    and $qM' \neq t\_tm\ M$
    and $qM' \neq r\_tm\ M$
    and $\forall k{<}k\_tm\ M.\ ae\_tape\_correspondence\ (le\_tm\ M)$
          $(mt\_tape\ cM\ k)\ (mt\_tape\ c'\ k)$
    and $\forall k{<}k\_tm\ M.\ mt\_pos\ cM\ k = ae\_decode\_pos\ (mt\_pos\ c'\ k)\ (ofs\ k)$
    and $ae\_tape\_in\_gamma\_block\ M\ c'$
    and $ae\_inv\_ss1\ M\ c'$
    and $ae\_position\_link\ M\ c'$
⟨*proof*⟩

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 $c' :: ('c :: enum \Rightarrow 'a,$
    $'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
  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 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: $qM' \neq r\_tm\ M$
obtains *c1 c2 c3* **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*⟩

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*⟩

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*⟩

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* $\Rightarrow$ '*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 *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 *c4_pos*: $\bigwedge$*kk*. *kk* < *k_tm M* $\Longrightarrow$ *mt_pos c4 kk* = *mt_pos c' kk*
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*: (*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 *step8_sub*: (*c7*, *c8*) $\in$ *mttm_step* (*ae_delta_ss8_ss1 M*)
and *step8*: (*c7*, *c8*) $\in$ *mttm_step* (*alphabet_enlarge_delta M*)
and *c5_pos_for_pos_ge2*:
$\bigwedge$*kk*. *kk* < *k_tm M* $\Longrightarrow$ *mt_pos c' kk* $\geq$ *2* $\Longrightarrow$ *dest5 kk* $\neq$ *AE_Left*
$\Longrightarrow$ *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 *left_not_le_c'_steady*:
$\forall k.\ mt\_pos\ c'\ k \geq 2$
$\longrightarrow mt\_tape\ c'\ k\ (mt\_pos\ c'\ k - 1) \neq LE\_block\ (le\_tm\ M)$
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_pos_ge2*:
$\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk \geq 2$
$\Longrightarrow mt\_pos\ c7\ kk = (if\ dest5\ kk = AE\_Left$
$then\ mt\_pos\ c'\ kk - 1$
$else\ mt\_pos\ c'\ kk + 1)$
shows $\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk \geq 2$
$\Longrightarrow mt\_tape\ c8\ kk\ (mt\_pos\ c'\ kk - 1) = fst\ (buf5\ kk)$
⟨*proof*⟩

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 $c'\ c4\ c5\ c6\ c7\ c8 :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
and $buf5 :: nat \Rightarrow ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$
and $dest5 :: nat \Rightarrow ae\_dest$
assumes *tape_corr*: $\forall k{<}k\_tm\ M.\ ae\_tape\_correspondence\ (le\_tm\ M)$
$(mt\_tape\ cM\ k)\ (mt\_tape\ c'\ k)$
and *c_ge_1*: $1 \leq card\ (UNIV :: 'c\ set)$
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 *c5_pos_for_pos1_dest_left*:
$\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk = 1 \Longrightarrow dest5\ kk = AE\_Left$
$\Longrightarrow mt\_pos\ c5\ kk = 2$
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 *cM_k_zero*: $\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_tape\ cM\_k\ kk\ 0 = le\_tm\ M$
and *m_tape_off_window_le0*:
$\bigwedge kk\ p.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk = 0$
$\Longrightarrow p > card\ (UNIV :: 'c\ set)$
$\Longrightarrow mt\_tape\ cM\_k\ kk\ p = mt\_tape\ cM\ kk\ p$
and *m_tape_off_window_le1*:
$\bigwedge kk\ p.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk = 1$

$\Longrightarrow p > 2 * card\ (UNIV :: 'c\ set)$
$\Longrightarrow mt\_tape\ cM\_k\ kk\ p = mt\_tape\ cM\ kk\ p$
and *m_tape_off_window_steady*:
$\bigwedge kk\ p.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk \geq 2$
$\Longrightarrow p < (mt\_pos\ c'\ kk - 2) * card\ (UNIV :: 'c\ set) + 1$
$\vee\ p \geq (mt\_pos\ c'\ kk - 2) * card\ (UNIV :: 'c\ set) + 1 + 3 * card\ (UNIV :: 'c\ set)$
$\Longrightarrow mt\_tape\ cM\_k\ kk\ p = mt\_tape\ cM\ kk\ p$
and *c5_pos_for_pos0*:
$\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk = 0 \Longrightarrow mt\_pos\ c5\ kk = 0$
and *c5_pos_for_pos_ge2*:
$\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk \geq 2 \Longrightarrow dest5\ kk \neq AE\_Left$
$\Longrightarrow mt\_pos\ c5\ kk = mt\_pos\ c'\ kk - 1$
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* ≥ *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* ≥ *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*
$\Longrightarrow$ *mt_tape cM_k kk* (*Suc* (*c_idx* (*i* :: *′c*)))
= (*fst* (*snd* (*buf5 kk*))) *i*
and *tape_cM_k_at_two_for_pos1*:
$\bigwedge$*kk i*. *kk* < *k_tm M* $\Longrightarrow$ *mt_pos c′ kk* = *1*
$\Longrightarrow$ *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*:
$\bigwedge$*kk i*. *kk* < *k_tm M* $\Longrightarrow$ *mt_pos c′ kk* ≥ *2*
$\Longrightarrow$ *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*:
$\bigwedge$*kk i*. *kk* < *k_tm M* $\Longrightarrow$ *mt_pos c′ kk* ≥ *2*
$\Longrightarrow$ *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*:
$\bigwedge$*kk i*. *kk* < *k_tm M* $\Longrightarrow$ *mt_pos c′ kk* ≥ *2*
$\Longrightarrow$ *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*⟩

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*⟩

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*). Uniformly *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 $c'\ c4\ c5\ c6\ c7\ c8 :: ('c :: enum \Rightarrow 'a, 'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
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 *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*: $\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c4\ kk = mt\_pos\ c'\ kk$
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*: $(c6, c7) \in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
and *c7_state*: $mt\_state\ c7 = (q6, ofs6, buf6, dest6, SS8)$
and *buf6_eq_buf5*: $buf6 = buf5$
and *dest6_eq_dest5*: $dest6 = dest5$
and *step8_sub*: $(c7, c8) \in mttm\_step\ (ae\_delta\_ss8\_ss1\ M)$
and *step8*: $(c7, c8) \in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
and *c5_pos_for_pos_ge2*:
$\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk \geq 2 \Longrightarrow dest5\ kk \neq AE\_Left \Longrightarrow mt\_pos\ c5\ kk = mt\_pos\ c'\ kk - 1$
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_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_pos_ge2*:
$\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk \geq 2 \Longrightarrow mt\_pos\ c7\ kk = (if\ dest5\ kk = AE\_Left\ then\ mt\_pos\ c'\ kk - 1\ else\ mt\_pos\ c'\ kk + 1)$

shows $\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk \geq 2$
$\Longrightarrow mt\_tape\ c8\ kk\ (mt\_pos\ c'\ kk + 1) = snd\ (snd\ (buf5\ kk))$
⟨*proof*⟩

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 $c'\ c4\ c5\ c6\ c7\ c8 :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
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 *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 *step8_sub*: $(c7, c8) \in mttm\_step\ (ae\_delta\_ss8\_ss1\ M)$
and *c5_pos_for_pos_ge2*:
$\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk \geq 2 \Longrightarrow dest5\ kk \neq AE\_Left$
$\Longrightarrow mt\_pos\ c5\ kk = mt\_pos\ c'\ kk - 1$
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 *left_not_le_c′_steady*:
$\forall k.\ mt\_pos\ c'\ k \geq 2$
$\longrightarrow mt\_tape\ c'\ k\ (mt\_pos\ c'\ k - 1) \neq LE\_block\ (le\_tm\ M)$
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_pos_ge2*:
$\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk \geq 2$
$\Longrightarrow mt\_pos\ c7\ kk = (\mathit{if}\ dest5\ kk = AE\_Left$
$\mathit{then}\ mt\_pos\ c'\ kk - 1$
$\mathit{else}\ mt\_pos\ c'\ kk + 1)$

shows $\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk \geq 2$
$\Longrightarrow mt\_pos\ c8\ kk = (\mathit{case}\ dest5\ kk\ \mathit{of}$
$AE\_Left\ \Rightarrow mt\_pos\ c'\ kk - 1$
$\mid AE\_Home\ \Rightarrow mt\_pos\ c'\ kk$
$\mid AE\_Right \Rightarrow mt\_pos\ c'\ kk + 1)$

