A Nondeterministic Multitape Turing Machine Substrate

Formal Foundations in Isabelle/HOL

András Z. Salamon Michael Wehar

August 26, 2026

Abstract

A self-contained, reusable formalisation of nondeterministic multitape Turing machines in Isabelle/HOL, depending only on `HOL-Library`.

A machine's transition is a *relation*, not a function: it can allow several moves at each step, so a machine may follow many different computations on one input — the machine is *nondeterministic*. The machine accepts when at least one computation reaches an accepting state. Deterministic machines, which have a single move everywhere, are the special case. The number of tapes is stored as ordinary data rather than fixed in the machine's type, so a single theorem can speak about machines with any number of tapes. Hierarchy theorems need exactly this.

The entry also justifies its least conventional modelling choice. Each tape has a marker at its left end, and the model lets a machine move that marker to the right, discarding a stretch of tape it can never return to. We prove that this changes nothing the machine computes: the run before the move reaches an accepting state exactly when the matching run after it does. This turns an assumption that such models usually leave implicit into a proved fact, and it gives a reusable tool for arguments that relocate or trim a tape.

This entry is the shared foundation for several companion developments: alphabet enlargement (`Multitape_Alphabet_Enlargement`) and alphabet reduction (`Multitape_Alphabet_Reduction`), with tape reduction and a universal machine to follow. It is also self-contained and usable on its own.

The model began as an adaptation of Dalvit and Thiemann's multitape Turing machine formalisation [2] (the AFP entry `Multitape_To_Singletape_TM`), and extends it in two ways: the number of tapes is part of a machine's data rather than fixed in its type; and a machine is itself an ordinary value that theorems can range over, rather than a fixed context — a locale — inside which each theorem is proved. Like their formalisation, a machine's transition is a relation, so machines may be nondeterministic; that sets both developments apart

from the Archive's other Turing-machine entries, whose transitions are functions.

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 The substrate

This session defines a reusable model of nondeterministic multitape Turing machines, self-contained and usable on its own. A machine's transition is a *relation*, not a function: a machine may have several possible moves at each step, so nondeterminism is the default, and a deterministic machine is the special case in which every situation offers at most one move. Dalvit and Thiemann model the transition relationally in the same way; two further design choices set this substrate apart from their `Multitape_To_Singletape_TM`, and are fixed here once.

First, the number of tapes is part of a machine's data rather than fixed in its type. A single theorem can then range over machines with any number of tapes.

Second, a machine is itself an ordinary value (a piece of data), so it can be built, compared, listed one after another, and even encoded as a word and fed to another machine as input. A hierarchy theorem needs exactly this last ability: it builds one machine that runs every other machine in turn and disagrees with each. A model that instead fixes a single machine as a background setting, and proves each theorem inside that setting, can still state facts that hold for every machine, but it cannot handle a machine as data in this way.

The model and its predicates live in `Multitape_Substrate.thy`. Companion theories add the origin-float justification (`Multitape_Origin_Float.thy`, described below) and the finite-control and time-bound convention layers the speedup entries reuse (`Multitape_Finite_Control.thy`, `Multitape_Finite_Patch.thy`, `Multitape_Time_Convention.thy`). The companion `Multitape_Alphabet_Enlargement` and `Multitape_Alphabet_Reduction` entries reuse this vocabulary unchanged, and their overviews take it as given.

Two other Turing-machine developments in the Archive of Formal Proofs sit alongside this substrate. Both give a machine a single move at each step. Balbach's `Cook_Levin` [1] formalises multitape machines, with the number of tapes a value as here, for the Cook–Levin theorem and NP-completeness. Xu, Zhang, Urban, and Joosten's `Universal_Turing_Machine` [4] builds a single-tape universal machine for computability, without any timing analysis. In neither may a machine take several moves at once, so nondeterminism has to be added on top rather than being built in. That is the main way this substrate differs from both.

1.2 Machines, configurations, and runs

Machines. A machine is a value of the datatype mttm, the ten-component descriptor MTTM $Q\ \Sigma\ \Gamma\ bl\ le\ \delta\ s\ t\ r\ k$: a state set Q, an input alphabet Σ and a tape alphabet Γ (both ordinary *value* sets over an element type $'a$), a blank bl and left-end marker le (two special *tape* symbols that validity keeps in Γ but not Σ, so both are usable on a tape yet never as input), a transition relation δ, start, accept and reject states s, t, r, and a tape count $k :: \mathbb{N}$. The components are read by the projections `Q_tm`, `Sigma_tm`, `Γ_tm`, `bl_tm`, `le_tm`, `delta_tm`, `s_tm`, `t_tm`, `r_tm`, `k_tm`.

Configurations. A configuration is a value of mt_config, written $\mathrm{Config_M}\ q\ ts\ n$: the current state q, the tape contents ts (a function $\mathbb{N} \to \mathbb{N} \to 'a$ giving the symbol in cell j of tape i as $ts\,i\,j$), and the head positions n (a function $\mathbb{N} \to \mathbb{N}$). Tapes and cells are indexed by $\mathbb{N}$; in a valid run, every tape beyond the machine's count k stays blank. The start configuration is init_config_mttm $M\,w$: state s, tape 0 holding the endmarker le at cell 0 followed by the input word w, every other in-range tape holding just le, and all heads at cell 0.

The step relation. mttm_step δ is the one-step relation on configurations, parametrised by δ alone. From $\mathrm{Config_M}\,q\,ts\,n$, let the *read vector* $\lambda k.\ ts\,k\,(n\,k)$ collect the symbol under each head; if $(q,\ \lambda k.\ ts\,k\,(n\,k),\ q',\ a,\ dir) \in \delta$, the machine may step to $\mathrm{Config_M}\,q'\,ts'\,n'$, where ts' overwrites cell $n\,k$ of each tape k with $a\,k$ and n' moves each head by $dir\,k \in \{L, R, N\}$. Because δ is a relation, several tuples may match one configuration. That is exactly where nondeterminism enters.

Validity, language, and time. The remaining predicates, the ones the transformation theorems carry as hypotheses and conclusions, are built on top:

valid_mttm M *"M is a legal machine".* The structural invariant: Q and Γ finite, $\Sigma \subseteq \Gamma$, the states s, t, r in Q, blank and endmarker tape-but-not-input symbols, $t \neq r$, $k > 0$, δ correctly typed, the *left-endmarker read discipline* (a transition reading le rewrites le and never moves left of it), and every tape beyond k frozen blank. Notably it does *not* constrain how le is *written*; that is the separate le_unique below, so a machine that plants a fresh le stays valid.

well_formed_mttm M *a legal machine that also starts cleanly and keeps its endmarker.* valid_mttm M plus non-degeneracy ($s \neq t$, $s \neq r$, $le \neq bl$) and the endmarker write discipline le_unique M. This is the carrier by which the alphabet-transformation theorems obtain LE-uniqueness of the machine they simulate.

le_unique M *the left endmarker is never freshly planted.* A machine property in the same family as det_mttm: no transition writes le where it was not already reading it ($a'\,j = le \Rightarrow a\,j = le$ over δ). Folded into well_formed_mttm; the origin-floating wrapper for the faithful $(1+\varepsilon)n$ speed-up is the one construction that forgoes it (planting a fresh le), hence is valid_mttm but not well_formed_mttm.

det_mttm M *M is deterministic.* δ is a partial function: each pair of a state and a read vector has at most one successor triple (q', a, dir).

$L(M) = \text{Lang\_mttm}\,M$ *the language M accepts.* The input words w (with set $w \subseteq \Sigma_M$) from which some run reaches an accepting configuration:

$$(\text{init\_config\_mttm}\,M\,w,\ c) \in (\text{mttm\_step}\,\delta_M)^* \quad \text{with} \quad \text{mt\_state}\,c = t.$$

The reflexive-transitive closure makes acceptance *existential* over runs.

accepts_in_time_mttm $M\,w\,m$ *M accepts w within m steps.* The time-bounded variant: some run of length at most m (an iterated step $(\text{mttm\_step}\,\delta_M)^n$ with $n \le m$) reaches the accept state.

These six are exactly the entries the `Multitape_Alphabet_Enlargement` and `Multitape_Alphabet_Reduction` glossaries list under "shared substrate predicates"; their definitions live here.

1.3 Scope and modelling choices

This entry is *foundational infrastructure.* It fixes the machine model and its invariants and proves no complexity results about specific languages; those live in the companion entries. It does prove two facts its companions rely on. The first is the structural left-end-marker fact (origin floating, below). The second is the shared time-bound convention vocabulary the linear-speedup entry consumes: the $\max(n+2, \cdot)$ "convention" predicate and its "eventual" companion, the finite-control `time_cleanup` that upgrades an eventual bound to a convention bound on all inputs, and a minimal counterexample `clean_convention_unattainable` showing that the clean constant-free convention form is *not* attainable in general — a valid machine can accept its language yet overshoot the convention floor on its shortest input, which is why the linear-speedup headline in `Multitape_Alphabet_Enlargement` keeps an explicit residual constant. Three modelling choices are worth knowing before building on it.

- *No separate input tape.* The input word occupies tape 0, an ordinary work tape, so `valid_mttm` requires $\Sigma \subseteq \Gamma$ (every input symbol is also a tape symbol) and a transformation may re-encode the input *in place* rather than copy it to a fresh tape.

- *One left-end marker, two rules.* A single left-end marker *le* obeys a *read* rule: reading *le* rewrites it and never moves off the left edge. This rule is part of `valid_mttm`. The stronger *write* rule le_unique (*le* is never freshly planted) belongs instead to `well_formed_mttm`, so a machine that plants a fresh *le* is still valid. The origin-floating speed-up wrapper is exactly such a machine.

- *One-way-infinite tapes.* Cells are indexed by $\mathbb{N}$ with the endmarker pinning cell 0; every tape beyond the machine's count k is frozen blank.

1.4 Origin floating: the left-endmarker write, validated

Of the choices above, moving the left-end marker is the one with real proof content, and `Multitape_Origin_Float.thy` supplies it. The model lets a machine place a fresh marker at some cell d, which cuts off cells 0 to $d-1$. That prefix is then unreachable, because the read rule forbids the machine from ever stepping left of the marker. Let shift_rel d be the operation that shifts every tape d cells to the right, leaving the state and heads unchanged and moving the symbol at cell p to cell $p+d$. The theory then shows that a run and its shifted copy match step for step, in both directions and at every level:

- single steps — mttm_step_shift_forward and mttm_step_shift_reverse;

- whole runs — mttm_relpow_shift_forward and mttm_relpow_shift_reverse;

- acceptance — shift_reach_state_forward and shift_reach_state_reverse: a run reaches a nominated state iff its shift does.

So placing a fresh marker neither gains nor loses behaviour. Allowing the machine to write this marker is therefore sound, and an assumption that such a model would usually leave implicit becomes a theorem here. The faithful $(1+\varepsilon)n$ speed-up wrapper in `Multitape_Alphabet_Enlargement` is the first user of these lemmas: it moves the marker to reclaim the space taken up by the consumed input. The lemmas themselves are stated only in terms of a single step of the machine, independently of that use, so they serve as a reusable building block for any tape argument that relocates or trims a tape, such as simulating a two-dimensional tape on ordinary tapes.

theory *Multitape_Substrate_Core*
  imports *HOL−Library.FuncSet*
begin

2 Substrate core (Dalvit–Thiemann definition surface)

The *definition surface* of the substrate, deliberately isolated in this theory: the datatypes, selectors, step relation, and the validity / configuration / language definitions — and no proofs. This is the part of the development that corresponds to the Dalvit–Thiemann AFP entry *Multitape_To_Singletape_TM* (its files *Multitape_TM* and *TM_Common*, about 180 lines), kept small so a reader can diff it directly against that source. Every lemma — the *valid_mttm* axiom-extraction toolkit, the validity-preservation and reachability results, and the displacement / left-endmarker tape tools, all *our* additions — lives in the parent theory *Multitape_Substrate*, which imports this one.

The datatypes (*mttm*, *mt_config*, *dir*) and the step relation are adapted from that entry [2]; *valid_mttm* is a direct conjunction of the substrate's structural axioms (no locale wrapping), tightened beyond the AFP entry's δLE axiom by an additional LE-write conjunct (see the validity predicate below).

The number of tapes is a *value*: a machine carries its tape count as a *nat* field k, and head contents are a total function $nat \Rightarrow 'a$ with a *blank tail* beyond k (the support invariant). This departs from the AFP source, where the tape count is trapped in a finite type parameter $'k$; the value form is what makes $\exists\ M$ / $\forall\ M$ over machines of all arities a single HOL proposition.

2.1 TM-direction primitive (forked from AFP *TM_Common*)

Three-valued TM head movement: right (R), left (L), or neutral / stay (N). Used positionally as the fifth component of every transition tuple; functions $nat \Rightarrow dir$ encode the per-tape head movement.

datatype $dir = R \mid L \mid N$

fun $go\_dir :: dir \Rightarrow nat \Rightarrow nat$ **where**
  $go\_dir\ R\ n = Suc\ n$
$\mid$ $go\_dir\ L\ n = n - 1$
$\mid$ $go\_dir\ N\ n = n$

2.2 Multitape-TM datatypes (forked from AFP *Multi-tape_TM*)

Multi-tape Turing-machine descriptor. Ten components: state set Q, input alphabet Σ, tape alphabet Γ, blank symbol, left endmarker, transition relation δ, start state, accept state, reject state, and tape count k. The shape mirrors the AFP source (the *mttm* type abbreviations are a conscious near-copy), except the tape count, trapped in the AFP source's finite type parameter $'k$, is here the value-level *nat* field k; transition tuples index the tapes by *nat*, blank beyond k. Only the *Q_tm* and Γ*_tm* accessors are record-style the others are positional with custom selectors below.

datatype $('q, 'a)$ *mttm* = *MTTM*
  (*Q_tm*: $'q$ *set*) — Q states
  $'a$ *set* — Σ input alphabet
  (Γ*_tm*: $'a$ *set*) — Γ tape alphabet
  $'a$ — blank
  $'a$ — left endmarker
  $('q \times (nat \Rightarrow 'a) \times 'q \times (nat \Rightarrow 'a) \times (nat \Rightarrow dir))$ *set*
  — transitions δ
  $'q$ — start state
  $'q$ — accept state
  $'q$ — reject state
  nat — k tape count

Multitape-TM configuration: state, per-tape contents (each a function $nat \Rightarrow 'a$ of cell positions), per-tape head positions. Tapes and heads are indexed by *nat*; tapes beyond the machine's count k are all-blank in a valid configuration.

datatype $('a, 'q)$ *mt_config* = $Config_M$
  (*mt_state*: $'q$)
  $nat \Rightarrow nat \Rightarrow 'a$
  (*mt_pos*: $nat \Rightarrow nat$)

2.3 Substrate selectors

Positional selectors for the *mttm* datatype's components. The datatype declares record-style accessors for *Q_tm* and Γ*_tm* only; the others (Σ, *blank*, *LE*, δ, s, t, r, k) are positional. These selectors are simple positional pattern matches, named on the $*\_tm$ convention to match *Q_tm* / Γ*_tm*.

fun *bl_tm* :: $('q, 'a)$ *mttm* ⇒ $'a$ **where**
  bl_tm (*MTTM* _ _ _ *bl* _ _ _ _ _ _) = *bl*

fun *le_tm* :: $('q, 'a)$ *mttm* ⇒ $'a$ **where**
  le_tm (*MTTM* _ _ _ _ *le* _ _ _ _ _) = *le*

fun *delta_tm* ::
  $('q, 'a)$ *mttm*

```
    ⇒ ('q × (nat ⇒ 'a) × 'q × (nat ⇒ 'a) × (nat ⇒ dir)) set where
  delta_tm (MTTM _ _ _ _ _ δ _ _ _ _) = δ

fun s_tm :: ('q, 'a) mttm ⇒ 'q where
  s_tm (MTTM _ _ _ _ _ _ s _ _ _) = s

fun t_tm :: ('q, 'a) mttm ⇒ 'q where
  t_tm (MTTM _ _ _ _ _ _ _ t _ _) = t

fun r_tm :: ('q, 'a) mttm ⇒ 'q where
  r_tm (MTTM _ _ _ _ _ _ _ _ r _) = r

fun Sigma_tm :: ('q, 'a) mttm ⇒ 'a set where
  Sigma_tm (MTTM _ Σ _ _ _ _ _ _ _ _) = Σ

fun k_tm :: ('q, 'a) mttm ⇒ nat where
  k_tm (MTTM _ _ _ _ _ _ _ _ _ k) = k
```

Tape-content selector for substrate configurations. The substrate datatype names *mt_state* and *mt_pos* via record-style accessors but leaves the tape function unnamed; we add it positionally for symmetry.

```
fun mt_tape :: ('a, 'q) mt_config ⇒ nat ⇒ nat ⇒ 'a where
  mt_tape (Config_M _ ts _) = ts
```

2.4 Step relation

Inductive characterisation of the multitape-TM step relation, parametrised by the transition relation δ alone. Body verbatim equivalent to the AFP source (only the tape index moves from the type $'k$ to *nat*); functional layer, so downstream Hoare-triple proofs invoke *mttm_step.intros* directly without any locale routing.

```
inductive_set mttm_step ::
  ('q × (nat ⇒ 'a) × 'q × (nat ⇒ 'a) × (nat ⇒ dir)) set
    ⇒ ('a, 'q) mt_config rel
  for δ
where
  step: (q, (λk. ts k (n k)), q', a, dir) ∈ δ ⟹
   (Config_M q ts n,
    Config_M q' (λk. (ts k)(n k := a k)) (λk. go_dir (dir k) (n k)))
    ∈ mttm_step δ
```

2.5 Substrate validity (functional axiom bundle)

Functional bundle of the substrate's structural axioms. This predicate enumerates the standard well-formedness conditions of Hopcroft and Ullman [3, §7.3]: finiteness of Q and Γ, alphabet inclusion, state membership for the start, accept, and reject states, blank / left-endmarker tape-alphabet membership (with Σ disjointness), accept $\neq$ reject, a positive tape count (the

input tape *0* must exist), the range typing of the transition relation, the LE read-discipline, and the *support invariant*.

The LE-discipline kept here is the read direction only: every δ-transition reading the left endmarker on tape j rewrites the LE position with itself and moves only N or R (boundary preservation). The converse write direction — a transition writes the left endmarker on tape j only where it was already reading it — is *not* a validity requirement: it is factored out as the separate predicate *le_unique* (below), which well-formed machines carry but a machine that deliberately plants a fresh *le* — the origin-floating wrapper for the faithful Hopcroft–Ullman speed-up bound [3, Theorem 12.4] — may forgo while staying valid. Keeping the write direction out of validity is exactly what lets that wrapper's output be a legal machine.

The support invariant — every transition is blank on reads and writes and stationary (N) on moves at every tape index $j \geq k$ — is the value-level replacement for the AFP source's finite tape type: it confines a k-tape machine's transitions to tapes $0 \ldots k - 1$ and, with Γ finite, makes δ finite (*valid_mttm_finite_delta*).

fun *valid_mttm* :: $('q, 'a)$ *mttm* $\Rightarrow$ *bool* **where**
valid_mttm (*MTTM* $Q\ \Sigma\ \Gamma$ *bl le* δ *s t r k*) =
(*finite* $Q \wedge$
finite $\Gamma \wedge$
$\Sigma \subseteq \Gamma \wedge$
$s \in Q \wedge$
$t \in Q \wedge$
$r \in Q \wedge$
bl $\in \Gamma \wedge$
bl $\notin \Sigma \wedge$
le $\in \Gamma \wedge$
le $\notin \Sigma \wedge$
$t \neq r \wedge$
$0 < k \wedge$
$\delta \subseteq (Q - \{t, r\}) \times (UNIV \to \Gamma) \times Q \times (UNIV \to \Gamma) \times (UNIV \to UNIV)$
$\wedge$
$(\forall\, q\ a\ q'\ a'\ d\ j.\ (q, a, q', a', d) \in \delta \longrightarrow a\ j = le \longrightarrow$
$a'\ j = le \wedge d\ j \in \{dir.N, dir.R\}) \wedge$
$(\forall\, q\ a\ q'\ a'\ d.\ (q, a, q', a', d) \in \delta \longrightarrow$
$(\forall\, j \geq k.\ a\ j = bl \wedge a'\ j = bl \wedge d\ j = dir.N)))$

The left-endmarker write discipline, as a standalone predicate: a transition writes the left endmarker *le_tm M* on a tape only where it was already reading it — the endmarker is never freshly planted. This was formerly a clause of *valid_mttm*; it is kept separate precisely so a machine that *does* plant a fresh *le* — the origin-floating wrapper for the faithful Theorem 12.4 bound — is still *valid_mttm*, while the alphabet-transformation chain, which needs the property of the machine it simulates, carries it explicitly (folded into *well_formed_mttm* and threaded to the simulation lemmas through

valid_mttm_deltaLE_no_write).

definition *le_unique* :: ('*q*, '*a*) *mttm* ⇒ *bool* **where**
le_unique M =
(∀ *q a q' a' d j*. (*q*, *a*, *q'*, *a'*, *d*) ∈ *delta_tm M* ⟶
a' j = *le_tm M* ⟶ *a j* = *le_tm M*)

Strengthening of *valid_mttm* with the three non-degeneracy conditions used by every linear-speedup-style theorem distinct start / accept / reject states and distinct left-endmarker / blank tape symbols plus the left-endmarker write discipline *le_unique*. Bundles *valid_mttm M*, *s_tm M* ≠ *t_tm M*, *s_tm M* ≠ *r_tm M*, *le_tm M* ≠ *bl_tm M*, and *le_unique M* into a single predicate so downstream statements need only one assumption clause instead of five. Since *le_unique* is no longer implied by *valid_mttm* it is exactly the clause the faithful-12.4 wrapper forgoes it is load-bearing here: *well_formed_mttm* is the alphabet-transformation chain's carrier of LE-uniqueness, threaded to the simulation lemmas that need it.

abbreviation *well_formed_mttm*
:: ('*q*, '*a*) *mttm* ⇒ *bool*
where
well_formed_mttm M ≡
valid_mttm M
∧ *s_tm M* ≠ *t_tm M*
∧ *s_tm M* ≠ *r_tm M*
∧ *le_tm M* ≠ *bl_tm M*
∧ *le_unique M*

2.6 Configuration validity

Functional re-exposition of the substrate's per-configuration validity invariant: state in Q, every tape's contents drawn from Γ, every *active* tape (index $i < k$) carries the left endmarker at position 0, and every *inactive* tape (index $i \geq k$) is all-blank (the configuration-level support invariant inactive tapes carry the blank symbol everywhere, not the left endmarker, since *bl* ≠ *le*).

fun *valid_config_mttm* ::
('*q*, '*a*) *mttm* ⇒ ('*a*, '*q*) *mt_config* ⇒ *bool*
where
valid_config_mttm (*MTTM Q* _ Γ *bl le* _ _ _ _ *k*) (*Config*$_M$ *q ts* _) =
(*q* ∈ *Q*
∧ (∀ *i*. *range* (*ts i*) ⊆ Γ)
∧ (∀ *i*<*k*. *ts i 0* = *le*)
∧ (∀ *i*≥*k*. ∀ *p*. *ts i p* = *bl*))

Functional initial configuration: take M as parameter, pull start state, blank, left endmarker, and tape count positionally. Active tapes (index $i < k$) carry the left endmarker at position 0 and, on the input tape *0*, the input w; inactive tapes (index $i \geq k$) are all-blank.

```
fun init_config_mttm ::
  ('q, 'a) mttm ⇒ 'a list ⇒ ('a, 'q) mt_config
where
  init_config_mttm (MTTM _ _ _ bl le _ s _ _ k) w =
    Config_M s
      (λi n. if i < k
          then (if n = 0 then le
              else if i = 0 ∧ n ≤ length w then w ! (n − 1)
              else bl)
          else bl)
      (λ_. 0)
```

2.7 Language and time-bound predicates

Functional analogues of the AFP's *Lang_mttm* and *det_mttm* top-level wrappers, plus the weak-acceptance predicate *accepts_in_time_mttm*. *Lang_mttm* is the set of inputs over *Sigma_tm M* that drive *M* from its initial configuration to the accept state *t_tm M*. *det_mttm* says the transition relation is single-valued: each source state together with the symbols read admits at most one outcome one next state, one written-symbol tuple, one head-move tuple so a configuration has at most one successor. This is the determinism hypothesis on which the alphabet-enlargement reverse direction turns. Bodies routed through the functional *init_config_mttm* and *mttm_step* rather than through locale-internal definitions. Used by the AE / AR language- and time-preservation theorems.

```
definition Lang_mttm :: ('q, 'a) mttm ⇒ 'a list set where
  Lang_mttm M =
    {w. set w ⊆ Sigma_tm M ∧
      (∃ w' n. (init_config_mttm M w, Config_M (t_tm M) w' n)
          ∈ (mttm_step (delta_tm M))*)}
```

```
definition det_mttm :: ('q, 'a) mttm ⇒ bool where
  det_mttm M =
    (∀ q a p1 b1 d1 p2 b2 d2.
      (q, a, p1, b1, d1) ∈ delta_tm M ⟶
      (q, a, p2, b2, d2) ∈ delta_tm M ⟶
      (p1, b1, d1) = (p2, b2, d2))
```

Weak time-bounded acceptance: M accepts input w in time at most t iff some accepting path of length $n \leq t$ exists from *init_config_mttm* to a config in state *t_tm M*. This matches the existential accepting-path shape that the alphabet-enlargement top-level theorems preserve. Non-accepting paths are not constrained, in contrast to a universal worst-case bound that would constrain every path regardless of acceptance.

Used by *alphabet_enlarge_language* and *alphabet_enlarge_time*.

```
definition accepts_in_time_mttm ::
  ('q, 'a) mttm ⇒ 'a list ⇒ nat ⇒ bool where
```

```
  accepts_in_time_mttm M w t =
    (∃ n cM_n. n ≤ t
          ∧ (init_config_mttm M w, cM_n) ∈ (mttm_step (delta_tm M))^^n
          ∧ mt_state cM_n = t_tm M)

end
theory Multitape_Substrate
  imports Multitape_Substrate_Core
begin
```

3 Substrate metatheory

The substrate metatheory — *our* additions over the Dalvit–Thiemann definition surface isolated in *Multitape_Substrate_Core*: relation-power and finiteness utilities, the *valid_mttm* axiom-extraction toolkit, per-step and reachability validity preservation, and the displacement / left-endmarker / no-write tape tools. The core (*Multitape_Substrate_Core*) carries the datatypes, accessors, step relation, and the validity / language definitions.

3.1 Relation-power and finiteness utilities

```
lemma finite_UNIV_dir [simp, intro]: finite (UNIV :: dir set)
proof −
  have id: UNIV = {L, R, N}
    using dir.exhaust by auto
  show ?thesis unfolding id by auto
qed

hide_const (open) L R N
```

Blank-tail finiteness: the set of total functions $nat \Rightarrow 'b$ ranging in a finite codomain B and *constant c beyond an index k* is finite. This replaces the function-space finiteness lemmas (*fin_funcsetI* / *finite_UNIV_fun_dir*) that the AFP source relied on: those are false at the value-level *nat* index, where the domain is infinite. Finiteness is recovered from the support bound a blank-tail function is determined by its restriction to $\{..<k\}$, of which there are finitely many. Used to re-derive *finite δ* for constructed machines (*valid_mttm_finite_delta*), with codomain Γ (blank tail) for read/write tuples and *dir* (N tail) for the move tuples.

```
lemma finite_tail_const_funcs:
  fixes B :: 'b set and k :: nat and c :: 'b
  assumes finB: finite B
  shows finite {f :: nat ⇒ 'b. (∀ j. f j ∈ B) ∧ (∀ j ≥ k. f j = c)}
proof −
  let ?S = {f :: nat ⇒ 'b. (∀ j. f j ∈ B) ∧ (∀ j ≥ k. f j = c)}
  have inj: inj_on (λf. restrict f {..<k}) ?S
  proof (rule inj_onI)
```

```
    fix f g
    assume f: f ∈ ?S and g: g ∈ ?S
      and eq: restrict f {..<k} = restrict g {..<k}
    show f = g
    proof
      fix j
      show f j = g j
      proof (cases j < k)
        case True
        have restrict f {..<k} j = restrict g {..<k} j using eq by simp
        thus ?thesis using True by (simp add: restrict_def)
      next
        case False
        hence k ≤ j by simp
        with f g show ?thesis by simp
      qed
    qed
  qed
  have rng: (λf. restrict f {..<k}) ` ?S ⊆ ({..<k} →_E B)
  proof
    fix h assume h ∈ (λf. restrict f {..<k}) ` ?S
    then obtain f where f: f ∈ ?S and h: h = restrict f {..<k} by blast
    show h ∈ {..<k} →_E B
      using f unfolding h by (simp add: restrict_PiE Pi_iff)
  qed
  have finPiE: finite ({..<k} →_E B) using finB by (simp add: finite_PiE)
  have finite ((λf. restrict f {..<k}) ` ?S)
    by (rule finite_subset[OF rng finPiE])
  thus finite ?S using inj by (rule finite_imageD)
qed

lemma relpow_transI:
  (x, y) ∈ R^^n ⟹ (y, z) ∈ R^^m ⟹ (x, z) ∈ R^^(n + m)
  by (simp add: relcomp.intros relpow_add)

lemma relpow_mono: fixes R :: 'a rel
  shows R ⊆ S ⟹ R^^n ⊆ S^^n
  by (induct n, auto)
```

Bounded-iteration combinator: given a single-step law that, from any config satisfying an index-parameterised invariant $P\ i$ with $i < n$, takes one R-step to a config satisfying $P\ (Suc\ i)$, iterate it: from $P\ 0\ c_0$ reach a c' with $(c_0, c') \in R^n$ and $P\ n\ c'$. A loop whose counter starts at some $off > 0$ instantiates P to re-index by that offset. Proved by induction generalising *both* the start config and the invariant, so the hypothesis applies to the shifted predicate $\lambda j.\ P\ (Suc\ j)$ after peeling the first step (prepended via *relpow_Suc_I2*).

```
lemma relpow_invariant_chain:
  fixes R :: ('s × 's) set
```

```
    and P :: nat ⇒ 's ⇒ bool
  assumes step: ⋀i c. ⟦ i < n; P i c ⟧
                      ⟹ ∃ c'. (c, c') ∈ R ∧ P (Suc i) c'
      and base: P 0 c0
  shows ∃ c'. (c0, c') ∈ R ^^ n ∧ P n c'
  using assms
proof (induction n arbitrary: c0)
  case 0
  show ?case using 0.prems(2) by auto
next
  case (Suc m)
  have step_m:
      ⋀i c. ⟦ i < m; P i c ⟧
            ⟹ ∃ c'. (c, c') ∈ R ∧ P (Suc i) c'
  proof −
    fix i :: nat and c :: 's
    assume i < m and P i c
    thus ∃ c'. (c, c') ∈ R ∧ P (Suc i) c'
      using Suc.prems(1)[of i c] by simp
  qed
  obtain c_m where chain_m: (c0, c_m) ∈ R ^^ m and Pcm: P m c_m
    using Suc.IH[OF step_m Suc.prems(2)] by blast
  obtain c' where last_step: (c_m, c') ∈ R and Pc': P (Suc m) c'
    using Suc.prems(1)[of m c_m] Pcm by auto
  have (c0, c') ∈ R ^^ Suc m
    using chain_m last_step by (rule relpow_Suc_I)
  thus ?case using Pc' by blast
qed
```

3.2 Functional axiom-extraction toolkit

Convenience lemmas projecting the substrate's structural axioms out of *valid_mttm* at the functional layer. Each one is a one-shot *by* (*cases M*) *auto*: the case-decomposition rewrites M into MTTM-form, exposes the *valid_mttm.simps* conjunction, and *auto* extracts the relevant conjunct.

