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*⟩

hide_const (**open**) *L R N*

Blank-tail finiteness: the set of total functions *nat* ⇒ *'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*⟩

lemma *relpow_transI*:
(*x*, *y*) ∈ *R*^^*n* ⟹ (*y*, *z*) ∈ *R*^^*m* ⟹ (*x*, *z*) ∈ *R*^^(*n* + *m*)
⟨*proof*⟩

lemma *relpow_mono*: **fixes** *R* :: *'a rel*
shows *R* ⊆ *S* ⟹ *R*^^*n* ⊆ *S*^^*n*

⟨*proof*⟩

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 $R :: ('s \times 's)\ set$
    and $P :: nat \Rightarrow 's \Rightarrow bool$
  assumes *step*: $\bigwedge i\ c.\ [\![\ i < n;\ P\ i\ c\ ]\!] \Longrightarrow \exists\, c'.\ (c,\ c') \in R \wedge P\ (Suc\ i)\ c'$
    and *base*: $P\ 0\ c0$
  shows $\exists\, c'.\ (c0,\ c') \in R\ \widehat{\ }\ n \wedge P\ n\ c'$
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *valid_mttm_finite_Gamma*:
  assumes *valid_mttm M*
  shows *finite* (Γ_*tm M*)
  ⟨*proof*⟩

lemma *valid_mttm_Sigma_sub_Gamma*:
  assumes *valid_mttm M*
  shows *Sigma_tm M* ⊆ Γ_*tm M*
  ⟨*proof*⟩

lemma *valid_mttm_s_in_Q*:
  assumes *valid_mttm M*
  shows *s_tm M* ∈ *Q_tm M*
  ⟨*proof*⟩

lemma *valid_mttm_t_in_Q*:
assumes *valid_mttm M*
shows *t_tm M* ∈ *Q_tm M*
⟨*proof*⟩

lemma *valid_mttm_r_in_Q*:
assumes *valid_mttm M*
shows *r_tm M* ∈ *Q_tm M*
⟨*proof*⟩

lemma *valid_mttm_blank_in_Gamma*:
assumes *valid_mttm M*
shows *bl_tm M* ∈ Γ*_tm M*
⟨*proof*⟩

lemma *valid_mttm_blank_not_Sigma*:
assumes *valid_mttm M*
shows *bl_tm M* ∉ *Sigma_tm M*
⟨*proof*⟩

lemma *valid_mttm_LE_in_Gamma*:
assumes *valid_mttm M*
shows *le_tm M* ∈ Γ*_tm M*
⟨*proof*⟩

lemma *valid_mttm_LE_not_Sigma*:
assumes *valid_mttm M*
shows *le_tm M* ∉ *Sigma_tm M*
⟨*proof*⟩

lemma *valid_mttm_t_neq_r*:
assumes *valid_mttm M*
shows *t_tm M* ≠ *r_tm M*
⟨*proof*⟩

lemma *valid_mttm_k_pos*:
assumes *valid_mttm M*
shows *0* < *k_tm M*
⟨*proof*⟩

lemma *valid_mttm_delta_set*:
assumes *valid_mttm M*
shows *delta_tm M* ⊆
(*Q_tm M* − {*t_tm M*, *r_tm M*})
× (*UNIV* → Γ*_tm M*)
× *Q_tm M*
× (*UNIV* → Γ*_tm M*)
× (*UNIV* → *UNIV*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*$\}$
⟨*proof*⟩

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*
⟨*proof*⟩

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* ... *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* $\wedge$ *a' j* = *bl_tm M* $\wedge$ *d j* = *dir.N*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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* $\subseteq$ *Sigma_tm M*
and *reach*: $(init\_config\_mttm\ M\ w,\ c) \in (mttm\_step\ (delta\_tm\ M))^*$
shows *valid_config_mttm M c*
⟨*proof*⟩