⟨*proof*⟩

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 $c'\ c8 :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
and $cM\_k :: ('a, 'q)\ mt\_config$
and $q6 :: 'q$
and $ofs5\ ofs6 :: nat \Rightarrow 'c$
and $buf5\ buf6 :: nat \Rightarrow ('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a)$
and $dest5 :: nat \Rightarrow ae\_dest$
assumes *ofs6_eq_ofs5*: $ofs6 = ofs5$
and *c8_state*: $mt\_state\ c8 = (q6, \mathit{if}\ q6 \in \{t\_tm\ M, r\_tm\ M\}$
$\mathit{then}\ init\_stage\ (le\_tm\ M)$
$\mathit{else}\ (ofs6, buf6, init\_dest, SS1))$
and *cM_k_window_general*:
$\forall kk{<}k\_tm\ M.\ ae\_window\_invariant\_general$
$(mt\_tape\ cM\_k\ kk)\ (mt\_pos\ cM\_k\ kk)$
$(dest5\ kk, ofs5\ kk)\ (buf5\ kk)$
$(mt\_pos\ c'\ kk)\ (le\_tm\ M)$
and *c8_pos_for_pos0*:
$\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk = 0$
$\Longrightarrow mt\_pos\ c8\ kk = (\mathit{if}\ dest5\ kk = AE\_Right\ \mathit{then}\ 1\ \mathit{else}\ 0)$
and *c8_pos_for_pos_ge2*:
$\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk \geq 2$
$\Longrightarrow mt\_pos\ c8\ kk = (\mathit{case}\ dest5\ kk\ \mathit{of}$
$AE\_Left\ \Rightarrow mt\_pos\ c'\ kk - 1$
$\mid AE\_Home\ \Rightarrow mt\_pos\ c'\ kk$
$\mid AE\_Right \Rightarrow mt\_pos\ c'\ kk + 1)$
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⟩
```

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 ⇒ '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 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: (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 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*⟩

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 $c'\ c4\ c5\ c6\ c7\ c8 :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
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 *step6*: $(c5, c6) \in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
and *step7_sub*: $(c6, c7) \in mttm\_step\ (ae\_delta\_ss7\_ss8\ M)$

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*:
∀ k<k_tm M. mt_pos c' k = 0 ⟶ fst (snd (buf5 k)) = LE_block (le_tm M)
and *c6_pos_for_pos0*:
⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 0 ⟹ mt_pos c6 kk = 1
and *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)
shows *⋀kk. kk < k_tm M ⟹ mt_pos c' kk = 0*
⟹ mt_tape c8 kk 1 = snd (snd (buf5 kk))
⟨*proof*⟩

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*: $\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c4\ kk = mt\_pos\ c'\ kk$
and *step6_sub*: $(c5,\ c6) \in mttm\_step\ (ae\_delta\_ss6\_ss7\ M)$
and *c5_pos_for_pos_ge2*:
$\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk \geq 2 \Longrightarrow dest5\ kk \neq AE\_Left$
$\Longrightarrow mt\_pos\ c5\ kk = mt\_pos\ c'\ kk - 1$
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 *left_not_le_c'_steady*:
$\forall k.\ mt\_pos\ c'\ k \geq 2$
$\longrightarrow mt\_tape\ c'\ k\ (mt\_pos\ c'\ k - 1) \neq LE\_block\ (le\_tm\ M)$
shows $\bigwedge kk.\ kk < k\_tm\ M \Longrightarrow mt\_pos\ c'\ kk \geq 2 \Longrightarrow mt\_pos\ c6\ kk = mt\_pos\ c'\ kk$
⟨*proof*⟩

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 $c'\ c1\ c2\ c3\ c4 :: ('c :: enum \Rightarrow\ 'a,$
$'q \times ('a,\ 'c)\ ae\_stage)\ mt\_config$
assumes *vM*: *valid_mttm M*
and *le_anchor*:
$\forall k{<}k\_tm\ M.\ mt\_tape\ c'\ k\ 0 = LE\_block\ (le\_tm\ M)$
and *no_le_per_tape*:
$\forall k.\ (mt\_pos\ c'\ k \geq 2$
$\longrightarrow (\forall i.\ i < 3 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt\_tape\ cM\ k$
$((mt\_pos\ c'\ k - 2) * card\ (UNIV :: 'c\ set) + 1 + i)$
$\neq le\_tm\ M))$

$\wedge$ ($mt\_pos\ c'\ k = 1$
$\longrightarrow$ ($\forall\, i.\ i < 2 * card\ (UNIV :: 'c\ set)$
$\longrightarrow\ mt\_tape\ cM\ k\ (Suc\ i) \neq le\_tm\ M$))
$\wedge$ ($mt\_pos\ c'\ k = 0$
$\longrightarrow$ ($\forall\, i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow\ mt\_tape\ cM\ k\ (Suc\ i) \neq le\_tm\ M$))
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*⟩

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*) ∈ *mttm_step* (*delta_tm M*) ^^ k
and *end_or_halt*:
$k =$ *card* (*UNIV* :: 'c *set*)
∨ *mt_state cM_k* ∈ {*t_tm M*, *r_tm M*}
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*

$((mt\_pos\ c'\ k - 2) * card\ (UNIV :: 'c\ set) + 1 + i)$
$\neq le\_tm\ M))$
$\wedge\ (mt\_pos\ c'\ k = 1$
$\longrightarrow\ (\forall\, i.\ i < 2 * card\ (UNIV :: 'c\ set)$
$\longrightarrow\ mt\_tape\ cM\ k\ (Suc\ i) \neq le\_tm\ M))$
$\wedge\ (mt\_pos\ c'\ k = 0$
$\longrightarrow\ (\forall\, i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow\ mt\_tape\ cM\ k\ (Suc\ i) \neq le\_tm\ M))$

obtains *c'' c1 c2 c3 c4 c5 c6 c7 ofs_f buf_f dest_f* **where**
$(c', c'') \in mttm\_step\ (alphabet\_enlarge\_delta\ M)\ \widehat{\ }\ 8$
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*⟩

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 $c' :: ('c :: enum \Rightarrow 'a,$

```
            '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⟩
```

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⟩
```

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*) $^{\frown\frown}$ n
    and *case mt_state* c' *of* (qM', _, _, _, *idx*) $\Rightarrow$
          idx = *SS1* $\vee$ qM' = *r_tm M*
⟨*proof*⟩

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' :: $('c :: enum \Rightarrow 'a,$
          $'q \times ('a, 'c)$ *ae_stage*) *mt_config*
    and n :: *nat*
  assumes *vM*: *valid_mttm M*
      and *lu*: *le_unique M*
      and *w_sub*: *set w* $\subseteq$ *Sigma_tm M*
      and *le_neq_bl*: *le_tm M* $\neq$ *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 *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))
⟨proof⟩
```

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⟩
```

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 $T :: nat \Rightarrow nat$
assumes *wf*: *well_formed_mttm M*
shows $\forall w.\ set\ w \subseteq Sigma\_tm\ M$
$\longrightarrow$ *accepts_in_time_mttm M w (T (length w))*
$\longrightarrow$ *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*⟩

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 $T :: nat \Rightarrow nat$
assumes *wf*: *well_formed_mttm M*
obtains $\alpha\ f :: nat$
where $\forall w.\ set\ w \subseteq Sigma\_tm\ M$
$\longrightarrow$ *accepts_in_time_mttm M w (T (length w))*
$\longrightarrow$ *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*⟩

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* × ('*a*, ('*c* :: *enum*)) *ae_stage*,
          '*c* ⇒ '*a*) *mttm*)
⟨*proof*⟩

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⟩
```

Auxiliary induction for the reverse lift: given an n-step *m_step_buffered* chain (with $n \leq c$) 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 ($c \leq bp\_linear + n$, $bp\_linear < 2c + n$).

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)
```

$$\neq le\_tm\ M))$$

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 $\exists cM\_n.$
$(cM,\ cM\_n) \in mttm\_step\ (delta\_tm\ M)\ ^{\frown\frown}\ n$
$\wedge\ mt\_state\ cM\_n = qM\_n$
$\wedge\ (\forall k{<}k\_tm\ M.\ ae\_window\_invariant\_general$
$(mt\_tape\ cM\_n\ k)\ (mt\_pos\ cM\_n\ k)$
$(end\_pos\_n\ k)\ (buf\_n\ k)\ (pos\ k)\ (le\_tm\ M))$
$\wedge\ (\forall k.\ (pos\ k = 0$
$\longrightarrow\ (\forall i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow mt\_tape\ cM\_n\ k\ (Suc\ i) \neq le\_tm\ M))$
$\wedge\ (pos\ k = 1$
$\longrightarrow\ (\forall i.\ i < 2 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt\_tape\ cM\_n\ k\ (Suc\ i) \neq le\_tm\ M))$
$\wedge\ (pos\ k \geq 2$
$\longrightarrow\ (\forall i.\ i < 3 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt\_tape\ cM\_n\ k$
$((pos\ k - 2) * card\ (UNIV :: 'c\ set)$
$+ 1 + i)$
$\neq le\_tm\ M)))$
$\wedge\ (\forall k{<}k\_tm\ M.\ pos\ k = 0$
$\longrightarrow\ fst\ (buf\_n\ k) \neq LE\_block\ (le\_tm\ M))$
$\wedge\ (\forall k.\ card\ (UNIV :: 'c\ set) \leq bp\_linear\ (end\_pos\_n\ k) + n$
$\wedge\ bp\_linear\ (end\_pos\_n\ k)$
$< 2 * card\ (UNIV :: 'c\ set) + n)$

⟨*proof*⟩

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 $ofs :: nat \Rightarrow 'c :: enum$
and *buf_full* ::
$nat \Rightarrow (('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a))$
and $pos :: nat \Rightarrow nat$
and $qM\_end :: 'q$
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*⟩

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*⟩

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 $c'\ c3 :: ('c :: enum \Rightarrow 'a,$
      $'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
    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 *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*: $(c', c3)$
        $\in mttm\_step\ (alphabet\_enlarge\_delta\ M)\ ^{\frown\frown}\ 3$
  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*⟩

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: $(cM, cM\_new) \in mttm\_step\ (delta\_tm$

