

A Verified Translation of Multitape Turing Machines into Singletape Turing Machines

Christian Dalvit and René Thiemann

March 24, 2023

Abstract

We define single- and multitape Turing machines (TMs) and verify a translation from multitape TMs to singletape TMs. In particular, the following results have been formalized: the accepted languages coincide, and whenever the multitape TM runs in $\mathcal{O}(f(n))$ time, then the singletape TM has a worst-time complexity of $\mathcal{O}(f(n)^2 + n)$. The translation is applicable both on deterministic and non-deterministic TMs.

Contents

1	Introduction	2
2	Preparations	2
3	Multitape Turing Machines	3
4	Singletape Turing Machines	6
5	Renamings for Singletape Turing Machines	10
6	Translating Multitape TMs to Singletape TMs	14
6.1	Definition of the Translation	15
6.2	Soundness of the Translation	21
6.2.1	R-Phase	22
6.2.2	S-Phase	27
6.2.3	E-Phase	30
6.2.4	Simulation of multitape TM by singletape TM	41
6.2.5	Simulation of singletape TM by multitape TM	43
6.2.6	Main Results	46
6.3	Main Results with Proper Renamings	47

1 Introduction

In 1965 Hartmanis and Stearns proved that multitape Turing machines (TMs) can be simulated by singletape Turing machines [1]. Since then, alternative approaches for translating multitape TMs to singletape TMs have been formulated [2, 3]. In this AFP entry we define a translation which has the usual quadratic overhead in running time.

For the design of the translation we had to choose between the approach how to encode the k tapes of a multitape TM onto a single tape.

In the textbooks [2, 3] the k tapes t_1, \dots, t_n are stored sequentially onto a single tape $t_1\#\dots\#t_n$ via a separator $\#$. The technical problem with this definition is that once a tape t_i needs to be enlarged to the right, the later tape content $\#t_{i+1}\#\dots\#t_n$ needs to be shifted correspondingly.

To avoid this problem, we followed the idea in the original work of Hartmanis et al. where the k -tapes are stored on top of each other, i.e., basically for the tape alphabet Γ of the multitape TM we switch to Γ^k in the singletape TM. As a consequence, the formal translation could be kept simple, in particular no tape shifts need to be performed.

2 Preparations

```
theory TM-Common
```

```
  imports
```

```
    HOL-Library.FuncSet
```

```
begin
```

A direction of a TM: go right, go left, or neutral (stay)

```
datatype dir = R | L | N
```

```
fun go-dir :: dir  $\Rightarrow$  nat  $\Rightarrow$  nat where
```

```
  go-dir R n = Suc n
```

```
  | go-dir L n = n - 1
```

```
  | go-dir N n = n
```

```
lemma finite-UNIV-dir[simp, intro]: finite (UNIV :: dir set)
```

```
proof -
```

```
  have id: UNIV = {L,R,N}
```

```
    using dir.exhaust by auto
```

```
  show ?thesis unfolding id by auto
```

```
qed
```

```
hide-const (open) L R N
```

```
lemma fin-funcsetI[intro]: finite A  $\Longrightarrow$  finite ((UNIV :: 'a :: finite set)  $\rightarrow$  A)
```

```
  by (metis PiE-UNIV-domain finite-PiE finite-code)
```

```

lemma finite-UNIV-fun-dir[simp,intro]: finite (UNIV :: ('k :: finite  $\Rightarrow$  dir) set)
  using fin-funcsetI[OF finite-UNIV-dir] by auto

lemma relpow-transI:  $(x,y) \in R^{\sim n} \Longrightarrow (y,z) \in R^{\sim m} \Longrightarrow (x,z) \in R^{\sim (n+m)}$ 
  by (simp add: relcomp.intros relpow-add)

lemma relpow-mono: fixes R :: 'a rel shows  $R \subseteq S \Longrightarrow R^{\sim n} \subseteq S^{\sim n}$ 
  by (induct n, auto)

lemma finite-infinite-inj-on: assumes A: finite (A :: 'a set) and inf: infinite
(UNIV :: 'b set)
  shows  $\exists f :: 'a \Rightarrow 'b. \text{inj-on } f \ A$ 
proof -
  from inf obtain B :: 'b set where B: finite B card B = card A
    by (meson infinite-arbitrarily-large)
  from A B obtain f :: 'a  $\Rightarrow$  'b where bij-betw f A B
    by (metis bij-betw-iff-card)
  thus ?thesis by (intro exI[of - f], auto simp: bij-betw-def)
qed

lemma gauss-sum-nat2:  $(\sum i < (n :: \text{nat}). i) = (n - 1) * n \text{ div } 2$ 
proof (cases n)
  case (Suc m)
    hence id:  $\{..<n\} = \{0..m\}$  by auto
    show ?thesis unfolding id unfolding gauss-sum-nat unfolding Suc by auto
qed auto

lemma aux-sum-formula:  $(\sum i < n. 10 + 5 * i) \leq 3 * n^2 + 7 * (n :: \text{nat})$ 
proof -
  have  $(\sum i < n. 10 + 5 * i) = 10 * n + 5 * (\sum i < n. i)$ 
    by (subst sum.distrib, auto simp: sum-distrib-left)
  also have  $\dots \leq 10 * n + 3 * ((n - 1) * n)$ 
    by (unfold gauss-sum-nat2, rule add-left-mono, cases n, auto)
  also have  $\dots = 3 * n^2 + 7 * n$ 
    unfolding power2-eq-square by (cases n, auto)
  finally show ?thesis .
qed

end