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*: $i \geq k\_tm\ M$
shows *mt_tape c i p* = *bl_tm M*
⟨*proof*⟩

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*: $(c,\ c') \in mttm\_step\ (delta\_tm\ M)\ \widehat{\ }\ n$
shows *valid_config_mttm M c'*
⟨*proof*⟩

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* $\subseteq$ *Sigma_tm M*
and *reach*: $(init\_config\_mttm\ M\ w,\ Config_M\ q\ ts\ n)$
$\in (mttm\_step\ (delta\_tm\ M))^*$
and *kK*: $k < k\_tm\ M$
shows *ts k 0* = *le_tm M*
⟨*proof*⟩

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* $\subseteq$ *Sigma_tm M*
    and *reach*: $(init\_config\_mttm\ M\ w,\ c) \in (mttm\_step\ (delta\_tm\ M))^*$
    and *p_ne_0*: $p \neq 0$
    and *bl_neq_le*: *bl_tm M* $\neq$ *le_tm M*
  shows *mt_tape c k p* $\neq$ *le_tm M*
⟨*proof*⟩

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*⟩

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*⟩

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

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

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 $(c,\ c') \in mttm\_step\ \delta\ ^{\frown\frown}\ n$
and $p > mt\_pos\ c\ k + n \vee p + n < mt\_pos\ c\ k$
shows $mt\_tape\ c'\ k\ p = mt\_tape\ c\ k\ p$
⟨*proof*⟩

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*: $(c,\ c') \in mttm\_step\ (delta\_tm\ M)$
and *LE_at0*: $mt\_tape\ c\ k\ 0 = le\_tm\ M$
shows $mt\_tape\ c'\ k\ 0 = le\_tm\ M$
⟨*proof*⟩

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*: $(c,\ c') \in mttm\_step\ (delta\_tm\ M)\ ^{\frown\frown}\ n$
and *LE_at0*: $mt\_tape\ c\ k\ 0 = le\_tm\ M$
shows $mt\_tape\ c'\ k\ 0 = le\_tm\ M$
⟨*proof*⟩

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*: $(c,\ c') \in mttm\_step\ \delta$
  obtains $q\ q'\ a\ dr$ **where**
    $mt\_state\ c = q$
   and $mt\_state\ c' = q'$
   and $(q,\ \lambda k.\ mt\_tape\ c\ k\ (mt\_pos\ c\ k),\ q',\ a,\ dr) \in \delta$
   and $\bigwedge kk.\ mt\_pos\ c'\ kk = go\_dir\ (dr\ kk)\ (mt\_pos\ c\ kk)$
   and $\bigwedge kk\ p.\ p \neq mt\_pos\ c\ kk \Longrightarrow mt\_tape\ c'\ kk\ p = mt\_tape\ c\ kk\ p$
   and $\bigwedge kk.\ mt\_tape\ c'\ kk\ (mt\_pos\ c\ kk) = a\ kk$
⟨*proof*⟩

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*: $(c,\ c') \in mttm\_step\ \delta$
   and *no_write*: $\bigwedge q\ a\ q'\ a'\ d.\ (q,\ a,\ q',\ a',\ d) \in \delta \Longrightarrow a' = a$
  shows $mt\_tape\ c' = mt\_tape\ c$
⟨*proof*⟩

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*⟩

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

lemma *accepts_in_time_mttm_mono*:
  $accepts\_in\_time\_mttm\ M\ w\ t \Longrightarrow t \leq t' \Longrightarrow accepts\_in\_time\_mttm\ M\ w\ t'$
  ⟨*proof*⟩

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 $\Longrightarrow$ *mt_state c2 = mt_state c1*
⟨*proof*⟩

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 ≠ 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*: ∀ *k*. *d k ≠ 0* ⟶ *mt_tape c1 k 0 = le_tm M*
    and *rel*: *shift_rel d c1 c2*
    and *step*: (*c1*, *c1'*) ∈ *mttm_step* (*delta_tm M*)
  shows ∃ *c2'*. (*c2*, *c2'*) ∈ *mttm_step* (*delta_tm M*) ∧ *shift_rel d c1' c2'*
⟨*proof*⟩