M) $\frown\frown$ n for some $n \leq |c|$, 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⟩
```

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*⟩

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*⟩

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*⟩

(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*⟩

(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*⟩

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*)
  ⟨*proof*⟩

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*)))
  ⟨*proof*⟩

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*)))
  ⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

General window inversion (all *pos*). The home and right blocks are determined by the tape window at every *pos*. For *pos* ≠ *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* ⟹ *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*)))

$\wedge$ *snd* (*snd b1*) = *snd* (*snd b2*)
$\wedge$ (*pos* $\neq$ *0* $\longrightarrow$ *bp1* = *bp2* $\wedge$ *b1* = *b2*)
⟨*proof*⟩

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* $\neq$ *0* $\longrightarrow$ *bp1 k* = *bp2 k* ∧ *b1 k* = *b2 k*)
⟨*proof*⟩

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 $a = \lambda k.\ ts\ k\ (n\ k)$, 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*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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*⟩

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* ⇒ ((′*c* :: *enum* ⇒ ′*a*) × (′*c* ⇒ ′*a*) × (′*c* ⇒ ′*a*))
⇒ ((′*c* ⇒ ′*a*) × (′*c* ⇒ ′*a*) × (′*c* ⇒ ′*a*)) ⇒ *bool* **where**

```
  ae_blk_agree le b1 b2 ⟷
    fst (snd b1) = fst (snd b2)
    ∧ snd (snd b1) = snd (snd b2)
    ∧ (fst (snd b1) ≠ LE_block le ⟶ fst b1 = fst b2)

lemma ae_blk_agree_action_eq:
  assumes ae_blk_agree le b1 b2
  shows b1 = b2 ∨ (fst (snd b1) = LE_block le ∧ fst (snd b2) = LE_block le
                   ∧ snd (snd b1) = snd (snd b2))
⟨proof⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

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
  ⟨proof⟩

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
  ⟨proof⟩
```

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:  ∀ 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⟩

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⟩

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⟩
```

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*: $q \notin \{t\_tm\ M,\ r\_tm\ M\}$
and *s1*: $(c1,\ c1') \in mttm\_step\ (ae\_delta\_ss8\_ss1\ M)$
and *s2*: $(c2,\ c2') \in mttm\_step\ (ae\_delta\_ss8\_ss1\ M)$
shows $mt\_tape\ c1' = mt\_tape\ c2' \wedge mt\_pos\ c1' = mt\_pos\ c2'$
$\wedge\ mt\_state\ c1' = (q,\ ofs1,\ buf1,\ init\_dest,\ SS1)$
$\wedge\ mt\_state\ c2' = (q,\ ofs2,\ buf2,\ init\_dest,\ SS1)$
⟨*proof*⟩

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 $c5a\ c5b\ ca\ cb :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a,\ 'c)\ ae\_stage)\ mt\_config$
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*: $\forall k.\ ae\_blk\_agree\ (le\_tm\ M)\ (buf1\ k)\ (buf2\ k)$
and *qnh*: $q \notin \{t\_tm\ M,\ r\_tm\ M\}$
and *a56*: $(c5a,\ d6a) \in mttm\_step\ (ae\_delta\_ss5\_ss6\ M)$
and *a67*: $(d6a,\ d7a) \in mttm\_step\ (ae\_delta\_ss6\_ss7\ M)$
and *a78*: $(d7a,\ d8a) \in mttm\_step\ (ae\_delta\_ss7\_ss8\ M)$
and *a81*: $(d8a,\ ca) \in mttm\_step\ (ae\_delta\_ss8\_ss1\ M)$
and *b56*: $(c5b,\ d6b) \in mttm\_step\ (ae\_delta\_ss5\_ss6\ M)$
and *b67*: $(d6b,\ d7b) \in mttm\_step\ (ae\_delta\_ss6\_ss7\ M)$
and *b78*: $(d7b,\ d8b) \in mttm\_step\ (ae\_delta\_ss7\_ss8\ M)$
and *b81*: $(d8b,\ cb) \in mttm\_step\ (ae\_delta\_ss8\_ss1\ M)$
shows $mt\_tape\ ca = mt\_tape\ cb \wedge mt\_pos\ ca = mt\_pos\ cb$
$\wedge\ mt\_state\ ca = (q,\ ofs1,\ buf1,\ init\_dest,\ SS1)$
$\wedge\ mt\_state\ cb = (q,\ ofs2,\ buf2,\ init\_dest,\ SS1)$
⟨*proof*⟩

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*:
$ae\_decode\_pos\ s\ i = 0 \Longrightarrow s = 0$

⟨*proof*⟩

lemma *ae_decode_pos_zero_eq*: *ae_decode_pos 0 i = 0*
⟨*proof*⟩

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*⟩

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*⟩

Halt branch of the closing SS8→SS1 step. When *q* ∈ {*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 $c1\ c2\ c1'\ c2' :: ('c :: enum \Rightarrow 'a,$
    $'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
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*: $\forall k.\ ae\_blk\_agree\ (le\_tm\ M)\ (buf1\ k)\ (buf2\ k)$
  and *qh*: $q \in \{t\_tm\ M, r\_tm\ M\}$
  and *s1*: $(c1, c1') \in mttm\_step\ (ae\_delta\_ss8\_ss1\ M)$
  and *s2*: $(c2, c2') \in mttm\_step\ (ae\_delta\_ss8\_ss1\ M)$
shows $mt\_tape\ c1' = mt\_tape\ c2' \wedge mt\_pos\ c1' = mt\_pos\ c2'$
    $\wedge\ mt\_state\ c1' = (q, init\_stage\ (le\_tm\ M))$
    $\wedge\ mt\_state\ c2' = (q, init\_stage\ (le\_tm\ M))$
⟨*proof*⟩

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 $c5a\ c5b\ c8a\ c8b :: ('c :: enum \Rightarrow 'a,$
    $'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
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*: $\forall k.\ ae\_blk\_agree\ (le\_tm\ M)\ (buf1\ k)\ (buf2\ k)$
  and *a56*: $(c5a, d6a) \in mttm\_step\ (ae\_delta\_ss5\_ss6\ M)$
  and *a67*: $(d6a, d7a) \in mttm\_step\ (ae\_delta\_ss6\_ss7\ M)$
  and *a78*: $(d7a, c8a) \in mttm\_step\ (ae\_delta\_ss7\_ss8\ M)$
  and *b56*: $(c5b, d6b) \in mttm\_step\ (ae\_delta\_ss5\_ss6\ M)$
  and *b67*: $(d6b, d7b) \in mttm\_step\ (ae\_delta\_ss6\_ss7\ M)$
  and *b78*: $(d7b, c8b) \in mttm\_step\ (ae\_delta\_ss7\_ss8\ M)$
shows $mt\_tape\ c8a = mt\_tape\ c8b \wedge mt\_pos\ c8a = mt\_pos\ c8b$
    $\wedge\ mt\_state\ c8a = (q, ofs1, buf1, dest, SS8)$
    $\wedge\ mt\_state\ c8b = (q, ofs2, buf2, dest, SS8)$
⟨*proof*⟩

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 $c''\ c''' :: ('c :: enum \Rightarrow 'a, 'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
assumes *sim'*: *ae_simulates M cM c'''*
and *cM_halt*: $mt\_state\ cM \in \{t\_tm\ M, r\_tm\ M\}$
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*⟩

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 $pos \neq 0$ 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 $tM :: nat \Rightarrow nat \Rightarrow 'a$ **and** $nM\ pos :: nat \Rightarrow nat$
and $bp1\ bp2 :: nat \Rightarrow ('c :: enum)\ bp$
and $b1\ b2 :: nat \Rightarrow (('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a))$
and $le :: 'a$ **and** $K :: nat$
assumes $h1$: $\forall k{<}K.$ *ae_window_invariant_general* $(tM\ k)\ (nM\ k)$ $(bp1\ k)\ (b1\ k)\ (pos\ k)\ le$
and $h2$: $\forall k{<}K.$ *ae_window_invariant_general* $(tM\ k)\ (nM\ k)$ $(bp2\ k)\ (b2\ k)\ (pos\ k)\ le$
shows $\forall k{<}K.\ nM\ k = 0 \vee snd\ (bp1\ k) = snd\ (bp2\ k)$
⟨*proof*⟩

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 $c3\ c4\ c5\ c6\ c7\ c'' :: ('c :: enum \Rightarrow 'a,$

```
                                '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⟩
```

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''* $:: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)$ *ae_stage*$)$ *mt_config*
and *c4f c5f c6f c7f c'''* $:: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)$ *ae_stage*$)$ *mt_config*
and *ofs_r ofs_f* $:: nat \Rightarrow 'c$
and *buf_r buf_f* $:: nat \Rightarrow (('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a))$
and *dest_r dest_f* $:: nat \Rightarrow$ *ae_dest* **and** *pos* $:: nat \Rightarrow nat$
assumes *halt*: *mt_state cM_new* $\in \{$*t_tm M*, *r_tm M*$\}$
and *run45*: $(c3, c4) \in$ *mttm_step* (*ae_delta_ss4_ss5 M*)
and *fwd45*: $(c3, c4f) \in$ *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*: $\forall k{<}$*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*) (*pos k*) (*le_tm M*)
and *fwd_window*: $\forall k{<}$*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*: $\forall k. \neg k < k\_tm\ M \longrightarrow ae\_blk\_agree\ (le\_tm\ M)\ (buf\_r\ k)\ (buf\_f\ k)$
and *dest_tail*: $\forall k. \neg k < k\_tm\ M \longrightarrow dest\_r\ k = dest\_f\ k$
and *run5*: $(c4, c5) \in$ *mttm_step* (*ae_delta_ss5_ss6 M*)
and *run6*: $(c5, c6) \in$ *mttm_step* (*ae_delta_ss6_ss7 M*)
and *run7*: $(c6, c7) \in$ *mttm_step* (*ae_delta_ss7_ss8 M*)
and *run8*: $(c7, c'') \in$ *mttm_step* (*ae_delta_ss8_ss1 M*)
and *fwd5*: $(c4f, c5f) \in$ *mttm_step* (*ae_delta_ss5_ss6 M*)
and *fwd6*: $(c5f, c6f) \in$ *mttm_step* (*ae_delta_ss6_ss7 M*)
and *fwd7*: $(c6f, c7f) \in$ *mttm_step* (*ae_delta_ss7_ss8 M*)
and *fwd8*: $(c7f, c''') \in$ *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*⟩