```

3 Multitape Turing Machines

```

theory Multitape-TM
  imports
    TM-Common
begin

```

Turing machines can be either defined via a datatype or via a locale. We use TMs with left endmarker and dedicated accepting and rejecting state

from which no further transitions are allowed. Deterministic TMs can be partial.

Having multiple tapes, tape positions, directions, etc. is modelled via functions of type $'k \Rightarrow 'whatever$ for some finite index type $'k$.

The input will always be provided on the first tape, indexed by $0::'k$.

```
datatype ('q,'a,'k)mttm = MTTM
  (Q-tm: 'q set)
  'a set
  ( $\Gamma$ -tm: 'a set)
  'a
  'a
  ('q  $\times$  ('k  $\Rightarrow$  'a)  $\times$  'q  $\times$  ('k  $\Rightarrow$  'a)  $\times$  ('k  $\Rightarrow$  dir)) set
  'q
  'q
  'q
```

```
datatype ('a,'q,'k) mt-config = ConfigM
  (mt-state: 'q)
  'k  $\Rightarrow$  nat  $\Rightarrow$  'a
  (mt-pos: 'k  $\Rightarrow$  nat)
```

locale multitape-tm =

fixes

```
  Q :: 'q set and
   $\Sigma$  :: 'a set and
   $\Gamma$  :: 'a set and
  blank :: 'a and
  LE :: 'a and
   $\delta$  :: ('q  $\times$  ('k  $\Rightarrow$  'a)  $\times$  'q  $\times$  ('k  $\Rightarrow$  'a)  $\times$  ('k :: {finite,zero}  $\Rightarrow$  dir)) set and
  s :: 'q and
  t :: 'q and
  r :: 'q
```

assumes

```
  fin-Q: finite Q and
  fin- $\Gamma$ : finite  $\Gamma$  and
   $\Sigma$ -sub- $\Gamma$ :  $\Sigma \subseteq \Gamma$  and
  sQ: s  $\in$  Q and
  tQ: t  $\in$  Q and
  rQ: r  $\in$  Q and
  blank: blank  $\in$   $\Gamma$  blank  $\notin$   $\Sigma$  and
  LE: LE  $\in$   $\Gamma$  LE  $\notin$   $\Sigma$  and
  tr: t  $\neq$  r and
   $\delta$ -set:  $\delta \subseteq (Q - \{t,r\}) \times (UNIV \rightarrow \Gamma) \times Q \times (UNIV \rightarrow \Gamma) \times (UNIV \rightarrow UNIV)$  and
   $\delta$ LE: (q, a, q', a', d)  $\in$   $\delta \implies a k = LE \implies a' k = LE \wedge d k \in \{dir.N, dir.R\}$ 
```

begin

lemma δ : **assumes** (q,a,q',b,d) \in δ

shows $q \in Q \ a \ k \in \Gamma \ q' \in Q \ b \ k \in \Gamma$
using *assms* δ -set **by** *auto*

lemma *fin- Σ* : *finite* Σ
using *fin- Γ* Σ -sub- Γ **by** (*metis finite-subset*)

lemma *fin- δ* : *finite* δ
by (*intro finite-subset*[*OF* δ -set] *finite-cartesian-product fin-funcsetI*, *insert fin-Q fin- Γ* , *auto*)

lemmas *tm* = *sQ* Σ -sub- Γ *blank(1)* *LE(1)*

fun *valid-config* :: ('a, 'q, 'k) *mt-config* \Rightarrow *bool* **where**
valid-config (*Config_M* $q \ w \ n$) = ($q \in Q \wedge (\forall k. \text{range } (w \ k) \subseteq \Gamma) \wedge (\forall k. w \ k \ 0 = LE)$)