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*⟩

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 $[\![\forall k.\ d\ k \neq 0 \longrightarrow mt\_tape\ c1\ k\ 0 = le\_tm\ M;\ shift\_rel\ d\ c1\ c2;$
$(c1,\ c1') \in (mttm\_step\ (delta\_tm\ M))\ ^^\ n\ ]\!]$
$\Longrightarrow \exists c2'.\ (c2,\ c2') \in (mttm\_step\ (delta\_tm\ M))\ ^^\ n$
$\wedge\ shift\_rel\ d\ c1'\ c2'$
⟨*proof*⟩

lemma *mttm_relpow_shift_reverse*:
fixes $M :: ('q, 'a)\ mttm$
assumes *vM*: *valid_mttm M*
shows $[\![\forall k.\ d\ k \neq 0 \longrightarrow mt\_tape\ c1\ k\ 0 = le\_tm\ M;\ shift\_rel\ d\ c1\ c2;$
$(c2,\ c2') \in (mttm\_step\ (delta\_tm\ M))\ ^^\ n\ ]\!]$
$\Longrightarrow \exists c1'.\ (c1,\ c1') \in (mttm\_step\ (delta\_tm\ M))\ ^^\ n$
$\wedge\ shift\_rel\ d\ c1'\ c2'$
⟨*proof*⟩

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*: $\forall k.\ d\ k \neq 0 \longrightarrow mt\_tape\ c1\ k\ 0 = le\_tm\ M$
and *rel*: *shift_rel d c1 c2*
and *run*: $(c1,\ ca) \in (mttm\_step\ (delta\_tm\ M))\ ^^\ n$
and *acc*: *mt_state ca = qa*
shows $\exists cb.\ (c2,\ cb) \in (mttm\_step\ (delta\_tm\ M))\ ^^\ n \wedge mt\_state\ cb = qa$
⟨*proof*⟩

lemma *shift_reach_state_reverse*:
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 *run*: $(c2,\ cb) \in (mttm\_step\ (delta\_tm\ M))\ ^^\ n$
and *acc*: *mt_state cb = qa*
shows $\exists ca.\ (c1,\ ca) \in (mttm\_step\ (delta\_tm\ M))\ ^^\ n \wedge mt\_state\ ca = qa$
⟨*proof*⟩

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 = (λ*j*. *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 = (λ*j*. *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
⟨*proof*⟩

lemma *finite_recogniser_Sigma_tm*:
Sigma_tm (finite_recogniser Sg Gamma bl le F k) = Sg
⟨*proof*⟩

lemma *finite_recogniser_Gamma_tm*:
Γ_tm (finite_recogniser Sg Gamma bl le F k) = Gamma
⟨*proof*⟩

lemma *finite_recogniser_bl_tm*:
bl_tm (finite_recogniser Sg Gamma bl le F k) = bl
⟨*proof*⟩

lemma *finite_recogniser_le_tm*:
le_tm (finite_recogniser Sg Gamma bl le F k) = le
⟨*proof*⟩

lemma *finite_recogniser_s_tm*:
s_tm (finite_recogniser Sg Gamma bl le F k) = FR_Scan []
⟨*proof*⟩

lemma *finite_recogniser_t_tm*:
t_tm (finite_recogniser Sg Gamma bl le F k) = FR_Acc
⟨*proof*⟩

lemma *finite_recogniser_r_tm*:
r_tm (finite_recogniser Sg Gamma bl le F k) = FR_Rej
⟨*proof*⟩

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*}
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*) ∈ *fr_delta Sg bl le F k N* ⟹ *d j* ∈ {*dir.N*, *dir.R*}
⟨*proof*⟩