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''* $:: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)$ *ae_stage*) *mt_config*
and *c4f c5f c6f c7f c'''* $:: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)$ *ae_stage*) *mt_config*
and *ofs_r ofs_f* $::$ *nat* $\Rightarrow 'c$
and *buf_r buf_f* $::$ *nat* $\Rightarrow (('c \Rightarrow 'a) \times ('c \Rightarrow 'a) \times ('c \Rightarrow 'a))$
and *dest_r dest_f* $::$ *nat* $\Rightarrow$ *ae_dest* **and** *pos* $::$ *nat* $\Rightarrow$ *nat*
assumes *q_in_Q*: *mt_state cM_new* $\in$ *Q_tm M*
and *run45*: $(c3, c4) \in$ *mttm_step* (*ae_delta_ss4_ss5 M*)
and *fwd45*: $(c3, c4f) \in$ *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*: $\forall k{<}$*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*) (*pos k*) (*le_tm M*)
and *fwd_window*: $\forall k{<}$*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*: $\forall k. \neg k < k\_tm\ M \longrightarrow$ *ae_blk_agree* (*le_tm M*) (*buf_r k*) (*buf_f k*)
and *dest_tail*: $\forall k. \neg k < k\_tm\ M \longrightarrow$ *dest_r k* $=$ *dest_f k*
and *run5*: $(c4, c5) \in$ *mttm_step* (*ae_delta_ss5_ss6 M*)
and *run6*: $(c5, c6) \in$ *mttm_step* (*ae_delta_ss6_ss7 M*)
and *run7*: $(c6, c7) \in$ *mttm_step* (*ae_delta_ss7_ss8 M*)
and *run8*: $(c7, c'') \in$ *mttm_step* (*ae_delta_ss8_ss1 M*)
and *fwd5*: $(c4f, c5f) \in$ *mttm_step* (*ae_delta_ss5_ss6 M*)
and *fwd6*: $(c5f, c6f) \in$ *mttm_step* (*ae_delta_ss6_ss7 M*)
and *fwd7*: $(c6f, c7f) \in$ *mttm_step* (*ae_delta_ss7_ss8 M*)
and *fwd8*: $(c7f, c''') \in$ *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*⟩

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*: $\bigwedge s\ a\ s1\ a1\ d1\ s2\ a2\ d2.$
$(s, a, s1, a1, d1) \in R \Longrightarrow (s, a, s2, a2, d2) \in R$

$\Longrightarrow s1 = s2 \wedge a1 = a2 \wedge d1 = d2$
and *h1*: $(c, c1) \in mttm\_step\ R$
and *h2*: $(c, c2) \in mttm\_step\ R$
shows $c1 = c2$
⟨*proof*⟩

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*: $qM' \neq r\_tm\ M$
and *no_le_per_tape*:
$\forall k.\ (mt\_pos\ c'\ k \geq 2$
$\longrightarrow (\forall i.\ i < 3 * card\ (UNIV :: 'c\ set)$
$\longrightarrow mt\_tape\ cM\ k$
$((mt\_pos\ c'\ k - 2) * card\ (UNIV :: 'c\ set)$
$+ 1 + i)$
$\neq le\_tm\ M))$

$\wedge$ (*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*))
$\wedge$ (*mt_pos* c' *k* = *0*
  $\longrightarrow$ ($\forall$ *i*. *i* < *card* (*UNIV* :: *'c set*)
    $\longrightarrow$ *mt_tape cM k* (*Suc i*) $\neq$ *le_tm M*))

and *run*: (*c'*, *c''*)
  $\in$ *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*) $\in$ *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 (*cM*, *cM_new*) $\in$ *mttm_step* (*delta_tm M*) ^^ *n*
and *n* $\leq$ *card* (*UNIV* :: *'c set*)
and *n* = *card* (*UNIV* :: *'c set*)
  $\vee$ *mt_state cM_new* $\in$ {*t_tm M*, *r_tm M*}
and *fst* (*mt_state c''*) = *mt_state cM_new*
and *mt_state cM_new* $\notin$ {*t_tm M*, *r_tm M*}
  $\longrightarrow$ *ae_inv_ss1 M c''*
and *mt_state cM_new* $\in$ {*t_tm M*, *r_tm M*}
  $\longrightarrow$ *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 $\forall$ *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 *ae_buffer_in_gamma_block M c4*
and *mt_state cM_new* $\in$ *Q_tm M*

⟨*proof*⟩

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'') $\in R^{\frown\frown}8$, produces the corresponding substrate trace (*cM*, *cM_new*) $\in$ *step*$\hat{}$*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
```

$\longrightarrow (\forall\, i.\ i < card\ (UNIV :: 'c\ set)$
$\longrightarrow mt\_tape\ cM\ k\ (Suc\ i) \neq le\_tm\ M))$
and *run*: $(c', c'') \in mttm\_step\ (alphabet\_enlarge\_delta\ M)\ ^^\ 8$
obtains *cM_new n* **where**
$(cM, cM\_new) \in mttm\_step\ (delta\_tm\ M)\ ^^\ n$
and $n \leq card\ (UNIV :: 'c\ set)$
and $n = card\ (UNIV :: 'c\ set)$
$\vee\ mt\_state\ cM\_new \in \{t\_tm\ M, r\_tm\ M\}$
and *ae_simulates M cM_new c''*
and *ae_buffer_in_gamma_block M c''*
and $\forall\, kk{<}k\_tm\ M.\ mt\_tape\ c''\ kk\ 0 = LE\_block\ (le\_tm\ M)$
⟨*proof*⟩

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*⟩

lemma *ae_step_from_SS2_lands_at_SS3*:
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')))) = SS2*
and *step*: $(c', c'') \in mttm\_step\ (alphabet\_enlarge\_delta\ M)$
shows *snd (snd (snd (snd (mt_state c'')))) = SS3*
⟨*proof*⟩

lemma *ae_step_from_SS3_lands_at_SS4*:
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')))) = SS3*

and *step*: $(c', c'') \in$ *mttm_step* (*alphabet_enlarge_delta* M)
shows *snd* (*snd* (*snd* (*snd* (*mt_state* c'')))) = *SS4*
⟨*proof*⟩

lemma *ae_step_from_SS4_lands_at_SS5*:
fixes M :: $('q, 'a)$ *mttm*
and c' c'' :: $('c$ :: *enum* ⇒ $'a,$
$'q \times ('a, 'c)$ *ae_stage*) *mt_config*
assumes *idx*: *snd* (*snd* (*snd* (*snd* (*mt_state* c')))) = *SS4*
and *step*: $(c', c'') \in$ *mttm_step* (*alphabet_enlarge_delta* M)
shows *snd* (*snd* (*snd* (*snd* (*mt_state* c'')))) = *SS5*
⟨*proof*⟩

lemma *ae_step_from_SS5_lands_at_SS6*:
fixes M :: $('q, 'a)$ *mttm*
and c' c'' :: $('c$ :: *enum* ⇒ $'a,$
$'q \times ('a, 'c)$ *ae_stage*) *mt_config*
assumes *idx*: *snd* (*snd* (*snd* (*snd* (*mt_state* c')))) = *SS5*
and *step*: $(c', c'') \in$ *mttm_step* (*alphabet_enlarge_delta* M)
shows *snd* (*snd* (*snd* (*snd* (*mt_state* c'')))) = *SS6*
⟨*proof*⟩

lemma *ae_step_from_SS6_lands_at_SS7*:
fixes M :: $('q, 'a)$ *mttm*
and c' c'' :: $('c$ :: *enum* ⇒ $'a,$
$'q \times ('a, 'c)$ *ae_stage*) *mt_config*
assumes *idx*: *snd* (*snd* (*snd* (*snd* (*mt_state* c')))) = *SS6*
and *step*: $(c', c'') \in$ *mttm_step* (*alphabet_enlarge_delta* M)
shows *snd* (*snd* (*snd* (*snd* (*mt_state* c'')))) = *SS7*
⟨*proof*⟩

lemma *ae_step_from_SS7_lands_at_SS8*:
fixes M :: $('q, 'a)$ *mttm*
and c' c'' :: $('c$ :: *enum* ⇒ $'a,$
$'q \times ('a, 'c)$ *ae_stage*) *mt_config*
assumes *idx*: *snd* (*snd* (*snd* (*snd* (*mt_state* c')))) = *SS7*
and *step*: $(c', c'') \in$ *mttm_step* (*alphabet_enlarge_delta* M)
shows *snd* (*snd* (*snd* (*snd* (*mt_state* c'')))) = *SS8*
⟨*proof*⟩

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* ∨ *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* ⇒ $'a,$

```
                'q × ('a, 'c) ae_stage) mt_config
  assumes idx: snd (snd (snd (snd (mt_state c')))) = SS8
      and step: (c', c'') ∈ mttm_step (alphabet_enlarge_delta M)
  shows snd (snd (snd (snd (mt_state c'')))) ∈ {SS1, VFwd}
⟨proof⟩
```

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⟩
```