These replace what used to be locale-routed retrievals of the form *multitape_tm.X*[*OF loc*].

```
lemma valid_mttm_finite_Q:
  assumes valid_mttm M
  shows finite (Q_tm M)
  using assms by (cases M) auto

lemma valid_mttm_finite_Gamma:
  assumes valid_mttm M
  shows finite (Γ_tm M)
  using assms by (cases M) auto

lemma valid_mttm_Sigma_sub_Gamma:
```

assumes *valid_mttm M*
shows *Sigma_tm M* $\subseteq$ Γ*_tm M*
using *assms* **by** (*cases M*) *auto*

lemma *valid_mttm_s_in_Q*:
assumes *valid_mttm M*
shows *s_tm M* $\in$ *Q_tm M*
using *assms* **by** (*cases M*) *auto*

lemma *valid_mttm_t_in_Q*:
assumes *valid_mttm M*
shows *t_tm M* $\in$ *Q_tm M*
using *assms* **by** (*cases M*) *auto*

lemma *valid_mttm_r_in_Q*:
assumes *valid_mttm M*
shows *r_tm M* $\in$ *Q_tm M*
using *assms* **by** (*cases M*) *auto*

lemma *valid_mttm_blank_in_Gamma*:
assumes *valid_mttm M*
shows *bl_tm M* $\in$ Γ*_tm M*
using *assms* **by** (*cases M*) *auto*

lemma *valid_mttm_blank_not_Sigma*:
assumes *valid_mttm M*
shows *bl_tm M* $\notin$ *Sigma_tm M*
using *assms* **by** (*cases M*) *auto*

lemma *valid_mttm_LE_in_Gamma*:
assumes *valid_mttm M*
shows *le_tm M* $\in$ Γ*_tm M*
using *assms* **by** (*cases M*) *auto*

lemma *valid_mttm_LE_not_Sigma*:
assumes *valid_mttm M*
shows *le_tm M* $\notin$ *Sigma_tm M*
using *assms* **by** (*cases M*) *auto*

lemma *valid_mttm_t_neq_r*:
assumes *valid_mttm M*
shows *t_tm M* $\neq$ *r_tm M*
using *assms* **by** (*cases M*) *auto*

lemma *valid_mttm_k_pos*:
assumes *valid_mttm M*
shows *0* $<$ *k_tm M*
using *assms* **by** (*cases M*) *auto*

lemma *valid_mttm_delta_set*:
assumes *valid_mttm M*
shows *delta_tm M* $\subseteq$
$(Q\_tm\ M - \{t\_tm\ M, r\_tm\ M\})$
$\times\ (UNIV \to \Gamma\_tm\ M)$
$\times\ Q\_tm\ M$
$\times\ (UNIV \to \Gamma\_tm\ M)$
$\times\ (UNIV \to UNIV)$
using *assms* **by** (*cases M*) *auto*

Range typing for transition tuples: pulls the four per-component facts from a single transition membership.

lemma *valid_mttm_delta*:
assumes *vM*: *valid_mttm M*
and *tr*: $(q, a, q', b, d) \in$ *delta_tm M*
shows $q \in Q\_tm\ M\ a\ k \in \Gamma\_tm\ M\ q' \in Q\_tm\ M\ b\ k \in \Gamma\_tm\ M$
using *valid_mttm_delta_set*[*OF vM*] *tr* **by** *auto*

Left-endmarker discipline: transitions reading *le* must rewrite *le* with itself and move only *N* or *R*.

lemma *valid_mttm_deltaLE*:
assumes *vM*: *valid_mttm M*
and *tr*: $(q, a, q', a', d) \in$ *delta_tm M*
and *LE*: $a\ k = le\_tm\ M$
shows $a'\ k = le\_tm\ M \wedge d\ k \in \{dir.N, dir.R\}$
using *vM tr LE* **by** (*cases M*) *auto*

Left-endmarker write discipline: a transition writes *le* on tape *k* only when it was reading *le* on the same tape. This is exactly the content of *le_unique* specialised to one tape / transition; it is the sole place that fact is extracted. Needed by *ae_coupled_run_aux* to preserve the "no LE in window" invariant across an M-step. Takes *le_unique* (not *valid_mttm*): it is the property a machine may forgo, so consumers thread it explicitly rather than reading it off *valid_mttm*.

lemma *valid_mttm_deltaLE_no_write*:
assumes *lu*: *le_unique M*
and *tr*: $(q, a, q', a', d) \in$ *delta_tm M*
and *LE'*: $a'\ k = le\_tm\ M$
shows $a\ k = le\_tm\ M$
using *lu*[*unfolded le_unique_def*] *tr LE'* **by** *blast*

Support invariant: every transition of a valid machine is blank on reads and writes, and stationary, at every tape index $j \geq k\_tm\ M$. The value-level confinement of a k-tape machine's action to tapes $0 \ldots k - 1$.

lemma *valid_mttm_delta_support*:
assumes *vM*: *valid_mttm M*
and *tr*: $(q, a, q', a', d) \in$ *delta_tm M*
and *j*: $j \geq k\_tm\ M$

```
  shows a j = bl_tm M ∧ a' j = bl_tm M ∧ d j = dir.N
proof −
  obtain Q Σ Γ bl le δ s t r K where M_eq:
      M = MTTM Q Σ Γ bl le δ s t r K
    by (cases M)
  have supp:
      ∀ q a q' a' d. (q, a, q', a', d) ∈ δ ⟶
                     (∀ j ≥ K. a j = bl ∧ a' j = bl ∧ d j = dir.N)
    using vM[unfolded M_eq valid_mttm.simps] by blast
  from tr M_eq have tr_delta: (q, a, q', a', d) ∈ δ by simp
  from j M_eq have jK: j ≥ K by simp
  have bleq: bl_tm M = bl using M_eq by simp
  from supp tr_delta jK have a j = bl ∧ a' j = bl ∧ d j = dir.N by blast
  thus ?thesis using bleq by simp
qed
```

Finiteness of δ from bounded support: a valid machine's transition relation is finite. The value-level replacement for the AFP source's function-space finiteness over a finite tape type. Each transition's read / write tuples range in the finite Γ and are blank beyond k, each move tuple is N beyond k; by *finite_tail_const_funcs* there are finitely many of each, so δ embeds in a finite product.

```
lemma valid_mttm_finite_delta:
  assumes vM: valid_mttm M
  shows finite (delta_tm M)
proof −
  obtain Q Σ Γ bl le δ s t r K where M_eq:
      M = MTTM Q Σ Γ bl le δ s t r K
    by (cases M)
  have finQ: finite Q using valid_mttm_finite_Q[OF vM] M_eq by simp
  have finG: finite Γ using valid_mttm_finite_Gamma[OF vM] M_eq by simp
  let ?A = {a. (∀ j. a j ∈ Γ) ∧ (∀ j ≥ K. a j = bl)}
  let ?D = {d. (∀ j. d j ∈ (UNIV :: dir set)) ∧ (∀ j ≥ K. d j = dir.N)}
  have finA: finite ?A by (rule finite_tail_const_funcs[OF finG, of K bl])
  have finD: finite ?D by (rule finite_tail_const_funcs[OF finite_UNIV_dir, of
K dir.N])
  have finprod: finite ((Q − {t, r}) × ?A × Q × ?A × ?D)
    using finQ finA finD by (intro finite_cartesian_product) auto
  have sub: delta_tm M ⊆ (Q − {t, r}) × ?A × Q × ?A × ?D
  proof
    fix x assume xd: x ∈ delta_tm M
    obtain q a q' a' d where x: x = (q, a, q', a', d)
      by (cases x)
    from valid_mttm_delta_set[OF vM] xd x M_eq
    have q_mem: q ∈ Q − {t, r} and a_pi: a ∈ UNIV → Γ
      and q'_mem: q' ∈ Q and a'_pi: a' ∈ UNIV → Γ
      by auto
    from a_pi have aG: ∀ j. a j ∈ Γ by (auto simp: Pi_iff)
    from a'_pi have a'G: ∀ j. a' j ∈ Γ by (auto simp: Pi_iff)
```

```
    have trM: (q, a, q', a', d) ∈ delta_tm M using xd x by simp
    have supp: ∀j ≥ K. a j = bl ∧ a' j = bl ∧ d j = dir.N
    proof (intro allI impI)
      fix j assume K ≤ j
      hence j ≥ k_tm M using M_eq by simp
      thus a j = bl ∧ a' j = bl ∧ d j = dir.N
        using valid_mttm_delta_support[OF vM trM] M_eq by simp
    qed
    have a ∈ ?A using aG supp by auto
    moreover have a' ∈ ?A using a'G supp by auto
    moreover have d ∈ ?D using supp by auto
    ultimately show x ∈ (Q − {t, r}) × ?A × Q × ?A × ?D
      using x q_mem q'_mem by auto
  qed
  show ?thesis using sub finprod by (rule finite_subset)
qed
```

State membership at the source of an *mttm_step*: the source state is in *Q_tm M* and is neither the accept nor the reject state. Direct consequence of the *mttm_step* introduction rule plus *valid_mttm_delta_set*'s range typing (which excludes halting states from the source projection of *delta_tm M*).

Used by the chunked-induction engine *ae_simulation_phase_chunked*: when the M-trace (*cM*, *cM_final*) ∈ *mttm_step* (*delta_tm M*) ^^ (*Suc n*) is non-empty, the first step exists and forces *cM*'s state to be non-halt and in *Q*, which feeds *ae_simulates_forward_stage_general*'s *qM_in_Q* / *q_neq_t* / *q_neq_r* hypotheses.

```
lemma mttm_step_src:
  fixes M :: ('q, 'a) mttm
  assumes vM:   valid_mttm M
    and step: (c, c') ∈ mttm_step (delta_tm M)
  shows mttm_step_src_in_Q:    mt_state c ∈ Q_tm M
    and mttm_step_src_neq_t:   mt_state c ≠ t_tm M
    and mttm_step_src_neq_r:   mt_state c ≠ r_tm M
proof −
  from step obtain q ts n q' a dir where
      c_eq: c = Config_M q ts n
    and tr: (q, λk. ts k (n k), q', a, dir) ∈ delta_tm M
    by (auto elim: mttm_step.cases)
  have q_eq: mt_state c = q using c_eq by simp
  have q_in_strict: q ∈ Q_tm M − {t_tm M, r_tm M}
    using valid_mttm_delta_set[OF vM] tr by auto
  show mt_state c ∈ Q_tm M  using q_eq q_in_strict by simp
  show mt_state c ≠ t_tm M using q_eq q_in_strict by simp
  show mt_state c ≠ r_tm M using q_eq q_in_strict by simp
qed
```

3.3 Functional validity preservation

Per-step preservation of *valid_config_mttm*: a valid configuration steps only to valid configurations. Re-derived directly from *valid_mttm_delta* (range typing), *valid_mttm_deltaLE* (left-endmarker discipline), and *valid_mttm_delta_support* (the new inactive-tape blank-tail case).

```
lemma valid_step_mttm:
  fixes M :: ('q, 'a) mttm
  assumes vM:    valid_mttm M
    and step:    (c, c') ∈ mttm_step (delta_tm M)
    and val_c:   valid_config_mttm M c
  shows valid_config_mttm M c'
proof −
  obtain Q Σ Γ bl le δ s t r K where M_eq:
      M = MTTM Q Σ Γ bl le δ s t r K
    by (cases M)
  obtain q ts n where c_eq: c = Config_M q ts n
    by (cases c)
  from step M_eq c_eq have step_delta:
      (Config_M q ts n, c') ∈ mttm_step δ
    by simp
  obtain q' a dir where c'_eq:
      c' = Config_M q' (λk. (ts k)(n k := a k))
                       (λk. go_dir (dir k) (n k))
    and tr: (q, (λk. ts k (n k)), q', a, dir) ∈ δ
    using step_delta by (auto elim: mttm_step.cases)
  from val_c c_eq M_eq have q_in: q ∈ Q
    and ts_Gamma: ⋀k. range (ts k) ⊆ Γ
    and ts_LE: ⋀k. k < K ⟹ ts k 0 = le
    and ts_blank: ⋀k p. k ≥ K ⟹ ts k p = bl
    by auto
  from tr M_eq have tr_M: (q, (λk. ts k (n k)), q', a, dir) ∈ delta_tm M
    by simp
  have q'_in: q' ∈ Q_tm M and a_Gamma: ⋀k. a k ∈ Γ_tm M
    using valid_mttm_delta[OF vM tr_M] by auto
  hence q'_in_Q: q' ∈ Q and a_Gamma_set: ⋀k. a k ∈ Γ
    using M_eq by auto
  have new_ts_Gamma: ⋀k. range ((ts k)(n k := a k)) ⊆ Γ
    using ts_Gamma a_Gamma_set by auto
  have new_LE: ⋀k. k < K ⟹ ((ts k)(n k := a k)) 0 = le
  proof −
    fix k assume kK: k < K
    show ((ts k)(n k := a k)) 0 = le
    proof (cases n k = 0)
      case True
      have read_LE: (λj. ts j (n j)) k = le
        using ts_LE[OF kK] True by simp
      have read_LE_le: (λj. ts j (n j)) k = le_tm M
        using read_LE M_eq by simp
```

```
    have a k = le_tm M
      using valid_mttm_deltaLE[OF vM tr_M read_LE_le] by simp
    hence a k = le using M_eq by simp
    with True show ?thesis by simp
  next
    case False
    thus ?thesis using ts_LE[OF kK] by simp
  qed
qed
have new_blank: ⋀k p. k ≥ K ⟹ ((ts k)(n k := a k)) p = bl
proof −
  fix k p assume kK: k ≥ K
  have ak_bl: a k = bl
  proof −
    have k ≥ k_tm M using kK M_eq by simp
    hence a k = bl_tm M
      using valid_mttm_delta_support[OF vM tr_M] by simp
    thus ?thesis using M_eq by simp
  qed
  show ((ts k)(n k := a k)) p = bl
    using ts_blank[OF kK] ak_bl by (cases p = n k) auto
qed
show ?thesis
  unfolding M_eq c'_eq valid_config_mttm.simps
proof (intro conjI allI impI)
  show q' ∈ Q using q'_in_Q .
next
  fix i
  show range ((λk. (ts k)(n k := a k)) i) ⊆ Γ
    using new_ts_Gamma by simp
next
  fix i assume i < K
  show (λk. (ts k)(n k := a k)) i 0 = le
    using new_LE[OF ‹i < K›] by simp
next
  fix i p assume K ≤ i
  show (λk. (ts k)(n k := a k)) i p = bl
    using new_blank[OF ‹K ≤ i›] by simp
qed
qed
```

Initial-configuration validity: a valid M's initial configuration on a valid input is itself valid.

```
lemma valid_init_config_mttm:
  fixes M :: ('q, 'a) mttm
  assumes vM: valid_mttm M
    and w:    set w ⊆ Sigma_tm M
  shows valid_config_mttm M (init_config_mttm M w)
proof −
```

```
  obtain Q Σ Γ bl le δ s t r K where M_eq:
      M = MTTM Q Σ Γ bl le δ s t r K
    by (cases M)
  have w_Sigma: set w ⊆ Σ
    using w M_eq by simp
  have s_in: s ∈ Q using valid_mttm_s_in_Q[OF vM] M_eq by simp
  have Sigma_sub: Σ ⊆ Γ using valid_mttm_Sigma_sub_Gamma[OF vM] M_eq
by simp
   have bl_in: bl ∈ Γ using valid_mttm_blank_in_Gamma[OF vM] M_eq by
simp
  have le_in: le ∈ Γ using valid_mttm_LE_in_Gamma[OF vM] M_eq by simp
  show ?thesis
    unfolding M_eq init_config_mttm.simps valid_config_mttm.simps
  proof (intro conjI allI impI)
    show s ∈ Q using s_in .
  next
    fix i
    show range (λn. if i < K
                  then (if n = 0 then le
                        else if i = 0 ∧ n ≤ length w then w ! (n − 1)
                        else bl)
                  else bl) ⊆ Γ
    proof (cases i < K)
      case True
      show ?thesis
        using bl_in le_in Sigma_sub w_Sigma True
        by (force simp: set_conv_nth)
    next
      case False
      show ?thesis using bl_in False by simp
    qed
  next
    fix i assume i < K
    show (if i < K
          then (if (0::nat) = 0 then le
                else if i = 0 ∧ 0 ≤ length w then w ! (0 − 1)
                else bl)
          else bl) = le
      using ‹i < K› by simp
  next
    fix i p assume K ≤ i
    show (if i < K
          then (if p = 0 then le
                else if i = 0 ∧ p ≤ length w then w ! (p − 1)
                else bl)
          else bl) = bl
      using ‹K ≤ i› by simp
  qed
qed
```

Reachability lift: every configuration reachable from a valid initial configuration is itself valid.

```
lemma valid_reach_mttm:
  fixes M :: ('q, 'a) mttm
  assumes vM:    valid_mttm M
    and w:       set w ⊆ Sigma_tm M
    and reach:   (init_config_mttm M w, c) ∈ (mttm_step (delta_tm M))*
  shows valid_config_mttm M c
  using reach
proof induction
  case base
  show ?case using valid_init_config_mttm[OF vM w] .
next
  case (step y z)
  show ?case using valid_step_mttm[OF vM step.hyps(2) step.IH] .
qed
```

Blank-tail accessor for a valid configuration: beyond the machine's tape count $k\_tm\ M$ every cell holds the blank $bl\_tm\ M$. Used on the reverse lift's padding tapes, where the per-tape window invariant is unavailable (it constrains the LE home cell, which is blank on padding) and the read-match must instead come from the substrate blank-tail.

```
lemma valid_config_mttm_blank_tail:
  assumes valc: valid_config_mttm M c
      and kge:  i ≥ k_tm M
  shows mt_tape c i p = bl_tm M
proof −
  obtain Q Σ Γ bl le δ s t r K where
      M_eq: M = MTTM Q Σ Γ bl le δ s t r K
    by (cases M)
  obtain q ts n where c_eq: c = Config_M q ts n by (cases c)
  from valc kge show ?thesis
    unfolding M_eq c_eq by auto
qed
```

Validity preservation along an n-step substrate trace from an arbitrary valid configuration (generic relpow closure of *valid_step_mttm*; *valid_reach_mttm* anchors only at *init_config_mttm*). Threads the blank-tail validity of the reverse lift's intermediate configs cM_n' through the chain induction.

```
lemma valid_reach_relpow_mttm:
  assumes vM:    valid_mttm M
      and valc:  valid_config_mttm M c
      and reach: (c, c') ∈ mttm_step (delta_tm M) ^^ n
  shows valid_config_mttm M c'
  using reach
proof (induction n arbitrary: c')
  case 0
```

```
  thus ?case using valc by simp
next
  case (Suc n)
  from Suc.prems obtain c'' where
      mid: (c, c'') ∈ mttm_step (delta_tm M) ^^ n
    and lst: (c'', c') ∈ mttm_step (delta_tm M)
    by (rule relpow_Suc_E)
  have valid_config_mttm M c'' using Suc.IH[OF mid] .
  thus ?case by (rule valid_step_mttm[OF vM lst])
qed
```

Per-tape LE-pinning corollary: along any reachable trace from a valid initial configuration, every *active* tape (index $k < k\_tm\ M$) carries $le\_tm\ M$ at position 0.

Note: *valid_config_mttm* only encodes the position-0 = LE constraint on active tapes, not the converse (positions $p \neq 0$ may legally hold LE in an arbitrary valid config). The "LE only at position 0" property is reach-specific and proved separately in the companion lemma below.

```
lemma valid_reach_LE_pos0_mttm:
  fixes M :: ('q, 'a) mttm
  assumes vM:  valid_mttm M
    and w:     set w ⊆ Sigma_tm M
    and reach: (init_config_mttm M w, Config_M q ts n)
                 ∈ (mttm_step (delta_tm M))*
    and kK:    k < k_tm M
  shows ts k 0 = le_tm M
proof −
  have val: valid_config_mttm M (Config_M q ts n)
    using valid_reach_mttm[OF vM w reach] .
  obtain Q Σ Γ bl le δ sM t r K where M_eq:
      M = MTTM Q Σ Γ bl le δ sM t r K
    by (cases M)
  have k < K using kK M_eq by simp
  thus ?thesis using val M_eq by simp
qed
```

Dual to *valid_reach_LE_pos0_mttm*: along any reachable trace from a valid initial configuration whose blank symbol differs from the left endmarker, every cell at a position $p \neq 0$ on any tape is not the left endmarker. Proof: induction on the reachability relation. Base: *init_config_mttm* places *le* only at position 0 of active tapes (input cells are in Σ hence $\neq le$; other active cells and all inactive cells are *bl* which by hypothesis $\neq le$). Step: cells away from the head are preserved (substrate update is $(ts\ k)(n\ k := a\ k)$); at the head, the substrate's δLE-no-write rule says the write equals *le* only if the read did, which by IH is impossible since the read sits at the head position $n\ k$ (and if $n\ k = p \neq 0$ the IH gives the read is not *le*; if $n\ k = 0$ then $p \neq n\ k$ and the off-head case fires).

```
lemma valid_reach_LE_only_pos0_mttm:
```

```
  fixes M :: ('q, 'a) mttm
    and c :: ('a, 'q) mt_config
    and k :: nat
    and p :: nat
  assumes vM:        valid_mttm M
    and lu:          le_unique M
    and w:           set w ⊆ Sigma_tm M
    and reach:       (init_config_mttm M w, c) ∈ (mttm_step (delta_tm M))*
    and p_ne_0:      p ≠ 0
    and bl_neq_le:   bl_tm M ≠ le_tm M
  shows mt_tape c k p ≠ le_tm M
proof −
  obtain Q Σ Γ bl le δ sM tt rr K where M_eq:
      M = MTTM Q Σ Γ bl le δ sM tt rr K
    by (cases M)
  have le_eq:  le_tm M = le  using M_eq by simp
  have bl_eq:  bl_tm M = bl  using M_eq by simp
  have bl_ne_le: bl ≠ le using bl_neq_le le_eq bl_eq by simp
  have le_not_Sigma: le ∉ Σ
    using valid_mttm_LE_not_Sigma[OF vM] M_eq by simp
  have w_Sigma: set w ⊆ Σ using w M_eq by simp
  have main:
    ∀ c'. (init_config_mttm M w, c') ∈ (mttm_step (delta_tm M))*
          ⟶ mt_tape c' k p ≠ le_tm M
  proof (intro allI impI)
    fix c'
    assume r: (init_config_mttm M w, c') ∈ (mttm_step (delta_tm M))*
    show mt_tape c' k p ≠ le_tm M
      using r
    proof induction
      case base
      have tape_p_init:
          mt_tape (init_config_mttm M w) k p
            = (if k < K
               then (if p = 0 then le
                     else if k = 0 ∧ p ≤ length w then w ! (p − 1)
                     else bl)
               else bl)
        unfolding M_eq init_config_mttm.simps by simp
      show ?case
      proof (cases k = 0 ∧ p ≤ length w)
        case True
        have K_pos: 0 < K using valid_mttm_k_pos[OF vM] M_eq by simp
        have kK: k < K using True K_pos by simp
        have p_ge_1: p ≥ 1 using p_ne_0 by simp
        have idx_lt: p − 1 < length w using True p_ge_1 by linarith
        have in_w: w ! (p − 1) ∈ set w
          using idx_lt by (auto simp: set_conv_nth)
        with w_Sigma have w ! (p − 1) ∈ Σ by blast
```

```
      with le_not_Sigma have w ! (p - 1) ≠ le by blast
      thus ?thesis using tape_p_init True kK p_ne_0 le_eq by simp
    next
      case False
      have mt_tape (init_config_mttm M w) k p = bl
        using tape_p_init False p_ne_0 by (cases k < K) auto
      thus ?thesis using bl_ne_le le_eq by simp
    qed
  next
    case (step y z)
    have ne_y: mt_tape y k p ≠ le_tm M using step.IH .
    obtain qy ts n where y_eq: y = Config_M qy ts n
      by (cases y)
    from step.hyps(2) y_eq obtain q' a dr where
        z_eq: z = Config_M q'
                  (λkk. (ts kk)(n kk := a kk))
                  (λkk. go_dir (dr kk) (n kk))
      and tr: (qy, (λkk. ts kk (n kk)), q', a, dr) ∈ delta_tm M
      by (auto elim: mttm_step.cases)
    show ?case
    proof (cases n k = p)
      case False
      have mt_tape z k p = ((ts k)(n k := a k)) p using z_eq by simp
      also have ... = ts k p using False by simp
      also have ... = mt_tape y k p using y_eq by simp
      finally have mt_tape z k p = mt_tape y k p .
      thus ?thesis using ne_y by simp
    next
      case True
      have read_at_p: ts k (n k) = mt_tape y k p
        using y_eq True by simp
      have read_ne: ts k (n k) ≠ le_tm M
        using ne_y read_at_p by simp
      have a_ne: a k ≠ le_tm M
      proof
        assume a_le: a k = le_tm M
        have (λkk. ts kk (n kk)) k = le_tm M
          using valid_mttm_deltaLE_no_write[OF lu tr a_le] by simp
        hence ts k (n k) = le_tm M by simp
        thus False using read_ne by simp
      qed
      have mt_tape z k p = ((ts k)(n k := a k)) p using z_eq by simp
      also have ... = a k using True by simp
      finally have mt_tape z k p = a k .
      thus ?thesis using a_ne by simp
    qed
  qed
qed
show ?thesis using main reach by blast
```

qed

3.4 Step locality: non-head cells preserved, head moves by ≤ 1

Two structural facts about *mttm_step* that fall directly out of the single rule's body $(ts\ k)(n\ k := a\ k)$ and $go\_dir\ (dir\ k)\ (n\ k)$: a single step modifies only the head cell on each tape, and the head displaces by at most one position per tape per step. These are generic over δ and used by alphabet-enlargement / -reduction to argue that cells outside a bounded window are unchanged after n steps.

lemma *mttm_step_tape_off_head*:
  assumes *step*: $(c,\ c') \in mttm\_step\ \delta$
    and *ne*: $p \neq mt\_pos\ c\ k$
  shows $mt\_tape\ c'\ k\ p = mt\_tape\ c\ k\ p$
proof −
  from *step* **obtain** $q\ ts\ n\ q'\ a\ dr$ **where**
      c_eq: $c = Config_M\ q\ ts\ n$
    and c'_eq: $c' = Config_M\ q'$
                  $(\lambda k.\ (ts\ k)(n\ k := a\ k))$
                  $(\lambda k.\ go\_dir\ (dr\ k)\ (n\ k))$
    by (*auto elim*: *mttm_step.cases*)
  from *ne c_eq* **have** $p \neq n\ k$ **by** *simp*
  thus *?thesis* **by** (*simp add*: *c_eq c'_eq*)
qed

lemma *mttm_step_pos_displacement*:
  assumes *step*: $(c,\ c') \in mttm\_step\ \delta$
  shows $mt\_pos\ c'\ k \leq mt\_pos\ c\ k + 1$
        $\wedge\ mt\_pos\ c\ k \leq mt\_pos\ c'\ k + 1$
proof −
  from *step* **obtain** $q\ ts\ n\ q'\ a\ dr$ **where**
      c_eq: $c = Config_M\ q\ ts\ n$
    and c'_eq: $c' = Config_M\ q'$
                  $(\lambda k.\ (ts\ k)(n\ k := a\ k))$
                  $(\lambda k.\ go\_dir\ (dr\ k)\ (n\ k))$
    by (*auto elim*: *mttm_step.cases*)
  have *pos*: $mt\_pos\ c\ k = n\ k$ **by** (*simp add*: *c_eq*)
  have *pos'*: $mt\_pos\ c'\ k = go\_dir\ (dr\ k)\ (n\ k)$ **by** (*simp add*: *c'_eq*)
  show *?thesis*
    unfolding *pos pos'* **by** (*cases dr k*) *auto*
qed

n-step lift: after n steps, head displacement on each tape is at most n.