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*) ∈ *fr_delta Sg bl le F k N*
and *a j* = *le*

shows $a'\ j = le \wedge\ d\ j \in \{dir.N,\ dir.R\}$
⟨*proof*⟩

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*⟩

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*: $Sg \subseteq Gamma$
and *blG*: $bl \in Gamma$
and *blS*: $bl \notin Sg$
and *leG*: $le \in Gamma$
and *leS*: $le \notin Sg$
and *blle*: $bl \neq le$
and *kpos*: $0 < k$
shows *well_formed_mttm* (*finite_recogniser Sg Gamma bl le F k*)
⟨*proof*⟩

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* $\Longrightarrow$ *?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*: $le \notin Sg$ **and** *blS*: $bl \notin Sg$ **and** *blle*: $bl \neq le$
shows *det_mttm* (*finite_recogniser Sg Gamma bl le F k*)

⟨*proof*⟩

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))
      (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)

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*: *∀ w∈F. set w ⊆ Sg* **and** *finF*: *finite F*
shows *F ⊆ Lang_mttm* (*finite_recogniser Sg Gamma bl le F k*)
⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

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* $\subseteq$ *Sg*
  shows $\exists\, c'.$ (*fr_run_config bl le k w m*, c')
          $\in$ *mttm_step* (*fr_delta Sg bl le F k* (*fr_N F*)) $\wedge$ *mt_state* $c' =$ *FR_Rej*
⟨*proof*⟩

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

lemma *fr_reject_step*:
  assumes *wnF*: $w \notin F$ **and** *wSg*: *set w* $\subseteq$ *Sg* **and** *lenN*: *length w* $\le$ *fr_N F*
  shows $\exists\, c'.$ (*fr_run_config bl le k w* (*length w*), c')
          $\in$ *mttm_step* (*fr_delta Sg bl le F k* (*fr_N F*)) $\wedge$ *mt_state* $c' =$ *FR_Rej*
⟨*proof*⟩

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 ⟷
    ($\exists\, m.\ m \le$ *length w* $\wedge\ m \le$ *fr_N F* $\wedge\ c =$ *fr_run_config bl le k w m*)
    $\vee\ c =$ *init_config_mttm* (*finite_recogniser Sg Gamma bl le F k*) *w*
    $\vee$ *mt_state c* $=$ *FR_Rej*
    $\vee$ (*mt_state c* $=$ *FR_Acc* $\wedge\ w \in F$)

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* $\notin$ *Sg* **and** *blS*: *bl* $\notin$ *Sg* **and** *blle*: *bl* $\neq$ *le*
    and *kpos*: $0 < k$ **and** *finF*: *finite F* **and** *wSg*: *set w* $\subseteq$ *Sg*
    and *inv*: *fr_inv Sg Gamma bl le F k w c*
    and *step*: $(c,\ c') \in$ *mttm_step* (*fr_delta Sg bl le F k* (*fr_N F*))
  shows *fr_inv Sg Gamma bl le F k w c'*

⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *mt_state_init_finite_recogniser*:
mt_state (*init_config_mttm* (*finite_recogniser Sg Gamma bl le F k*) *w*) = *FR_Scan* []
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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* ::
  $'a\ set \Rightarrow 'a \Rightarrow 'a \Rightarrow nat \Rightarrow 'q \Rightarrow 'q \Rightarrow ('a\ list \Rightarrow bool) \Rightarrow nat$
    $\Rightarrow ('q \times (nat \Rightarrow 'a) \times 'q \times (nat \Rightarrow 'a) \times (nat \Rightarrow dir))\ set$
    $\Rightarrow (('q, 'a)\ fp\_state \times (nat \Rightarrow 'a) \times ('q, 'a)\ fp\_state$
        $\times (nat \Rightarrow 'a) \times (nat \Rightarrow dir))\ set$
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*) $\}$
    $\cup$ $\{$ (*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* $\subseteq$ *Sg* $\wedge$ *length u* $<$ *N* $\wedge$ $x \in$ *Sg* $\}$
    $\cup$ $\{$ (*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* $\subseteq$ *Sg* $\wedge$ *length u* $=$ *N* $\wedge$ $x \in$ *Sg* $\}$
    $\cup$ $\{$ (*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* $\subseteq$ *Sg* $\wedge$ *length u* $\leq$ *N* $\wedge$ *table u* $\}$
    $\cup$ $\{$ (*FP_Scan u*, *fr_read bl le k bl*, *FP_Rej*,
        fr_read bl le k bl, *fr_move dir.N*)
      | $u.$ *set u* $\subseteq$ *Sg* $\wedge$ *length u* $\leq$ *N* $\wedge$ $\neg$ *table u* $\}$
    $\cup$ $\{$ (*FP_Rewind*, *fr_read bl le k x*, *FP_Rewind*,
        fr_read bl le k x, *fr_move dir.L*)
      | $x.$ $x \in$ *Sg* $\}$
    $\cup$ $\{$ (*FP_Rewind*, *fr_read bl le k le*, *FP_Run st*,
        fr_read bl le k le, *fr_move dir.N*) $\}$
    $\cup$ $\{$ (*FP_Run q*, *a*, *FP_Run q'*, *a'*, *d*)
      | $q\ a\ q'\ a'\ d.$ $(q, a, q', a', d) \in$ *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* ::
  $('q, 'a)\ mttm \Rightarrow ('a\ list \Rightarrow bool) \Rightarrow nat \Rightarrow (('q, 'a)\ fp\_state, 'a)\ mttm$
where
  finite_patch M table N =
    MTTM
      (*FP_Scan* ‘ $\{u.$ *set u* $\subseteq$ *Sigma_tm M* $\wedge$ *length u* $\leq$ *N*$\}$
        $\cup$ *FP_Run* ‘ *Q_tm M* $\cup$ $\{$*FP_Rewind*, *FP_Rej*$\}$)
      (*Sigma_tm M*)
      (Γ*_tm M*)

$(bl\_tm\ M)$
$(le\_tm\ M)$
$(fp\_delta\ (Sigma\_tm\ M)\ (bl\_tm\ M)\ (le\_tm\ M)\ (k\_tm\ M)$
$\quad (s\_tm\ M)\ (t\_tm\ M)\ table\ N\ (delta\_tm\ M))$
$(FP\_Scan\ [])$
$(FP\_Run\ (t\_tm\ M))$
FP_Rej
$(k\_tm\ M)$

6.4 Component accessors

lemma *finite_patch_k_tm*:
$k\_tm\ (finite\_patch\ M\ table\ N) = k\_tm\ M$
⟨*proof*⟩

lemma *finite_patch_Sigma_tm*:
$Sigma\_tm\ (finite\_patch\ M\ table\ N) = Sigma\_tm\ M$
⟨*proof*⟩

lemma *finite_patch_Gamma_tm*:
$\Gamma\_tm\ (finite\_patch\ M\ table\ N) = \Gamma\_tm\ M$
⟨*proof*⟩

lemma *finite_patch_bl_tm*:
$bl\_tm\ (finite\_patch\ M\ table\ N) = bl\_tm\ M$
⟨*proof*⟩

lemma *finite_patch_le_tm*:
$le\_tm\ (finite\_patch\ M\ table\ N) = le\_tm\ M$
⟨*proof*⟩

lemma *finite_patch_s_tm*:
$s\_tm\ (finite\_patch\ M\ table\ N) = FP\_Scan\ []$
⟨*proof*⟩

lemma *finite_patch_t_tm*:
$t\_tm\ (finite\_patch\ M\ table\ N) = FP\_Run\ (t\_tm\ M)$
⟨*proof*⟩

lemma *finite_patch_r_tm*:
$r\_tm\ (finite\_patch\ M\ table\ N) = FP\_Rej$
⟨*proof*⟩

lemma *finite_patch_Q_tm*:
$Q\_tm\ (finite\_patch\ M\ table\ N)$
$\quad = FP\_Scan\ `\ \{u.\ set\ u \subseteq Sigma\_tm\ M \wedge length\ u \leq N\}$
$\quad\quad \cup\ FP\_Run\ `\ Q\_tm\ M \cup \{FP\_Rewind,\ FP\_Rej\}$
⟨*proof*⟩