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)
```

$$
\begin{aligned}
&\quad * \mathit{card}\ (\mathit{UNIV} :: 'c\ \mathit{set}) \\
&\quad + 1 + i) \\
&\neq \mathit{le\_tm}\ M)) \\
\wedge\ &(\mathit{mt\_pos}\ c'\ k = 1 \\
&\longrightarrow (\forall i.\ i < 2 * \mathit{card}\ (\mathit{UNIV} :: 'c\ \mathit{set}) \\
&\longrightarrow \mathit{mt\_tape}\ cM\ k\ (\mathit{Suc}\ i) \neq \mathit{le\_tm}\ M)) \\
\wedge\ &(\mathit{mt\_pos}\ c'\ k = 0 \\
&\longrightarrow (\forall i.\ i < \mathit{card}\ (\mathit{UNIV} :: 'c\ \mathit{set}) \\
&\longrightarrow \mathit{mt\_tape}\ cM\ k\ (\mathit{Suc}\ i) \neq \mathit{le\_tm}\ M))
\end{aligned}
$$

⟨*proof*⟩

Halt-branch contradiction: when *mt_state* c' is at a halt-coerced state (qM', *init_stage le*) with $qM' \in \{t\_tm\ M, r\_tm\ M\}$, 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 $M :: ('q, 'a)\ \mathit{mttm}$
and $qM' :: 'q$
and $c'\ c'' :: (('c :: \mathit{enum}) \Rightarrow 'a,$
$'q \times ('a, 'c)\ \mathit{ae\_stage})\ \mathit{mt\_config}$
assumes *vM*: *valid_mttm M*
and *qM_in_tr*: $qM' \in \{t\_tm\ M, r\_tm\ M\}$
and *state_c'*: *mt_state* $c' = (qM', \mathit{init\_stage}\ (\mathit{le\_tm}\ M))$
and *step*: (c', c'')
$\in \mathit{mttm\_step}\ (\mathit{alphabet\_enlarge\_delta}\ M)$
shows *False*
⟨*proof*⟩

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 $c'\ c\_acc :: ('c :: enum \Rightarrow 'a,$
$'q \times ('a, 'c)\ ae\_stage)\ mt\_config$
and $m :: nat$
assumes *vM*: $valid\_mttm\ M$
and *lu*: $le\_unique\ M$
and *w_sub*: $set\ w \subseteq Sigma\_tm\ M$
and *le_neq_bl*: $le\_tm\ M \neq bl\_tm\ M$
and *reach_M*: $(init\_config\_mttm\ M\ w, cM)$
$\in (mttm\_step\ (delta\_tm\ M))^*$
and *trace*: (c', c_acc)
$\in mttm\_step\ (alphabet\_enlarge\_delta\ M)\ ^^\ m$
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*: $\forall kk{<}k\_tm\ M.\ mt\_tape\ c'\ kk\ 0 = LE\_block\ (le\_tm\ M)$
shows $\exists cM\_final.\ (cM, cM\_final) \in (mttm\_step\ (delta\_tm\ M))^*$
$\wedge\ mt\_state\ cM\_final = t\_tm\ M$
⟨*proof*⟩

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 bicon-

ditional.

theorem *alphabet_enlarge_language*:
  fixes $M :: ('q, 'a)\ mttm$
  assumes *wf*: *well_formed_mttm M*
  shows $\forall w.\ set\ w \subseteq Sigma\_tm\ M$
    $\longrightarrow (encode\_input\ (bl\_tm\ M)\ w \in Lang\_mttm$
      $(alphabet\_enlarge\ M$
        $:: ('q \times ('a, ('c :: enum))\ ae\_stage,$
          $'c \Rightarrow 'a)\ mttm))$
      $= (w \in Lang\_mttm\ M)$
⟨*proof*⟩

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 $T :: nat \Rightarrow nat$
  assumes *wf*: *well_formed_mttm M*
  obtains $\alpha\ f :: nat$
  where $\forall w.\ set\ w \subseteq Sigma\_tm\ M$
    $\longrightarrow (encode\_input\ (bl\_tm\ M)\ w \in Lang\_mttm$
      $(alphabet\_enlarge\ M$
        $:: ('q \times ('a, ('c :: enum))\ ae\_stage,$
          $'c \Rightarrow 'a)\ mttm))$
      $= (w \in Lang\_mttm\ M)$
    and $\forall w.\ set\ w \subseteq Sigma\_tm\ M$
    $\longrightarrow accepts\_in\_time\_mttm\ M\ w\ (T\ (length\ w))$
    $\longrightarrow accepts\_in\_time\_mttm$
      $(alphabet\_enlarge\ M$
        $:: ('q \times ('a, 'c)\ ae\_stage, 'c \Rightarrow 'a)\ mttm)$
      $(encode\_input\ (bl\_tm\ M)\ w)$
      $(\alpha * ((length\ w + card\ (UNIV :: 'c\ set) - 1)$
          $div\ card\ (UNIV :: 'c\ set))$
       $+ 8 * ((T\ (length\ w) + card\ (UNIV :: 'c\ set) - 1)$

div card (*UNIV* :: *'c set*))
+ *f*)
⟨*proof*⟩

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* ⇒ *'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 Σ*u* =
    { (*qf*, *sym*, *qf*, *sym*, λ*t*. *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* ' Σ*u* ∪ *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))
  ⟨proof⟩
termination
  ⟨proof⟩
```

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⟩
```

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 $(Suc\ i) * c \leq length\ w$
shows $wrap\_enc\ pack\ c\ w\ !\ i = pack\ (take\ c\ (drop\ (i * c)\ w))$
⟨*proof*⟩

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 $length\ (wrap\_enc\ pack\ c\ w) =$
$length\ w\ div\ c + (if\ length\ w\ mod\ c = 0\ then\ 0\ else\ 1)$
⟨*proof*⟩

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 $length\ w\ mod\ c \neq 0$
shows $wrap\_enc\ pack\ c\ w\ !\ (length\ w\ div\ c) =$
$pack\ (drop\ ((length\ w\ div\ c) * c)\ w)$
⟨*proof*⟩

Derived safety lemmas for entries of *wrap_enc*: under the pack contracts ($pack\ (take\ c\ (drop\ (i * c)\ w)) \in$ 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* $i * c \leq$ *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*) $\subseteq$ *Sigma_tm M* (from *wrap_enc pack c w* $\in$ *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*)) $\in$ *set* (*wrap_enc pack c w*)
⟨*proof*⟩

lemma *wrap_enc_in_Gamma*:
  assumes *c_pos*: $0 < c$
    and *pack_contract_in*:
      $\bigwedge i.\ i * c <$ *length w* $\Longrightarrow$ *pack* (*take c* (*drop* ($i * c$) *w*)) $\in$ Γ_*tm M*
    and *j_lt*: $j <$ *length* (*wrap_enc pack c w*)
  shows *wrap_enc pack c w* ! $j \in$ Γ_*tm M*
⟨*proof*⟩

lemma *wrap_enc_ne_le*:
  assumes *c_pos*: $0 < c$
    and *pack_contract_ne_le*:
      $\bigwedge i.\ i * c <$ *length w* $\Longrightarrow$ *pack* (*take c* (*drop* ($i * c$) *w*)) $\neq$ *le_tm M*
    and *j_lt*: $j <$ *length* (*wrap_enc pack c w*)
  shows *wrap_enc pack c w* ! $j \neq$ *le_tm M*
⟨*proof*⟩

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*) $\subseteq$ *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*:
$\bigwedge i.\ i * c < length\ w \Longrightarrow pack\ (take\ c\ (drop\ (i * c)\ w)) \in Sigma\_tm\ M$
and *j_lt*: $j < length\ (wrap\_enc\ pack\ c\ w)$
shows $wrap\_enc\ pack\ c\ w\ !\ j \in Sigma\_tm\ M$
⟨*proof*⟩

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 $1 \ldots k$, 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 $wrap\_tau :: nat \Rightarrow nat$ **where**
$wrap\_tau\ p =$
$(case\ p\ of\ 0 \Rightarrow Suc\ 0 \mid Suc\ 0 \Rightarrow 0 \mid Suc\ (Suc\ k) \Rightarrow Suc\ (Suc\ k))$

lemma *wrap_tau_0* [*simp*]: $wrap\_tau\ 0 = Suc\ 0$
⟨*proof*⟩

lemma *wrap_tau_1* [*simp*]: $wrap\_tau\ (Suc\ 0) = 0$
⟨*proof*⟩

lemma *wrap_tau_ge2* [*simp*]: $wrap\_tau\ (Suc\ (Suc\ k)) = Suc\ (Suc\ k)$
⟨*proof*⟩

lemma *wrap_tau_invol* [*simp*]: *wrap_tau* (*wrap_tau p*) = *p*
⟨*proof*⟩

lemma *wrap_tau_ge2_id*: *2* ≤ *p* ⟹ *wrap_tau p* = *p*
⟨*proof*⟩

lemma *wrap_tau_lt2*: *wrap_tau p* < *2* ⟷ *p* < *2*
⟨*proof*⟩

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*,
λ*t*. *if t* < *K then Enc* (*le_tm M*) *else Enc* (*bl_tm M*),
W_Buf [],
λ*t*. *if t* < *K then Enc* (*le_tm M*) *else Enc* (*bl_tm M*),
λ*t*. *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 Σ*u* =
{ (*W_Buf ws*, *sym*, *W_Buf* (*ws* @ [*a*]), *sym*,
λ*t*. *case t of 0* ⇒ *dir.R* | _ ⇒ *dir.N*)
| *ws a sym*.
length ws + *1* < *c* ∧ *a* ∈ Σ*u* ∧ *set ws* ⊆ Σ*u*
∧ *sym 0* = *Raw a*
∧ (∀ *j*≥*K*. *sym j* = *Enc* (*bl_tm M*))

$\wedge$ *sym* $\in$ *UNIV* $\rightarrow$ *Raw* ‘ $\Sigma u \cup$ *Enc* ‘ Γ*_tm M* }

definition *wrap_buf_close_delta_gen* ::
nat $\Rightarrow$ (*'q*, *'b*) *mttm* $\Rightarrow$ (*'a list* $\Rightarrow$ *'b*) $\Rightarrow$ *nat* $\Rightarrow$ *'a set*
$\Rightarrow$ ((*'q*, *'a*, *'b*) *wrap_state*
$\times$ (*nat* $\Rightarrow$ (*'a*, *'b*) *wrap_alphabet*)
$\times$ (*'q*, *'a*, *'b*) *wrap_state*
$\times$ (*nat* $\Rightarrow$ (*'a*, *'b*) *wrap_alphabet*)
$\times$ (*nat* $\Rightarrow$ *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* $\Rightarrow$ (*if k = 0 then Enc* (*pack* (*ws* @ [*a*]))
else sym (*Suc k*))
| *0* $\Rightarrow$ *sym 0*),
λt. *case t of 0* $\Rightarrow$ *dir.R*
| *Suc k* $\Rightarrow$ (*if k = 0 then dir.R else dir.N*))
| *ws a sym*.
length ws + *1* = *c* $\wedge$ *a* $\in$ Σu $\wedge$ *set ws* $\subseteq$ Σu
$\wedge$ *sym 0* = *Raw a*
$\wedge$ *sym* (*Suc 0*) $\neq$ *Enc* (*le_tm M*)
$\wedge$ *pack* (*ws* @ [*a*]) $\in$ *Sigma_tm M*
$\wedge$ ($\forall$ *j*$\geq$*K*. *sym j* = *Enc* (*bl_tm M*))
$\wedge$ *sym* $\in$ *UNIV* $\rightarrow$ *Raw* ‘ $\Sigma u \cup$ *Enc* ‘ Γ*_tm M* }

definition *wrap_buf_empty_end_delta_gen* ::
nat $\Rightarrow$ (*'q*, *'b*) *mttm* $\Rightarrow$ *'a set*
$\Rightarrow$ ((*'q*, *'a*, *'b*) *wrap_state*
$\times$ (*nat* $\Rightarrow$ (*'a*, *'b*) *wrap_alphabet*)
$\times$ (*'q*, *'a*, *'b*) *wrap_state*
$\times$ (*nat* $\Rightarrow$ (*'a*, *'b*) *wrap_alphabet*)
$\times$ (*nat* $\Rightarrow$ *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*)
$\wedge$ ($\forall$ *j*$\geq$*K*. *sym j* = *Enc* (*bl_tm M*))
$\wedge$ *sym* $\in$ *UNIV* $\rightarrow$ *Raw* ‘ $\Sigma u \cup$ *Enc* ‘ Γ*_tm M* }

definition *wrap_buf_nonempty_end_delta_gen* ::
nat $\Rightarrow$ (*'q*, *'b*) *mttm* $\Rightarrow$ (*'a list* $\Rightarrow$ *'b*) $\Rightarrow$ *nat* $\Rightarrow$ *'a set*
$\Rightarrow$ ((*'q*, *'a*, *'b*) *wrap_state*
$\times$ (*nat* $\Rightarrow$ (*'a*, *'b*) *wrap_alphabet*)
$\times$ (*'q*, *'a*, *'b*) *wrap_state*
$\times$ (*nat* $\Rightarrow$ (*'a*, *'b*) *wrap_alphabet*)
$\times$ (*nat* $\Rightarrow$ *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.
```