lemma *mttm_relpow_pos_displacement*:
  assumes $(c,\ c') \in mttm\_step\ \delta\ \widehat{\ }\widehat{\ }\ n$
  shows $mt\_pos\ c'\ k \leq mt\_pos\ c\ k + n$
        $\wedge\ mt\_pos\ c\ k \leq mt\_pos\ c'\ k + n$

```
  using assms
proof (induction n arbitrary: c')
  case 0
  thus ?case by simp
next
  case (Suc n)
  from Suc.prems obtain c'' where
      ih: (c, c'') ∈ mttm_step δ ^^ n
    and step: (c'', c') ∈ mttm_step δ
    by (auto elim: relpow_Suc_E)
  have h1: mt_pos c'' k ≤ mt_pos c k + n
         ∧ mt_pos c k ≤ mt_pos c'' k + n
    using Suc.IH[OF ih] .
  have h2: mt_pos c' k ≤ mt_pos c'' k + 1
         ∧ mt_pos c'' k ≤ mt_pos c' k + 1
    using mttm_step_pos_displacement[OF step] .
  from h1 h2 show ?case by linarith
qed
```

n-step lift: cells more than n away from the start head position are unchanged after n steps.

```
lemma mttm_relpow_tape_off_window:
  assumes (c, c') ∈ mttm_step δ ^^ n
    and p > mt_pos c k + n ∨ p + n < mt_pos c k
  shows mt_tape c' k p = mt_tape c k p
  using assms
proof (induction n arbitrary: c')
  case 0
  thus ?case by simp
next
  case (Suc n)
  from Suc.prems(1) obtain c'' where
      ih: (c, c'') ∈ mttm_step δ ^^ n
    and step: (c'', c') ∈ mttm_step δ
    by (auto elim: relpow_Suc_E)
  have far: p > mt_pos c k + n ∨ p + n < mt_pos c k
    using Suc.prems(2) by linarith
  have eq_ih: mt_tape c'' k p = mt_tape c k p
    using Suc.IH[OF ih far] .
  have disp: mt_pos c'' k ≤ mt_pos c k + n
           ∧ mt_pos c k ≤ mt_pos c'' k + n
    using mttm_relpow_pos_displacement[OF ih] .
  have ne: p ≠ mt_pos c'' k
    using Suc.prems(2) disp by linarith
  have eq_step: mt_tape c' k p = mt_tape c'' k p
    using mttm_step_tape_off_head[OF step ne] .
  show ?case using eq_step eq_ih by simp
qed
```

3.5 Left-endmarker pinning at position 0 along execution

Local LE-pinning: any step of a valid M from a config whose tape k already has *le_tm M* at position 0 ends with a config whose tape k still has *le_tm M* at position 0. Standalone variant of *valid_step_mttm*'s position-0-= LE conjunct, stripped of the *valid_config_mttm* precondition: only the per-tape LE-at-0 fact is needed (not Q-membership or Γ-typing). Case-split on whether the head is at position 0: at-head fires δLE (read LE forces write LE); off-head uses *mttm_step_tape_off_head* directly.

lemma *mttm_step_LE_pos0_preserve*:
  fixes *M* :: (*'q*, *'a*) *mttm*
  assumes *vM*: *valid_mttm M*
    and *step*: (*c*, *c'*) ∈ *mttm_step* (*delta_tm M*)
    and *LE_at0*: *mt_tape c k 0* = *le_tm M*
  shows *mt_tape c' k 0* = *le_tm M*
proof (*cases mt_pos c k* = *0*)
  case *True*
  from *step* **obtain** *q ts n q' a dr* **where**
      c_eq: *c* = $Config_M$ *q ts n*
    and *c'_eq*: *c'* = $Config_M$ *q'*
                  (λ*kk*. (*ts kk*)(*n kk* := *a kk*))
                  (λ*kk*. *go_dir* (*dr kk*) (*n kk*))
    and *tr*: (*q*, λ*kk*. *ts kk* (*n kk*), *q'*, *a*, *dr*) ∈ *delta_tm M*
    by (*auto elim*: *mttm_step.cases*)
  have *nk_0*: *n k* = *0* **using** *True c_eq* **by** *simp*
  have *ts_k_0_le*: *ts k 0* = *le_tm M* **using** *LE_at0 c_eq* **by** *simp*
  have *read_LE*: (λ*kk*. *ts kk* (*n kk*)) *k* = *le_tm M*
    using *ts_k_0_le nk_0* **by** *simp*
  have *write_LE*: *a k* = *le_tm M*
    using *valid_mttm_deltaLE*[*OF vM tr read_LE*] **by** *simp*
  have *mt_tape c' k 0* = ((*ts k*)(*n k* := *a k*)) *0*
    using *c'_eq* **by** *simp*
  also have ... = *a k* **using** *nk_0* **by** *simp*
  also have ... = *le_tm M* **using** *write_LE* **.**
  finally show *?thesis* **.**
next
  case *False*
  have *ne_0*: (*0* :: *nat*) ≠ *mt_pos c k* **using** *False* **by** *simp*
  have *unchanged*: *mt_tape c' k 0* = *mt_tape c k 0*
    using *mttm_step_tape_off_head*[*OF step ne_0*] **.**
  show *?thesis* **using** *unchanged LE_at0* **by** *simp*
qed

n-step lift of *mttm_step_LE_pos0_preserve*: along any chain of M-steps, *le_tm M* at position 0 is preserved on every tape. Used in AE's LE-edge forward stage to derive *cM_k*'s position-0 = LE without recourse to reachability from init.

lemma *mttm_relpow_LE_pos0_preserve*:

```
  fixes M :: ('q, 'a) mttm
  assumes vM:    valid_mttm M
    and chain:   (c, c') ∈ mttm_step (delta_tm M) ^^ n
    and LE_at0:  mt_tape c k 0 = le_tm M
  shows mt_tape c' k 0 = le_tm M
  using chain
proof (induction n arbitrary: c')
  case 0
  thus ?case using LE_at0 by simp
next
  case (Suc n)
  from Suc.prems obtain c'' where
      ih: (c, c'') ∈ mttm_step (delta_tm M) ^^ n
    and step: (c'', c') ∈ mttm_step (delta_tm M)
    by (auto elim: relpow_Suc_E)
  have at_c'': mt_tape c'' k 0 = le_tm M using Suc.IH[OF ih] .
  show ?case
    using mttm_step_LE_pos0_preserve[OF vM step at_c''] .
qed
```

3.6 Step structural decomposition

Packages *mttm_step.cases* with the post-state position + tape equations on each tape: a single *obtains* rule that yields all four useful facts (pre-state's state, post-state's state, the per-tape position equation *go_dir* (*dr k*) (*n k*), and the per-tape head-cell update). Chain proofs in AE / AR / TR avoid re-doing the case-analysis at every per-substep position-trajectory step.

```
lemma mttm_step_obtain_action:
  assumes step: (c, c') ∈ mttm_step δ
  obtains q q' a dr where
      mt_state c = q
    and mt_state c' = q'
    and (q, λk. mt_tape c k (mt_pos c k), q', a, dr) ∈ δ
    and ⋀kk. mt_pos c' kk = go_dir (dr kk) (mt_pos c kk)
    and ⋀kk p. p ≠ mt_pos c kk
                ⟹ mt_tape c' kk p = mt_tape c kk p
    and ⋀kk. mt_tape c' kk (mt_pos c kk) = a kk
proof −
  from step obtain q ts n q' a dr where
      c_eq:  c = Config_M q ts n
    and c'_eq: c' = Config_M q'
                    (λk. (ts k)(n k := a k))
                    (λk. go_dir (dr k) (n k))
    and tr: (q, λk. ts k (n k), q', a, dr) ∈ δ
    by (auto elim: mttm_step.cases)
  have st_c: mt_state c = q using c_eq by simp
  have st_c': mt_state c' = q' using c'_eq by simp
  have pos_c: ⋀kk. mt_pos c kk = n kk using c_eq by simp
```

```
  have pos_c': ⋀kk. mt_pos c' kk = go_dir (dr kk) (n kk)
    using c'_eq by simp
  have tape_c: ⋀kk. mt_tape c kk = ts kk using c_eq by simp
  have tape_c': ⋀kk. mt_tape c' kk = (ts kk)(n kk := a kk)
    using c'_eq by simp
  have tr': (q, λk. mt_tape c k (mt_pos c k), q', a, dr) ∈ δ
    using tr c_eq by simp
  show thesis
  proof (rule that[OF st_c st_c' tr'])
    fix kk show mt_pos c' kk = go_dir (dr kk) (mt_pos c kk)
      using pos_c' pos_c by simp
  next
    fix kk p assume p ≠ mt_pos c kk
    thus mt_tape c' kk p = mt_tape c kk p
      using tape_c' tape_c pos_c by simp
  next
    fix kk show mt_tape c' kk (mt_pos c kk) = a kk
      using tape_c' pos_c by simp
  qed
qed
```

3.7 No-write deltas: tape preserved across a step

For a transition relation in which every tuple's read-component equals its write-component, a single step preserves the entire tape function (the substrate update $f(x := f\,x)$ is the identity). Used by the alphabet- enlargement chain proof for the read-only buffer-loading substeps SS1→SS2, SS2→SS3, SS3→SS4, SS4→SS5.

```
lemma mttm_step_no_write_tape:
  assumes step: (c, c') ∈ mttm_step δ
    and no_write: ⋀q a q' a' d. (q, a, q', a', d) ∈ δ ⟹ a' = a
  shows mt_tape c' = mt_tape c
proof −
  from step obtain q ts n q' a dr where
      c_eq:  c = Config_M q ts n
    and c'_eq: c' = Config_M q'
                    (λk. (ts k)(n k := a k))
                    (λk. go_dir (dr k) (n k))
    and tr: (q, λk. ts k (n k), q', a, dr) ∈ δ
    by (auto elim: mttm_step.cases)
  from no_write[OF tr] have a = (λk. ts k (n k)) by simp
  hence (λk. (ts k)(n k := a k)) = ts by auto
  thus ?thesis by (simp add: c_eq c'_eq)
qed
```

3.8 Step determinism and acceptance monotonicity

Two general facts promoted from the finite-control layer: a step of a functional transition relation is deterministic (a configuration has at most one successor), and weak time-bounded acceptance is monotone in the time budget.

lemma *mttm_step_functional*:
  assumes *fdet*: $\forall\, q\ a\ p_1\ b_1\ d_1\ p_2\ b_2\ d_2.$
      $(q,\ a,\ p_1,\ b_1,\ d_1) \in \delta \longrightarrow (q,\ a,\ p_2,\ b_2,\ d_2) \in \delta$
        $\longrightarrow (p_1,\ b_1,\ d_1) = (p_2,\ b_2,\ d_2)$
    and *step1*: $(c,\ c1) \in mttm\_step\ \delta$
    and *step2*: $(c,\ c2) \in mttm\_step\ \delta$
  shows $c1 = c2$
proof −
  from *step1* **obtain** $q\ ts\ n\ q1'\ a1\ d1$ **where**
      c_eq: $c = Config_M\ q\ ts\ n$
    and *c1_eq*: $c1 = Config_M\ q1'\ (\lambda k.\ (ts\ k)(n\ k := a1\ k))\ (\lambda k.\ go\_dir\ (d1\ k)\ (n\ k))$
    and *tr1*: $(q,\ \lambda k.\ ts\ k\ (n\ k),\ q1',\ a1,\ d1) \in \delta$
    by (*auto elim*: *mttm_step.cases*)
  from *step2* **obtain** $q2'\ a2\ d2$ **where**
      c2_eq: $c2 = Config_M\ q2'\ (\lambda k.\ (ts\ k)(n\ k := a2\ k))\ (\lambda k.\ go\_dir\ (d2\ k)\ (n\ k))$
    and *tr2*: $(q,\ \lambda k.\ ts\ k\ (n\ k),\ q2',\ a2,\ d2) \in \delta$
    using *c_eq* **by** (*auto elim*: *mttm_step.cases*)
  have $(q1',\ a1,\ d1) = (q2',\ a2,\ d2)$ **using** *fdet tr1 tr2* **by** *blast*
  thus *?thesis* **using** *c1_eq c2_eq* **by** *simp*
qed

Weak time-bounded acceptance is monotone in the time budget.

lemma *accepts_in_time_mttm_mono*:
  $accepts\_in\_time\_mttm\ M\ w\ t \Longrightarrow t \le t' \Longrightarrow accepts\_in\_time\_mttm\ M\ w\ t'$
  unfolding *accepts_in_time_mttm_def* **by** (*meson order_trans*)

end
theory *Multitape_Origin_Float*
  imports *Multitape_Substrate*
begin

4 Origin floating: an unreachable left prefix is invisible

A machine that has planted a fresh left endmarker at some cell d can never read the cells to its left: the left-endmarker discipline (clause 1, *valid_mttm_deltaLE*) forbids a leftward move off *le*, so the head is penned in the semi-tape $[d, \infty)$. The content below d is therefore dead weight. This theory makes that precise as a per-tape left *shift* relation and shows the substrate step relation translates across it in both directions.

The intended use is origin floating on a singly-infinite tape: to reuse a spent input tape as a blank work tape without the linear rewind, a machine plants *le* at the current head cell d and treats d as the new origin. The tape from d onward (*le* then blanks) is then indistinguishable from a fresh blank tape whose origin sits at cell 0 — which is exactly a shift by d.

Not a bisimulation. The machines here may be nondeterministic, so the two step relations are *not* bisimilar (their successor sets do not correspond). Each single-step lemma below translates *one* transition; chained along a path it carries a single witnessing run from one origin to the other, which is all that language inclusion needs. The forward and reverse lemmas together give inclusion in both directions — never a claim that the run trees match.

4.1 The per-tape left-shift relation

shift_rel d c1 c2: configuration *c2* is *c1* with each tape k shifted right by d k cells. The state agrees, every head sits d k cells further right, and every cell of *c1* reappears d k cells further right in *c2*. The cells of *c2* below d k are unconstrained — that is the discarded prefix. Setting $d\ k = 0$ leaves tape k untouched, so a single-tape float is the instance $d = (\lambda j.\ \textit{if}\ j = k0\ \textit{then off else}\ 0)$.

```
definition shift_rel ::
  (nat ⇒ nat) ⇒ ('a, 'q) mt_config ⇒ ('a, 'q) mt_config ⇒ bool where
  shift_rel d c1 c2 ⟷
    mt_state c2 = mt_state c1
    ∧ (∀k. mt_pos c2 k = mt_pos c1 k + d k)
    ∧ (∀k p. mt_tape c2 k (p + d k) = mt_tape c1 k p)

lemma shift_rel_state:
  shift_rel d c1 c2 ⟹ mt_state c2 = mt_state c1
  by (simp add: shift_rel_def)
```

4.2 Single-step translation, both directions

Forward: a step of M from the base config *c1* lifts to a step from the shifted config *c2*, landing in the shift of the base successor. The one delicate point is the head position at a floated origin: were the head at cell 0 of a floated tape ($d\ k > 0$) to move left, the base clamps at 0 while the shift lands at $d\ k - 1$, breaking sync. But cell 0 of the base carries *le* (hypothesis *le0*), so clause 1 forbids that leftward move. *le0* is required only on the *floated* tapes ($d\ k \neq 0$): where $d\ k = 0$ the shift is the identity and a leftward move at cell 0 stays in sync, so no *le* is needed there. This matters when the base is a lift with all-blank out-of-range tapes (no *le* at their cell 0): those tapes are never floated, so *le0* does not constrain them. On the floated tapes *le0* holds of every configuration reachable from an initial one

(*valid_reach_LE_pos0_mttm*).

lemma *mttm_step_shift_forward*:
  fixes M :: ('*q*, '*a*) *mttm*
  assumes *vM*: *valid_mttm M*
    and *le0*: $\forall k.\ d\ k \neq 0 \longrightarrow mt\_tape\ c1\ k\ 0 = le\_tm\ M$
    and *rel*: *shift_rel d c1 c2*
    and *step*: $(c1, c1') \in mttm\_step\ (delta\_tm\ M)$
  shows $\exists c2'.\ (c2, c2') \in mttm\_step\ (delta\_tm\ M) \wedge shift\_rel\ d\ c1'\ c2'$
proof −
  from *step* **obtain** *q ts1 n1 q' a dr* **where**
      c1_eq: $c1 = Config_M\ q\ ts1\ n1$
    and *c1'_eq*: $c1' = Config_M\ q'\ (\lambda k.\ (ts1\ k)(n1\ k := a\ k))$
                                $(\lambda k.\ go\_dir\ (dr\ k)\ (n1\ k))$
    and *tr*: $(q, \lambda k.\ ts1\ k\ (n1\ k), q', a, dr) \in delta\_tm\ M$
    by (*auto elim*: *mttm_step.cases*)
  obtain *sc ts2 n2* **where** *c2_eq*: $c2 = Config_M\ sc\ ts2\ n2$
    by (*cases c2*) *auto*
  have *unpacked*:
    $sc = q \wedge (\forall k.\ n2\ k = n1\ k + d\ k) \wedge (\forall k\ p.\ ts2\ k\ (p + d\ k) = ts1\ k\ p)$
    using *rel* **by** (*simp add*: *shift_rel_def c1_eq c2_eq*)
  from *unpacked* **have** *st2*: $sc = q$ **by** *simp*
  from *unpacked* **have** *pos2*: $\bigwedge k.\ n2\ k = n1\ k + d\ k$ **by** *simp*
  from *unpacked* **have** *tape2*: $\bigwedge k\ p.\ ts2\ k\ (p + d\ k) = ts1\ k\ p$ **by** *simp*
  — Reads match: the shifted head reads what the base head reads.
  have *read_eq*: $(\lambda k.\ ts2\ k\ (n2\ k)) = (\lambda k.\ ts1\ k\ (n1\ k))$
  proof
    fix *k*
    have $ts2\ k\ (n2\ k) = ts2\ k\ (n1\ k + d\ k)$ **using** *pos2* **by** *simp*
    also have $\ldots = ts1\ k\ (n1\ k)$ **using** *tape2* **by** *simp*
    finally show $ts2\ k\ (n2\ k) = ts1\ k\ (n1\ k)$ **.**
  qed
  have *tr2*: $(q, \lambda k.\ ts2\ k\ (n2\ k), q', a, dr) \in delta\_tm\ M$
    using *tr read_eq* **by** *simp*
  let *?c2'* = $Config_M\ q'\ (\lambda k.\ (ts2\ k)(n2\ k := a\ k))$
                    $(\lambda k.\ go\_dir\ (dr\ k)\ (n2\ k))$
  have *step2*: $(c2, ?c2') \in mttm\_step\ (delta\_tm\ M)$
    unfolding *c2_eq st2* **by** (*rule mttm_step.step*, *rule tr2*)
  — No leftward move at a floated origin: reading *le* forbids *L*. Only floated tapes ($d\ k \neq 0$) need this; unfloated ones stay in sync regardless.
  have *no_L_at0*: $\bigwedge k.\ d\ k \neq 0 \Longrightarrow n1\ k = 0 \Longrightarrow dr\ k \neq dir.L$
  proof −
    fix *k* **assume** *dk*: $d\ k \neq 0$ **and** *nk0*: $n1\ k = 0$
    have $(\lambda k.\ ts1\ k\ (n1\ k))\ k = le\_tm\ M$ **using** *le0 dk nk0 c1_eq* **by** *simp*
    from *valid_mttm_deltaLE*[*OF vM tr this*] **show** $dr\ k \neq dir.L$ **by** *auto*
  qed
  have *pos_sync*: $\bigwedge k.\ go\_dir\ (dr\ k)\ (n2\ k) = go\_dir\ (dr\ k)\ (n1\ k) + d\ k$
  proof −
    fix *k*
    show $go\_dir\ (dr\ k)\ (n2\ k) = go\_dir\ (dr\ k)\ (n1\ k) + d\ k$

```
    proof (cases dr k)
      case N thus ?thesis using pos2 by simp
    next
      case R thus ?thesis using pos2 by simp
    next
      case L
      show ?thesis
      proof (cases d k = 0)
        case True thus ?thesis using L pos2 by simp
      next
        case False
        hence n1 k ≠ 0 using no_L_at0 L by blast
        then obtain m where n1 k = Suc m by (cases n1 k) auto
        thus ?thesis using L pos2 by simp
      qed
    qed
  qed
  have tape_sync:
    ⋀k p. ((ts2 k)(n2 k := a k)) (p + d k) = ((ts1 k)(n1 k := a k)) p
  proof −
    fix k p
    show ((ts2 k)(n2 k := a k)) (p + d k) = ((ts1 k)(n1 k := a k)) p
    proof (cases p = n1 k)
      case True
      hence p + d k = n2 k using pos2 by simp
      thus ?thesis using True by simp
    next
      case False
      hence p + d k ≠ n2 k using pos2 by simp
      thus ?thesis using False tape2 by simp
    qed
  qed
  have rel': shift_rel d c1' ?c2'
    unfolding shift_rel_def c1'_eq using pos_sync tape_sync by simp
  from step2 rel' show ?thesis by blast
qed
```

Reverse: a step of M from the shifted config *c2* descends to a step from the base config *c1*. The transition witnessing the *c2* step reads exactly what *c1*'s head reads (the shift relation), so the same transition fires from *c1*; the boundary argument is identical (*le* at the base origin forbids the desyncing leftward move).

```
lemma mttm_step_shift_reverse:
  fixes M :: ('q, 'a) mttm
  assumes vM:   valid_mttm M
    and le0:    ∀k. d k ≠ 0 ⟶ mt_tape c1 k 0 = le_tm M
    and rel:    shift_rel d c1 c2
    and step:   (c2, c2') ∈ mttm_step (delta_tm M)
  shows ∃c1'. (c1, c1') ∈ mttm_step (delta_tm M) ∧ shift_rel d c1' c2'
```

```
proof −
  obtain q ts1 n1 where c1_eq: c1 = Config_M q ts1 n1
    by (cases c1) auto
  from step obtain sc ts2 n2 q' a dr where
      c2_eq:  c2 = Config_M sc ts2 n2
    and c2'_eq: c2' = Config_M q' (λk. (ts2 k)(n2 k := a k))
                                  (λk. go_dir (dr k) (n2 k))
    and tr:   (sc, λk. ts2 k (n2 k), q', a, dr) ∈ delta_tm M
    by (auto elim: mttm_step.cases)
  have unpacked:
    sc = q ∧ (∀k. n2 k = n1 k + d k) ∧ (∀k p. ts2 k (p + d k) = ts1 k p)
    using rel by (simp add: shift_rel_def c1_eq c2_eq)
  from unpacked have st2:  sc = q by simp
  from unpacked have pos2: ⋀k. n2 k = n1 k + d k by simp
  from unpacked have tape2: ⋀k p. ts2 k (p + d k) = ts1 k p by simp
  — The witnessing transition reads what the base head reads.
  have read_eq: (λk. ts2 k (n2 k)) = (λk. ts1 k (n1 k))
  proof
    fix k
    have ts2 k (n2 k) = ts2 k (n1 k + d k) using pos2 by simp
    also have ... = ts1 k (n1 k) using tape2 by simp
    finally show ts2 k (n2 k) = ts1 k (n1 k) .
  qed
  have tr1: (q, λk. ts1 k (n1 k), q', a, dr) ∈ delta_tm M
    using tr read_eq st2 by simp
  let ?c1' = Config_M q' (λk. (ts1 k)(n1 k := a k))
                         (λk. go_dir (dr k) (n1 k))
  have step1: (c1, ?c1') ∈ mttm_step (delta_tm M)
    unfolding c1_eq by (rule mttm_step.step, rule tr1)
  have no_L_at0: ⋀k. d k ≠ 0 ⟹ n1 k = 0 ⟹ dr k ≠ dir.L
  proof −
    fix k assume dk: d k ≠ 0 and nk0: n1 k = 0
    have (λk. ts1 k (n1 k)) k = le_tm M using le0 dk nk0 c1_eq by simp
    from valid_mttm_deltaLE[OF vM tr1 this] show dr k ≠ dir.L by auto
  qed
  have pos_sync: ⋀k. go_dir (dr k) (n2 k) = go_dir (dr k) (n1 k) + d k
  proof −
    fix k
    show go_dir (dr k) (n2 k) = go_dir (dr k) (n1 k) + d k
    proof (cases dr k)
      case N thus ?thesis using pos2 by simp
    next
      case R thus ?thesis using pos2 by simp
    next
      case L
      show ?thesis
      proof (cases d k = 0)
        case True thus ?thesis using L pos2 by simp
      next
```

```
      case False
      hence n1 k ≠ 0 using no_L_at0 L by blast
      then obtain m where n1 k = Suc m by (cases n1 k) auto
      thus ?thesis using L pos2 by simp
    qed
  qed
 qed
 have tape_sync:
  ⋀k p. ((ts2 k)(n2 k := a k)) (p + d k) = ((ts1 k)(n1 k := a k)) p
 proof −
  fix k p
  show ((ts2 k)(n2 k := a k)) (p + d k) = ((ts1 k)(n1 k := a k)) p
  proof (cases p = n1 k)
   case True
   hence p + d k = n2 k using pos2 by simp
   thus ?thesis using True by simp
  next
   case False
   hence p + d k ≠ n2 k using pos2 by simp
   thus ?thesis using False tape2 by simp
  qed
 qed
 have rel': shift_rel d ?c1' c2'
  unfolding shift_rel_def c2'_eq using pos_sync tape_sync by simp
 from step1 rel' show ?thesis by blast
qed
```

4.3 Path translation, both directions

Chaining the single-step lemmas along a path. The *le*-at-0 invariant is re-established at each intermediate configuration by *mttm_step_LE_pos0_preserve*, so the same base config hypothesis *le0* drives the whole chain. These translate *one* path of length n; there is no claim that the base and shifted machines have matching successor sets.

```
lemma mttm_relpow_shift_forward:
 fixes M :: ('q, 'a) mttm
 assumes vM: valid_mttm M
 shows ⟦ ∀ k. d k ≠ 0 ⟶ mt_tape c1 k 0 = le_tm M; shift_rel d c1 c2;
         (c1, c1') ∈ (mttm_step (delta_tm M)) ^^ n ⟧
       ⟹ ∃ c2'. (c2, c2') ∈ (mttm_step (delta_tm M)) ^^ n
              ∧ shift_rel d c1' c2'
proof (induction n arbitrary: c1 c2 c1')
 case 0
 then show ?case by auto
next
 case (Suc n)
 from Suc.prems(3) obtain cmid where
    first: (c1, cmid) ∈ mttm_step (delta_tm M)
```

```
    and rest: (cmid, c1') ∈ (mttm_step (delta_tm M)) ^^ n
    by (blast dest: relpow_Suc_D2)
  obtain cmid2 where
      first2:  (c2, cmid2) ∈ mttm_step (delta_tm M)
    and rel_mid: shift_rel d cmid cmid2
    using mttm_step_shift_forward[OF vM Suc.prems(1) Suc.prems(2) first] by
blast
  have le0_mid: ∀ k. d k ≠ 0 ⟶ mt_tape cmid k 0 = le_tm M
  proof (intro allI impI)
    fix k assume d k ≠ 0
    have mt_tape c1 k 0 = le_tm M using Suc.prems(1) ‹d k ≠ 0› by blast
    from mttm_step_LE_pos0_preserve[OF vM first this]
    show mt_tape cmid k 0 = le_tm M .
  qed
  from Suc.IH[OF le0_mid rel_mid rest] obtain c2' where
      tail2:   (cmid2, c2') ∈ (mttm_step (delta_tm M)) ^^ n
    and rel_end: shift_rel d c1' c2'
    by blast
  have (c2, c2') ∈ (mttm_step (delta_tm M)) ^^ (Suc n)
    using first2 tail2 by (rule relpow_Suc_I2)
  then show ?case using rel_end by blast
qed

lemma mttm_relpow_shift_reverse:
  fixes M :: ('q, 'a) mttm
  assumes vM: valid_mttm M
  shows ⟦ ∀ k. d k ≠ 0 ⟶ mt_tape c1 k 0 = le_tm M; shift_rel d c1 c2;
          (c2, c2') ∈ (mttm_step (delta_tm M)) ^^ n ⟧
        ⟹ ∃ c1'. (c1, c1') ∈ (mttm_step (delta_tm M)) ^^ n
                 ∧ shift_rel d c1' c2'
proof (induction n arbitrary: c1 c2 c2')
  case 0
  then show ?case by auto
next
  case (Suc n)
  from Suc.prems(3) obtain cmid2 where
      first2: (c2, cmid2) ∈ mttm_step (delta_tm M)
    and rest2: (cmid2, c2') ∈ (mttm_step (delta_tm M)) ^^ n
    by (blast dest: relpow_Suc_D2)
  obtain cmid where
      first:   (c1, cmid) ∈ mttm_step (delta_tm M)
    and rel_mid: shift_rel d cmid cmid2
    using mttm_step_shift_reverse[OF vM Suc.prems(1) Suc.prems(2) first2] by
blast
  have le0_mid: ∀ k. d k ≠ 0 ⟶ mt_tape cmid k 0 = le_tm M
  proof (intro allI impI)
    fix k assume d k ≠ 0
    have mt_tape c1 k 0 = le_tm M using Suc.prems(1) ‹d k ≠ 0› by blast
    from mttm_step_LE_pos0_preserve[OF vM first this]
```

```
    show mt_tape cmid k 0 = le_tm M .
  qed
  from Suc.IH[OF le0_mid rel_mid rest2] obtain c1' where
      tail:    (cmid, c1') ∈ (mttm_step (delta_tm M)) ^^ n
    and rel_end: shift_rel d c1' c2'
    by blast
  have (c1, c1') ∈ (mttm_step (delta_tm M)) ^^ (Suc n)
    using first tail by (rule relpow_Suc_I2)
  then show ?case using rel_end by blast
qed
```

4.4 Acceptance transfers across the shift, both directions

The consumer-facing interface. A run reaching a nominated state *qa* (the accept state, in use) in n steps from the base config yields one of the *same length* reaching the same state from the shifted config, and conversely. Length preservation carries the time bound; state preservation (*shift_rel_state*) carries acceptance. The two directions give language inclusion each way — exactly what a nondeterministic machine admits, with no bisimulation claim. These *relpow* (fixed-length) forms feed *accepts_in_time_mttm* directly; a caller working with *Lang_mttm* unfolds *rtrancl_power* to a fixed length first, then applies them.

```
lemma shift_reach_state_forward:
  fixes M :: ('q, 'a) mttm
  assumes vM:  valid_mttm M
    and le0:   ∀k. d k ≠ 0 ⟶ mt_tape c1 k 0 = le_tm M
    and rel:   shift_rel d c1 c2
    and run:   (c1, ca) ∈ (mttm_step (delta_tm M)) ^^ n
    and acc:   mt_state ca = qa
  shows ∃cb. (c2, cb) ∈ (mttm_step (delta_tm M)) ^^ n ∧ mt_state cb = qa
proof −
  from mttm_relpow_shift_forward[OF vM le0 rel run] obtain cb where
      run2: (c2, cb) ∈ (mttm_step (delta_tm M)) ^^ n
    and rel2: shift_rel d ca cb by blast
  have mt_state cb = mt_state ca using rel2 by (rule shift_rel_state)
  thus ?thesis using run2 acc by auto
qed

lemma shift_reach_state_reverse:
  fixes M :: ('q, 'a) mttm
  assumes vM:  valid_mttm M
    and le0:   ∀k. d k ≠ 0 ⟶ mt_tape c1 k 0 = le_tm M
    and rel:   shift_rel d c1 c2
    and run:   (c2, cb) ∈ (mttm_step (delta_tm M)) ^^ n
    and acc:   mt_state cb = qa
  shows ∃ca. (c1, ca) ∈ (mttm_step (delta_tm M)) ^^ n ∧ mt_state ca = qa
proof −
  from mttm_relpow_shift_reverse[OF vM le0 rel run] obtain ca where
```

```
    run1: (c1, ca) ∈ (mttm_step (delta_tm M)) ^^ n
   and rel2: shift_rel d ca cb by blast
  have mt_state cb = mt_state ca using rel2 by (rule shift_rel_state)
  thus ?thesis using run1 acc by auto
qed

end
theory Multitape_Finite_Control
  imports Multitape_Substrate
begin
```

5 Finite-control small-input recogniser

A reusable substrate construction for the ubiquitous low-level move: finitely many bounded-length inputs are decided directly by the finite control, reading the input in a fixed number of moves; only longer inputs need run a real machine.