definition *init-config* :: 'a list \Rightarrow ('a,'q,'k)*mt-config* **where**
init-config $w = (\text{Config}_M \ s \ (\lambda k \ n. \text{if } n = 0 \text{ then } LE \text{ else if } k = 0 \wedge n \leq \text{length } w \text{ then } w \ ! \ (n-1) \text{ else blank}) \ (\lambda \cdot. 0))$

lemma *valid-init-config*: *set* $w \subseteq \Sigma \Longrightarrow \text{valid-config } (\text{init-config } w)$
unfolding *init-config-def* *valid-config.simps* **using** *tm* **by** (*force simp: set-conv-nth*)

inductive-set *step* :: ('a, 'q, 'k) *mt-config* *rel* **where**
step: ($q, (\lambda k. \text{ts } k \ (n \ k)), q', a, \text{dir}$) $\in \delta \Longrightarrow$
($\text{Config}_M \ q \ \text{ts } n, \text{Config}_M \ q' \ (\lambda k. (\text{ts } k)(n \ k := a \ k)) \ (\lambda k. \text{go-dir } (\text{dir } k) \ (n \ k))$) $\in \text{step}$

lemma *valid-step*: **assumes** $(\alpha, \beta) \in \text{step}$
and *val*: *valid-config* α
shows *valid-config* β
using *step*
proof (*cases rule: step.cases*)
case (*step* $q \ \text{ts } n \ q' \ a \ \text{dir}$)
from δ [*OF* *step(3)*] *val* δ *LE* *step(3)*
show *?thesis* **unfolding** *step(1-2)* **by** *fastforce*
qed

definition *Lang* :: 'a list set **where**
Lang = $\{w . \text{set } w \subseteq \Sigma \wedge (\exists w' \ n. (\text{init-config } w, \text{Config}_M \ t \ w' \ n) \in \text{step}^{\wedge *})\}$

definition *deterministic* **where**
deterministic = $(\forall q \ a \ p1 \ b1 \ d1 \ p2 \ b2 \ d2. (q, a, p1, b1, d1) \in \delta \longrightarrow (q, a, p2, b2, d2) \in \delta \longrightarrow (p1, b1, d1) = (p2, b2, d2))$

definition *upper-time-bound* :: (*nat* \Rightarrow *nat*) \Rightarrow *bool* **where**
upper-time-bound $f = (\forall w \ c \ n. \text{set } w \subseteq \Sigma \longrightarrow (\text{init-config } w, c) \in \text{step}^{\wedge n} \longrightarrow n \leq f \ (\text{length } w))$
end

```

fun valid-mttm :: ('q,'a,'k :: {finite,zero})mttm  $\Rightarrow$  bool where
  valid-mttm (MTTM Q  $\Sigma$   $\Gamma$  bl le  $\delta$  s t r) = multitape-tm Q  $\Sigma$   $\Gamma$  bl le  $\delta$  s t r

fun Lang-mttm :: ('q,'a,'k :: {finite,zero})mttm  $\Rightarrow$  'a list set where
  Lang-mttm (MTTM Q  $\Sigma$   $\Gamma$  bl le  $\delta$  s t r) = multitape-tm.Lang  $\Sigma$  bl le  $\delta$  s t

fun det-mttm :: ('q,'a,'k :: {finite,zero})mttm  $\Rightarrow$  bool where
  det-mttm (MTTM Q  $\Sigma$   $\Gamma$  bl le  $\delta$  s t r) = multitape-tm.deterministic  $\delta$ 

fun upperb-time-mttm :: ('q,'a,'k :: {finite, zero})mttm  $\Rightarrow$  (nat  $\Rightarrow$  nat)  $\Rightarrow$  bool
where
  upperb-time-mttm (MTTM Q  $\Sigma$   $\Gamma$  bl le  $\delta$  s t r) f = multitape-tm.upper-time-bound
   $\Sigma$  bl le  $\delta$  s f

end

```

4 Singletape Turing Machines

```

theory Singletape-TM
  imports
    TM-Common
  begin

```

Turing machines can be either defined via a datatype or via a locale. We use TMs with left endmarker and dedicated accepting and rejecting state from which no further transitions are allowed. Deterministic TMs can be partial.

```

datatype ('q,'a)tm = TM
  (Q-tm: 'q set)
  'a set
  ( $\Gamma$ -tm: 'a set)
  'a
  'a
  ('q  $\times$  'a  $\times$  'q  $\times$  'a  $\times$  dir) set
  'q
  'q
  'q

datatype ('a, 'q) st-config = Configs
  'q
  nat  $\Rightarrow$  'a
  nat

locale singletape-tm =
  fixes