$(q, \sigma, q', \sigma', d) \in delta\_tm\ M$
$\wedge\ (\forall p.\ sym\ p = Enc\ (\sigma\ (wrap\_tau\ p)))$
$\wedge\ sym \in UNIV \to Raw\ `\ \Sigma u \cup Enc\ `\ \Gamma\_tm\ M\ \}$

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* $\rightarrow$ *W_Disp* $\rightarrow$ *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 ⇒ ('b, 'q) mt_config
   ⇒ (('a, 'b) wrap_alphabet, ('q, 'a, 'b) wrap_state) mt_config
where
  lift_M_config M cM =
     Config_M (W_Run (mt_state cM))
       (λp n. if p < k_tm M then Enc (mt_tape cM (wrap_tau p) n)
              else Enc (bl_tm M))
       (λp. 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 ⇒ ('q, 'b) mttm ⇒ ('a list ⇒ 'b) ⇒ nat ⇒ 'a list ⇒ nat ⇒ nat
   ⇒ (('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
       (λt n. if t < K
              then (case t of
```

```
      0 ⇒ (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 ⇒ (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 ⇒ m0
    | Suc k ⇒ (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* $\wedge$ $n \leq$ *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)))
  ⟨proof⟩
```

```
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⟩
```

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 ⇒ '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*⟩

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
```

⟨*proof*⟩

Obligation 1 — well-formedness of the faithful k-tape plant-*le* wrap at tape count *k_tm M*. Hypotheses: $2 \leq$ *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* $\neq$ *bl_tm M*
    and *finSu*: *finite* Σu
    and *c_pos*: $0 < c$
    and *k2*: $2 \leq$ *k_tm M*
  shows *valid_mttm* (*encoding_wrap M pack c* Σu)
⟨*proof*⟩

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)
  ⟨*proof*⟩

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* $\subseteq \Sigma u$
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*⟩

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*⟩

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*⟩

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*⟩

The dispatch step: one *wrap_disp_delta_gen* transition lifts *W_Disp*

$\rightarrow$ *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* $\leq$ *k_tm M*
shows (*mid_plant_disp_config M pack c w*, *post_plant_dispatch_config M pack c w*)
$\in$ *mttm_step* (*wrap_delta M pack c* Σu)
⟨*proof*⟩

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* $\subseteq$ 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 (*post_encoder_config_gen* (*k_tm M*) *M pack c w*,
post_plant_dispatch_config M pack c w)
$\in$ (*mttm_step* (*wrap_delta M pack c* Σu))
$\widehat{\ }\widehat{\ }$ (*Suc* (*length* (*wrap_enc pack c w*)) + *2*)
⟨*proof*⟩

lemma *plant_reset_dispatch_rtrancl*:
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 (*post_encoder_config_gen* (*k_tm M*) *M pack c w*,
post_plant_dispatch_config M pack c w)
$\in$ (*mttm_step* (*wrap_delta M pack c* Σu))*
⟨*proof*⟩

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*⟩

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*⟩

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 $(W\_Run\ q, sym, qs', a', dir) \in wrap\_delta\ M\ pack\ c\ \Sigma u$
  shows $(W\_Run\ q, sym, qs', a', dir) \in wrap\_run\_delta\ M\ \Sigma u$
  ⟨*proof*⟩

10.3 Reverse simulation: a wrap-step projects to an M-step

lemma *run_step_reverse*:
  assumes *vM*: *valid_mttm M*
      and *k2*: $2 \le k\_tm\ M$
      and *val_cM*: *valid_config_mttm M cM*
      and *step_wrap*:
        $(lift\_M\_config\ M\ cM, C') \in mttm\_step\ (wrap\_delta\ M\ pack\ c\ \Sigma u)$
    shows $\exists\, cM'.\ C' = lift\_M\_config\ M\ cM'$
          $\wedge\ (cM, cM') \in mttm\_step\ (delta\_tm\ M)$
⟨*proof*⟩

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*: $2 \le k\_tm\ M$
      and *val_cM*: *valid_config_mttm M cM*
      and *steps_M*: $(cM, cM') \in (mttm\_step\ (delta\_tm\ M))^*$
    shows $(lift\_M\_config\ M\ cM, lift\_M\_config\ M\ cM')$
        $\in (mttm\_step\ (wrap\_delta\ M\ pack\ c\ \Sigma u))^*$
⟨*proof*⟩

lemma *run_steps_forward_relpow*:
  assumes *vM*: *valid_mttm M*
      and *k2*: $2 \le k\_tm\ M$
      and *val_cM*: *valid_config_mttm M cM*
      and *steps_M*: $(cM, cM') \in (mttm\_step\ (delta\_tm\ M))\ ^{\frown\frown}\ n$
    shows $(lift\_M\_config\ M\ cM, lift\_M\_config\ M\ cM')$

$\in (mttm\_step\ (wrap\_delta\ M\ pack\ c\ \Sigma u))\ ^{\frown\frown}\ n$
⟨*proof*⟩

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* Σ*u*))*
shows ∃ *cM′*. *C′* = *lift_M_config M cM′*
∧ (*cM*, *cM′*) ∈ (*mttm_step* (*delta_tm M*))*
∧ *valid_config_mttm M cM′*
⟨*proof*⟩

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* Σ*u*) (*map Raw w*),
mid_encoder_config_gen (*k_tm M*) *M pack c w 0*)
∈ *mttm_step* (*wrap_delta M pack c* Σ*u*)
⟨*proof*⟩

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* $\subseteq \Sigma u$
    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*))
    $\in$ *mttm_step* (*wrap_delta M pack c* Σu)
⟨*proof*⟩

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* $\subseteq \Sigma u$
    and (*Suc i*) $* c \le$ *length w*
    and *pack* (*take c* (*drop* ($i * c$) *w*)) $\in$ *Sigma_tm M*
    and *bl_tm M* $\neq$ *le_tm M*
    and *k2*: $2 \le$ *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*))
    $\in$ *mttm_step* (*wrap_delta M pack c* Σu)
⟨*proof*⟩