This theory provides the atom *finite recogniser*: given a finite input alphabet *Sg*, a concrete tape alphabet *Gamma* with a blank and a left endmarker, a finite set F of words over *Sg*, and a tape count k, it builds a deterministic machine that decides F — accepting each member reading only its input. The combinator *finite patch*, which reuses the recogniser on short inputs and hands off to an arbitrary machine on long inputs, is built on top.

The construction is read-only: it never modifies the tape (the write component of every transition equals its read component), so the left-endmarker write discipline is vacuous and only the head movement carries the left-endmarker read discipline.

5.1 Recogniser state space

Three phases: *FR_Scan u* has read the prefix u of the input so far, *FR_Acc* has accepted, *FR_Rej* has rejected. The scan payload ranges over words of length at most the recogniser's cutoff, so the state set is finite.

```
datatype 'a fr_state = FR_Scan 'a list | FR_Acc | FR_Rej
```

5.2 Recogniser transition components

The read/write tuple for a step reading symbol x on the input tape *0*: tape *0* carries x, every other active tape (index $1 \le j < k$) reads the left endmarker *le* (its head never leaves position *0*), and every inactive tape (index $j \ge k$) reads the blank *bl* (the support invariant).

```
definition fr_read :: 'a ⇒ 'a ⇒ nat ⇒ 'a ⇒ (nat ⇒ 'a) where
  fr_read bl le k x = (λj. if j = 0 then x else if j < k then le else bl)
```

The head-move tuple: tape *0* moves by d, every other tape stays put. This meets the support invariant (N beyond k) and, paired with *fr_read*, the left-endmarker read discipline.

```
definition fr_move :: dir ⇒ (nat ⇒ dir) where
  fr_move d = (λj. if j = 0 then d else dir.N)
```

5.3 Recogniser cutoff and transition relation

The scan cutoff: an upper bound on the length of every word in F (the maximum length, or *0* when F is empty). Inputs longer than the cutoff cannot be in F, so the scan rejects them on the overflow read.

```
definition fr_N :: 'a list set ⇒ nat where
  fr_N F = (if F = {} then 0 else Max (length ` F))
```

The recogniser transition relation. Five families, all with write component equal to read component:

- advance past the left endmarker (start step);
- extend the scanned prefix by one symbol while below the cutoff;
- overflow: reading a symbol at the cutoff rejects (input too long);
- end-of-input on a member prefix accepts;
- end-of-input on a non-member prefix rejects.

```
definition fr_delta ::
  'a set ⇒ 'a ⇒ 'a ⇒ 'a list set ⇒ nat ⇒ nat
    ⇒ ('a fr_state × (nat ⇒ 'a) × 'a fr_state
        × (nat ⇒ 'a) × (nat ⇒ dir)) set
where
  fr_delta Sg bl le F k N =
    { (FR_Scan [], fr_read bl le k le, FR_Scan [],
       fr_read bl le k le, fr_move dir.R) }
    ∪ { (FR_Scan u, fr_read bl le k x, FR_Scan (u @ [x]),
         fr_read bl le k x, fr_move dir.R)
        | u x. set u ⊆ Sg ∧ length u < N ∧ x ∈ Sg }
    ∪ { (FR_Scan u, fr_read bl le k x, FR_Rej,
         fr_read bl le k x, fr_move dir.N)
        | u x. set u ⊆ Sg ∧ length u = N ∧ x ∈ Sg }
    ∪ { (FR_Scan u, fr_read bl le k bl, FR_Acc,
         fr_read bl le k bl, fr_move dir.N)
        | u. set u ⊆ Sg ∧ length u ≤ N ∧ u ∈ F }
    ∪ { (FR_Scan u, fr_read bl le k bl, FR_Rej,
         fr_read bl le k bl, fr_move dir.N)
        | u. set u ⊆ Sg ∧ length u ≤ N ∧ u ∉ F }
```

5.4 The recogniser machine

The finite recogniser over input alphabet *Sg*, tape alphabet *Gamma*, blank *bl*, left endmarker *le*, word set *F*, and tape count *k*. Its state set is the scannable prefixes (words over *Sg* up to the cutoff) plus the two halting states; the start state is the empty scan.

definition *finite_recogniser* ::
  'a set ⇒ 'a set ⇒ 'a ⇒ 'a ⇒ 'a list set ⇒ nat
    ⇒ ('a fr_state, 'a) mttm
where
  finite_recogniser Sg Gamma bl le F k =
    MTTM
      (FR_Scan ' {u. set u ⊆ Sg ∧ length u ≤ fr_N F} ∪ {FR_Acc, FR_Rej})
      Sg
      Gamma
      bl
      le
      (fr_delta Sg bl le F k (fr_N F))
      (FR_Scan [])
      FR_Acc
      FR_Rej
      k

5.5 Component accessors

lemma *finite_recogniser_k_tm*:
  k_tm (finite_recogniser Sg Gamma bl le F k) = k
  by *(simp add: finite_recogniser_def)*

lemma *finite_recogniser_Sigma_tm*:
  Sigma_tm (finite_recogniser Sg Gamma bl le F k) = Sg
  by *(simp add: finite_recogniser_def)*

lemma *finite_recogniser_Gamma_tm*:
  Γ_tm (finite_recogniser Sg Gamma bl le F k) = Gamma
  by *(simp add: finite_recogniser_def)*

lemma *finite_recogniser_bl_tm*:
  bl_tm (finite_recogniser Sg Gamma bl le F k) = bl
  by *(simp add: finite_recogniser_def)*

lemma *finite_recogniser_le_tm*:
  le_tm (finite_recogniser Sg Gamma bl le F k) = le
  by *(simp add: finite_recogniser_def)*

lemma *finite_recogniser_s_tm*:
  s_tm (finite_recogniser Sg Gamma bl le F k) = FR_Scan []
  by *(simp add: finite_recogniser_def)*

lemma *finite_recogniser_t_tm*:
t_tm (finite_recogniser Sg Gamma bl le F k) = FR_Acc
by (*simp add*: *finite_recogniser_def*)

lemma *finite_recogniser_r_tm*:
r_tm (finite_recogniser Sg Gamma bl le F k) = FR_Rej
by (*simp add*: *finite_recogniser_def*)

lemma *finite_recogniser_Q_tm*:
Q_tm (finite_recogniser Sg Gamma bl le F k)
= FR_Scan ‘ {u. set u ⊆ Sg ∧ length u ≤ fr_N F} ∪ {FR_Acc, FR_Rej}
by (*simp add*: *finite_recogniser_def*)

lemma *finite_recogniser_delta_tm*:
delta_tm (finite_recogniser Sg Gamma bl le F k)
= fr_delta Sg bl le F k (fr_N F)
by (*simp add*: *finite_recogniser_def*)

5.6 Transition-relation discipline

Every recogniser transition writes back exactly what it reads: the recogniser never modifies the tape. This makes the left-endmarker *write* discipline (*le_unique*) vacuous.

lemma *fr_delta_no_write*:
(q, a, q′, a′, d) ∈ fr_delta Sg bl le F k N ⟹ a′ = a
by (*auto simp*: *fr_delta_def*)

The read tuple is injective in the tape-*0* symbol (its value there), so a transition's read component determines the symbol scanned. This is the discriminator behind determinism.

lemma *fr_read_inj*:
fr_read bl le k x = fr_read bl le k y ⟹ x = y
proof −
assume *fr_read bl le k x = fr_read bl le k y*
hence *fr_read bl le k x 0 = fr_read bl le k y 0* **by** *simp*
thus *x = y* **by** (*simp add*: *fr_read_def*)
qed

The support invariant: at every inactive tape index $j \geq k$, each transition reads and writes the blank and stays put.

lemma *fr_delta_support*:
assumes *0 < k*
and *(q, a, q′, a′, d) ∈ fr_delta Sg bl le F k N*
and *k ≤ j*
shows *a j = bl ∧ a′ j = bl ∧ d j = dir.N*
using *assms* **by** (*auto simp*: *fr_delta_def fr_read_def fr_move_def*)

No recogniser transition ever moves left: the input tape moves R or stays, and every other tape stays put.

lemma *fr_delta_move_no_L*:
  $(q,\ a,\ q',\ a',\ d) \in$ *fr_delta Sg bl le F k N* $\Longrightarrow$ *d j* $\in \{$*dir.N*, *dir.R*$\}$
  by (*auto simp*: *fr_delta_def fr_move_def split*: *if_splits*)

The left-endmarker read discipline: a transition reading *le* on tape j writes *le* back there (read-only) and moves only N or R (never-left). Both halves fall out of the two facts above, needing no alphabet side-conditions.

lemma *fr_delta_LE*:
  assumes $(q,\ a,\ q',\ a',\ d) \in$ *fr_delta Sg bl le F k N*
    and *a j* $=$ *le*
  shows *a′ j* $=$ *le* $\wedge$ *d j* $\in \{$*dir.N*, *dir.R*$\}$
   using *fr_delta_no_write*[*OF assms*(*1*)] *fr_delta_move_no_L*[*OF assms*(*1*)] *assms*(*2*)
  by *simp*

5.7 Well-formedness

The recogniser is a valid substrate machine. The alphabet must be finite with the blank and left endmarker inside *Gamma* but outside *Sg*, and the tape count positive — the standard machine hypotheses, exactly the data a concrete machine (or the combinator *finite_patch* built on this one) already carries.

lemma *finite_recogniser_valid*:
  assumes *finG*: *finite Gamma*
    and *SgG*: *Sg* $\subseteq$ *Gamma*
    and *blG*: *bl* $\in$ *Gamma*
    and *blS*: *bl* $\notin$ *Sg*
    and *leG*: *le* $\in$ *Gamma*
    and *leS*: *le* $\notin$ *Sg*
    and *kpos*: *0* $<$ *k*
  shows *valid_mttm* (*finite_recogniser Sg Gamma bl le F k*)
proof −
  have *finSg*: *finite Sg* **using** *finite_subset*[*OF SgG finG*] **.**
  have *finP*: *finite* $\{$*u. set u* $\subseteq$ *Sg* $\wedge$ *length u* $\leq$ *fr_N F*$\}$
    using *finite_lists_length_le*[*OF finSg*] **by** *blast*
  have *finQ*: *finite* (*FR_Scan* ‘ $\{$*u. set u* $\subseteq$ *Sg* $\wedge$ *length u* $\leq$ *fr_N F*$\}$
                  $\cup$ $\{$*FR_Acc*, *FR_Rej*$\}$)
    using *finP* **by** *simp*
  have *shape*: *fr_delta Sg bl le F k* (*fr_N F*)
     $\subseteq$ (*FR_Scan* ‘ $\{$*u. set u* $\subseteq$ *Sg* $\wedge$ *length u* $\leq$ *fr_N F*$\}$ $\cup$ $\{$*FR_Acc*, *FR_Rej*$\}$
       − $\{$*FR_Acc*, *FR_Rej*$\}$)
      $\times$ (*UNIV* $\to$ *Gamma*)
     $\times$ (*FR_Scan* ‘ $\{$*u. set u* $\subseteq$ *Sg* $\wedge$ *length u* $\leq$ *fr_N F*$\}$ $\cup$ $\{$*FR_Acc*, *FR_Rej*$\}$)
      $\times$ (*UNIV* $\to$ *Gamma*)
      $\times$ (*UNIV* $\to$ *UNIV*)
    unfolding *fr_delta_def fr_read_def*
    using *SgG blG leG* **by** (*auto simp*: *Pi_iff*)
  have *LEc*: $\forall$ *q a q′ a′ d j*.

```
    (q, a, q', a', d) ∈ fr_delta Sg bl le F k (fr_N F)
      ⟶ a j = le ⟶ a' j = le ∧ d j ∈ {dir.N, dir.R}
  by (auto dest: fr_delta_LE)
 have suppc: ∀ q a q' a' d.
    (q, a, q', a', d) ∈ fr_delta Sg bl le F k (fr_N F)
      ⟶ (∀ j ≥ k. a j = bl ∧ a' j = bl ∧ d j = dir.N)
  by (auto dest: fr_delta_support[OF kpos])
 show ?thesis
  unfolding finite_recogniser_def valid_mttm.simps
  using finQ finG SgG blG blS leG leS kpos shape LEc suppc
  by (intro conjI) auto
qed
```

The recogniser is well-formed in the strengthened sense (*well_formed_mttm*): distinct start / accept / reject and distinct endmarker / blank, plus the left-endmarker write discipline — the last free from read-only-ness.

```
lemma finite_recogniser_well_formed:
 assumes finG: finite Gamma
   and SgG:  Sg ⊆ Gamma
   and blG:  bl ∈ Gamma
   and blS:  bl ∉ Sg
   and leG:  le ∈ Gamma
   and leS:  le ∉ Sg
   and blle: bl ≠ le
   and kpos: 0 < k
 shows well_formed_mttm (finite_recogniser Sg Gamma bl le F k)
proof −
 have lu: le_unique (finite_recogniser Sg Gamma bl le F k)
  unfolding le_unique_def finite_recogniser_delta_tm
  by (auto dest: fr_delta_no_write)
 have vd: valid_mttm (finite_recogniser Sg Gamma bl le F k)
  by (rule finite_recogniser_valid[OF finG SgG blG blS leG leS kpos])
 show ?thesis
  using vd lu blle
  by (auto simp: finite_recogniser_s_tm finite_recogniser_t_tm
                 finite_recogniser_r_tm finite_recogniser_le_tm
                 finite_recogniser_bl_tm)
qed
```

5.8 Determinism

The recogniser is deterministic. The source state fixes the scanned prefix (constructor injectivity) and the read component fixes the scanned symbol (*fr_read ?bl ?le ?k ?x = fr_read ?bl ?le ?k ?y ⟹ ?x = ?y*); the five transition families are then pairwise disjoint because the left endmarker, the blank, and the input alphabet are pairwise distinct.

lemma *finite_recogniser_det*:

assumes *leS*: *le* ∉ *Sg* **and** *blS*: *bl* ∉ *Sg* **and** *blle*: *bl* ≠ *le*
shows *det_mttm* (*finite_recogniser Sg Gamma bl le F k*)
unfolding *det_mttm_def finite_recogniser_delta_tm fr_delta_def*
using *leS blS blle* **by** (*auto dest*: *fr_read_inj*)

5.9 The accepting run

The configuration after the recogniser has read the left endmarker and the first m input symbols: state *FR_Scan* (*take m w*), the (read-only, hence unchanging) input tape, and the input head at position $m + 1$.

definition *fr_run_config* ::
'a ⇒ *'a* ⇒ *nat* ⇒ *'a list* ⇒ *nat* ⇒ (*'a*, *'a fr_state*) *mt_config*
where
fr_run_config bl le k w m =
Config$_M$ (*FR_Scan* (*take m w*))
(λ*i n*. *if i* < *k*
then (*if n* = *0 then le*
else if i = *0* ∧ *n* ≤ *length w then w* ! (*n* − *1*) *else bl*)
else bl)
(λ*i*. *if i* = *0 then m* + *1 else 0*)

One scan step: from the length-m scan configuration, reading the $(m+1)$-th input symbol advances to the length-(*Suc m*) one, provided the input still has a symbol there ($m <$ *length w*) and the cutoff is not yet reached ($m <$ *fr_N F*).

lemma *fr_scan_step*:
assumes *kpos*: *0* < *k* **and** *mlt*: *m* < *length w* **and** *mN*: *m* < *fr_N F*
and *wSg*: *set w* ⊆ *Sg*
shows (*fr_run_config bl le k w m*, *fr_run_config bl le k w* (*Suc m*))
∈ *mttm_step* (*fr_delta Sg bl le F k* (*fr_N F*))
proof −
let *?x* = *w* ! *m*
let *?ts* = λ*i n*. *if i* < *k*
then (*if n* = *0 then le*
else if i = *0* ∧ *n* ≤ *length w then w* ! (*n* − *1*) *else bl*)
else bl
let *?nn* = λ*i*::*nat*. *if i* = *0 then m* + *1 else 0*
have *xSg*: *?x* ∈ *Sg* **using** *nth_mem*[*OF mlt*] *wSg* **by** *blast*
have *lenu*: *length* (*take m w*) = *m* **using** *mlt* **by** *simp*
have *takeq*: *take* (*Suc m*) *w* = *take m w* @ [*?x*]
using *mlt* **by** (*simp add*: *take_Suc_conv_app_nth*)
have *setu*: *set* (*take m w*) ⊆ *Sg*
using *wSg* **by** (*meson set_take_subset subset_trans*)
have *read_eq*: (λ*i*. *?ts i* (*?nn i*)) = *fr_read bl le k ?x*
proof (*rule ext*)
fix *i* **show** *?ts i* (*?nn i*) = *fr_read bl le k ?x i*
using *kpos mlt* **by** (*cases i* = *0*) (*auto simp*: *fr_read_def*)
qed

```
  have tr: (FR_Scan (take m w), fr_read bl le k ?x, FR_Scan (take (Suc m) w),
          fr_read bl le k ?x, fr_move dir.R) ∈ fr_delta Sg bl le F k (fr_N F)
    unfolding fr_delta_def takeq using setu lenu mN xSg by auto
  have src: fr_run_config bl le k w m = Config_M (FR_Scan (take m w)) ?ts ?nn
    by (simp add: fr_run_config_def)
  have step: (Config_M (FR_Scan (take m w)) ?ts ?nn,
              Config_M (FR_Scan (take (Suc m) w))
                (λi. (?ts i)(?nn i := fr_read bl le k ?x i))
                (λi. go_dir (fr_move dir.R i) (?nn i)))
             ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
  proof (rule mttm_step.step)
    show (FR_Scan (take m w), λi. ?ts i (?nn i), FR_Scan (take (Suc m) w),
          fr_read bl le k ?x, fr_move dir.R) ∈ fr_delta Sg bl le F k (fr_N F)
      using tr by (simp add: read_eq)
  qed
  have tgt: fr_run_config bl le k w (Suc m)
            = Config_M (FR_Scan (take (Suc m) w))
                (λi. (?ts i)(?nn i := fr_read bl le k ?x i))
                (λi. go_dir (fr_move dir.R i) (?nn i))
  proof −
    have tape: (λi. (?ts i)(?nn i := fr_read bl le k ?x i)) = ?ts
    proof (rule ext)
      fix i
      have eq: fr_read bl le k ?x i = ?ts i (?nn i)
        using kpos mlt by (cases i = 0) (auto simp: fr_read_def)
      show (?ts i)(?nn i := fr_read bl le k ?x i) = ?ts i
        unfolding eq by (rule fun_upd_triv)
    qed
    have head: (λi. go_dir (fr_move dir.R i) (?nn i))
                 = (λi::nat. if i = 0 then Suc m + 1 else 0)
    proof (rule ext)
      fix i show go_dir (fr_move dir.R i) (?nn i) = (if i = 0 then Suc m + 1 else
0)
        by (cases i = 0) (simp_all add: fr_move_def)
    qed
    show ?thesis
      by (simp add: fr_run_config_def tape head)
  qed
  show ?thesis unfolding src tgt by (rule step)
qed
```

Every word in F is no longer than the cutoff.

```
lemma fr_length_le_N:
  assumes w ∈ F and finite F
  shows length w ≤ fr_N F
proof −
  have F ≠ {} using assms(1) by auto
  hence fr_N F = Max (length ‘ F) by (simp add: fr_N_def)
  moreover have length w ∈ length ‘ F using assms(1) by simp
```

```
  ultimately show ?thesis using assms(2) by simp
qed
```

The start step: from the initial configuration, reading the left endmarker advances into the length-*0* scan configuration.

```
lemma fr_le_step:
  assumes kpos: 0 < k
  shows (init_config_mttm (finite_recogniser Sg Gamma bl le F k) w,
         fr_run_config bl le k w 0)
          ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
proof −
  let ?ts = λi n. if i < k
                  then (if n = 0 then le
                        else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
                  else bl
  have read_eq: (λi. ?ts i ((λ_. 0) i)) = fr_read bl le k le
  proof (rule ext)
    fix i show ?ts i ((λ_. 0) i) = fr_read bl le k le i
      using kpos by (cases i = 0) (auto simp: fr_read_def)
  qed
  have tr: (FR_Scan [], fr_read bl le k le, FR_Scan [],
            fr_read bl le k le, fr_move dir.R) ∈ fr_delta Sg bl le F k (fr_N F)
    unfolding fr_delta_def by auto
  have init_eq: init_config_mttm (finite_recogniser Sg Gamma bl le F k) w
                  = Config_M (FR_Scan []) ?ts (λ_. 0)
    by (simp add: finite_recogniser_def)
  have step: (Config_M (FR_Scan []) ?ts (λ_. 0),
              Config_M (FR_Scan [])
                (λi. (?ts i)((λ_. 0) i := fr_read bl le k le i))
                (λi. go_dir (fr_move dir.R i) ((λ_. 0) i)))
             ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
  proof (rule mttm_step.step)
    show (FR_Scan [], λi. ?ts i ((λ_. 0) i), FR_Scan [],
          fr_read bl le k le, fr_move dir.R) ∈ fr_delta Sg bl le F k (fr_N F)
      using tr by (simp add: read_eq)
  qed
  have tgt: fr_run_config bl le k w 0
              = Config_M (FR_Scan [])
                  (λi. (?ts i)((λ_. 0) i := fr_read bl le k le i))
                  (λi. go_dir (fr_move dir.R i) ((λ_. 0) i))
  proof −
    have tape: (λi. (?ts i)((λ_. 0) i := fr_read bl le k le i)) = ?ts
    proof (rule ext)
      fix i
      have eq: fr_read bl le k le i = ?ts i ((λ_. 0) i)
        using kpos by (cases i = 0) (auto simp: fr_read_def)
      show (?ts i)((λ_. 0) i := fr_read bl le k le i) = ?ts i
        unfolding eq by (rule fun_upd_triv)
    qed
```

```
  have head: (λi. go_dir (fr_move dir.R i) ((λ_. 0) i))
                = (λi::nat. if i = 0 then 0 + 1 else 0)
  proof (rule ext)
    fix i show go_dir (fr_move dir.R i) ((λ_. 0) i) = (if i = 0 then 0 + 1 else
0)
      by (cases i = 0) (simp_all add: fr_move_def)
  qed
  show ?thesis by (simp add: fr_run_config_def tape head)
 qed
 show ?thesis unfolding init_eq tgt by (rule step)
qed
```

The accept step: from the fully-scanned configuration (prefix w), reading the end-of-input blank accepts, provided $w \in F$.

```
lemma fr_accept_step:
 assumes wF: w ∈ F and wSg: set w ⊆ Sg and finF: finite F
 shows (fr_run_config bl le k w (length w),
        Config_M FR_Acc
         (λi n. if i < k
                then (if n = 0 then le
                      else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
                else bl)
         (λi::nat. if i = 0 then length w + 1 else 0))
        ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
proof −
 let ?ts = λi n. if i < k
                 then (if n = 0 then le
                       else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
                 else bl
 let ?nn = λi::nat. if i = 0 then length w + 1 else 0
 have lenN: length w ≤ fr_N F using fr_length_le_N[OF wF finF] .
 have read_eq: (λi. ?ts i (?nn i)) = fr_read bl le k bl
 proof (rule ext)
   fix i show ?ts i (?nn i) = fr_read bl le k bl i
     by (cases i = 0) (auto simp: fr_read_def)
 qed
 have tr: (FR_Scan w, fr_read bl le k bl, FR_Acc,
           fr_read bl le k bl, fr_move dir.N) ∈ fr_delta Sg bl le F k (fr_N F)
   unfolding fr_delta_def using wSg lenN wF by auto
 have src: fr_run_config bl le k w (length w) = Config_M (FR_Scan w) ?ts ?nn
   by (simp add: fr_run_config_def)
 have step: (Config_M (FR_Scan w) ?ts ?nn,
             Config_M FR_Acc
               (λi. (?ts i)(?nn i := fr_read bl le k bl i))
               (λi. go_dir (fr_move dir.N i) (?nn i)))
            ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
 proof (rule mttm_step.step)
   show (FR_Scan w, λi. ?ts i (?nn i), FR_Acc,
         fr_read bl le k bl, fr_move dir.N) ∈ fr_delta Sg bl le F k (fr_N F)
```

```
    using tr by (simp add: read_eq)
  qed
  have tape: (λi. (?ts i)(?nn i := fr_read bl le k bl i)) = ?ts
  proof (rule ext)
    fix i
    have eq: fr_read bl le k bl i = ?ts i (?nn i)
      by (cases i = 0) (auto simp: fr_read_def)
    show (?ts i)(?nn i := fr_read bl le k bl i) = ?ts i
      unfolding eq by (rule fun_upd_triv)
  qed
  have head: (λi. go_dir (fr_move dir.N i) (?nn i)) = ?nn
    by (rule ext) (simp add: fr_move_def)
  show ?thesis
    unfolding src using step by (simp add: tape head)
qed
```

The scan phase: iterating the scan step reads all of w, reaching the fully-scanned configuration in *length w* steps. Assembled by the substrate's bounded-iteration combinator *relpow_invariant_chain*.

```
lemma fr_scan_chain:
  assumes kpos: 0 < k and wF: w ∈ F and wSg: set w ⊆ Sg and finF: finite F
  shows (fr_run_config bl le k w 0, fr_run_config bl le k w (length w))
        ∈ (mttm_step (fr_delta Sg bl le F k (fr_N F))) ^^ (length w)
proof −
  have lenN: length w ≤ fr_N F using fr_length_le_N[OF wF finF] .
  have ∃ c'. (fr_run_config bl le k w 0, c')
             ∈ (mttm_step (fr_delta Sg bl le F k (fr_N F))) ^^ (length w)
           ∧ c' = fr_run_config bl le k w (length w)
  proof (rule relpow_invariant_chain[where P = λi c. c = fr_run_config bl le k
w i])
    fix i c assume ilt: i < length w and Pi: c = fr_run_config bl le k w i
    have iN: i < fr_N F using ilt lenN by linarith
    have (fr_run_config bl le k w i, fr_run_config bl le k w (Suc i))
         ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
      by (rule fr_scan_step[OF kpos ilt iN wSg])
    thus ∃ c'. (c, c') ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
               ∧ c' = fr_run_config bl le k w (Suc i)
      using Pi by blast
  next
    show fr_run_config bl le k w 0 = fr_run_config bl le k w 0 by (rule refl)
  qed
  thus ?thesis by blast
qed
```