```

$Q :: 'q \text{ set and}$
 $\Sigma :: 'a \text{ set and}$
 $\Gamma :: 'a \text{ set and}$
 $\text{blank} :: 'a \text{ and}$
 $LE :: 'a \text{ and}$
 $\delta :: ('q \times 'a \times 'q \times 'a \times \text{dir}) \text{ set and}$
 $s :: 'q \text{ and}$
 $t :: 'q \text{ and}$
 $r :: 'q$
assumes
 $\text{fin-}Q: \text{finite } Q \text{ and}$
 $\text{fin-}\Gamma: \text{finite } \Gamma \text{ and}$
 $\Sigma\text{-sub-}\Gamma: \Sigma \subseteq \Gamma \text{ and}$
 $sQ: s \in Q \text{ and}$
 $tQ: t \in Q \text{ and}$
 $rQ: r \in Q \text{ and}$
 $\text{blank}: \text{blank} \in \Gamma \text{ blank} \notin \Sigma \text{ and}$
 $LE: LE \in \Gamma \text{ } LE \notin \Sigma \text{ and}$
 $tr: t \neq r \text{ and}$
 $\delta\text{-set}: \delta \subseteq (Q - \{t,r\}) \times \Gamma \times Q \times \Gamma \times \text{UNIV} \text{ and}$
 $\delta LE: (q, LE, q', a', d) \in \delta \implies a' = LE \wedge d \in \{\text{dir}.N, \text{dir}.R\}$
begin

lemma δ : **assumes** $(q, a, q', b, d) \in \delta$
shows $q \in Q \ a \in \Gamma \ q' \in Q \ b \in \Gamma$
using *assms* $\delta\text{-set}$ **by** *auto*

lemma $\text{fin}\Sigma$: *finite* Σ
using $\text{fin-}\Gamma \ \Sigma\text{-sub-}\Gamma$ **by** (*metis finite-subset*)

lemmas $tm = sQ \ \Sigma\text{-sub-}\Gamma \ \text{blank}(1) \ LE(1)$

fun *valid-config* :: $('a, 'q) \text{ st-config} \Rightarrow \text{bool}$ **where**
 $\text{valid-config } (\text{Configs } q \ w \ n) = (q \in Q \wedge \text{range } w \subseteq \Gamma)$

definition *init-config* :: $'a \text{ list} \Rightarrow ('a, 'q) \text{ st-config}$ **where**
 $\text{init-config } w = (\text{Configs } s \ (\lambda \ n. \ \text{if } n = 0 \ \text{then } LE \ \text{else if } n \leq \text{length } w \ \text{then } w !$
 $(n-1) \ \text{else blank}) \ 0)$

lemma *valid-init-config*: $\text{set } w \subseteq \Sigma \implies \text{valid-config } (\text{init-config } w)$
unfolding *init-config-def valid-config.simps* **using** *tm* **by** (*force simp: set-conv-nth*)

inductive-set *step* :: $('a, 'q) \text{ st-config rel}$ **where**
 $\text{step}: (q, \text{ts } n, q', a, \text{dir}) \in \delta \implies$
 $(\text{Configs } q \ \text{ts } n, \text{Configs } q' \ (\text{ts } n := a)) \ (\text{go-dir } \text{dir } n) \in \text{step}$