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* $\subseteq \Sigma u$
    and *block_fits*: (*Suc i*) $* c \le$ *length w*
    and *pack_in_Sigma*: *pack* (*take c* (*drop* ($i * c$) *w*)) $\in$ *Sigma_tm M*

and *bl_ne_le*: $bl\_tm\ M \neq le\_tm\ M$
and *k2*: $2 \leq k\_tm\ M$
shows $(mid\_encoder\_config\_gen\ (k\_tm\ M)\ M\ pack\ c\ w\ i,$
$mid\_encoder\_config\_gen\ (k\_tm\ M)\ M\ pack\ c\ w\ (Suc\ i))$
$\in (mttm\_step\ (wrap\_delta\ M\ pack\ c\ \Sigma u))^*$
⟨*proof*⟩

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*: $set\ w \subseteq \Sigma u$
and *block_fits*: $(Suc\ i) * c \leq length\ w$
and *pack_in_Sigma*: $pack\ (take\ c\ (drop\ (i * c)\ w)) \in Sigma\_tm\ M$
and *bl_ne_le*: $bl\_tm\ M \neq le\_tm\ M$
and *k2*: $2 \leq k\_tm\ M$
shows $(mid\_encoder\_config\_gen\ (k\_tm\ M)\ M\ pack\ c\ w\ i,$
$mid\_encoder\_config\_gen\ (k\_tm\ M)\ M\ pack\ c\ w\ (Suc\ i))$
$\in (mttm\_step\ (wrap\_delta\ M\ pack\ c\ \Sigma u))\ ^\frown\!\!^\frown\ c$
⟨*proof*⟩

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*: $set\ w \subseteq \Sigma u$
and *bl_ne_le*: $bl\_tm\ M \neq le\_tm\ M$
and *k2*: $2 \leq k\_tm\ M$
and *pack_contract_in_Sigma*:
$\bigwedge i.\ i * c < length\ w \Longrightarrow$
$pack\ (take\ c\ (drop\ (i * c)\ w)) \in Sigma\_tm\ M$
and *i_le*: $i \leq length\ w\ div\ c$
shows $(mid\_encoder\_config\_gen\ (k\_tm\ M)\ M\ pack\ c\ w\ 0,$
$mid\_encoder\_config\_gen\ (k\_tm\ M)\ M\ pack\ c\ w\ i)$
$\in (mttm\_step\ (wrap\_delta\ M\ pack\ c\ \Sigma u))\ ^\frown\!\!^\frown\ (c * i)$
⟨*proof*⟩

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* Σ*u*)
⟨*proof*⟩

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* ⊆ Σ*u*
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* Σ*u*))∗
⟨*proof*⟩

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* ⊆ Σ*u*
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* Σ*u*)) ^^ *Suc* (*length w mod c*)
⟨*proof*⟩

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* ⊆ Σ*u*

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* Σ*u*))*
⟨*proof*⟩

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* ⊆ 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*))*
⟨*proof*⟩

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*⟩

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* $\rightarrow$ *W_Buf* $\rightarrow$ *W_Reset* $\rightarrow$ *W_Disp* $\rightarrow$ *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* $\Rightarrow$ $('a, 'b)$ *wrap_alphabet*
    and a' :: *nat* $\Rightarrow$ $('a, 'b)$ *wrap_alphabet*
    and *dir* :: *nat* $\Rightarrow$ *dir*
  assumes *tuple_in*: $(q,$ *sym*, *W_Run* $q',$ $a',$ *dir*$) \in$ *wrap_delta M pack c* Σu
      and *non_run_src*: $\forall\, q_M.\ q \neq$ *W_Run* q_M
    shows $q =$ *W_Disp* $\wedge$ $q' =$ *s_tm M*
        $\wedge\ (q,$ *sym*, *W_Run* $q',$ $a',$ *dir*$) \in$ *wrap_disp_delta_gen* $(k\_tm\ M)\ M\ \Sigma u$
  ⟨*proof*⟩

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*: $(q,\ a,\ p_1,\ b_1,\ d_1) \in$ *wrap_delta M pack c* Σu
      and *t2*: $(q,\ a,\ p_2,\ b_2,\ d_2) \in$ *wrap_delta M pack c* Σu
      and *non_run*: $\forall\, q_M.\ q \neq$ *W_Run* q_M
    shows $(p_1,\ b_1,\ d_1) = (p_2,\ b_2,\ d_2)$
⟨*proof*⟩

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*: $(C,\ C') \in (mttm\_step\ (wrap\_delta\ M\ pack\ c\ \Sigma u))^*$

and *src*: *mt_state C = W_Run q*
shows ∃ *q'*. *mt_state C' = W_Run q'*
⟨*proof*⟩

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* Σ*u*
shows (*W_Disp*, *inp*, *qs'*, *a'*, *dir*) ∈ *wrap_disp_delta_gen* (*k_tm M*) *M* Σ*u*
⟨*proof*⟩

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* Σ*u*)
and *src*: *mt_state C = W_Disp*
shows *mt_state C' = W_Run* (*s_tm M*)
⟨*proof*⟩

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'*) ∈ (*mttm_step* (*wrap_delta M pack c* Σ*u*)) ^^ *n*
and *src*: *mt_state C = W_Disp*
and *n_pos*: *0 < n*
shows ∃ *q*. *mt_state C' = W_Run q*
⟨*proof*⟩

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* Σ*u*) (*map Raw w*), *C_acc*)
∈ (*mttm_step* (*wrap_delta M pack c* Σ*u*))*
and *accept*: ∃ *q*. *mt_state C_acc = W_Run q*
shows ∃ *C_pre C_post*.
(*init_config_mttm* (*encoding_wrap M pack c* Σ*u*) (*map Raw w*), *C_pre*)
∈ (*mttm_step* (*wrap_delta M pack c* Σ*u*))*
∧ *mt_state C_pre = W_Disp*
∧ (*C_pre*, *C_post*) ∈ *mttm_step* (*wrap_delta M pack c* Σ*u*)
∧ *mt_state C_post = W_Run* (*s_tm M*)

$\wedge$ (C_post, C_acc) $\in$ ($mttm\_step$ ($wrap\_delta$ M $pack$ c Σu))*
⟨*proof*⟩

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, $C1$) $\in$ $mttm\_step$ ($wrap\_delta$ M $pack$ c Σu)
and *step2*: (C, $C2$) $\in$ $mttm\_step$ ($wrap\_delta$ M $pack$ c Σu)
and *non_run*: $\forall$ q_M. mt_state $C \neq W\_Run$ q_M
shows $C1 = C2$
⟨*proof*⟩

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*: ($C0$, $C1$) $\in$ ($mttm\_step$ ($wrap\_delta$ M $pack$ c Σu)) $^{\frown\frown}$ n
and *path2*: ($C0$, $C2$) $\in$ ($mttm\_step$ ($wrap\_delta$ M $pack$ c Σu)) $^{\frown\frown}$ n
and *non_run_C1*: $\forall$ q_M. mt_state $C1 \neq W\_Run$ q_M
and *non_run_C2*: $\forall$ q_M. mt_state $C2 \neq W\_Run$ q_M
shows $C1 = C2$
⟨*proof*⟩

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 :: $nat \Rightarrow$ ('a, 'b) $wrap\_alphabet$
and a' :: $nat \Rightarrow$ ('a, 'b) $wrap\_alphabet$
and dir' :: $nat \Rightarrow dir$
assumes *tuple_in*: (W_Buf ws, inp, qs', a', dir') $\in$ $wrap\_delta$ M $pack$ c Σu
and *ws_full*: $length$ $ws + 1 = c$
and *inp_W_User_Raw*: inp $0 = Raw$ a
shows (W_Buf ws, inp, qs', a', dir') $\in$ $wrap\_buf\_close\_delta\_gen$ (k_tm M) M $pack$ c Σu
⟨*proof*⟩

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 :: $nat \Rightarrow$ ('a, 'b) $wrap\_alphabet$
and a' :: $nat \Rightarrow$ ('a, 'b) $wrap\_alphabet$

and $dir' :: nat \Rightarrow dir$
assumes *tuple_in*: $(W\_Buf\ ws,\ inp,\ qs',\ a',\ dir') \in wrap\_delta\ M\ pack\ c\ \Sigma u$
and *ws_partial*: $length\ ws < c$
and *ws_nonempty*: $ws \neq []$
and *inp_W_User_bl*: $inp\ 0 = Enc\ (bl\_tm\ M)$
shows $(W\_Buf\ ws,\ inp,\ qs',\ a',\ dir') \in wrap\_buf\_nonempty\_end\_delta\_gen\ (k\_tm\ M)\ M\ pack\ c\ \Sigma u$
⟨*proof*⟩

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 $Suc\ i * c \leq length\ w$, any firing wrap-step forces both the pack contract $pack\ (take\ c\ (drop\ (i * c)\ w)) \in Sigma\_tm\ M$ 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 $2 \leq k\_tm\ M$).

lemma *wrap_close_step_forces_pack*:
fixes $M :: ('q, 'b)\ mttm$
assumes *vM*: *valid_mttm M*
and *c_pos*: $0 < c$
and *w_alpha*: $set\ w \subseteq \Sigma u$
and *block_full*: $(Suc\ i) * c \leq length\ w$
and *bl_ne_le*: $bl\_tm\ M \neq le\_tm\ M$
and *k2*: $2 \leq k\_tm\ M$
and *step*: $(mid\_buf\_extending\_config\_gen\ (k\_tm\ M)\ M\ pack\ c\ w\ i\ (c - 1), C') \in mttm\_step\ (wrap\_delta\ M\ pack\ c\ \Sigma u)$
shows $pack\ (take\ c\ (drop\ (i * c)\ w)) \in Sigma\_tm\ M \wedge C' = mid\_encoder\_config\_gen\ (k\_tm\ M)\ M\ pack\ c\ w\ (Suc\ i)$
⟨*proof*⟩