The full accepting run: for a member w, the recogniser reaches its accept state from the initial configuration in *length w + 2* steps — the start step, the *length w* scan steps, and the accept step.

```
lemma fr_accepting_run:
  assumes kpos: 0 < k and wF: w ∈ F and wSg: set w ⊆ Sg and finF: finite F
```

```
  shows ∃ c. (init_config_mttm (finite_recogniser Sg Gamma bl le F k) w, c)
             ∈ (mttm_step (fr_delta Sg bl le F k (fr_N F))) ^^ (length w + 2)
           ∧ mt_state c = FR_Acc
proof −
  let ?R = mttm_step (fr_delta Sg bl le F k (fr_N F))
  let ?init = init_config_mttm (finite_recogniser Sg Gamma bl le F k) w
  let ?acc = Config_M FR_Acc
               (λi n. if i < k
                      then (if n = 0 then le
                            else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
                      else bl)
               (λi::nat. if i = 0 then length w + 1 else 0)
  have le1: (?init, fr_run_config bl le k w 0) ∈ ?R ^^ 1
    using fr_le_step[OF kpos] by simp
  have sc: (fr_run_config bl le k w 0, fr_run_config bl le k w (length w)) ∈ ?R
^^ (length w)
    using fr_scan_chain[OF kpos wF wSg finF] .
  have ac: (fr_run_config bl le k w (length w), ?acc) ∈ ?R ^^ 1
    using fr_accept_step[OF wF wSg finF] by simp
  have t1: (?init, fr_run_config bl le k w (length w)) ∈ ?R ^^ (1 + length w)
    by (rule relpow_transI[OF le1 sc])
  have t2: (?init, ?acc) ∈ ?R ^^ (1 + length w + 1)
    by (rule relpow_transI[OF t1 ac])
 have (?init, ?acc) ∈ ?R ^^ (length w + 2) using t2 by (simp add: add.commute)
  moreover have mt_state ?acc = FR_Acc by simp
  ultimately show ?thesis by blast
qed
```

Consequently a member is accepted within *length w + 2* steps. (The extra *+2* over the classical *n+1* is the substrate's left endmarker read, which the textbook model has no counterpart for.)

```
lemma finite_recogniser_accepts:
  assumes kpos: 0 < k and wF: w ∈ F and wSg: set w ⊆ Sg and finF: finite F
  shows accepts_in_time_mttm (finite_recogniser Sg Gamma bl le F k) w (length
w + 2)
proof −
  obtain c where c: (init_config_mttm (finite_recogniser Sg Gamma bl le F k)
w, c)
                    ∈ (mttm_step (fr_delta Sg bl le F k (fr_N F))) ^^ (length w
+ 2)
          and cst: mt_state c = FR_Acc
    using fr_accepting_run[OF kpos wF wSg finF] by blast
  show ?thesis
       unfolding accepts_in_time_mttm_def finite_recogniser_delta_tm fi-
nite_recogniser_t_tm
    using c cst by blast
qed
```

Completeness half of the language characterisation: every member is accepted, hence in the language.

lemma *finite_recogniser_lang_complete*:
  assumes *kpos*: $0 < k$ **and** *Fsub*: $\forall\, w{\in}F.\ set\ w \subseteq Sg$ **and** *finF*: *finite F*
  shows $F \subseteq Lang\_mttm\ (finite\_recogniser\ Sg\ Gamma\ bl\ le\ F\ k)$
proof
  fix *w* **assume** *wF*: $w \in F$
  have *wSg*: $set\ w \subseteq Sg$ **using** *Fsub wF* **by** *blast*
  obtain *c* **where** *c*: $(init\_config\_mttm\ (finite\_recogniser\ Sg\ Gamma\ bl\ le\ F\ k)\ w,\ c)$
                    $\in (mttm\_step\ (fr\_delta\ Sg\ bl\ le\ F\ k\ (fr\_N\ F)))\ ^{\frown\frown}\ (length\ w + 2)$
          and *cst*: $mt\_state\ c = FR\_Acc$
    using *fr_accepting_run*[*OF kpos wF wSg finF*] **by** *blast*
  obtain $w'\ n$ **where** *c_eq*: $c = Config_M\ FR\_Acc\ w'\ n$
    using *cst* **by** (*cases c*) *auto*
  have *reach*: $(init\_config\_mttm\ (finite\_recogniser\ Sg\ Gamma\ bl\ le\ F\ k)\ w,$
              $Config_M\ FR\_Acc\ w'\ n)$
               $\in (mttm\_step\ (fr\_delta\ Sg\ bl\ le\ F\ k\ (fr\_N\ F)))^*$
    using *c c_eq* **by** (*metis relpow_imp_rtrancl*)
  show $w \in Lang\_mttm\ (finite\_recogniser\ Sg\ Gamma\ bl\ le\ F\ k)$
              unfolding *Lang_mttm_def finite_recogniser_Sigma_tm finite_recogniser_t_tm*
            finite_recogniser_delta_tm
    using *wSg reach* **by** *blast*
qed

5.10 Soundness of the run

Every recogniser transition, and hence every recogniser step, starts from a scan state — so the two halting states are sinks.

lemma *fr_delta_src_scan*:
  $(q,\ a,\ q',\ a',\ d) \in fr\_delta\ Sg\ bl\ le\ F\ k\ N \Longrightarrow \exists\, u.\ q = FR\_Scan\ u$
  by (*auto simp*: *fr_delta_def*)

lemma *fr_step_src_scan*:
  assumes $(c,\ c') \in mttm\_step\ (fr\_delta\ Sg\ bl\ le\ F\ k\ N)$
  shows $\exists\, u.\ mt\_state\ c = FR\_Scan\ u$
proof −
  from *assms* **obtain** *q ts n q' a d* **where**
      c_eq: $c = Config_M\ q\ ts\ n$
    and *tr*: $(q,\ \lambda k.\ ts\ k\ (n\ k),\ q',\ a,\ d) \in fr\_delta\ Sg\ bl\ le\ F\ k\ N$
    by (*auto elim*: *mttm_step.cases*)
  from *fr_delta_src_scan*[*OF tr*] **obtain** *u* **where** $q = FR\_Scan\ u$ **by** *blast*
  thus *?thesis* **using** *c_eq* **by** *auto*
qed

The overflow step: at the cutoff ($m = fr\_N\ F$) with input still remaining ($m < length\ w$), reading the next symbol rejects.

lemma *fr_overflow_step*:
  assumes *kpos*: $0 < k$ **and** *mlt*: $m < length\ w$ **and** *mN*: $m = fr\_N\ F$

```
  and wSg: set w ⊆ Sg
 shows ∃ c'. (fr_run_config bl le k w m, c')
         ∈ mttm_step (fr_delta Sg bl le F k (fr_N F)) ∧ mt_state c' = FR_Rej
proof −
 let ?x = w ! m
 let ?ts = λi n. if i < k
                 then (if n = 0 then le
                       else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
                 else bl
 let ?nn = λi::nat. if i = 0 then m + 1 else 0
 let ?tgt = Config_M FR_Rej
             (λi. (?ts i)(?nn i := fr_read bl le k ?x i))
             (λi. go_dir (fr_move dir.N i) (?nn i))
 have xSg: ?x ∈ Sg using nth_mem[OF mlt] wSg by blast
 have lenu: length (take m w) = m using mlt by simp
 have setu: set (take m w) ⊆ Sg
  using wSg by (meson set_take_subset subset_trans)
 have read_eq: (λi. ?ts i (?nn i)) = fr_read bl le k ?x
 proof (rule ext)
  fix i show ?ts i (?nn i) = fr_read bl le k ?x i
   using kpos mlt by (cases i = 0) (auto simp: fr_read_def)
 qed
 have tr: (FR_Scan (take m w), fr_read bl le k ?x, FR_Rej,
           fr_read bl le k ?x, fr_move dir.N) ∈ fr_delta Sg bl le F k (fr_N F)
  unfolding fr_delta_def using setu lenu mN xSg by auto
 have src: fr_run_config bl le k w m = Config_M (FR_Scan (take m w)) ?ts ?nn
  by (simp add: fr_run_config_def)
 have step: (fr_run_config bl le k w m, ?tgt)
             ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
  unfolding src
 proof (rule mttm_step.step)
  show (FR_Scan (take m w), λi. ?ts i (?nn i), FR_Rej,
        fr_read bl le k ?x, fr_move dir.N) ∈ fr_delta Sg bl le F k (fr_N F)
   using tr by (simp add: read_eq)
 qed
 have (fr_run_config bl le k w m, ?tgt)
       ∈ mttm_step (fr_delta Sg bl le F k (fr_N F)) ∧ mt_state ?tgt = FR_Rej
  using step by simp
 thus ?thesis by blast
qed
```

The end-reject step: reading the end-of-input blank on a non-member prefix rejects.

```
lemma fr_reject_step:
 assumes wnF: w ∉ F and wSg: set w ⊆ Sg and lenN: length w ≤ fr_N F
 shows ∃ c'. (fr_run_config bl le k w (length w), c')
         ∈ mttm_step (fr_delta Sg bl le F k (fr_N F)) ∧ mt_state c' = FR_Rej
proof −
 let ?ts = λi n. if i < k
```

```
                then (if n = 0 then le
                      else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
                else bl
  let ?nn = λi::nat. if i = 0 then length w + 1 else 0
  let ?tgt = Config_M FR_Rej
              (λi. (?ts i)(?nn i := fr_read bl le k bl i))
              (λi. go_dir (fr_move dir.N i) (?nn i))
  have read_eq: (λi. ?ts i (?nn i)) = fr_read bl le k bl
  proof (rule ext)
    fix i show ?ts i (?nn i) = fr_read bl le k bl i
      by (cases i = 0) (auto simp: fr_read_def)
  qed
  have tr: (FR_Scan w, fr_read bl le k bl, FR_Rej,
            fr_read bl le k bl, fr_move dir.N) ∈ fr_delta Sg bl le F k (fr_N F)
    unfolding fr_delta_def using wSg lenN wnF by auto
  have src: fr_run_config bl le k w (length w) = Config_M (FR_Scan w) ?ts ?nn
    by (simp add: fr_run_config_def)
  have step: (fr_run_config bl le k w (length w), ?tgt)
              ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
    unfolding src
  proof (rule mttm_step.step)
    show (FR_Scan w, λi. ?ts i (?nn i), FR_Rej,
          fr_read bl le k bl, fr_move dir.N) ∈ fr_delta Sg bl le F k (fr_N F)
      using tr by (simp add: read_eq)
  qed
  have (fr_run_config bl le k w (length w), ?tgt)
        ∈ mttm_step (fr_delta Sg bl le F k (fr_N F)) ∧ mt_state ?tgt = FR_Rej
    using step by simp
  thus ?thesis by blast
qed
```

The reachability invariant: from the initial configuration on an input over Sg, every reachable configuration is the initial one, a scan configuration below the cutoff, a reject, or an accept that certifies $w \in F$.

```
definition fr_inv ::
  'a set ⇒ 'a set ⇒ 'a ⇒ 'a ⇒ 'a list set ⇒ nat ⇒ 'a list
    ⇒ ('a, 'a fr_state) mt_config ⇒ bool
where
  fr_inv Sg Gamma bl le F k w c ⟷
    (∃ m. m ≤ length w ∧ m ≤ fr_N F ∧ c = fr_run_config bl le k w m)
    ∨ c = init_config_mttm (finite_recogniser Sg Gamma bl le F k) w
    ∨ mt_state c = FR_Rej
    ∨ (mt_state c = FR_Acc ∧ w ∈ F)
```

Closure of the invariant under a step. Each canonical configuration has a single successor (determinism, via *mttm_step_functional*), given by the matching step lemma; the two halting states are sinks (*fr_step_src_scan*).

```
lemma fr_inv_closed:
  assumes leS: le ∉ Sg and blS: bl ∉ Sg and blle: bl ≠ le
```

```
    and kpos: 0 < k and finF: finite F and wSg: set w ⊆ Sg
    and inv: fr_inv Sg Gamma bl le F k w c
    and step: (c, c') ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
  shows fr_inv Sg Gamma bl le F k w c'
proof −
  have fdet: ∀ q a p1 b1 d1 p2 b2 d2.
      (q, a, p1, b1, d1) ∈ fr_delta Sg bl le F k (fr_N F)
        ⟶ (q, a, p2, b2, d2) ∈ fr_delta Sg bl le F k (fr_N F)
        ⟶ (p1, b1, d1) = (p2, b2, d2)
    using finite_recogniser_det[OF leS blS blle]
    unfolding det_mttm_def finite_recogniser_delta_tm by blast
  from inv[unfolded fr_inv_def] show ?thesis
  proof (elim disjE)
    assume ∃ m. m ≤ length w ∧ m ≤ fr_N F ∧ c = fr_run_config bl le k w m
    then obtain m where mlw: m ≤ length w and mN: m ≤ fr_N F
      and c_eq: c = fr_run_config bl le k w m by blast
    show ?thesis
    proof (cases m < length w)
      case True
      show ?thesis
      proof (cases m < fr_N F)
        case True
        have s: (c, fr_run_config bl le k w (Suc m))
                  ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
          using fr_scan_step[OF kpos ‹m < length w› True wSg] c_eq by simp
        have c' = fr_run_config bl le k w (Suc m)
          using mttm_step_functional[OF fdet step s] .
        moreover have Suc m ≤ length w using ‹m < length w› by simp
        moreover have Suc m ≤ fr_N F using True by simp
        ultimately show ?thesis unfolding fr_inv_def by blast
      next
        case False
        hence mEq: m = fr_N F using mN by simp
        obtain d where d: (fr_run_config bl le k w m, d)
                            ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
          and dst: mt_state d = FR_Rej
          using fr_overflow_step[OF kpos ‹m < length w› mEq wSg] by blast
        have c' = d using mttm_step_functional[OF fdet step] d c_eq by simp
        thus ?thesis unfolding fr_inv_def using dst by simp
      qed
    next
      case False
      hence mEq: m = length w using mlw by simp
      have lenN: length w ≤ fr_N F using mEq mN by simp
      show ?thesis
      proof (cases w ∈ F)
        case True
        have acc: (c, Config_M FR_Acc
                    (λi n. if i < k
```

```
                    then (if n = 0 then le
                          else if i = 0 ∧ n ≤ length w
                               then w ! (n − 1) else bl)
                    else bl)
                (λi::nat. if i = 0 then length w + 1 else 0))
              ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
      using fr_accept_step[OF True wSg finF] c_eq mEq by simp
    have c' = Config_M FR_Acc
              (λi n. if i < k
                    then (if n = 0 then le
                          else if i = 0 ∧ n ≤ length w
                               then w ! (n − 1) else bl)
                    else bl)
              (λi::nat. if i = 0 then length w + 1 else 0)
      using mttm_step_functional[OF fdet step acc] .
    thus ?thesis unfolding fr_inv_def using True by simp
   next
    case False
    obtain d where d: (fr_run_config bl le k w (length w), d)
                    ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
      and dst: mt_state d = FR_Rej
      using fr_reject_step[OF False wSg lenN] by blast
    have (c, d) ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
      using d c_eq mEq by simp
    hence c' = d using mttm_step_functional[OF fdet step] by blast
    thus ?thesis unfolding fr_inv_def using dst by simp
   qed
  qed
 next
  assume ini: c = init_config_mttm (finite_recogniser Sg Gamma bl le F k) w
  have s: (c, fr_run_config bl le k w 0)
          ∈ mttm_step (fr_delta Sg bl le F k (fr_N F))
    using fr_le_step[OF kpos] ini by simp
  have c' = fr_run_config bl le k w 0
    using mttm_step_functional[OF fdet step s] .
  thus ?thesis unfolding fr_inv_def by auto
 next
  assume mt_state c = FR_Rej
  thus ?thesis using fr_step_src_scan[OF step] by auto
 next
  assume mt_state c = FR_Acc ∧ w ∈ F
  thus ?thesis using fr_step_src_scan[OF step] by auto
 qed
qed
```

State of the two canonical non-halting configurations.

```
lemma mt_state_fr_run_config:
  mt_state (fr_run_config bl le k w m) = FR_Scan (take m w)
  by (simp add: fr_run_config_def)
```

lemma *mt_state_init_finite_recogniser*:
mt_state (*init_config_mttm* (*finite_recogniser Sg Gamma bl le F k*) *w*) = *FR_Scan* []
by (*simp add*: *finite_recogniser_def*)

The invariant holds at every reachable configuration (*rtrancl_induct* from the initial configuration, using closure).

lemma *fr_inv_reach*:
assumes *leS*: *le* ∉ *Sg* **and** *blS*: *bl* ∉ *Sg* **and** *blle*: *bl* ≠ *le*
and *kpos*: *0* < *k* **and** *finF*: *finite F* **and** *wSg*: *set w* ⊆ *Sg*
and *reach*: (*init_config_mttm* (*finite_recogniser Sg Gamma bl le F k*) *w*, *c*)
∈ (*mttm_step* (*fr_delta Sg bl le F k* (*fr_N F*)))*
shows *fr_inv Sg Gamma bl le F k w c*
using *reach*
proof (*induction rule*: *rtrancl_induct*)
case *base*
show *?case* **unfolding** *fr_inv_def* **by** *simp*
next
case (*step y z*)
show *?case*
by (*rule fr_inv_closed*[*OF leS blS blle kpos finF wSg step.IH step.hyps*(*2*)])
qed

An accepting reachable configuration certifies membership: the other invariant disjuncts have a non-accept state.

lemma *fr_inv_acc_imp_mem*:
assumes *fr_inv Sg Gamma bl le F k w c* **and** *mt_state c* = *FR_Acc*
shows *w* ∈ *F*
using *assms* **unfolding** *fr_inv_def*
by (*auto simp*: *mt_state_fr_run_config mt_state_init_finite_recogniser*)

Soundness half of the language characterisation: only members are accepted.

lemma *finite_recogniser_lang_sound*:
assumes *leS*: *le* ∉ *Sg* **and** *blS*: *bl* ∉ *Sg* **and** *blle*: *bl* ≠ *le*
and *kpos*: *0* < *k* **and** *finF*: *finite F*
shows *Lang_mttm* (*finite_recogniser Sg Gamma bl le F k*) ⊆ *F*
proof
fix *w* **assume** *wL*: *w* ∈ *Lang_mttm* (*finite_recogniser Sg Gamma bl le F k*)
from *wL*[*unfolded Lang_mttm_def finite_recogniser_Sigma_tm finite_recogniser_t_tm*]
obtain *w' n* **where** *wSg*: *set w* ⊆ *Sg*
and *r*: (*init_config_mttm* (*finite_recogniser Sg Gamma bl le F k*) *w*,
Config$_M$ *FR_Acc w' n*)
∈ (*mttm_step* (*delta_tm* (*finite_recogniser Sg Gamma bl le F k*)))*
by *auto*
have *r'*: (*init_config_mttm* (*finite_recogniser Sg Gamma bl le F k*) *w*,
Config$_M$ *FR_Acc w' n*)

```
        ∈ (mttm_step (fr_delta Sg bl le F k (fr_N F)))*
    using r by (simp add: finite_recogniser_delta_tm)
  have inv: fr_inv Sg Gamma bl le F k w (Config_M FR_Acc w' n)
    by (rule fr_inv_reach[OF leS blS blle kpos finF wSg r'])
  show w ∈ F using fr_inv_acc_imp_mem[OF inv] by simp
qed
```

The language characterisation: the recogniser decides exactly F.

```
theorem finite_recogniser_language:
  assumes leS: le ∉ Sg and blS: bl ∉ Sg and blle: bl ≠ le
    and kpos: 0 < k and finF: finite F and Fsub: ∀ w∈F. set w ⊆ Sg
  shows Lang_mttm (finite_recogniser Sg Gamma bl le F k) = F
  using finite_recogniser_lang_complete[OF kpos Fsub finF]
        finite_recogniser_lang_sound[OF leS blS blle kpos finF]
  by blast


end
theory Multitape_Finite_Patch
  imports Multitape_Finite_Control
begin
```

6 Finite-control patch combinator

The combinator *finite_patch* wraps an arbitrary machine M with a finite-control front end deciding the short inputs (length at most a cutoff N) by a caller-supplied table, while long inputs (length above N) are handed to M unchanged. Unlike the alphabet combinators it performs *no encoding*: it keeps M's input alphabet, tape alphabet, blank, left endmarker, and tape count, so running M is a pure state relabelling (*FP_Run q* mirrors M's state q) on the very same tapes.

The front end scans the input read-only as the recogniser does; once it has read past the cutoff it walks the head back to the origin (phase *FP_Rewind*) and enters M's start state, so the handoff configuration is exactly M's initial configuration. Short inputs are decided in place: the scan ends on the end-of-input blank and jumps straight to M's accept state (identify-with-run, so no extra halting funnel) when the table holds, or to a dedicated reject sink otherwise.

6.1 Patch state space

Four phases: *FP_Scan u* has read the prefix u; *FP_Rewind* is walking the head back to the origin after a long-input overflow; *FP_Run q* is running the wrapped machine in (relabelled) state q; *FP_Rej* is the short-input rejection sink. Acceptance is identified with *reaching M's accept state under the run relabelling*, *FP_Run* (*t_tm M*).

datatype $('q, 'a)$ *fp_state* =
  FP_Scan $'a$ *list* | *FP_Rewind* | *FP_Run* $'q$ | *FP_Rej*

6.2 Patch transition relation

Eight families, reusing the recogniser's read/move tuples *fr_read* / *fr_move* (so the front end is read-only, slots 2 and 4 equal). Parameters: input alphabet *Sg*, blank *bl*, left endmarker *le*, tape count *k*, the wrapped machine's start state *st* and accept state *tt*, the short-input decision table *table*, the cutoff *N*, and the wrapped machine's transition relation *deltaM*.

- advance-le — read past the left endmarker (start step);
- scan-extend — extend the scanned prefix while below the cutoff;
- overflow — at the cutoff with input remaining, enter the rewind;
- short-accept — end-of-input on a table member jumps to *M*'s accept;
- short-reject — end-of-input on a non-member rejects;
- rewind-walk — walk the head left over the input (the only left move);
- rewind-done — reading the left endmarker enters *M*'s start state;
- run-lift — *M*'s own transitions, relabelled by *FP_Run*.

definition *fp_delta* ::
  $'a$ *set* ⇒ $'a$ ⇒ $'a$ ⇒ *nat* ⇒ $'q$ ⇒ $'q$ ⇒ ($'a$ *list* ⇒ *bool*) ⇒ *nat*
    ⇒ ($'q$ × (*nat* ⇒ $'a$) × $'q$ × (*nat* ⇒ $'a$) × (*nat* ⇒ *dir*)) *set*
    ⇒ (($'q$, $'a$) *fp_state* × (*nat* ⇒ $'a$) × ($'q$, $'a$) *fp_state*
        × (*nat* ⇒ $'a$) × (*nat* ⇒ *dir*)) *set*
where
  fp_delta Sg bl le k st tt table N deltaM =
    { (*FP_Scan* [], *fr_read bl le k le*, *FP_Scan* [],
       fr_read bl le k le, *fr_move dir.R*) }
    ∪ { (*FP_Scan u*, *fr_read bl le k x*, *FP_Scan* (*u* @ [*x*]),
         fr_read bl le k x, *fr_move dir.R*)
        | *u x. set u* ⊆ *Sg* ∧ *length u* < *N* ∧ *x* ∈ *Sg* }
    ∪ { (*FP_Scan u*, *fr_read bl le k x*, *FP_Rewind*,
         fr_read bl le k x, *fr_move dir.N*)
        | *u x. set u* ⊆ *Sg* ∧ *length u* = *N* ∧ *x* ∈ *Sg* }
    ∪ { (*FP_Scan u*, *fr_read bl le k bl*, *FP_Run tt*,
         fr_read bl le k bl, *fr_move dir.N*)
        | *u. set u* ⊆ *Sg* ∧ *length u* ≤ *N* ∧ *table u* }
    ∪ { (*FP_Scan u*, *fr_read bl le k bl*, *FP_Rej*,
         fr_read bl le k bl, *fr_move dir.N*)
        | *u. set u* ⊆ *Sg* ∧ *length u* ≤ *N* ∧ ¬ *table u* }
    ∪ { (*FP_Rewind*, *fr_read bl le k x*, *FP_Rewind*,

```
      fr_read bl le k x, fr_move dir.L)
    | x. x ∈ Sg }
  ∪ { (FP_Rewind, fr_read bl le k le, FP_Run st,
      fr_read bl le k le, fr_move dir.N) }
  ∪ { (FP_Run q, a, FP_Run q', a', d)
    | q a q' a' d. (q, a, q', a', d) ∈ deltaM }
```

6.3 The patch machine

The finite patch of M at cutoff N with short-input table *table*. Its state set is the scannable prefixes (words over M's input alphabet up to the cutoff), the relabelled states of M, the rewind state, and the reject sink; alphabet, blank, endmarker, and tape count are inherited from M.

```
definition finite_patch ::
  ('q, 'a) mttm ⇒ ('a list ⇒ bool) ⇒ nat ⇒ (('q, 'a) fp_state, 'a) mttm
where
  finite_patch M table N =
    MTTM
      (FP_Scan ` {u. set u ⊆ Sigma_tm M ∧ length u ≤ N}
        ∪ FP_Run ` Q_tm M ∪ {FP_Rewind, FP_Rej})
      (Sigma_tm M)
      (Γ_tm M)
      (bl_tm M)
      (le_tm M)
      (fp_delta (Sigma_tm M) (bl_tm M) (le_tm M) (k_tm M)
        (s_tm M) (t_tm M) table N (delta_tm M))
      (FP_Scan [])
      (FP_Run (t_tm M))
      FP_Rej
      (k_tm M)
```

6.4 Component accessors

```
lemma finite_patch_k_tm:
  k_tm (finite_patch M table N) = k_tm M
  by (simp add: finite_patch_def)

lemma finite_patch_Sigma_tm:
  Sigma_tm (finite_patch M table N) = Sigma_tm M
  by (simp add: finite_patch_def)

lemma finite_patch_Gamma_tm:
  Γ_tm (finite_patch M table N) = Γ_tm M
  by (simp add: finite_patch_def)

lemma finite_patch_bl_tm:
  bl_tm (finite_patch M table N) = bl_tm M
  by (simp add: finite_patch_def)
```

lemma *finite_patch_le_tm*:
le_tm (finite_patch M table N) = le_tm M
by (*simp add*: *finite_patch_def*)

lemma *finite_patch_s_tm*:
s_tm (finite_patch M table N) = FP_Scan []
by (*simp add*: *finite_patch_def*)

lemma *finite_patch_t_tm*:
t_tm (finite_patch M table N) = FP_Run (t_tm M)
by (*simp add*: *finite_patch_def*)

lemma *finite_patch_r_tm*:
r_tm (finite_patch M table N) = FP_Rej
by (*simp add*: *finite_patch_def*)

lemma *finite_patch_Q_tm*:
Q_tm (finite_patch M table N)
= FP_Scan ‘ {u. set u ⊆ Sigma_tm M ∧ length u ≤ N}
∪ FP_Run ‘ Q_tm M ∪ {FP_Rewind, FP_Rej}
by (*simp add*: *finite_patch_def*)

lemma *finite_patch_delta_tm*:
delta_tm (finite_patch M table N)
= fp_delta (Sigma_tm M) (bl_tm M) (le_tm M) (k_tm M)
(s_tm M) (t_tm M) table N (delta_tm M)
by (*simp add*: *finite_patch_def*)

6.5 Transition-relation discipline

The support invariant: at every inactive tape index $j \geq k$, each transition reads and writes the blank and stays put. The front families are read-only with *fr_read* / *fr_move*, so $j \geq k$ reads and writes *bl* and moves N; the run-lift family inherits it from the wrapped machine's own support hypothesis.

lemma *fp_delta_support*:
assumes *kpos*: *0 < k*
and *dM*: *∀ q a q′ a′ d. (q, a, q′, a′, d) ∈ deltaM*
⟶ (∀ j≥k. a j = bl ∧ a′ j = bl ∧ d j = dir.N)
and *tr*: *(q, a, q′, a′, d) ∈ fp_delta Sg bl le k st tt table N deltaM*
and *j*: *k ≤ j*
shows *a j = bl ∧ a′ j = bl ∧ d j = dir.N*
proof −
have *jn0*: *j ≠ 0* **using** *j kpos* **by** *simp*
have *jk*: *¬ j < k* **using** *j* **by** *simp*
from *tr* **consider**
(*front*) *rsym mdir* **where** *a = fr_read bl le k rsym*
a′ = fr_read bl le k rsym d = fr_move mdir
| (*run*) *q0 q0′* **where** *(q0, a, q0′, a′, d) ∈ deltaM*
unfolding *fp_delta_def* **by** *blast*

```
  then show ?thesis
  proof cases
    case front — read-only front families: bl on the support region
    show ?thesis using front jn0 jk by (simp add: fr_read_def fr_move_def)
  next
    case run — the run-lift family inherits M's support invariant
    show ?thesis using dM run j by blast
  qed
qed
```

A tape reading *fr_read*'s input symbol $x \in Sg$ as *le* must be tape *0*'s neighbour, not tape *0* itself: tape *0* carries x, and $x \neq le$ since $le \notin Sg$. This is what keeps the rewind walk's left move off any endmarker-reading tape.

```
lemma fr_read_le_pos:
  le ∉ Sg ⟹ x ∈ Sg ⟹ fr_read bl le k x j = le ⟹ j ≠ 0
  by (auto simp: fr_read_def split: if_splits)
```