lemma *stepI*: $(q, a, q', b, \text{dir}) \in \delta \implies \text{ts } n = a \implies \text{ts}' = \text{ts}(n := b) \implies n' =$
 $\text{go-dir } \text{dir } n \implies q1 = q \implies q2 = q'$
 $\implies (\text{Configs } q1 \ \text{ts } n, \text{Configs } q2 \ \text{ts}' \ n') \in \text{step}$

```

using step[of q ts n q' b dir] by auto

lemma valid-step: assumes step:  $(\alpha, \beta) \in \text{step}$ 
and val: valid-config  $\alpha$ 
shows valid-config  $\beta$ 
using step
proof (cases rule: step.cases)
case (step q ts n q' a dir)
from  $\delta[\text{OF } \text{step}(\beta)] \text{ val}$ 
show ?thesis unfolding step(1-2) by auto
qed

definition Lang :: 'a list set where
  Lang =  $\{w . \text{set } w \subseteq \Sigma \wedge (\exists w' n. (\text{init-config } w, \text{Config}_S t w' n) \in \text{step}^{\widehat{*}})\}$ 

definition deterministic where
  deterministic =  $(\forall q a p1 b1 d1 p2 b2 d2. (q, a, p1, b1, d1) \in \delta \longrightarrow (q, a, p2, b2, d2) \in \delta \longrightarrow (p1, b1, d1) = (p2, b2, d2))$ 

definition upper-time-bound ::  $(\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{bool}$  where
  upper-time-bound  $f = (\forall w c n. \text{set } w \subseteq \Sigma \longrightarrow (\text{init-config } w, c) \in \text{step}^{\sim n} \longrightarrow n \leq f (\text{length } w))$ 
end

fun valid-tm :: 'q, 'a)tm  $\Rightarrow$  bool where
  valid-tm  $(TM Q \Sigma \Gamma \text{bl le } \delta s t r) = \text{singletape-tm } Q \Sigma \Gamma \text{bl le } \delta s t r$ 

fun Lang-tm :: 'q, 'a)tm  $\Rightarrow$  'a list set where
  Lang-tm  $(TM Q \Sigma \Gamma \text{bl le } \delta s t r) = \text{singletape-tm.Lang } \Sigma \text{bl le } \delta s t r$ 

fun det-tm :: 'q, 'a)tm  $\Rightarrow$  bool where
  det-tm  $(TM Q \Sigma \Gamma \text{bl le } \delta s t r) = \text{singletape-tm.deterministic } \delta$ 

fun upperb-time-tm :: 'q, 'a)tm  $\Rightarrow$   $(\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{bool}$  where
  upperb-time-tm  $(TM Q \Sigma \Gamma \text{bl le } \delta s t r) f = \text{singletape-tm.upper-time-bound } \Sigma \text{bl le } \delta s f$ 

context singletape-tm
begin

A deterministic step (in a potentially non-deterministic TM) is a step without alternatives. This will be useful in the translation of multitape TMs. The simulation is mostly deterministic, and only at very specific points it is non-deterministic, namely at the points where the multitape-TM transition is chosen.

inductive-set dstep :: 'a, 'q) st-config rel where
  dstep:  $(q, ts n, q', a, dir) \in \delta \Longrightarrow$ 
   $(\bigwedge q1' a1 dir1. (q, ts n, q1', a1, dir1) \in \delta \Longrightarrow (q1', a1, dir1) = (q', a, dir)) \Longrightarrow$ 
   $(\text{Config}_S q ts n, \text{Config}_S q' (ts(n := a)) (\text{go-dir } dir n)) \in \text{dstep}$ 

```


lemma *dstepI*: $(q, a, q', b, dir) \in \delta \implies ts\ n = a \implies ts' = ts(n := b) \implies n' =$
go-dir $dir\ n \implies q1 = q \implies q2 = q'$
 $\implies (\bigwedge q''\ b'\ dir'. (q, a, q'', b', dir') \in \delta \implies (q'', b', dir') = (q', b, dir))$
 $\implies (Config_S\ q1\ ts\ n, Config_S\ q2\ ts'\ n') \in dstep$
using *dstep[of q ts n q' b dir]* **by** *blast*

lemma *dstep-step*: $dstep \subseteq step$

proof

fix *st*

assume *dstep*: $st \in dstep$

obtain *s t* **where** *st*: $st = (s, t)$ **by** *force*

have $(s, t) \in step$ **using** *dstep[unfolded st]*

proof (*cases rule: dstep.cases*)

case 1: (*dstep q ts n q' a dir*)

show *?thesis* **unfolding** 1(1–2) **by** (*rule stepI[OF 1(3)], auto*)

qed

thus $st \in step$ **using** *st* **by** *auto*

qed

lemma *dstep-inj*: **assumes** $(x, y) \in dstep$

and $(x, z) \in step$

shows $z = y$

using *assms(2)*

proof (*cases*)

case 1: (*step q ts n p a d*)

show *?thesis* **using** *assms(1)* **unfolding** 1(1)

proof *cases*

case 2: (*dstep p' a' d'*)

from 2(3)[*OF 1(3)*] **have** *id*: $p' = p\ a' = a\ d' = d$ **by** *auto*

show *?thesis* **unfolding** 1 2 *id ..*

qed

qed

lemma *dsteps-inj*: **assumes** $(x, y) \in dstep^{\sim n}$

and $(x, z) \in step^{\sim m}$

and $\neg (\exists u. (z, u) \in step)$

shows $\exists k. m = n + k \wedge (y, z) \in step^{\sim k}$

using *assms(1–2)*

proof (*induct n arbitrary: m x*)

case (*Suc n m x*)