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 $2 \leq k\_tm\ M$ 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*: $set\ w \subseteq \Sigma u$
and *bl_ne_le*: $bl\_tm\ M \neq le\_tm\ M$
and *k2*: $2 \leq k\_tm\ M$
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*: $(mid\_buf\_extending\_config\_gen\ (k\_tm\ M)\ M\ pack\ c\ w\ i\ (length$

$w - i * c)$, $C')$
$\in$ *mttm_step* (*wrap_delta M pack c* Σu)
shows *pack* (*take c* (*drop* ($i * c$) *w*)) $\in$ *Sigma_tm M*
$\wedge$ $C' =$ *post_encoder_config_gen* (*k_tm M*) *M pack c w*
⟨*proof*⟩
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* $\leq$ *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* $\subseteq$ Σu
and *bl_ne_le*: *bl_tm M* $\neq$ *le_tm M*
and *k2*: *2* $\leq$ *k_tm M*
and *i_full*: (*Suc i*) $*$ *c* $\leq$ *length w*
and *pack_ih*: $\bigwedge j.\ j < i \Longrightarrow$ *pack* (*take c* (*drop* ($j * c$) *w*)) $\in$ *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*))
$\in$ (*mttm_step* (*wrap_delta M pack c* Σu))*
⟨*proof*⟩

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* $\subseteq$ Σu
and *bl_ne_le*: *bl_tm M* $\neq$ *le_tm M*
and *k2*: *2* $\leq$ *k_tm M*
and *i_lt*: *i* $*$ *c* $<$ *length w*
and *i_partial*: *length w* $<$ (*Suc i*) $*$ *c*
and *pack_ih*: $\bigwedge j.\ j < i \Longrightarrow$ *pack* (*take c* (*drop* (*j* $*$ *c*) *w*)) $\in$ *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*))
$\in$ (*mttm_step* (*wrap_delta M pack c* Σu))*
⟨*proof*⟩

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* $\neq$ *le_tm M*
and *w_alpha*: *set w* $\subseteq$ Σu
and *k2*: *2* $\leq$ *k_tm M*
and *trace*: (*init_config_mttm* (*encoding_wrap M pack c* Σu) (*map Raw w*), *C_acc*)
$\in$ (*mttm_step* (*wrap_delta M pack c* Σu))*
and *accept*: $\exists q.$ *mt_state C_acc* $=$ *W_Run q*
and *i_full*: (*Suc i*) $*$ *c* $\leq$ *length w*
shows *pack* (*take c* (*drop* (*i* $*$ *c*) *w*)) $\in$ *Sigma_tm M*
⟨*proof*⟩

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* $\neq$ *le_tm M*
and *w_alpha*: *set w* $\subseteq$ Σu
and *k2*: *2* $\leq$ *k_tm M*
and *trace*: (*init_config_mttm* (*encoding_wrap M pack c* Σu) (*map Raw w*), *C_acc*)
$\in$ (*mttm_step* (*wrap_delta M pack c* Σu))*
and *accept*: $\exists q.$ *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*: $\bigwedge j.\ j < i \Longrightarrow pack\ (take\ c\ (drop\ (j * c)\ w)) \in Sigma\_tm\ M$
shows $pack\ (take\ c\ (drop\ (i * c)\ w)) \in Sigma\_tm\ M$
⟨*proof*⟩

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*: $bl\_tm\ M \neq le\_tm\ M$
and *w_alpha*: $set\ w \subseteq \Sigma u$
and *k2*: $2 \leq k\_tm\ M$
and *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_state*: $\exists q.\ mt\_state\ C\_acc = W\_Run\ q$
shows $(post\_plant\_dispatch\_config\ M\ pack\ c\ w,\ C\_acc)$
$\in (mttm\_step\ (wrap\_delta\ M\ pack\ c\ \Sigma u))^*$
$\wedge\ (\forall i.\ i * c < length\ w$
$\longrightarrow pack\ (take\ c\ (drop\ (i * c)\ w)) \in Sigma\_tm\ M)$
⟨*proof*⟩
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: $Lang\_user\_wrap\ (encoding\_wrap\ M\ pack\ c\ \Sigma u) = \{w.\ set\ w \subseteq \Sigma u \wedge wrap\_enc\ pack\ c\ w \in Lang\_mttm\ M\}$. 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* $\neq$ *le_tm M*
    and *k2*: $2 \leq$ *k_tm M*
   shows $\{w.$ *set w* $\subseteq \Sigma u \wedge$ *wrap_enc pack c w* $\in$ *Lang_mttm M*$\}$
      $\subseteq$ *Lang_user_wrap* (*encoding_wrap M pack c* Σu)
⟨*proof*⟩

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* $\neq$ *le_tm M*
    and *k2*: $2 \leq$ *k_tm M*
  shows *Lang_user_wrap* (*encoding_wrap M pack c* Σu)
      $\subseteq \{w.$ *set w* $\subseteq \Sigma u \wedge$ *wrap_enc pack c w* $\in$ *Lang_mttm M*$\}$
⟨*proof*⟩

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* Σ*u*)
= {*w*. *set w* ⊆ Σ*u* ∧ *wrap_enc pack c w* ∈ *Lang_mttm M*}
⟨*proof*⟩

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* ∗ *length w*), wrapper B *never walks back over the raw input* — the plant-*le* reset rewinds only the storage tape (*length* (*wrap_enc pack c w*) packed cells, at most *length w*), then plants a fresh *le* at the input origin. This is the (*1* + *ε*)·*n* setup the linear-time speed-up [6, Theorem 12.4] needs; the transpose's ~*2*·*n* rewind is what the super-linear [6, Theorem 12.3] tolerates but 12.4 does not.

The per-input precondition *set* (*wrap_enc pack c w*) ⊆ *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* Σ*u*
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* ⊆ Σ*u*
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* Σ*u*) *w*
(*length w* + *length* (*wrap_enc pack c w*) + *5* + *t*)
⟨*proof*⟩

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 (*linear_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⟩
```

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⟩
```

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*⟩

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*⟩

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*

$\longrightarrow$ *N0* $\leq$ *length w*
$\longrightarrow$ *accepts_in_time_mttm M w* (*T* (*length w*))
$\longrightarrow$ *accepts_in_time_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*))
w
(*T* (*length w*) *div q* + *K*)

⟨*proof*⟩

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* $\Rightarrow$ *nat*
and *q* :: *nat*
assumes *wf*: *well_formed_mttm M*
and *det*: *det_mttm M*
and *growth*: $\forall\, d.\ \exists\, N.\ \forall\, n.\ N \leq n \longrightarrow d * n \leq T\ n$
and *q_pos*: *0* < *q*
and *c_large*: *16* $*$ *q* $\leq$ *card* (*UNIV* :: ('*c* :: *enum*) *set*)
and *k2*: *2* $\leq$ *k_tm M*
obtains *K N0* :: *nat*
where *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 *det_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 $\forall w.\ set\ w \subseteq Sigma\_tm\ M$
$\longrightarrow N0 \leq length\ w$
$\longrightarrow accepts\_in\_time\_mttm\ M\ w\ (T\ (length\ w))$
$\longrightarrow accepts\_in\_time\_user\_wrap$
$(encoding\_wrap$
$(alphabet\_enlarge\ M$
$:: ('q \times ('a, 'c)\ ae\_stage, 'c \Rightarrow 'a)\ mttm)$
$(ae\_pack\ (bl\_tm\ M) :: 'a\ list \Rightarrow ('c \Rightarrow 'a))$
$(card\ (UNIV :: 'c\ set))$
$(Sigma\_tm\ M))$
w
$(T\ (length\ w)\ div\ q + K)$
⟨*proof*⟩

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* Σu)
⟨*proof*⟩

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*: $q * (q + 1) * C \leq a$
  shows a *div* $(q + 1) + C \leq a$ *div* q
⟨*proof*⟩

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 $T :: nat \Rightarrow nat$
    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 $q * (3 + 8 * d\_0) \le card\ (UNIV :: 'c\ set)$
      $\longrightarrow$ *time_bounded_conv*
        (*encoding_wrap*
          (*alphabet_enlarge M*
            $:: ('q \times ('a, 'c)\ ae\_stage, 'c \Rightarrow 'a)\ mttm$)
          $(ae\_pack\ (bl\_tm\ M) :: 'a\ list \Rightarrow ('c \Rightarrow 'a))$
          $(card\ (UNIV :: 'c\ set))$
          $(Sigma\_tm\ M)$)
        $(\lambda n.\ n + n\ div\ q + (28 + 8 * d\_0 + 8 * b))$
⟨*proof*⟩

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 $(q + 1) * (3 + 8 * d\_0) \le card\ (UNIV :: 'c\ set)$
$\longrightarrow$ ($\forall v \in Lang\_mttm$
(*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*)).
$q * (q + 1) * (28 + 8 * d\_0 + 8 * b) \le length\ v$
$\longrightarrow$ *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*⟩

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*))
(λ*n*. *n* + *n div q* + (*28* + *8* ∗ *d_0* + *8* ∗ *b*))

⟨*proof*⟩

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⟩
```

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 $\lambda n.\ T\ n\ div\ q$. 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′* (λ*n*. *T n div q*)
⟨*proof*⟩

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′* (λ*n*. *T n div q*)
⟨*proof*⟩

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.