The left-endmarker read discipline: a transition reading *le* on tape j writes *le* back there and moves only N or R. The front families are read-only, so the write half is immediate; for the move half the only left move is the rewind walk (*fr_move L*), and there tape *0* reads an input symbol (never *le*, as $le \notin Sg$) while every other tape stays put. The run-lift family inherits it from the wrapped machine's own endmarker hypothesis.

```
lemma fp_delta_LE:
  assumes leS: le ∉ Sg
    and dM: ∀ q a q' a' d j. (q, a, q', a', d) ∈ deltaM ⟶ a j = le
              ⟶ a' j = le ∧ d j ∈ {dir.N, dir.R}
    and tr: (q, a, q', a', d) ∈ fp_delta Sg bl le k st tt table N deltaM
    and LE: a j = le
  shows a' j = le ∧ d j ∈ {dir.N, dir.R}
proof −
  — name the family up front, so each case reasons over clean (non-recursive) equations rather than letting the simplifier back-substitute le into an fr_read body
  from tr consider
      (front) a' = a d = fr_move dir.R ∨ d = fr_move dir.N
    | (rwalk) x where x ∈ Sg a = fr_read bl le k x a' = a
                     d = fr_move dir.L
    | (run) q0 q0' where (q0, a, q0', a', d) ∈ deltaM
    unfolding fp_delta_def by blast
  then show ?thesis
  proof cases
    case front — read-only front families with no left move
    from front(1) LE have aj: a' j = le by simp
    have d j ≠ dir.L using front(2) by (auto simp: fr_move_def split: if_splits)
    hence d j ∈ {dir.N, dir.R} by (cases d j) auto
    with aj show ?thesis by blast
  next
    case rwalk — the rewind walk: reading le forces the head off tape 0
    have aj: fr_read bl le k x j = le using LE rwalk(2) by simp
```

```
    have j ≠ 0 using fr_read_le_pos[OF leS rwalk(1) aj] .
    moreover have a' j = le using LE rwalk(3) by simp
    ultimately show ?thesis using rwalk(4) by (simp add: fr_move_def)
  next
    case run — the run-lift family inherits M's endmarker discipline
    show ?thesis using dM run LE by blast
  qed
qed
```

6.6 Well-formedness

The patch of a valid machine is a valid substrate machine. The front families supply the finite-control side of every structural axiom (finiteness of the scannable prefixes, the read/write alphabet, the endmarker and support disciplines) and the run-lift family forwards M's own structure through the substrate's projection lemmas.

```
lemma finite_patch_valid:
  assumes vM: valid_mttm M
  shows valid_mttm (finite_patch M table N)
proof −
  let ?Sg = Sigma_tm M
  let ?Ga = Γ_tm M
  let ?bl = bl_tm M
  let ?le = le_tm M
  let ?k = k_tm M
  let ?st = s_tm M
  let ?tt = t_tm M
  let ?QM = Q_tm M
  let ?Q = FP_Scan ‘ {u. set u ⊆ ?Sg ∧ length u ≤ N}
           ∪ FP_Run ‘ ?QM ∪ {FP_Rewind, FP_Rej}
  let ?dl = fp_delta ?Sg ?bl ?le ?k ?st ?tt table N (delta_tm M)
  — the machine's own structural facts, projected out of vM
  have SgG: ?Sg ⊆ ?Ga by (rule valid_mttm_Sigma_sub_Gamma[OF vM])
  have blG: ?bl ∈ ?Ga by (rule valid_mttm_blank_in_Gamma[OF vM])
  have blS: ?bl ∉ ?Sg by (rule valid_mttm_blank_not_Sigma[OF vM])
  have leG: ?le ∈ ?Ga by (rule valid_mttm_LE_in_Gamma[OF vM])
  have leS: ?le ∉ ?Sg by (rule valid_mttm_LE_not_Sigma[OF vM])
  have sQ: ?st ∈ ?QM by (rule valid_mttm_s_in_Q[OF vM])
  have tQ: ?tt ∈ ?QM by (rule valid_mttm_t_in_Q[OF vM])
  have kpos: 0 < ?k by (rule valid_mttm_k_pos[OF vM])
  have finGa: finite ?Ga by (rule valid_mttm_finite_Gamma[OF vM])
  have finQM: finite ?QM by (rule valid_mttm_finite_Q[OF vM])
  have dM_set: delta_tm M
     ⊆ (?QM − {?tt, r_tm M}) × (UNIV → ?Ga) × ?QM
         × (UNIV → ?Ga) × (UNIV → UNIV)
    by (rule valid_mttm_delta_set[OF vM])
  have dM_LE: ∀ q a q' a' d j. (q, a, q', a', d) ∈ delta_tm M ⟶ a j = ?le
                ⟶ a' j = ?le ∧ d j ∈ {dir.N, dir.R}
```

```
  using valid_mttm_deltaLE[OF vM] by blast
 have dM_supp: ∀ q a q' a' d. (q, a, q', a', d) ∈ delta_tm M
                ⟶ (∀ j≥?k. a j = ?bl ∧ a' j = ?bl ∧ d j = dir.N)
  using valid_mttm_delta_support[OF vM] by blast
— finiteness of the scannable-prefix component, hence of Q
 have finSg: finite ?Sg using finite_subset[OF SgG finGa] .
 have finP: finite {u. set u ⊆ ?Sg ∧ length u ≤ N}
  using finite_lists_length_le[OF finSg] by blast
 have finQ: finite ?Q using finP finQM by simp
— the transition-shape, endmarker, and support obligations
 have shape: ?dl ⊆ (?Q − {FP_Run ?tt, FP_Rej}) × (UNIV → ?Ga) × ?Q
                  × (UNIV → ?Ga) × (UNIV → UNIV)
  unfolding fp_delta_def fr_read_def
  using SgG blG leG sQ tQ dM_set by (auto simp: Pi_iff)
 have LEc: ∀ q a q' a' d j.
    (q, a, q', a', d) ∈ ?dl ⟶ a j = ?le ⟶ a' j = ?le ∧ d j ∈ {dir.N, dir.R}
  using fp_delta_LE[OF leS dM_LE] by blast
 have suppc: ∀ q a q' a' d.
    (q, a, q', a', d) ∈ ?dl ⟶ (∀ j≥?k. a j = ?bl ∧ a' j = ?bl ∧ d j = dir.N)
  using fp_delta_support[OF kpos dM_supp] by blast
 show ?thesis
  unfolding finite_patch_def valid_mttm.simps
  using finQ finGa SgG blG blS leG leS kpos tQ shape LEc suppc
  by (intro conjI) auto
qed
```

6.7 Determinism

The patch is deterministic when the wrapped machine is. The three phases are disjoint by source constructor (*FP_Scan* / *FP_Rewind* / *FP_Run*); within the scan and rewind phases the tape-*0* read (*fr_read_inj*) selects a unique family, the left endmarker, the blank, and the input alphabet being pairwise distinct; and the run phase is single-valued because M is.

```
lemma finite_patch_det:
 assumes vM: valid_mttm M and blle: bl_tm M ≠ le_tm M and dM: det_mttm
M
 shows det_mttm (finite_patch M table N)
proof −
 have leS: le_tm M ∉ Sigma_tm M by (rule valid_mttm_LE_not_Sigma[OF
vM])
 have blS: bl_tm M ∉ Sigma_tm M by (rule valid_mttm_blank_not_Sigma[OF
vM])
 show ?thesis
  unfolding det_mttm_def finite_patch_delta_tm fp_delta_def
  using leS blS blle dM[unfolded det_mttm_def]
  by (auto dest: fr_read_inj)
qed
```

6.8 Long-input simulation of the wrapped machine

The run-lift family embeds M's transitions verbatim, so the *run relabelling* — tagging M's state q as *FP_Run q* on the very same tapes — carries every M step to a patch step. This is the long-input arm's correctness, and it holds for a nondeterministic M (the run phase is where the patch's own nondeterminism lives).

```
fun fp_lift :: ('a, 'q) mt_config ⇒ ('a, ('q, 'a) fp_state) mt_config where
  fp_lift (Config_M q ts n) = Config_M (FP_Run q) ts n

lemma fp_run_step:
  assumes (c, c') ∈ mttm_step deltaM
  shows (fp_lift c, fp_lift c')
          ∈ mttm_step (fp_delta Sg bl le k st tt table N deltaM)
proof −
  from assms obtain q ts n q' a d where
      c: c = Config_M q ts n
    and c': c' = Config_M q' (λi. (ts i)(n i := a i)) (λi. go_dir (d i) (n i))
    and tr: (q, λi. ts i (n i), q', a, d) ∈ deltaM
    by (auto elim: mttm_step.cases)
  have tr': (FP_Run q, λi. ts i (n i), FP_Run q', a, d)
              ∈ fp_delta Sg bl le k st tt table N deltaM
    unfolding fp_delta_def using tr by blast
  have (Config_M (FP_Run q) ts n,
        Config_M (FP_Run q') (λi. (ts i)(n i := a i)) (λi. go_dir (d i) (n i)))
         ∈ mttm_step (fp_delta Sg bl le k st tt table N deltaM)
  proof (rule mttm_step.step)
    show (FP_Run q, λi. ts i (n i), FP_Run q', a, d)
           ∈ fp_delta Sg bl le k st tt table N deltaM
      by (rule tr')
  qed
  thus ?thesis using c c' by simp
qed

lemma fp_run_rtrancl:
  assumes (c, c') ∈ (mttm_step deltaM)*
  shows (fp_lift c, fp_lift c')
          ∈ (mttm_step (fp_delta Sg bl le k st tt table N deltaM))*
  using assms
proof (induction rule: rtrancl_induct)
  case base
  show ?case by simp
next
  case (step y z)
  from fp_run_step[OF step.hyps(2)] step.IH
  show ?case by (auto intro: rtrancl.rtrancl_into_rtrancl)
qed
```

6.9 The scanning front run

The configuration after the patch has read the left endmarker and the first m input symbols: state *FP_Scan* (*take m w*), the (read-only, hence unchanging) input tape, and the input head at position *m + 1*. Identical in tape shape to the recogniser's *fr_run_config*, differing only in the scan constructor.

```
definition fp_scan_config ::
  'a ⇒ 'a ⇒ nat ⇒ 'a list ⇒ nat ⇒ ('a, ('q, 'a) fp_state) mt_config
where
  fp_scan_config bl le k w m =
    Config_M (FP_Scan (take m w))
      (λi n. if i < k
           then (if n = 0 then le
                 else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
           else bl)
      (λi. if i = 0 then m + 1 else 0)
```

The start step: reading the left endmarker advances the patch's initial configuration into the length-*0* scan configuration.

```
lemma fp_le_step:
  assumes kpos: 0 < k_tm M
  shows (init_config_mttm (finite_patch M table N) w,
         fp_scan_config (bl_tm M) (le_tm M) (k_tm M) w 0)
         ∈ mttm_step (fp_delta (Sigma_tm M) (bl_tm M) (le_tm M) (k_tm M)
                       (s_tm M) (t_tm M) table N (delta_tm M))
proof −
  let ?bl = bl_tm M and ?le = le_tm M and ?k = k_tm M
  let ?ts = λi n. if i < ?k
                  then (if n = 0 then ?le
                        else if i = 0 ∧ n ≤ length w then w ! (n − 1) else ?bl)
                  else ?bl
  let ?dl = fp_delta (Sigma_tm M) ?bl ?le ?k (s_tm M) (t_tm M) table N
(delta_tm M)
  have read_eq: (λi. ?ts i ((λ_. 0) i)) = fr_read ?bl ?le ?k ?le
  proof (rule ext)
    fix i show ?ts i ((λ_. 0) i) = fr_read ?bl ?le ?k ?le i
      using kpos by (cases i = 0) (auto simp: fr_read_def)
  qed
  have tr: (FP_Scan [], fr_read ?bl ?le ?k ?le, FP_Scan [],
            fr_read ?bl ?le ?k ?le, fr_move dir.R) ∈ ?dl
    unfolding fp_delta_def by auto
  have init_eq: init_config_mttm (finite_patch M table N) w
                  = Config_M (FP_Scan []) ?ts (λ_. 0)
    by (simp add: finite_patch_def)
  have step: (Config_M (FP_Scan []) ?ts (λ_. 0),
              Config_M (FP_Scan [])
                (λi. (?ts i)((λ_. 0) i := fr_read ?bl ?le ?k ?le i))
                (λi. go_dir (fr_move dir.R i) ((λ_. 0) i))) ∈ mttm_step ?dl
```

```
  proof (rule mttm_step.step)
    show (FP_Scan [], λi. ?ts i ((λ_. 0) i), FP_Scan [],
          fr_read ?bl ?le ?k ?le, fr_move dir.R) ∈ ?dl
      using tr by (simp add: read_eq)
  qed
  have tgt: fp_scan_config ?bl ?le ?k w 0
            = Config_M (FP_Scan [])
                (λi. (?ts i)((λ_. 0) i := fr_read ?bl ?le ?k ?le i))
                (λi. go_dir (fr_move dir.R i) ((λ_. 0) i))
  proof −
    have tape: (λi. (?ts i)((λ_. 0) i := fr_read ?bl ?le ?k ?le i)) = ?ts
    proof (rule ext)
      fix i
      have eq: fr_read ?bl ?le ?k ?le i = ?ts i ((λ_. 0) i)
        using kpos by (cases i = 0) (auto simp: fr_read_def)
      show (?ts i)((λ_. 0) i := fr_read ?bl ?le ?k ?le i) = ?ts i
        unfolding eq by (rule fun_upd_triv)
    qed
    have head: (λi. go_dir (fr_move dir.R i) ((λ_. 0) i))
                 = (λi::nat. if i = 0 then 0 + 1 else 0)
      by (rule ext) (simp add: fr_move_def)
    show ?thesis by (simp add: fp_scan_config_def tape head)
  qed
  show ?thesis unfolding init_eq tgt by (rule step)
qed
```

One scan step: reading the next input symbol extends the scanned prefix, provided the input still has a symbol there and the cutoff is not yet reached.

```
lemma fp_scan_step:
  assumes kpos: 0 < k and mlt: m < length w and mN: m < N
    and wSg: set w ⊆ Sg
  shows (fp_scan_config bl le k w m, fp_scan_config bl le k w (Suc m))
          ∈ mttm_step (fp_delta Sg bl le k st tt table N deltaM)
proof −
  let ?x = w ! m
  let ?ts = λi n. if i < k
                  then (if n = 0 then le
                        else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
                  else bl
  let ?nn = λi::nat. if i = 0 then m + 1 else 0
  let ?dl = fp_delta Sg bl le k st tt table N deltaM
  have xSg: ?x ∈ Sg using nth_mem[OF mlt] wSg by blast
  have lenu: length (take m w) = m using mlt by simp
  have takeq: take (Suc m) w = take m w @ [?x]
    using mlt by (simp add: take_Suc_conv_app_nth)
  have setu: set (take m w) ⊆ Sg
    using wSg by (meson set_take_subset subset_trans)
  have read_eq: (λi. ?ts i (?nn i)) = fr_read bl le k ?x
  proof (rule ext)
```

```
    fix i show ?ts i (?nn i) = fr_read bl le k ?x i
      using kpos mlt by (cases i = 0) (auto simp: fr_read_def)
  qed
  have tr: (FP_Scan (take m w), fr_read bl le k ?x, FP_Scan (take (Suc m) w),
            fr_read bl le k ?x, fr_move dir.R) ∈ ?dl
    unfolding fp_delta_def takeq using setu lenu mN xSg by auto
  have src: fp_scan_config bl le k w m = Config_M (FP_Scan (take m w)) ?ts ?nn
    by (simp add: fp_scan_config_def)
  have step: (Config_M (FP_Scan (take m w)) ?ts ?nn,
              Config_M (FP_Scan (take (Suc m) w))
                (λi. (?ts i)(?nn i := fr_read bl le k ?x i))
                (λi. go_dir (fr_move dir.R i) (?nn i))) ∈ mttm_step ?dl
  proof (rule mttm_step.step)
    show (FP_Scan (take m w), λi. ?ts i (?nn i), FP_Scan (take (Suc m) w),
          fr_read bl le k ?x, fr_move dir.R) ∈ ?dl
      using tr by (simp add: read_eq)
  qed
  have tgt: fp_scan_config bl le k w (Suc m)
            = Config_M (FP_Scan (take (Suc m) w))
                (λi. (?ts i)(?nn i := fr_read bl le k ?x i))
                (λi. go_dir (fr_move dir.R i) (?nn i))
  proof −
    have tape: (λi. (?ts i)(?nn i := fr_read bl le k ?x i)) = ?ts
    proof (rule ext)
      fix i
      have eq: fr_read bl le k ?x i = ?ts i (?nn i)
        using kpos mlt by (cases i = 0) (auto simp: fr_read_def)
      show (?ts i)(?nn i := fr_read bl le k ?x i) = ?ts i
        unfolding eq by (rule fun_upd_triv)
    qed
    have head: (λi. go_dir (fr_move dir.R i) (?nn i))
                 = (λi::nat. if i = 0 then Suc m + 1 else 0)
      by (rule ext) (simp add: fr_move_def)
    show ?thesis by (simp add: fp_scan_config_def tape head)
  qed
  show ?thesis unfolding src tgt by (rule step)
qed
```

The scan phase: from the length-*0* scan configuration, iterating the scan step reaches the length-m one in m steps, for any m up to both the input length and the cutoff. Assembled by the substrate's *relpow_invariant_chain*.

```
lemma fp_scan_chain:
  assumes kpos: 0 < k and mle: m ≤ length w and mN: m ≤ N
    and wSg: set w ⊆ Sg
  shows (fp_scan_config bl le k w 0, fp_scan_config bl le k w m)
        ∈ (mttm_step (fp_delta Sg bl le k st tt table N deltaM)) ^^ m
  using mle mN
proof (induction m)
```

```
  case 0
  show ?case by simp
next
  case (Suc m)
  have mw: m < length w using Suc.prems(1) by simp
  have mN': m < N using Suc.prems(2) by simp
  have chain: (fp_scan_config bl le k w 0, fp_scan_config bl le k w m)
                ∈ (mttm_step (fp_delta Sg bl le k st tt table N deltaM)) ^^ m
    using Suc.IH Suc.prems by simp
  have step: (fp_scan_config bl le k w m, fp_scan_config bl le k w (Suc m))
               ∈ mttm_step (fp_delta Sg bl le k st tt table N deltaM)
    by (rule fp_scan_step[OF kpos mw mN' wSg])
  from chain step show ?case by (rule relpow_Suc_I)
qed
```

6.10 Short-input dispatch

At the end of a short input (length at most the cutoff), reading the end-of-input blank on a fully-scanned prefix jumps to M's accept state when the table holds — the short arm never runs M.

```
lemma fp_accept_step:
  assumes wtab: table w and wSg: set w ⊆ Sg and lenN: length w ≤ N
  shows (fp_scan_config bl le k w (length w),
         Config_M (FP_Run tt)
           (λi n. if i < k
                  then (if n = 0 then le
                        else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
                  else bl)
           (λi::nat. if i = 0 then length w + 1 else 0))
        ∈ mttm_step (fp_delta Sg bl le k st tt table N deltaM)
proof −
  let ?ts = λi n. if i < k
                  then (if n = 0 then le
                        else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
                  else bl
  let ?nn = λi::nat. if i = 0 then length w + 1 else 0
  let ?dl = fp_delta Sg bl le k st tt table N deltaM
  have read_eq: (λi. ?ts i (?nn i)) = fr_read bl le k bl
  proof (rule ext)
    fix i show ?ts i (?nn i) = fr_read bl le k bl i
      by (cases i = 0) (auto simp: fr_read_def)
  qed
  have tr: (FP_Scan w, fr_read bl le k bl, FP_Run tt,
            fr_read bl le k bl, fr_move dir.N) ∈ ?dl
    unfolding fp_delta_def using wSg lenN wtab by auto
  have src: fp_scan_config bl le k w (length w) = Config_M (FP_Scan w) ?ts ?nn
    by (simp add: fp_scan_config_def)
  have step: (Config_M (FP_Scan w) ?ts ?nn,
              Config_M (FP_Run tt)
```

```
      (λi. (?ts i)(?nn i := fr_read bl le k bl i))
      (λi. go_dir (fr_move dir.N i) (?nn i))) ∈ mttm_step ?dl
  proof (rule mttm_step.step)
    show (FP_Scan w, λi. ?ts i (?nn i), FP_Run tt,
          fr_read bl le k bl, fr_move dir.N) ∈ ?dl
      using tr by (simp add: read_eq)
  qed
  have tape: (λi. (?ts i)(?nn i := fr_read bl le k bl i)) = ?ts
  proof (rule ext)
    fix i
    have eq: fr_read bl le k bl i = ?ts i (?nn i)
      by (cases i = 0) (auto simp: fr_read_def)
    show (?ts i)(?nn i := fr_read bl le k bl i) = ?ts i
      unfolding eq by (rule fun_upd_triv)
  qed
  have head: (λi. go_dir (fr_move dir.N i) (?nn i)) = ?nn
    by (rule ext) (simp add: fr_move_def)
  show ?thesis unfolding src using step by (simp add: tape head)
qed
```

The dual short-input step: on a non-member prefix the scan rejects into the dedicated sink *FP_Rej*.

```
lemma fp_reject_step:
  assumes wtab: ¬ table w and wSg: set w ⊆ Sg and lenN: length w ≤ N
  shows ∃ c'. (fp_scan_config bl le k w (length w), c')
            ∈ mttm_step (fp_delta Sg bl le k st tt table N deltaM)
          ∧ mt_state c' = FP_Rej
proof −
  let ?ts = λi n. if i < k
                  then (if n = 0 then le
                        else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
                  else bl
  let ?nn = λi::nat. if i = 0 then length w + 1 else 0
  let ?dl = fp_delta Sg bl le k st tt table N deltaM
  let ?tgt = Config_M FP_Rej
             (λi. (?ts i)(?nn i := fr_read bl le k bl i))
             (λi. go_dir (fr_move dir.N i) (?nn i))
  have read_eq: (λi. ?ts i (?nn i)) = fr_read bl le k bl
  proof (rule ext)
    fix i show ?ts i (?nn i) = fr_read bl le k bl i
      by (cases i = 0) (auto simp: fr_read_def)
  qed
  have tr: (FP_Scan w, fr_read bl le k bl, FP_Rej,
            fr_read bl le k bl, fr_move dir.N) ∈ ?dl
    unfolding fp_delta_def using wSg lenN wtab by auto
  have src: fp_scan_config bl le k w (length w) = Config_M (FP_Scan w) ?ts ?nn
    by (simp add: fp_scan_config_def)
  have step: (fp_scan_config bl le k w (length w), ?tgt) ∈ mttm_step ?dl
    unfolding src
```

```
  proof (rule mttm_step.step)
    show (FP_Scan w, λi. ?ts i (?nn i), FP_Rej,
          fr_read bl le k bl, fr_move dir.N) ∈ ?dl
      using tr by (simp add: read_eq)
  qed
  have (fp_scan_config bl le k w (length w), ?tgt) ∈ mttm_step ?dl
        ∧ mt_state ?tgt = FP_Rej
    using step by simp
  thus ?thesis by blast
qed
```

6.11 Long-input rewind and handoff

The rewind configuration: state *FP_Rewind*, the unchanging input tape, and the input head at position p as it walks back to the origin.

```
definition fp_rewind_config ::
  'a ⇒ 'a ⇒ nat ⇒ 'a list ⇒ nat ⇒ ('a, ('q, 'a) fp_state) mt_config
where
  fp_rewind_config bl le k w p =
     Config_M FP_Rewind
      (λi n. if i < k
           then (if n = 0 then le
                 else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
           else bl)
      (λi. if i = 0 then p else 0)
```

Overflow: at the cutoff with input still remaining, reading the next symbol enters the rewind (the head does not move — the symbol under it is re-read on the first rewind step).

```
lemma fp_overflow_step:
  assumes kpos: 0 < k and Nlt: N < length w and wSg: set w ⊆ Sg
  shows (fp_scan_config bl le k w N, fp_rewind_config bl le k w (N + 1))
        ∈ mttm_step (fp_delta Sg bl le k st tt table N deltaM)
proof −
  let ?x = w ! N
  let ?ts = λi n. if i < k
                  then (if n = 0 then le
                        else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
                  else bl
  let ?nn = λi::nat. if i = 0 then N + 1 else 0
  let ?dl = fp_delta Sg bl le k st tt table N deltaM
  have xSg: ?x ∈ Sg using nth_mem[OF Nlt] wSg by blast
  have lenu: length (take N w) = N using Nlt by simp
  have setu: set (take N w) ⊆ Sg
    using wSg by (meson set_take_subset subset_trans)
  have read_eq: (λi. ?ts i (?nn i)) = fr_read bl le k ?x
  proof (rule ext)
    fix i show ?ts i (?nn i) = fr_read bl le k ?x i
      using kpos Nlt by (cases i = 0) (auto simp: fr_read_def)
```

```
  qed
  have tr: (FP_Scan (take N w), fr_read bl le k ?x, FP_Rewind,
          fr_read bl le k ?x, fr_move dir.N) ∈ ?dl
    unfolding fp_delta_def using setu lenu xSg by auto
  have src: fp_scan_config bl le k w N = Config_M (FP_Scan (take N w)) ?ts ?nn
    by (simp add: fp_scan_config_def)
  have step: (Config_M (FP_Scan (take N w)) ?ts ?nn,
              Config_M FP_Rewind
                (λi. (?ts i)(?nn i := fr_read bl le k ?x i))
                (λi. go_dir (fr_move dir.N i) (?nn i))) ∈ mttm_step ?dl
  proof (rule mttm_step.step)
    show (FP_Scan (take N w), λi. ?ts i (?nn i), FP_Rewind,
          fr_read bl le k ?x, fr_move dir.N) ∈ ?dl
      using tr by (simp add: read_eq)
  qed
  have tgt: fp_rewind_config bl le k w (N + 1)
            = Config_M FP_Rewind
                (λi. (?ts i)(?nn i := fr_read bl le k ?x i))
                (λi. go_dir (fr_move dir.N i) (?nn i))
  proof −
    have tape: (λi. (?ts i)(?nn i := fr_read bl le k ?x i)) = ?ts
    proof (rule ext)
      fix i
      have eq: fr_read bl le k ?x i = ?ts i (?nn i)
        using kpos Nlt by (cases i = 0) (auto simp: fr_read_def)
      show (?ts i)(?nn i := fr_read bl le k ?x i) = ?ts i
        unfolding eq by (rule fun_upd_triv)
    qed
    have head: (λi. go_dir (fr_move dir.N i) (?nn i)) = ?nn
      by (rule ext) (simp add: fr_move_def)
    show ?thesis by (simp add: fp_rewind_config_def tape head fun_eq_iff)
  qed
  show ?thesis unfolding src tgt by (rule step)
qed
```

One rewind step: reading an input symbol walks the head one cell left, leaving the tape unchanged.

```
lemma fp_rewind_walk_step:
  assumes kpos: 0 < k and qw: Suc q ≤ length w and wSg: set w ⊆ Sg
  shows (fp_rewind_config bl le k w (Suc q), fp_rewind_config bl le k w q)
        ∈ mttm_step (fp_delta Sg bl le k st tt table N deltaM)
proof −
  let ?x = w ! q
  let ?ts = λi n. if i < k
                  then (if n = 0 then le
                        else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
                  else bl
  let ?nn = λi::nat. if i = 0 then Suc q else 0
  let ?dl = fp_delta Sg bl le k st tt table N deltaM
```

```
  have xin: q < length w using qw by simp
  have xSg: ?x ∈ Sg using nth_mem[OF xin] wSg by blast
  have read_eq: (λi. ?ts i (?nn i)) = fr_read bl le k ?x
  proof (rule ext)
    fix i show ?ts i (?nn i) = fr_read bl le k ?x i
      using kpos qw by (cases i = 0) (auto simp: fr_read_def)
  qed
  have tr: (FP_Rewind, fr_read bl le k ?x, FP_Rewind,
            fr_read bl le k ?x, fr_move dir.L) ∈ ?dl
    unfolding fp_delta_def using xSg by auto
  have src: fp_rewind_config bl le k w (Suc q) = Config_M FP_Rewind ?ts ?nn
    by (simp add: fp_rewind_config_def)
  have step: (Config_M FP_Rewind ?ts ?nn,
              Config_M FP_Rewind
                (λi. (?ts i)(?nn i := fr_read bl le k ?x i))
                (λi. go_dir (fr_move dir.L i) (?nn i))) ∈ mttm_step ?dl
  proof (rule mttm_step.step)
    show (FP_Rewind, λi. ?ts i (?nn i), FP_Rewind,
          fr_read bl le k ?x, fr_move dir.L) ∈ ?dl
      using tr by (simp add: read_eq)
  qed
  have tgt: fp_rewind_config bl le k w q
            = Config_M FP_Rewind
                (λi. (?ts i)(?nn i := fr_read bl le k ?x i))
                (λi. go_dir (fr_move dir.L i) (?nn i))
  proof −
    have tape: (λi. (?ts i)(?nn i := fr_read bl le k ?x i)) = ?ts
    proof (rule ext)
      fix i
      have eq: fr_read bl le k ?x i = ?ts i (?nn i)
        using kpos qw by (cases i = 0) (auto simp: fr_read_def)
      show (?ts i)(?nn i := fr_read bl le k ?x i) = ?ts i
        unfolding eq by (rule fun_upd_triv)
    qed
    have head: (λi. go_dir (fr_move dir.L i) (?nn i))
                 = (λi::nat. if i = 0 then q else 0)
      by (rule ext) (simp add: fr_move_def)
    show ?thesis by (simp add: fp_rewind_config_def tape head)
  qed
  show ?thesis unfolding src tgt by (rule step)
qed
```