from *Suc(2)* **obtain** x' **where** *step*: $(x, x') \in dstep$ **and** *steps*: $(x', y) \in dstep^{\sim n}$
by (*metis relpow-Suc-E2*)

from *step dstep-step* **have** $(x, x') \in step$ **by** *auto*

with *assms(3)* **have** $x \neq z$ **by** *auto*

with *Suc(3)* **obtain** mm **where** $m = Suc\ mm$ **by** (*cases m, auto*)

from *Suc(3)[unfolded this]* **obtain** x'' **where** *step'*: $(x, x'') \in step$ **and** *steps'*:
 $(x'', z) \in step^{\sim mm}$ **by** (*metis relpow-Suc-E2*)

from *dstep-inj[OF step step']* **have** $x'' = x'$ **by** *auto*

from $Suc(1)[OF\ steps\ steps'[unfolding\ x'']]$ **obtain** k **where** $mm: mm = n + k$
and $steps: (y, z) \in step \sim k$ **by** *auto*
thus *?case unfolding m mm by auto*
qed *auto*

lemma $dsteps-inj'$: **assumes** $(x, y) \in dstep \sim n$
and $(x, z) \in step \sim m$
and $m \geq n$
shows $\exists k. m = n + k \wedge (y, z) \in step \sim k$
using $assms(1-3)$
proof (*induct n arbitrary: m x*)
case ($Suc\ n\ m\ x$)
from $Suc(2)$ **obtain** x' **where** $step: (x, x') \in dstep$ **and** $steps: (x', y) \in dstep \sim n$
by (*metis relpow-Suc-E2*)
from $step\ dstep-step$ **have** $(x, x') \in step$ **by** *auto*
with Suc **obtain** mm **where** $m: m = Suc\ mm$ **by** (*cases m, auto*)
from $Suc(3)[unfolding\ this]$ **obtain** x'' **where** $step': (x, x'') \in step$ **and** $steps':$
 $(x'', z) \in step \sim mm$ **by** (*metis relpow-Suc-E2*)
from $dstep-inj[OF\ step\ step']$ **have** $x'': x'' = x'$ **by** *auto*
from $Suc(1)[OF\ steps\ steps'[unfolding\ x'']]$ $m\ Suc$ **obtain** k **where** $mm: mm =$
 $n + k$ **and** $steps: (y, z) \in step \sim k$ **by** *auto*
thus *?case unfolding m mm by auto*
qed *auto*
end
end

5 Renamings for Singletape Turing Machines

theory *STM-Renaming*

imports

Singletape-TM

begin

locale *renaming-of-singletape-tm* = *singletape-tm* $Q\ \Sigma\ \Gamma\ blank\ LE\ \delta\ s\ t\ r$

for $Q :: 'q\ set$ **and** $\Sigma :: 'a\ set$ **and** $\Gamma\ blank\ LE\ \delta\ s\ t\ r$

+ **fixes** $ra :: 'a \Rightarrow 'b$

and $rq :: 'q \Rightarrow 'p$

assumes $ra: inj-on\ ra\ \Gamma$

and $rq: inj-on\ rq\ Q$

begin

abbreviation rd **where** $rd \equiv map-prod\ rq\ (map-prod\ ra\ (map-prod\ rq\ (map-prod\ ra\ (\lambda\ d :: dir.\ d))))$

sublocale $ren: singletape-tm\ rq\ 'Q\ ra\ ' \Sigma\ ra\ ' \Gamma\ ra\ blank\ ra\ LE\ rd\ ' \delta\ rq\ s\ rq\ t\ rq\ r$

proof (*unfold-locales; (intro finite-imageI imageI image-mono fin-Q fin-Γ tm sQ tQ rQ)?*)