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*)
⟨*proof*⟩

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*: $\forall\, q\ a\ q'\ a'\ d.\ (q,\ a,\ q',\ a',\ d) \in \mathit{deltaM}$
$\longrightarrow (\forall\, j{\geq}k.\ a\ j = \mathit{bl} \wedge a'\ j = \mathit{bl} \wedge d\ j = \mathit{dir}.N)$
and *tr*: $(q,\ a,\ q',\ a',\ d) \in$ *fp_delta Sg bl le k st tt table N deltaM*
and *j*: $k \leq j$
shows $a\ j = \mathit{bl} \wedge a'\ j = \mathit{bl} \wedge d\ j = \mathit{dir}.N$
⟨*proof*⟩

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*:
$le \notin Sg \Longrightarrow x \in Sg \Longrightarrow \mathit{fr\_read}\ \mathit{bl}\ \mathit{le}\ k\ x\ j = \mathit{le} \Longrightarrow j \neq 0$
⟨*proof*⟩

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*: $le \notin Sg$
and *dM*: $\forall\, q\ a\ q'\ a'\ d\ j.\ (q,\ a,\ q',\ a',\ d) \in \mathit{deltaM} \longrightarrow a\ j = \mathit{le}$
$\longrightarrow a'\ j = \mathit{le} \wedge d\ j \in \{\mathit{dir}.N,\ \mathit{dir}.R\}$
and *tr*: $(q,\ a,\ q',\ a',\ d) \in$ *fp_delta Sg bl le k st tt table N deltaM*
and *LE*: $a\ j = \mathit{le}$
shows $a'\ j = \mathit{le} \wedge d\ j \in \{\mathit{dir}.N,\ \mathit{dir}.R\}$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*))*
  ⟨*proof*⟩

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 unchang-

ing) 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⟩
```

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⟩
```