The rewind phase: from head position p (within the input) the walk reaches the origin in p steps.

```
lemma fp_rewind_chain:
  assumes kpos: 0 < k and pw: p ≤ length w and wSg: set w ⊆ Sg
  shows (fp_rewind_config bl le k w p, fp_rewind_config bl le k w 0)
        ∈ (mttm_step (fp_delta Sg bl le k st tt table N deltaM)) ^^ p
  using pw
```

```
proof (induction p)
  case 0
  show ?case by simp
next
  case (Suc p)
  have step: (fp_rewind_config bl le k w (Suc p), fp_rewind_config bl le k w p)
               ∈ mttm_step (fp_delta Sg bl le k st tt table N deltaM)
    by (rule fp_rewind_walk_step[OF kpos Suc.prems wSg])
  have chain: (fp_rewind_config bl le k w p, fp_rewind_config bl le k w 0)
                ∈ (mttm_step (fp_delta Sg bl le k st tt table N deltaM)) ^^ p
    using Suc.IH Suc.prems by simp
  from step chain show ?case by (rule relpow_Suc_I2)
qed
```

Rewind completion: reading the left endmarker at the origin enters M's start state, and the configuration is exactly the run relabelling of M's initial configuration.

```
lemma fp_rewind_done_step:
  assumes kpos: 0 < k_tm M
  shows (fp_rewind_config (bl_tm M) (le_tm M) (k_tm M) w 0,
         fp_lift (init_config_mttm M w))
         ∈ mttm_step (fp_delta (Sigma_tm M) (bl_tm M) (le_tm M) (k_tm M)
                       (s_tm M) (t_tm M) table N (delta_tm M))
proof (cases M)
  case (MTTM Q Sg Ga bl le d s t r k)
  — reduce every _tm M selector to a concrete component up front, so the tape
reasoning is over ground terms
  have kpos': 0 < k using kpos by (simp add: MTTM)
  let ?ts = λi n. if i < k
                  then (if n = 0 then le
                        else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl)
                  else bl
  let ?dl = fp_delta Sg bl le k s t table N d
  have read_eq: (λi. ?ts i ((λ_. 0) i)) = fr_read bl le k le
  proof (rule ext)
    fix i show ?ts i ((λ_. 0) i) = fr_read bl le k le i
      using kpos' by (cases i = 0) (auto simp: fr_read_def)
  qed
  have tr: (FP_Rewind, fr_read bl le k le, FP_Run s,
            fr_read bl le k le, fr_move dir.N) ∈ ?dl
    unfolding fp_delta_def by auto
  have src: fp_rewind_config (bl_tm M) (le_tm M) (k_tm M) w 0
              = Config_M FP_Rewind ?ts (λ_. 0)
    by (simp add: fp_rewind_config_def MTTM)
  have dl_eq: fp_delta (Sigma_tm M) (bl_tm M) (le_tm M) (k_tm M)
                (s_tm M) (t_tm M) table N (delta_tm M) = ?dl
    by (simp add: MTTM)
  have step: (Config_M FP_Rewind ?ts (λ_. 0),
              Config_M (FP_Run s)
```

```
            (λi. (?ts i)((λ_. 0) i := fr_read bl le k le i))
            (λi. go_dir (fr_move dir.N i) ((λ_. 0) i)))
          ∈ mttm_step ?dl
  proof (rule mttm_step.step)
    show (FP_Rewind, λi. ?ts i ((λ_. 0) i), FP_Run s,
          fr_read bl le k le, fr_move dir.N) ∈ ?dl
      using tr by (simp add: read_eq)
  qed
  have tgt: fp_lift (init_config_mttm M w)
          = Config_M (FP_Run s)
              (λi. (?ts i)((λ_. 0) i := fr_read bl le k le i))
              (λi. go_dir (fr_move dir.N i) ((λ_. 0) i))
  proof −
    have tape: (λi. (?ts i)((λ_. 0) i := fr_read bl le k le i)) = ?ts
    proof (rule ext)
      fix i
      have eq: fr_read bl le k le i = ?ts i ((λ_. 0) i)
        using kpos' by (cases i = 0) (auto simp: fr_read_def)
      show (?ts i)((λ_. 0) i := fr_read bl le k le i) = ?ts i
        unfolding eq by (rule fun_upd_triv)
    qed
    have head: (λi. go_dir (fr_move dir.N i) ((λ_. 0) i)) = (λ_::nat. 0)
      by (rule ext) (simp add: fr_move_def)
    have init: init_config_mttm M w = Config_M s ?ts (λ_. 0)
      by (simp add: MTTM)
    show ?thesis by (simp add: init tape head)
  qed
  show ?thesis unfolding dl_eq src tgt using step by simp
qed
```

6.12 Forward language direction

A short accepted input reaches M's accept state without running M: read the endmarker, scan the whole input, and dispatch on the table.

```
lemma fp_short_run:
  assumes kpos: 0 < k_tm M and wSg: set w ⊆ Sigma_tm M
    and lenN: length w ≤ N and tab: table w
  shows ∃ c. (init_config_mttm (finite_patch M table N) w, c)
          ∈ (mttm_step (delta_tm (finite_patch M table N)))*
        ∧ mt_state c = FP_Run (t_tm M)
proof −
  let ?dl = fp_delta (Sigma_tm M) (bl_tm M) (le_tm M) (k_tm M)
            (s_tm M) (t_tm M) table N (delta_tm M)
  let ?sc = fp_scan_config (bl_tm M) (le_tm M) (k_tm M) w
  let ?acc = Config_M (FP_Run (t_tm M))
              (λi n. if i < k_tm M
                     then (if n = 0 then le_tm M
                           else if i = 0 ∧ n ≤ length w then w ! (n − 1) else bl_tm M)
                     else bl_tm M)
```

```
      (λi::nat. if i = 0 then length w + 1 else 0)
    have dl: delta_tm (finite_patch M table N) = ?dl by (rule finite_patch_delta_tm)
  have le: (init_config_mttm (finite_patch M table N) w, ?sc 0) ∈ (mttm_step ?dl)*
    using fp_le_step[OF kpos] by (rule r_into_rtrancl)
  have sc: (?sc 0, ?sc (length w)) ∈ (mttm_step ?dl)*
        using fp_scan_chain[OF kpos order_refl lenN wSg] by (rule relpow_imp_rtrancl)
  have ac0: (?sc (length w), ?acc) ∈ mttm_step ?dl
    using tab wSg lenN by (rule fp_accept_step)
  have ac: (?sc (length w), ?acc) ∈ (mttm_step ?dl)*
    using ac0 by (rule r_into_rtrancl)
  have reach: (init_config_mttm (finite_patch M table N) w, ?acc) ∈ (mttm_step ?dl)*
    using le sc ac by (meson rtrancl_trans)
  have mt_state ?acc = FP_Run (t_tm M) by simp
  thus ?thesis using reach dl by auto
qed
```

A long accepted input (in M's language) reaches M's accept state by overflowing into the rewind, walking back to M's initial configuration, and then running M's accepting computation under the run relabelling.

```
lemma fp_long_run:
  assumes kpos: 0 < k_tm M and wSg: set w ⊆ Sigma_tm M
    and Nlt: N < length w and wL: w ∈ Lang_mttm M
  shows ∃c. (init_config_mttm (finite_patch M table N) w, c)
              ∈ (mttm_step (delta_tm (finite_patch M table N)))*
            ∧ mt_state c = FP_Run (t_tm M)
proof −
  let ?dl = fp_delta (Sigma_tm M) (bl_tm M) (le_tm M) (k_tm M)
              (s_tm M) (t_tm M) table N (delta_tm M)
  let ?init = init_config_mttm (finite_patch M table N) w
  let ?sc = fp_scan_config (bl_tm M) (le_tm M) (k_tm M) w
  let ?rw = fp_rewind_config (bl_tm M) (le_tm M) (k_tm M) w
    have dl: delta_tm (finite_patch M table N) = ?dl by (rule finite_patch_delta_tm)
  have Nle: N ≤ length w using Nlt by simp
  have N1le: N + 1 ≤ length w using Nlt by simp
  have le: (?init, ?sc 0) ∈ (mttm_step ?dl)*
    using fp_le_step[OF kpos] by (rule r_into_rtrancl)
  have sc: (?sc 0, ?sc N) ∈ (mttm_step ?dl)*
        using fp_scan_chain[OF kpos Nle order_refl wSg] by (rule relpow_imp_rtrancl)
  have ov: (?sc N, ?rw (N + 1)) ∈ (mttm_step ?dl)*
    using fp_overflow_step[OF kpos Nlt wSg] by (rule r_into_rtrancl)
  have rc: (?rw (N + 1), ?rw 0) ∈ (mttm_step ?dl)*
    using fp_rewind_chain[OF kpos N1le wSg] by (rule relpow_imp_rtrancl)
  have dn: (?rw 0, fp_lift (init_config_mttm M w)) ∈ (mttm_step ?dl)*
```

```
  using fp_rewind_done_step[OF kpos] by (rule r_into_rtrancl)
 from wL obtain w' n where mreach:
    (init_config_mttm M w, Config_M (t_tm M) w' n) ∈ (mttm_step (delta_tm
M))*
  unfolding Lang_mttm_def by auto
 have sim: (fp_lift (init_config_mttm M w), Config_M (FP_Run (t_tm M)) w'
n)
           ∈ (mttm_step ?dl)*
  using fp_run_rtrancl[OF mreach] by simp
 have r1: (?init, ?sc N) ∈ (mttm_step ?dl)* using le sc by (rule rtrancl_trans)
 have r2: (?init, ?rw (N + 1)) ∈ (mttm_step ?dl)* using r1 ov by (rule
rtrancl_trans)
 have r3: (?init, ?rw 0) ∈ (mttm_step ?dl)* using r2 rc by (rule rtrancl_trans)
 have r4: (?init, fp_lift (init_config_mttm M w)) ∈ (mttm_step ?dl)*
  using r3 dn by (rule rtrancl_trans)
 have reach: (?init, Config_M (FP_Run (t_tm M)) w' n) ∈ (mttm_step ?dl)*
  using r4 sim by (rule rtrancl_trans)
 have mt_state (Config_M (FP_Run (t_tm M)) w' n) = FP_Run (t_tm M) by
simp
 thus ?thesis using reach dl by auto
qed
```

Forward language inclusion: every input satisfying the dispatch specification (short inputs decided by the table, long inputs by M's language) is accepted by the patch.

```
theorem finite_patch_language_forward:
 assumes vM: valid_mttm M
 shows {w. set w ⊆ Sigma_tm M
           ∧ (if length w ≤ N then table w else w ∈ Lang_mttm M)}
       ⊆ Lang_mttm (finite_patch M table N)
proof
 fix w assume w ∈ {w. set w ⊆ Sigma_tm M
                    ∧ (if length w ≤ N then table w else w ∈ Lang_mttm M)}
 hence wSg: set w ⊆ Sigma_tm M
   and disp: if length w ≤ N then table w else w ∈ Lang_mttm M by auto
 have kpos: 0 < k_tm M by (rule valid_mttm_k_pos[OF vM])
 have ex: ∃ c. (init_config_mttm (finite_patch M table N) w, c)
               ∈ (mttm_step (delta_tm (finite_patch M table N)))*
             ∧ mt_state c = FP_Run (t_tm M)
 proof (cases length w ≤ N)
   case True
   hence tab: table w using disp by simp
   show ?thesis using kpos wSg True tab by (rule fp_short_run)
 next
   case False
   hence Nlt: N < length w by simp
   have wL: w ∈ Lang_mttm M using disp False by simp
   show ?thesis using kpos wSg Nlt wL by (rule fp_long_run)
 qed
```

```
  then obtain c where reach: (init_config_mttm (finite_patch M table N) w, c)
                           ∈ (mttm_step (delta_tm (finite_patch M table N)))*
    and cst: mt_state c = FP_Run (t_tm M) by blast
  obtain w' n where c_eq: c = Config_M (FP_Run (t_tm M)) w' n
    using cst by (cases c) auto
  show w ∈ Lang_mttm (finite_patch M table N)
    unfolding Lang_mttm_def finite_patch_Sigma_tm finite_patch_t_tm
    using wSg reach c_eq by auto
qed
```

6.13 Reverse language direction

The front (scan and rewind) is deterministic irrespective of M: its families are the only ones with a *FP_Scan* or *FP_Rewind* source, and they are pairwise disjoint on the read (the endmarker, the blank, and the input alphabet being distinct). The wrapped machine's nondeterminism is confined to the *FP_Run* phase, so soundness needs no *det_mttm M*.

```
lemma fp_delta_front_functional:
  assumes leS: le ∉ Sg and blS: bl ∉ Sg and blle: bl ≠ le
    and src: (∃ u. q = FP_Scan u) ∨ q = FP_Rewind
    and tr1: (q, a, p1, b1, d1) ∈ fp_delta Sg bl le k st tt table N deltaM
    and tr2: (q, a, p2, b2, d2) ∈ fp_delta Sg bl le k st tt table N deltaM
  shows (p1, b1, d1) = (p2, b2, d2)
  using src tr1 tr2 leS blS blle
  unfolding fp_delta_def
  by (auto dest: fr_read_inj)

lemma fp_step_front_functional:
  assumes leS: le ∉ Sg and blS: bl ∉ Sg and blle: bl ≠ le
    and src: (∃ u. mt_state c = FP_Scan u) ∨ mt_state c = FP_Rewind
    and step1: (c, c1) ∈ mttm_step (fp_delta Sg bl le k st tt table N deltaM)
    and step2: (c, c2) ∈ mttm_step (fp_delta Sg bl le k st tt table N deltaM)
  shows c1 = c2
proof −
  from step1 obtain q ts n q1' a1 d1 where
      c_eq: c = Config_M q ts n
    and c1_eq: c1 = Config_M q1' (λi. (ts i)(n i := a1 i)) (λi. go_dir (d1 i) (n i))
    and tr1: (q, λi. ts i (n i), q1', a1, d1)
             ∈ fp_delta Sg bl le k st tt table N deltaM
    by (auto elim: mttm_step.cases)
  from step2 obtain q2' a2 d2 where
      c2_eq: c2 = Config_M q2' (λi. (ts i)(n i := a2 i)) (λi. go_dir (d2 i) (n i))
    and tr2: (q, λi. ts i (n i), q2', a2, d2)
             ∈ fp_delta Sg bl le k st tt table N deltaM
    using c_eq by (auto elim: mttm_step.cases)
  have srcq: (∃ u. q = FP_Scan u) ∨ q = FP_Rewind using src c_eq by simp
  have (q1', a1, d1) = (q2', a2, d2)
    by (rule fp_delta_front_functional[OF leS blS blle srcq tr1 tr2])
```

```
  thus ?thesis using c1_eq c2_eq by simp
qed
```

Reverse simulation: a step out of a run-relabelled configuration is a run relabelling of an M step (only the run-lift family has an *FP_Run* source).

```
lemma fp_run_step_rev:
  assumes step: (fp_lift cM, c')
              ∈ mttm_step (fp_delta Sg bl le k st tt table N deltaM)
  shows ∃ cM'. c' = fp_lift cM' ∧ (cM, cM') ∈ mttm_step deltaM
proof −
  obtain q0 ts n where cM_eq: cM = Config_M q0 ts n by (cases cM)
  have lift_eq: fp_lift cM = Config_M (FP_Run q0) ts n using cM_eq by simp
  from step[unfolded lift_eq] obtain q' a d where
      c'_eq: c' = Config_M q' (λi. (ts i)(n i := a i)) (λi. go_dir (d i) (n i))
    and tr: (FP_Run q0, λi. ts i (n i), q', a, d)
              ∈ fp_delta Sg bl le k st tt table N deltaM
    by (auto elim: mttm_step.cases)
  from tr obtain q0' where q'_eq: q' = FP_Run q0'
    and trM: (q0, λi. ts i (n i), q0', a, d) ∈ deltaM
    unfolding fp_delta_def by auto
  let ?cM' = Config_M q0' (λi. (ts i)(n i := a i)) (λi. go_dir (d i) (n i))
  have c' = fp_lift ?cM' using c'_eq q'_eq by simp
  moreover have (cM, ?cM') ∈ mttm_step deltaM
    unfolding cM_eq
  proof (rule mttm_step.step)
    show (q0, λi. ts i (n i), q0', a, d) ∈ deltaM by (rule trM)
  qed
  ultimately show ?thesis by blast
qed
```

The two halting states are sinks: M's accept (relabelled) has no successor because M's transitions never leave it, and the reject sink is the source of no family.

```
lemma fp_run_t_sink:
  assumes vM: valid_mttm M
    and step: (c, c') ∈ mttm_step (fp_delta Sg bl le k st tt table N (delta_tm M))
    and st: mt_state c = FP_Run (t_tm M)
  shows False
proof −
  from step obtain q ts n q' a d where c_eq: c = Config_M q ts n
    and tr: (q, λi. ts i (n i), q', a, d)
              ∈ fp_delta Sg bl le k st tt table N (delta_tm M)
    by (auto elim: mttm_step.cases)
  have q_eq: q = FP_Run (t_tm M) using st c_eq by simp
  from tr q_eq obtain q0' where
      trM: (t_tm M, λi. ts i (n i), q0', a, d) ∈ delta_tm M
    unfolding fp_delta_def by auto
  have t_tm M ∈ Q_tm M − {t_tm M, r_tm M}
    using valid_mttm_delta_set[OF vM] trM by blast
```

```
  thus False by simp
qed

lemma fp_rej_sink:
  assumes step: (c, c') ∈ mttm_step (fp_delta Sg bl le k st tt table N deltaM)
    and st: mt_state c = FP_Rej
  shows False
proof −
  from step obtain q ts n q' a d where c_eq: c = Config_M q ts n
    and tr: (q, λi. ts i (n i), q', a, d)
          ∈ fp_delta Sg bl le k st tt table N deltaM
    by (auto elim: mttm_step.cases)
  have q = FP_Rej using st c_eq by simp
  thus False using tr unfolding fp_delta_def by auto
qed
```

State of each canonical configuration.

```
lemma mt_state_fp_scan_config:
  mt_state (fp_scan_config bl le k w m) = FP_Scan (take m w)
  by (simp add: fp_scan_config_def)

lemma mt_state_fp_rewind_config:
  mt_state (fp_rewind_config bl le k w p) = FP_Rewind
  by (simp add: fp_rewind_config_def)

lemma mt_state_init_finite_patch:
  mt_state (init_config_mttm (finite_patch M table N) w) = FP_Scan []
  by (simp add: finite_patch_def)

lemma mt_state_fp_lift:
  mt_state (fp_lift cM) = FP_Run (mt_state cM)
  by (cases cM) simp
```

The reachability invariant. From the initial configuration on an input over M's alphabet, every reachable configuration is: the initial one; a scan configuration below the cutoff; (long inputs only) a rewind configuration or a run relabelling of an M-reachable configuration; a short accept certifying the table; or a reject.

```
definition fp_inv ::
  ('q, 'a) mttm ⇒ ('a list ⇒ bool) ⇒ nat ⇒ 'a list
    ⇒ ('a, ('q, 'a) fp_state) mt_config ⇒ bool
where
  fp_inv M table N w c ⟷
    c = init_config_mttm (finite_patch M table N) w
    ∨ (∃ m. m ≤ length w ∧ m ≤ N
          ∧ c = fp_scan_config (bl_tm M) (le_tm M) (k_tm M) w m)
    ∨ (N < length w ∧ (∃ p. p ≤ N + 1
          ∧ c = fp_rewind_config (bl_tm M) (le_tm M) (k_tm M) w p))
    ∨ (N < length w ∧ (∃ cM. (init_config_mttm M w, cM)
```

$\in (mttm\_step\ (delta\_tm\ M))^* \wedge c = fp\_lift\ cM))$
$\vee\ (length\ w \leq N \wedge table\ w \wedge mt\_state\ c = FP\_Run\ (t\_tm\ M))$
$\vee\ mt\_state\ c = FP\_Rej$

Closure of the invariant under a step. Front configurations (init, scan, rewind) have a unique successor given by the matching step lemma (*fp_step_front_functional*); a run configuration steps to another run configuration (*fp_run_step_rev*); the two halting states are sinks.

lemma *fp_inv_closed*:
assumes *vM*: *valid_mttm M* **and** *blle*: *bl_tm M* $\neq$ *le_tm M*
and *wSg*: *set w* $\subseteq$ *Sigma_tm M*
and *inv*: *fp_inv M table N w c*
and *step*: (*c*, *c'*) $\in$ *mttm_step* (*fp_delta* (*Sigma_tm M*) (*bl_tm M*) (*le_tm M*) (*k_tm M*) (*s_tm M*) (*t_tm M*) *table N* (*delta_tm M*))
shows *fp_inv M table N w c'*
proof −
let *?bl* = *bl_tm M*
let *?le* = *le_tm M*
let *?k* = *k_tm M*
let *?dl* = *fp_delta* (*Sigma_tm M*) *?bl ?le ?k* (*s_tm M*) (*t_tm M*) *table N* (*delta_tm M*)
have *kpos*: *0* < *?k* **by** (*rule valid_mttm_k_pos*[*OF vM*])
have *leS*: *?le* $\notin$ *Sigma_tm M* **by** (*rule valid_mttm_LE_not_Sigma*[*OF vM*])
have *blS*: *?bl* $\notin$ *Sigma_tm M* **by** (*rule valid_mttm_blank_not_Sigma*[*OF vM*])
note *funct* = *fp_step_front_functional*[*OF leS blS blle*]
from *inv*[*unfolded fp_inv_def*] **show** *?thesis*
proof (*elim disjE*)
assume *c* = *init_config_mttm* (*finite_patch M table N*) *w*
note *cinit* = *this*
have *src*: ($\exists$ *u*. *mt_state c* = *FP_Scan u*) $\vee$ *mt_state c* = *FP_Rewind*
using *cinit* **by** (*auto simp*: *mt_state_init_finite_patch*)
have *cs*: (*c*, *fp_scan_config ?bl ?le ?k w 0*) $\in$ *mttm_step ?dl*
using *fp_le_step*[*OF kpos*] *cinit* **by** *simp*
have *c'* = *fp_scan_config ?bl ?le ?k w 0* **by** (*rule funct*[*OF src step cs*])
thus *?thesis* **unfolding** *fp_inv_def* **by** *auto*
next
assume $\exists$ *m*. *m* $\leq$ *length w* $\wedge$ *m* $\leq$ *N*
$\wedge$ *c* = *fp_scan_config ?bl ?le ?k w m*
then obtain *m* **where** *mlw*: *m* $\leq$ *length w* **and** *mN*: *m* $\leq$ *N*
and *c_eq*: *c* = *fp_scan_config ?bl ?le ?k w m* **by** *blast*
have *src*: ($\exists$ *u*. *mt_state c* = *FP_Scan u*) $\vee$ *mt_state c* = *FP_Rewind*
using *c_eq* **by** (*auto simp*: *mt_state_fp_scan_config*)
show *?thesis*
proof (*cases m* < *length w*)
case *True*
show *?thesis*
proof (*cases m* < *N*)
case *True*
have *cs*: (*c*, *fp_scan_config ?bl ?le ?k w* (*Suc m*)) $\in$ *mttm_step ?dl*

```
      using fp_scan_step[OF kpos ‹m < length w› True wSg] c_eq by simp
    have c' = fp_scan_config ?bl ?le ?k w (Suc m)
      by (rule funct[OF src step cs])
    moreover have Suc m ≤ length w using ‹m < length w› by simp
    moreover have Suc m ≤ N using True by simp
    ultimately show ?thesis unfolding fp_inv_def by blast
  next
    case False
    hence mEqN: m = N using mN by simp
    have Nlt: N < length w using ‹m < length w› mEqN by simp
    have cs: (c, fp_rewind_config ?bl ?le ?k w (N + 1)) ∈ mttm_step ?dl
      using fp_overflow_step[OF kpos Nlt wSg] c_eq mEqN by simp
    have c' = fp_rewind_config ?bl ?le ?k w (N + 1)
      by (rule funct[OF src step cs])
    moreover have N + 1 ≤ N + 1 by simp
    ultimately show ?thesis unfolding fp_inv_def using Nlt by blast
  qed
next
  case False
  hence mEq: m = length w using mlw by simp
  hence lenN: length w ≤ N using mN by simp
  show ?thesis
  proof (cases table w)
    case True
    let ?acc = Config_M (FP_Run (t_tm M))
                (λi n. if i < ?k
                       then (if n = 0 then ?le
                             else if i = 0 ∧ n ≤ length w then w ! (n − 1) else ?bl)
                       else ?bl)
                (λi::nat. if i = 0 then length w + 1 else 0)
    have cs: (c, ?acc) ∈ mttm_step ?dl
      using True wSg lenN unfolding c_eq mEq by (rule fp_accept_step)
    have c' = ?acc by (rule funct[OF src step cs])
    hence mt_state c' = FP_Run (t_tm M) by simp
    thus ?thesis unfolding fp_inv_def using lenN True by blast
  next
    case False
    have rex: ∃ d. (fp_scan_config ?bl ?le ?k w (length w), d) ∈ mttm_step ?dl
                   ∧ mt_state d = FP_Rej
      using False wSg lenN by (rule fp_reject_step)
    then obtain d where rstep: (fp_scan_config ?bl ?le ?k w (length w), d)
                               ∈ mttm_step ?dl and dst: mt_state d = FP_Rej
      by blast
    have cs: (c, d) ∈ mttm_step ?dl using rstep c_eq mEq by simp
    have c' = d by (rule funct[OF src step cs])
    thus ?thesis unfolding fp_inv_def using dst by blast
  qed
qed
next
```

```
    assume N < length w ∧ (∃ p. p ≤ N + 1
                            ∧ c = fp_rewind_config ?bl ?le ?k w p)
    then obtain p where Nlt: N < length w and pN1: p ≤ N + 1
      and c_eq: c = fp_rewind_config ?bl ?le ?k w p by blast
    have src: (∃ u. mt_state c = FP_Scan u) ∨ mt_state c = FP_Rewind
      using c_eq by (auto simp: mt_state_fp_rewind_config)
    have N1lw: N + 1 ≤ length w using Nlt by simp
    show ?thesis
    proof (cases p)
      case 0
      have cs: (c, fp_lift (init_config_mttm M w)) ∈ mttm_step ?dl
        using fp_rewind_done_step[OF kpos] c_eq 0 by simp
      have c' = fp_lift (init_config_mttm M w) by (rule funct[OF src step cs])
      moreover have (init_config_mttm M w, init_config_mttm M w)
                    ∈ (mttm_step (delta_tm M))* by simp
      ultimately show ?thesis unfolding fp_inv_def using Nlt by blast
    next
      case (Suc q)
      have Sqlw: Suc q ≤ length w using pN1 Suc N1lw by simp
      have cs: (c, fp_rewind_config ?bl ?le ?k w q) ∈ mttm_step ?dl
        using fp_rewind_walk_step[OF kpos Sqlw wSg] c_eq Suc by simp
      have c' = fp_rewind_config ?bl ?le ?k w q by (rule funct[OF src step cs])
      moreover have q ≤ N + 1 using pN1 Suc by simp
      ultimately show ?thesis unfolding fp_inv_def using Nlt by blast
    qed
  next
    assume N < length w ∧ (∃ cM. (init_config_mttm M w, cM)
                            ∈ (mttm_step (delta_tm M))* ∧ c = fp_lift cM)
    then obtain cM where Nlt: N < length w
      and reachM: (init_config_mttm M w, cM) ∈ (mttm_step (delta_tm M))*
      and c_eq: c = fp_lift cM by blast
    have ∃ cM'. c' = fp_lift cM' ∧ (cM, cM') ∈ mttm_step (delta_tm M)
      using fp_run_step_rev[OF step[unfolded c_eq]] .
    then obtain cM' where c'_eq: c' = fp_lift cM'
      and mstep: (cM, cM') ∈ mttm_step (delta_tm M) by blast
    have (init_config_mttm M w, cM') ∈ (mttm_step (delta_tm M))*
      using reachM mstep by (rule rtrancl_into_rtrancl)
    thus ?thesis unfolding fp_inv_def using Nlt c'_eq by blast
  next
    assume length w ≤ N ∧ table w ∧ mt_state c = FP_Run (t_tm M)
    hence mt_state c = FP_Run (t_tm M) by simp
    hence False using fp_run_t_sink[OF vM step] by simp
    thus ?thesis by simp
  next
    assume mt_state c = FP_Rej
    hence False using fp_rej_sink[OF step] by simp
    thus ?thesis by simp
  qed
qed
```

The invariant holds at every reachable configuration.

```
lemma fp_inv_reach:
  assumes vM: valid_mttm M and blle: bl_tm M ≠ le_tm M
    and wSg: set w ⊆ Sigma_tm M
    and reach: (init_config_mttm (finite_patch M table N) w, c)
                ∈ (mttm_step (delta_tm (finite_patch M table N)))*
  shows fp_inv M table N w c
  using reach
proof (induction rule: rtrancl_induct)
  case base
  show ?case unfolding fp_inv_def by simp
next
  case (step y z)
  have yz: (y, z) ∈ mttm_step (fp_delta (Sigma_tm M) (bl_tm M) (le_tm M)
            (k_tm M) (s_tm M) (t_tm M) table N (delta_tm M))
    using step.hyps(2) by (simp add: finite_patch_delta_tm)
  show ?case by (rule fp_inv_closed[OF vM blle wSg step.IH yz])
qed
```