show $ra\ LE \notin ra\ ' \Sigma$ **using** $ra\ tm\ LE$ **by** (*simp add: inj-on-image-mem-iff*)

```

show  $ra \text{ blank} \notin ra \text{ ' } \Sigma$  using  $ra \text{ tm blank}$  by (simp add: inj-on-image-mem-iff)
show  $rq \text{ t} \neq rq \text{ r}$  using  $rq \text{ tQ} \text{ rQ} \text{ tr}$  by (metis inj-on-contrad)
show  $rd \text{ ' } \delta \subseteq (rq \text{ ' } Q - \{rq \text{ t}, rq \text{ r}\}) \times ra \text{ ' } \Gamma \times rq \text{ ' } Q \times ra \text{ ' } \Gamma \times UNIV$ 
using  $\delta\text{-set} \text{ tQ} \text{ rQ} \text{ rq}$  by (auto simp: inj-on-def)
fix  $p1 \text{ p2} \text{ b2} \text{ d}$ 
assume  $(p1, ra \text{ LE}, p2, b2, d) \in rd \text{ ' } \delta$ 
then obtain  $q1 \text{ q2} \text{ a1} \text{ a2}$  where  $mem: (q1, a1, q2, a2, d) \in \delta$  and
   $id: p1 = rq \text{ q1} \text{ ra} \text{ LE} = ra \text{ a1} \text{ p2} = rq \text{ q2} \text{ b2} = ra \text{ a2}$  by auto
from  $\delta[OF \text{ mem}] \text{ id}(2) \text{ LE} \text{ ra}$  have  $a1 = LE$  by (simp add: inj-onD)
with  $\delta \text{ LE}[OF \text{ mem}[unfolding \text{ this}]]$ 
show  $b2 = ra \text{ LE} \wedge d \in \{dir.N, dir.R\}$  unfolding  $id$  by auto
qed

fun  $rc :: ('a, 'q) \text{ st-config} \Rightarrow ('b, 'p) \text{ st-config}$  where
   $rc (\text{Configs } q \text{ tc } pos) = \text{Configs} (rq \text{ q}) (ra \text{ o } tc) \text{ pos}$ 

lemma ren-init:  $rc (\text{init-config } w) = \text{ren.init-config} (\text{map } ra \text{ w})$ 
unfolding init-config-def ren.init-config-def  $rc.simps$ 
by auto

lemma ren-step: assumes  $(c, c') \in \text{step}$ 
shows  $(rc \text{ c}, rc \text{ c}') \in \text{ren.step}$ 
using assms
proof (cases rule: step.cases)
case (step q ts n q' a dir)
from step(3) have  $mem: (rq \text{ q}, ra \text{ (ts n)}, rq \text{ q'}, ra \text{ a}, dir) \in rd \text{ ' } \delta$  by force
show ?thesis unfolding step rc.simps
by (intro ren.stepI[OF mem], auto)
qed

lemma ren-steps: assumes  $(c, c') \in \text{step}^*$ 
shows  $(rc \text{ c}, rc \text{ c}') \in \text{ren.step}^*$ 
using assms by (induct, insert ren-step, force+)

lemma ren-steps-count: assumes  $(c, c') \in \text{step}^{\sim n}$ 
shows  $(rc \text{ c}, rc \text{ c}') \in \text{ren.step}^{\sim n}$ 
using assms by (induct n arbitrary: c c', insert ren-step, force+)

lemma ren-Lang-forward: assumes  $w \in \text{Lang}$ 
shows  $\text{map } ra \text{ w} \in \text{ren.Lang}$ 
proof –
from assms[unfolding Lang-def, simplified]
obtain  $w' \text{ n}$  where  $w: \text{set } w \subseteq \Sigma$  and  $\text{steps}: (\text{init-config } w, \text{Configs } t \text{ w}' \text{ n}) \in \text{step}^*$ 
by auto
from ren-steps[OF steps, unfolding ren-init, unfolding rc.simps]  $w$ 
show  $\text{map } ra \text{ w} \in \text{ren.Lang}$  unfolding ren.Lang-def by auto
qed

```

abbreviation *ira* **where** $ira \equiv the-inv-into \Gamma ra$
abbreviation *irq* **where** $irq \equiv the-inv-into Q rq$

interpretation *inv*: *renaming-of-singletape-tm* $rq \text{ ' } Q ra \text{ ' } \Sigma ra \text{ ' } \Gamma ra blank ra LE$
 $rd \text{ ' } \delta rq s rq t rq r ira irq$
by (*unfold-locales*, *insert ra rq inj-on-the-inv-into*, *auto*)

lemmas *inv-simps*[*simp*] = *the-inv-into-f-f*[*OF ra*] *the-inv-into-f-f*[*OF rq*]

lemma *inv-ren-Sigma*: $ira \text{ ' } ra \text{ ' } \Sigma = \Sigma$ **using** *inv-simps*(1)[*OF set-mp*[*OF* Σ -*sub*- Γ]]
by (*smt* (*verit*, *best*) *equalityI imageE image-subset-iff subsetI*)

lemma *inv-ren-Gamma*: $ira \text{ ' } ra \text{ ' } \Gamma = \Gamma$ **using** *inv-simps*(1)
by (*smt* (*verit*, *best*) *equalityI imageE image-subset-iff subsetI*)