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
  ⟨proof⟩
```

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*)
          (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*::*nat*. *if i* = *0 then length w* + *1 else 0*))
         ∈ *mttm_step* (*fp_delta Sg bl le k st tt table N deltaM*)
⟨*proof*⟩

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*⟩

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*
      (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 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*⟩

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*: $Suc\ q \le length\ w$ **and** *wSg*: $set\ w \subseteq Sg$
shows $(fp\_rewind\_config\ bl\ le\ k\ w\ (Suc\ q), fp\_rewind\_config\ bl\ le\ k\ w\ q)$
$\in mttm\_step\ (fp\_delta\ Sg\ bl\ le\ k\ st\ tt\ table\ N\ deltaM)$
⟨*proof*⟩

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*: $p \le length\ w$ **and** *wSg*: $set\ w \subseteq Sg$
shows $(fp\_rewind\_config\ bl\ le\ k\ w\ p, fp\_rewind\_config\ bl\ le\ k\ w\ 0)$
$\in (mttm\_step\ (fp\_delta\ Sg\ bl\ le\ k\ st\ tt\ table\ N\ deltaM))\ ^^\ p$
⟨*proof*⟩

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 $(fp\_rewind\_config\ (bl\_tm\ M)\ (le\_tm\ M)\ (k\_tm\ M)\ w\ 0,$
$fp\_lift\ (init\_config\_mttm\ M\ w))$
$\in mttm\_step\ (fp\_delta\ (Sigma\_tm\ M)\ (bl\_tm\ M)\ (le\_tm\ M)\ (k\_tm\ M)$
$(s\_tm\ M)\ (t\_tm\ M)\ table\ N\ (delta\_tm\ M))$
⟨*proof*⟩

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*: $set\ w \subseteq Sigma\_tm\ M$
and *lenN*: $length\ w \le N$ **and** *tab*: $table\ w$
shows $\exists\, c.\ (init\_config\_mttm\ (finite\_patch\ M\ table\ N)\ w,\ c)$
$\in (mttm\_step\ (delta\_tm\ (finite\_patch\ M\ table\ N)))^*$
$\wedge\ mt\_state\ c = FP\_Run\ (t\_tm\ M)$
⟨*proof*⟩

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*: $set\ w \subseteq Sigma\_tm\ M$
and *Nlt*: $N < length\ w$ **and** *wL*: $w \in Lang\_mttm\ M$
shows $\exists\, c.\ (init\_config\_mttm\ (finite\_patch\ M\ table\ N)\ w,\ c)$
$\in (mttm\_step\ (delta\_tm\ (finite\_patch\ M\ table\ N)))^*$
$\wedge\ mt\_state\ c = FP\_Run\ (t\_tm\ M)$
⟨*proof*⟩

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 $\{w.\ set\ w \subseteq Sigma\_tm\ M$
$\wedge\ (if\ length\ w \leq N\ then\ table\ w\ else\ w \in Lang\_mttm\ M)\}$
$\subseteq Lang\_mttm\ (finite\_patch\ M\ table\ N)$
⟨*proof*⟩

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*: $le \notin Sg$ **and** *blS*: $bl \notin Sg$ **and** *blle*: $bl \neq le$
and *src*: $(\exists u.\ q = FP\_Scan\ u) \vee q = FP\_Rewind$
and *tr1*: $(q, a, p1, b1, d1) \in fp\_delta\ Sg\ bl\ le\ k\ st\ tt\ table\ N\ deltaM$
and *tr2*: $(q, a, p2, b2, d2) \in fp\_delta\ Sg\ bl\ le\ k\ st\ tt\ table\ N\ deltaM$
shows $(p1, b1, d1) = (p2, b2, d2)$
⟨*proof*⟩

lemma *fp_step_front_functional*:
assumes *leS*: $le \notin Sg$ **and** *blS*: $bl \notin Sg$ **and** *blle*: $bl \neq le$
and *src*: $(\exists u.\ mt\_state\ c = FP\_Scan\ u) \vee mt\_state\ c = FP\_Rewind$
and *step1*: $(c, c1) \in mttm\_step\ (fp\_delta\ Sg\ bl\ le\ k\ st\ tt\ table\ N\ deltaM)$
and *step2*: $(c, c2) \in mttm\_step\ (fp\_delta\ Sg\ bl\ le\ k\ st\ tt\ table\ N\ deltaM)$
shows $c1 = c2$
⟨*proof*⟩

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*: $(fp\_lift\ cM, c')$
$\in mttm\_step\ (fp\_delta\ Sg\ bl\ le\ k\ st\ tt\ table\ N\ deltaM)$
shows $\exists cM'.\ c' = fp\_lift\ cM' \wedge (cM, cM') \in mttm\_step\ deltaM$
⟨*proof*⟩

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*: $(c, c') \in mttm\_step\ (fp\_delta\ Sg\ bl\ le\ k\ st\ tt\ table\ N\ (delta\_tm\ M))$

and *st*: *mt_state c* = *FP_Run* (*t_tm M*)