An accepting reachable configuration certifies the dispatch condition: a short input satisfies the table, a long input is in M's language.

```
lemma fp_inv_accept:
  assumes inv: fp_inv M table N w c and st: mt_state c = FP_Run (t_tm M)
    and wSg: set w ⊆ Sigma_tm M
  shows if length w ≤ N then table w else w ∈ Lang_mttm M
  using inv[unfolded fp_inv_def]
proof (elim disjE)
  assume c = init_config_mttm (finite_patch M table N) w
  hence mt_state c = FP_Scan [] by (simp add: mt_state_init_finite_patch)
  thus ?thesis using st by simp
next
  assume ∃ m. m ≤ length w ∧ m ≤ N
           ∧ c = fp_scan_config (bl_tm M) (le_tm M) (k_tm M) w m
  then obtain m where c = fp_scan_config (bl_tm M) (le_tm M) (k_tm M) w
m by blast
    hence mt_state c = FP_Scan (take m w) by (simp add:
mt_state_fp_scan_config)
  thus ?thesis using st by simp
next
  assume N < length w ∧ (∃ p. p ≤ N + 1
                          ∧ c = fp_rewind_config (bl_tm M) (le_tm M) (k_tm
M) w p)
  then obtain p where c = fp_rewind_config (bl_tm M) (le_tm M) (k_tm M)
w p by blast
  hence mt_state c = FP_Rewind by (simp add: mt_state_fp_rewind_config)
  thus ?thesis using st by simp
next
  assume N < length w ∧ (∃ cM. (init_config_mttm M w, cM)
                          ∈ (mttm_step (delta_tm M))* ∧ c = fp_lift cM)
```

```
  then obtain cM where Nlt: N < length w
    and reachM: (init_config_mttm M w, cM) ∈ (mttm_step (delta_tm M))*
    and c_eq: c = fp_lift cM by blast
  have mt_state cM = t_tm M using st c_eq by (simp add: mt_state_fp_lift)
  then obtain w' n where cM_eq: cM = Config_M (t_tm M) w' n by (cases cM)
auto
  have w ∈ Lang_mttm M
    unfolding Lang_mttm_def using wSg reachM cM_eq by auto
  thus ?thesis using Nlt by simp
next
  assume length w ≤ N ∧ table w ∧ mt_state c = FP_Run (t_tm M)
  thus ?thesis by simp
next
  assume mt_state c = FP_Rej
  thus ?thesis using st by simp
qed
```

Reverse language inclusion, hence the language characterisation: the patch decides exactly the dispatch specification — and this holds for a possibly-nondeterministic M.

```
theorem finite_patch_language_sound:
  assumes vM: valid_mttm M and blle: bl_tm M ≠ le_tm M
  shows Lang_mttm (finite_patch M table N)
        ⊆ {w. set w ⊆ Sigma_tm M
              ∧ (if length w ≤ N then table w else w ∈ Lang_mttm M)}
proof
  fix w assume w ∈ Lang_mttm (finite_patch M table N)
  from this[unfolded Lang_mttm_def finite_patch_Sigma_tm finite_patch_t_tm]
  obtain w' n where wSg: set w ⊆ Sigma_tm M
    and r: (init_config_mttm (finite_patch M table N) w,
            Config_M (FP_Run (t_tm M)) w' n)
           ∈ (mttm_step (delta_tm (finite_patch M table N)))*
    by auto
  have inv: fp_inv M table N w (Config_M (FP_Run (t_tm M)) w' n)
    by (rule fp_inv_reach[OF vM blle wSg r])
  have if length w ≤ N then table w else w ∈ Lang_mttm M
    by (rule fp_inv_accept[OF inv _ wSg]) simp
  thus w ∈ {w. set w ⊆ Sigma_tm M
               ∧ (if length w ≤ N then table w else w ∈ Lang_mttm M)}
    using wSg by simp
qed

theorem finite_patch_language:
  assumes vM: valid_mttm M and blle: bl_tm M ≠ le_tm M
  shows Lang_mttm (finite_patch M table N)
        = {w. set w ⊆ Sigma_tm M
              ∧ (if length w ≤ N then table w else w ∈ Lang_mttm M)}
    using finite_patch_language_forward[OF vM] finite_patch_language_sound[OF vM blle]
```

by *blast*

6.14 Strengthened well-formedness

The left-endmarker write discipline for the patch: the front families are read-only, and the run-lift family inherits it from M's own *le_unique*.

lemma *fp_delta_write_le*:
assumes *dM*: $\forall\, q\ a\ q'\ a'\ d\ j.\ (q,\ a,\ q',\ a',\ d) \in deltaM$
$\longrightarrow\ a'\ j = le \longrightarrow\ a\ j = le$
and *tr*: $(q,\ a,\ q',\ a',\ d) \in$ *fp_delta Sg bl le k st tt table N deltaM*
and *w*: $a'\ j = le$
shows $a\ j = le$
proof −
from *tr* **consider**
(*front*) $a' = a$
| (*run*) *q0 q0'* **where** $(q0,\ a,\ q0',\ a',\ d) \in deltaM$
unfolding *fp_delta_def* **by** *blast*
then show *?thesis*
proof *cases*
case *front* **thus** *?thesis* **using** *w* **by** *simp*
next
case *run* **thus** *?thesis* **using** *dM w* **by** *blast*
qed
qed

The patch of a well-formed machine is well-formed (distinct start / accept / reject and endmarker / blank, plus the endmarker write discipline).

lemma *finite_patch_well_formed*:
assumes *wfM*: *well_formed_mttm M*
shows *well_formed_mttm* (*finite_patch M table N*)
proof −
have *vM*: *valid_mttm M* **using** *wfM* **by** *simp*
have *luM*: *le_unique M* **using** *wfM* **by** *simp*
have *dM*: $\forall\, q\ a\ q'\ a'\ d\ j.\ (q,\ a,\ q',\ a',\ d) \in delta\_tm\ M$
$\longrightarrow\ a'\ j = le\_tm\ M \longrightarrow\ a\ j = le\_tm\ M$
using *luM* **unfolding** *le_unique_def* **by** *simp*
have *valid_mttm* (*finite_patch M table N*) **by** (*rule finite_patch_valid*[*OF vM*])
moreover have *s_tm* (*finite_patch M table N*) $\neq$ *t_tm* (*finite_patch M table N*)
by (*simp add*: *finite_patch_s_tm finite_patch_t_tm*)
moreover have *s_tm* (*finite_patch M table N*) $\neq$ *r_tm* (*finite_patch M table N*)
by (*simp add*: *finite_patch_s_tm finite_patch_r_tm*)
moreover have *le_tm* (*finite_patch M table N*) $\neq$ *bl_tm* (*finite_patch M table N*)
using *wfM* **by** (*simp add*: *finite_patch_le_tm finite_patch_bl_tm*)
moreover have *le_unique* (*finite_patch M table N*)
unfolding *le_unique_def finite_patch_delta_tm finite_patch_le_tm*
using *fp_delta_write_le*[*OF dM*] **by** *blast*

ultimately show *?thesis* **by** *simp*
qed

6.15 Time bounds

The step-counting analogue of *fp_run_rtrancl*: an n-step M computation lifts to an n-step patch computation.

lemma *fp_run_relpow*:
assumes $(c,\ c') \in (mttm\_step\ deltaM)\ ^^\ n$
shows $(fp\_lift\ c,\ fp\_lift\ c')$
$\in (mttm\_step\ (fp\_delta\ Sg\ bl\ le\ k\ st\ tt\ table\ N\ deltaM))\ ^^\ n$
using *assms*
proof (*induction n arbitrary*: c')
case *0*
show *?case* **using** *0* **by** *simp*
next
case (*Suc n*)
from *Suc.prems* **obtain** c'' **where** *sn*: $(c,\ c'') \in (mttm\_step\ deltaM)\ ^^\ n$
and *s1*: $(c'',\ c') \in mttm\_step\ deltaM$
by (*blast elim*: *relpow_Suc_E*)
have $(fp\_lift\ c,\ fp\_lift\ c'')$
$\in (mttm\_step\ (fp\_delta\ Sg\ bl\ le\ k\ st\ tt\ table\ N\ deltaM))\ ^^\ n$
by (*rule Suc.IH*[*OF sn*])
moreover have $(fp\_lift\ c'',\ fp\_lift\ c')$
$\in mttm\_step\ (fp\_delta\ Sg\ bl\ le\ k\ st\ tt\ table\ N\ deltaM)$
by (*rule fp_run_step*[*OF s1*])
ultimately show *?case* **by** (*rule relpow_Suc_I*)
qed

A short accepted input is decided in $length\ w + 2$ steps — the endmarker read, the $length\ w$ scan steps, and the dispatch — without running M (the same $+2$ as the recogniser).

lemma *fp_short_time*:
assumes *vM*: *valid_mttm M* **and** *wSg*: $set\ w \subseteq Sigma\_tm\ M$
and *lenN*: $length\ w \le N$ **and** *tab*: *table w*
shows *accepts_in_time_mttm* (*finite_patch M table N*) *w* (*length w + 2*)
proof −
let *?dl* = *fp_delta* (*Sigma_tm M*) (*bl_tm M*) (*le_tm M*) (*k_tm M*)
(*s_tm M*) (*t_tm M*) *table N* (*delta_tm M*)
let *?R* = *mttm_step ?dl*
let *?sc* = *fp_scan_config* (*bl_tm M*) (*le_tm M*) (*k_tm M*) *w*
let *?init* = *init_config_mttm* (*finite_patch M table N*) *w*
let *?acc* = $Config_M$ (*FP_Run* (*t_tm M*))
(λ*i n*. *if i* < *k_tm M*
then (*if n* = *0 then le_tm M*
else if i = *0* ∧ *n* ≤ *length w then w* ! (*n* − *1*) *else bl_tm M*)
else bl_tm M)
(λ*i*::*nat*. *if i* = *0 then length w* + *1 else 0*)
have *kpos*: *0* < *k_tm M* **by** (*rule valid_mttm_k_pos*[*OF vM*])

```
  have le: (?init, ?sc 0) ∈ ?R ^^ 1 using fp_le_step[OF kpos] by simp
  have sc: (?sc 0, ?sc (length w)) ∈ ?R ^^ (length w)
    by (rule fp_scan_chain[OF kpos order_refl lenN wSg])
    have ac1: (?sc (length w), ?acc) ∈ ?R using tab wSg lenN by (rule
fp_accept_step)
  have ac: (?sc (length w), ?acc) ∈ ?R ^^ 1 using ac1 by simp
  have t1: (?init, ?sc (length w)) ∈ ?R ^^ (1 + length w)
    by (rule relpow_transI[OF le sc])
  have t2: (?init, ?acc) ∈ ?R ^^ (1 + length w + 1)
    by (rule relpow_transI[OF t1 ac])
   have run: (?init, ?acc) ∈ ?R ^^ (length w + 2) using t2 by (simp add:
add.commute)
  have mt_state ?acc = FP_Run (t_tm M) by simp
  thus ?thesis
          unfolding accepts_in_time_mttm_def finite_patch_delta_tm fi-
nite_patch_t_tm
    using run by blast
qed
```

A long accepted input is decided in $t + c0$ steps, where t bounds M's own acceptance time and the additive constant $c0 = 2 * N + 4$ is the finite-control overhead — the endmarker read, the N scan steps to the cutoff, the overflow, the $N + 1$ rewind steps, and the handoff.

```
lemma fp_long_time:
  assumes vM: valid_mttm M and wSg: set w ⊆ Sigma_tm M
    and Nlt: N < length w and acc: accepts_in_time_mttm M w t
  shows accepts_in_time_mttm (finite_patch M table N) w (t + (2 * N + 4))
proof −
  let ?dl = fp_delta (Sigma_tm M) (bl_tm M) (le_tm M) (k_tm M)
              (s_tm M) (t_tm M) table N (delta_tm M)
  let ?R = mttm_step ?dl
  let ?sc = fp_scan_config (bl_tm M) (le_tm M) (k_tm M) w
  let ?rw = fp_rewind_config (bl_tm M) (le_tm M) (k_tm M) w
  let ?init = init_config_mttm (finite_patch M table N) w
  have kpos: 0 < k_tm M by (rule valid_mttm_k_pos[OF vM])
  have Nle: N ≤ length w using Nlt by simp
  have N1le: N + 1 ≤ length w using Nlt by simp
  have le: (?init, ?sc 0) ∈ ?R ^^ 1 using fp_le_step[OF kpos] by simp
  have sc: (?sc 0, ?sc N) ∈ ?R ^^ N
    by (rule fp_scan_chain[OF kpos Nle order_refl wSg])
  have ov: (?sc N, ?rw (N + 1)) ∈ ?R ^^ 1
    using fp_overflow_step[OF kpos Nlt wSg] by simp
  have rc: (?rw (N + 1), ?rw 0) ∈ ?R ^^ (N + 1)
    by (rule fp_rewind_chain[OF kpos N1le wSg])
  have dn: (?rw 0, fp_lift (init_config_mttm M w)) ∈ ?R ^^ 1
    using fp_rewind_done_step[OF kpos] by simp
  from acc obtain n cMn where nt: n ≤ t
    and mreach: (init_config_mttm M w, cMn) ∈ (mttm_step (delta_tm M)) ^^
n
```

and *mst*: *mt_state cMn = t_tm M*
unfolding *accepts_in_time_mttm_def* **by** *blast*
have *sim*: (*fp_lift* (*init_config_mttm M w*), *fp_lift cMn*) ∈ *?R* ^^ *n*
by (*rule fp_run_relpow*[*OF mreach*])
have *c1*: (*?init*, *?sc N*) ∈ *?R* ^^ (*1* + *N*)
by (*rule relpow_transI*[*OF le sc*])
have *c2*: (*?init*, *?rw* (*N* + *1*)) ∈ *?R* ^^ (*1* + *N* + *1*)
by (*rule relpow_transI*[*OF c1 ov*])
have *c3*: (*?init*, *?rw 0*) ∈ *?R* ^^ (*1* + *N* + *1* + (*N* + *1*))
by (*rule relpow_transI*[*OF c2 rc*])
have *c4*: (*?init*, *fp_lift* (*init_config_mttm M w*)) ∈ *?R* ^^ (*1* + *N* + *1* + (*N* + *1*) + *1*)
by (*rule relpow_transI*[*OF c3 dn*])
have *c5*: (*?init*, *fp_lift cMn*) ∈ *?R* ^^ (*1* + *N* + *1* + (*N* + *1*) + *1* + *n*)
by (*rule relpow_transI*[*OF c4 sim*])
— keep the raw step count (*2* ∗ *N* + *4* + *n*) as the witness rather than rewrite the relpow exponent, which the simplifier would peel into a relcomp chain
have *bound*: *1* + *N* + *1* + (*N* + *1*) + *1* + *n* ≤ *t* + (*2* ∗ *N* + *4*) **using** *nt* **by** *simp*
have *mt_state* (*fp_lift cMn*) = *FP_Run* (*t_tm M*) **using** *mst* **by** (*simp add*: *mt_state_fp_lift*)
thus *?thesis*
unfolding *accepts_in_time_mttm_def finite_patch_delta_tm finite_patch_t_tm*
using *c5 bound* **by** *blast*
qed

end
theory *Multitape_Time_Convention*
imports *Multitape_Finite_Patch*
begin

7 Time-bound class predicates and cleanup

Shared vocabulary for the linear-speedup consumers. An *eventual* bound holds for all long enough inputs in the language; a *convention* bound is the Hopcroft–Ullman [3, p. 291] *max*(*n*+*1*, ...) form on the substrate, where the floor is *n* + *2* (the substrate reads the left endmarker before the first input symbol).

definition *time_bounded_ev* :: ('*q*, '*a*) *mttm* ⇒ (*nat* ⇒ *nat*) ⇒ *bool* **where**
time_bounded_ev M g ⟷
(∃ *N*. ∀ *w*∈*Lang_mttm M*. *N* ≤ *length w*
⟶ *accepts_in_time_mttm M w* (*g* (*length w*)))

definition *time_bounded_conv* :: ('*q*, '*a*) *mttm* ⇒ (*nat* ⇒ *nat*) ⇒ *bool* **where**
time_bounded_conv M g ⟷
(∀ *w*∈*Lang_mttm M*.
accepts_in_time_mttm M w (*max* (*length w* + *2*) (*g* (*length w*))))

The cleanup corollary: an eventual bound is upgraded to a convention bound on *all* inputs, at the cost of the finite-control overhead $c0 = 2 * N + 4$, with language, tape count, and determinism preserved. Instantiates *finite_patch* with the (finite, hence unconditionally available) table $w \in Lang\_mttm\ M$.

Hypotheses are the weakest the proof uses — *valid_mttm* and $bl\_tm\ M \neq le\_tm\ M$, matching the *finite_patch* contract lemmas it composes (*finite_patch_language* / *_det* / *fp_short_time* / *fp_long_time*). In particular it does *not* require *le_unique* or the state distinctness of *well_formed_mttm*, so it applies to machines (e.g. the encoding wrap) whose *le*-uniqueness is not separately established.

theorem *time_cleanup*:
assumes *vM*: *valid_mttm M* **and** *blle*: $bl\_tm\ M \neq le\_tm\ M$
and *ev*: *time_bounded_ev M g*
obtains *N* **where**
$Lang\_mttm\ (finite\_patch\ M\ (\lambda w.\ w \in Lang\_mttm\ M)\ N) = Lang\_mttm\ M$
and $k\_tm\ (finite\_patch\ M\ (\lambda w.\ w \in Lang\_mttm\ M)\ N) = k\_tm\ M$
and $det\_mttm\ M \longrightarrow det\_mttm\ (finite\_patch\ M\ (\lambda w.\ w \in Lang\_mttm\ M)\ N)$
and $time\_bounded\_conv\ (finite\_patch\ M\ (\lambda w.\ w \in Lang\_mttm\ M)\ N)\ (\lambda n.\ g\ n + (2 * N + 4))$
proof −
from *ev* **obtain** *N* **where** evN: $\forall w{\in}Lang\_mttm\ M.\ N \leq length\ w \longrightarrow accepts\_in\_time\_mttm\ M\ w\ (g\ (length\ w))$
unfolding *time_bounded_ev_def* **by** *blast*
let $?M' = finite\_patch\ M\ (\lambda w.\ w \in Lang\_mttm\ M)\ N$
have *lang*: $Lang\_mttm\ ?M' = Lang\_mttm\ M$
proof −
have $Lang\_mttm\ ?M' = \{w.\ set\ w \subseteq Sigma\_tm\ M \wedge (if\ length\ w \leq N\ then\ w \in Lang\_mttm\ M\ else\ w \in Lang\_mttm\ M)\}$
by (*rule finite_patch_language*[*OF vM blle*])
also have $\ldots = \{w.\ set\ w \subseteq Sigma\_tm\ M \wedge w \in Lang\_mttm\ M\}$ **by** *simp*
also have $\ldots = Lang\_mttm\ M$ **unfolding** *Lang_mttm_def* **by** *auto*
finally show *?thesis* **.**
qed
have *kpres*: $k\_tm\ ?M' = k\_tm\ M$ **by** (*rule finite_patch_k_tm*)
have *detpres*: $det\_mttm\ M \longrightarrow det\_mttm\ ?M'$
using *finite_patch_det*[*OF vM blle*] **by** *blast*
have *conv*: $time\_bounded\_conv\ ?M'\ (\lambda n.\ g\ n + (2 * N + 4))$
unfolding *time_bounded_conv_def*
proof
fix *w* **assume** $w \in Lang\_mttm\ ?M'$
hence *wL*: $w \in Lang\_mttm\ M$ **using** *lang* **by** *simp*
have *wSg*: $set\ w \subseteq Sigma\_tm\ M$ **using** *wL* **unfolding** *Lang_mttm_def* **by** *simp*
show $accepts\_in\_time\_mttm\ ?M'\ w\ (max\ (length\ w + 2)\ ((\lambda n.\ g\ n + (2 * N + 4))\ (length\ w)))$
proof (*cases* $length\ w \leq N$)

```
    case True
    have accepts_in_time_mttm ?M' w (length w + 2)
      using vM wSg True wL by (rule fp_short_time)
    moreover have length w + 2 ≤ max (length w + 2) (g (length w) + (2 * N
+ 4))
      by simp
    ultimately show ?thesis by (auto elim: accepts_in_time_mttm_mono)
   next
    case False
    hence Nlt: N < length w by simp
    have accepts_in_time_mttm M w (g (length w)) using evN wL Nlt by simp
    hence accepts_in_time_mttm ?M' w (g (length w) + (2 * N + 4))
      by (rule fp_long_time[OF vM wSg Nlt])
    moreover have g (length w) + (2 * N + 4)
                  ≤ max (length w + 2) (g (length w) + (2 * N + 4)) by simp
    ultimately show ?thesis by (auto elim: accepts_in_time_mttm_mono)
   qed
  qed
  show thesis
  proof (rule that[of N])
   show Lang_mttm ?M' = Lang_mttm M by (rule lang)
   show k_tm ?M' = k_tm M by (rule kpres)
   show det_mttm M ⟶ det_mttm ?M' by (rule detpres)
   show time_bounded_conv ?M' (λn. g n + (2 * N + 4)) by (rule conv)
  qed
qed
```

7.1 The clean convention form is unattainable in general

Why the linear-T speedup theorems are stated with a residual $+K$ (all inputs) or an explicit threshold (eventual), never as the clean *time_bounded_conv M* ($\lambda n.\ n + n\ div\ q$) on *all* inputs: that clean form is not attainable for a machine whose small-input computation exceeds the convention floor $n + 2$. The minimal witness *CE* below is a deterministic, valid machine over the empty input alphabet that accepts the empty input in exactly *three* steps (state chain $0 \to 1 \to 2 \to 3$, reading and rewriting the left endmarker in place), one more than the floor *length* [] + 2 = 2. So [] ∈ *Lang_mttm CE* yet ¬ *accepts_in_time_mttm CE* [] 2, hence *time_bounded_conv CE* ($\lambda n.\ n + n\ div\ q$) fails at [] for every *q*. *CE* stands in for the encoding wrap, whose own setup phases (*W_Init* → *W_Buf* → *W_Reset* → *W_Disp*) likewise exceed the floor on tiny inputs; closing the small-input band needs the non-effective finite-exceptions table (an existence-only object; see *time_cleanup*), which is exactly what the clean form would demand and what the speedup construction does not build.

```
definition ce_read :: nat ⇒ nat where
  ce_read = (λj. if j = 0 then 1 else 0)
```

```
definition ce_delta ::
  (nat × (nat ⇒ nat) × nat × (nat ⇒ nat) × (nat ⇒ dir)) set where
  ce_delta = {(0, ce_read, 1, ce_read, λ_. dir.N),
              (1, ce_read, 2, ce_read, λ_. dir.N),
              (2, ce_read, 3, ce_read, λ_. dir.N)}

definition CE :: (nat, nat) mttm where
  CE = MTTM {0,1,2,3,4} {} {0,1} 0 1 ce_delta 0 3 4 1

definition ce_tape :: nat ⇒ nat ⇒ nat where
  ce_tape = (λi n. if i = 0 ∧ n = 0 then 1 else 0)

abbreviation ce_cfg :: nat ⇒ (nat, nat) mt_config where
  ce_cfg q ≡ Config_M q ce_tape (λ_. 0)

lemma ce_delta_tm: delta_tm CE = ce_delta by (simp add: CE_def)
lemma ce_t_tm: t_tm CE = 3 by (simp add: CE_def)

lemma ce_read_eq: (λk. ce_tape k ((λ_. 0::nat) k)) = ce_read
  by (simp add: ce_tape_def ce_read_def fun_eq_iff)

lemma ce_valid: valid_mttm CE
  by (auto simp: CE_def ce_delta_def ce_read_def Pi_iff)

lemma ce_det: det_mttm CE
  by (auto simp: det_mttm_def ce_delta_tm ce_delta_def)

lemma ce_init: init_config_mttm CE [] = ce_cfg 0
  by (auto simp: CE_def ce_tape_def fun_eq_iff)
```

Forward: each state $m \leq 2$ steps deterministically to $m + 1$, the tape and heads unchanged (the endmarker is re-read and re-written in place).

```
lemma ce_step_fwd:
  assumes m ≤ 2
  shows (ce_cfg m, ce_cfg (Suc m)) ∈ mttm_step ce_delta
proof −
  have mem: (m, ce_read, Suc m, ce_read, λ_. dir.N) ∈ ce_delta
    using assms unfolding ce_delta_def by (cases m; simp; presburger)
  have upd: (λk. (ce_tape k)((λ_. 0::nat) k := ce_read k)) = ce_tape
    by (simp add: ce_tape_def ce_read_def fun_eq_iff)
  have mov: (λk. go_dir ((λ_. dir.N) k) ((λ_. 0::nat) k)) = (λ_. 0)
    by simp
  have (Config_M m ce_tape (λ_. 0),
        Config_M (Suc m) (λk. (ce_tape k)((λ_. 0) k := ce_read k))
                         (λk. go_dir ((λ_. dir.N) k) ((λ_. 0) k)))
       ∈ mttm_step ce_delta
    by (rule mttm_step.step) (use mem ce_read_eq in simp)
  thus ?thesis by (simp add: upd mov)
qed
```

Backward: from state m the *only* successor is state $m + 1$ — the single *ce_delta* entry with source m, tape and heads fixed.

lemma *ce_delta_inv*:
(*m*, *ce_read*, *q'*, *a*, *dir*) ∈ *ce_delta* ⟹ *q'* = *Suc m* ∧ *a* = *ce_read* ∧ *dir* = (λ_. *dir.N*)
by (*auto simp*: *ce_delta_def*)

lemma *ce_step_unique*:
(*ce_cfg m*, *cM*) ∈ *mttm_step ce_delta* ⟹ *cM* = *ce_cfg* (*Suc m*)
by (*auto elim*!: *mttm_step.cases*
simp: *ce_delta_def ce_tape_def ce_read_def fun_eq_iff*)

Reachability: in $n \leq 2$ steps from the start the machine is in state $n \leq 2 \neq 3$ — so it cannot have accepted.

lemma *ce_reach*:
(*ce_cfg 0*, *cM*) ∈ (*mttm_step ce_delta*) ^^ *n* ⟹ *n* ≤ *2* ⟹ *cM* = *ce_cfg n*
proof (*induction n arbitrary*: *cM*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc m*)
from *Suc.prems*(*1*) **obtain** *c* **where**
rm: (*ce_cfg 0*, *c*) ∈ (*mttm_step ce_delta*) ^^ *m* **and**
st: (*c*, *cM*) ∈ *mttm_step ce_delta*
by (*auto elim*: *relpow_Suc_E*)
have *m* ≤ *2* **using** *Suc.prems*(*2*) **by** *simp*
from *Suc.IH*[*OF rm this*] **have** *c* = *ce_cfg m* **.**
with *st* **have** (*ce_cfg m*, *cM*) ∈ *mttm_step ce_delta* **by** *simp*
from *ce_step_unique*[*OF this*] **show** *?case* **.**
qed

lemma *ce_lang*: [] ∈ *Lang_mttm CE*
proof −
have *s0*: (*ce_cfg 0*, *ce_cfg* (*Suc 0*)) ∈ *mttm_step ce_delta*
by (*rule ce_step_fwd*) *simp*
have *s1*: (*ce_cfg* (*Suc 0*), *ce_cfg* (*Suc* (*Suc 0*))) ∈ *mttm_step ce_delta*
by (*rule ce_step_fwd*) *simp*
have *s2*: (*ce_cfg* (*Suc* (*Suc 0*)), *ce_cfg* (*Suc* (*Suc* (*Suc 0*)))) ∈ *mttm_step ce_delta*
by (*rule ce_step_fwd*) *simp*
have (*ce_cfg 0*, *ce_cfg* (*Suc* (*Suc* (*Suc 0*)))) ∈ (*mttm_step ce_delta*) ^^ (*Suc* (*Suc* (*Suc 0*)))
by (*rule relpow_Suc_I2*[*OF s0 relpow_Suc_I2*[*OF s1 relpow_Suc_I2*[*OF s2 relpow_0_I*]]])
hence (*ce_cfg 0*, *ce_cfg* (*Suc* (*Suc* (*Suc 0*)))) ∈ (*mttm_step ce_delta*)*
by (*rule relpow_imp_rtrancl*)
hence (*init_config_mttm CE* [], *Config*$_M$ (*t_tm CE*) *ce_tape* (λ_. *0*))
∈ (*mttm_step* (*delta_tm CE*))*
by (*simp add*: *ce_init ce_delta_tm ce_t_tm numeral_3_eq_3*)

```
  thus ?thesis
    unfolding Lang_mttm_def by (auto simp: CE_def)
qed

lemma ce_not_accepts: ¬ accepts_in_time_mttm CE [] 2
proof
  assume accepts_in_time_mttm CE [] 2
  then obtain n cM where nle: n ≤ 2
    and reach: (init_config_mttm CE [], cM) ∈ (mttm_step (delta_tm CE)) ^^ n
    and acc: mt_state cM = t_tm CE
    unfolding accepts_in_time_mttm_def by blast
  from reach have (ce_cfg 0, cM) ∈ (mttm_step ce_delta) ^^ n
    by (simp add: ce_init ce_delta_tm)
  from ce_reach[OF this nle] have mt_state cM = n by simp
  with acc nle show False by (simp add: ce_t_tm)
qed

theorem clean_convention_unattainable:
  ¬ time_bounded_conv CE (λn. n + n div q)
proof
  assume time_bounded_conv CE (λn. n + n div q)
  hence ∀ w∈Lang_mttm CE.
          accepts_in_time_mttm CE w (max (length w + 2) (length w + length w
div q))
    by (simp add: time_bounded_conv_def)
  from bspec[OF this ce_lang] have accepts_in_time_mttm CE [] 2
    by (simp add: numeral_2_eq_2)
  with ce_not_accepts show False by simp
qed

end
```

References

[1] F. J. Balbach. The Cook-Levin theorem. *Archive of Formal Proofs*, January 2023. https://isa-afp.org/entries/Cook_Levin.html, Formal proof development.

[2] 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.

[3] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[4] J. Xu, X. Zhang, C. Urban, S. J. C. Joosten, and F. Regensburger. Universal Turing machine. *Archive of Formal Proofs*, Febru-

ary 2019. https://isa-afp.org/entries/Universal_Turing_Machine.html, Formal proof development.