shows *False*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *mt_state_fp_rewind_config*:
mt_state (*fp_rewind_config bl le k w p*) = *FP_Rewind*
⟨*proof*⟩

lemma *mt_state_init_finite_patch*:
mt_state (*init_config_mttm* (*finite_patch M table N*) *w*) = *FP_Scan* []
⟨*proof*⟩

lemma *mt_state_fp_lift*:
mt_state (*fp_lift cM*) = *FP_Run* (*mt_state cM*)
⟨*proof*⟩

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*)
∈ (*mttm_step* (*delta_tm M*))* ∧ *c* = *fp_lift cM*))
∨ (*length w* ≤ *N* ∧ *table w* ∧ *mt_state c* = *FP_Run* (*t_tm M*))
∨ *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* ≠ *le_tm M*
    and *wSg*: *set w* ⊆ *Sigma_tm M*
    and *inv*: *fp_inv M table N w c*
    and *step*: (*c*, *c'*) ∈ *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*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*⟩

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*)}
  ⟨*proof*⟩

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*⟩

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*⟩

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 $(c,\ c') \in (mttm\_step\ deltaM)\ \widehat{\ }\ n$
  shows (*fp_lift c*, *fp_lift c'*)
      $\in (mttm\_step\ (fp\_delta\ Sg\ bl\ le\ k\ st\ tt\ table\ N\ deltaM))\ \widehat{\ }\ n$
  ⟨*proof*⟩

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* ⊆ *Sigma_tm M*
    and *lenN*: *length w* ≤ *N* **and** *tab*: *table w*
  shows *accepts_in_time_mttm* (*finite_patch M table N*) *w* (*length w* + *2*)
⟨*proof*⟩

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*⟩

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* $\Rightarrow$ $(nat \Rightarrow nat) \Rightarrow$ *bool* **where**
time_bounded_ev M g $\longleftrightarrow$
$(\exists\, N.\ \forall\, w{\in}Lang\_mttm\ M.\ N \leq length\ w$
$\longrightarrow$ *accepts_in_time_mttm M w* (*g* (*length w*)))

definition *time_bounded_conv* :: $('q, 'a)$ *mttm* $\Rightarrow$ $(nat \Rightarrow nat) \Rightarrow$ *bool* **where**
time_bounded_conv M g $\longleftrightarrow$
$(\forall\, w{\in}Lang\_mttm\ M.$
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*⟩

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 = (λ*j*. *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 = (λ*i 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* ⟨*proof*⟩
lemma *ce_t_tm*: *t_tm CE* = *3* ⟨*proof*⟩

lemma *ce_read_eq*: (λ*k*. *ce_tape k* ((λ_. *0*::*nat*) *k*)) = *ce_read*
⟨*proof*⟩

lemma *ce_valid*: *valid_mttm CE*
⟨*proof*⟩

lemma *ce_det*: *det_mttm CE*
⟨*proof*⟩

lemma *ce_init*: *init_config_mttm CE* [] = *ce_cfg 0*
⟨*proof*⟩

Forward: each state $m \le 2$ 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 $(ce\_cfg\ m,\ ce\_cfg\ (Suc\ m)) \in mttm\_step\ ce\_delta$
⟨*proof*⟩

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*:
$(m,\ ce\_read,\ q',\ a,\ dir) \in ce\_delta \Longrightarrow q' = Suc\ m \wedge a = ce\_read \wedge dir = (\lambda\_.\ dir.N)$
⟨*proof*⟩

lemma *ce_step_unique*:
$(ce\_cfg\ m,\ cM) \in mttm\_step\ ce\_delta \Longrightarrow cM = ce\_cfg\ (Suc\ m)$
⟨*proof*⟩

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*:
$(ce\_cfg\ 0,\ cM) \in (mttm\_step\ ce\_delta)\ ^{\frown\frown}\ n \Longrightarrow n \leq 2 \Longrightarrow cM = ce\_cfg\ n$
⟨*proof*⟩

lemma *ce_lang*: $[] \in Lang\_mttm\ CE$
⟨*proof*⟩

lemma *ce_not_accepts*: $\neg\ accepts\_in\_time\_mttm\ CE\ []\ 2$
⟨*proof*⟩

theorem *clean_convention_unattainable*:
$\neg\ time\_bounded\_conv\ CE\ (\lambda n.\ n + n\ div\ q)$
⟨*proof*⟩

end

References

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

[2] 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.