lemma *inv-ren-t*: $irq (rq t) = t$ **using** *tQ* **by** *simp*
lemma *inv-ren-s*: $irq (rq s) = s$ **using** *sQ* **by** *simp*
lemma *inv-ren-r*: $irq (rq r) = r$ **using** *rQ* **by** *simp*
lemma *inv-ren-blank*: $ira (ra blank) = blank$ **using** *tm* **by** *simp*
lemma *inv-ren-LE*: $ira (ra LE) = LE$ **using** *tm* **by** *simp*

lemma *inv-ren-delta*: $inv.rd \text{ ' } rd \text{ ' } \delta = \delta$

proof –

```
{
  fix trans :: ('q × 'a × 'q × 'a × dir)
  obtain q a p b d where trans: trans = (q,a,p,b,d) by (cases trans, auto)
  {
    assume t: trans ∈ δ
    note mem = δ[OF this[unfolded trans]]
    from t have inv.rd (rd trans) ∈ inv.rd ' rd ' δ by blast
    also have inv.rd (rd trans) = trans unfolding trans using mem by auto
    finally have trans ∈ inv.rd ' rd ' δ .
  }
  moreover
  {
    assume t: trans ∈ inv.rd ' rd ' δ
    then obtain t' where t'd: t' ∈ δ and tra: trans = inv.rd (rd t') by auto
    obtain q' a' p' b' d' where t': t' = (q',a',p',b',d') by (cases t', auto)
    note mem = δ[OF t'd[unfolded t']]
    from tra[unfolded trans t'] mem
    have id: q' = q a' = a p' = p b' = b d' = d by auto
    with trans t'd t' have trans ∈ δ by auto
  }
  ultimately have trans ∈ inv.rd ' rd ' δ ⟷ trans ∈ δ by blast
}
thus ?thesis by blast
qed
```

lemmas *inv-ren* = *inv-ren-t inv-ren-s inv-ren-r inv-ren-delta inv-ren-Gamma inv-ren-Sigma*

inv-ren-blank inv-ren-LE

lemma *inv-ren-Lang*: $inv-ren.Lang = Lang$ **unfolding** *inv-ren ..*

lemma *ren-Lang-backward*: **assumes** $v \in ren.Lang$

shows $\exists w. v = map\ ra\ w \wedge w \in Lang$

proof (*intro exI conjI*)

let $?w = map\ ira\ v$

from *inv-ren-Lang-forward*[*OF assms, unfolded inv-ren-Lang*]

show $?w \in Lang$.

show $v = map\ ra\ ?w$ **unfolding** *map-map o-def*

proof (*subst map-idI*)

fix b

assume $b \in set\ v$

with *assms*[*unfolded ren.Lang-def*] $\Sigma-sub-\Gamma$ **obtain** a **where** $a: a \in \Gamma$ **and** $b:$

$b = ra\ a$ **by** *blast*

show $ra\ (ira\ b) = b$ **unfolding** b **using** a **by** *simp*

qed *auto*

qed

lemma *ren-Lang*: $ren.Lang = map\ ra\ 'Lang$

proof

show $map\ ra\ 'Lang \subseteq ren.Lang$ **using** *ren-Lang-forward* **by** *blast*

show $ren.Lang \subseteq map\ ra\ 'Lang$ **using** *ren-Lang-backward* **by** *blast*

qed

lemma *ren-det*: **assumes** *deterministic*

shows *ren.deterministic*

unfolding *ren.deterministic-def*

proof (*intro allI impI, goal-cases*)

case $(1\ q\ a\ p1\ b1\ d1\ p2\ b2\ d2)$

let $?t1 = (q, a, p1, b1, d1)$

let $?t2 = (q, a, p2, b2, d2)$

from 1 **have** $t1: inv.rd\ ?t1 \in inv.rd\ 'rd\ ' \delta$ **by** *force*

from 1 **have** $t2: inv.rd\ ?t2 \in inv.rd\ 'rd\ ' \delta$ **by** *force*

from $t1\ t2$ **have** $(irq\ q, ira\ a, irq\ p1, ira\ b1, d1) \in \delta$ $(irq\ q, ira\ a, irq\ p2, ira\ b2, d2) \in \delta$

unfolding *inv-ren* **by** *auto*

from *assms*[*unfolded deterministic-def, rule-format, OF this*]

have $id: irq\ p1 = irq\ p2\ ira\ b1 = ira\ b2$ **and** $d: d1 = d2$ **by** *auto*

from *inj-onD*[*OF inv.rq id(1)*] *ren. δ* [*OF 1(1)*] *ren. δ* [*OF 1(2)*]

have $p: p1 = p2$ **by** *auto*

from *inj-onD*[*OF inv.ra id(2)*] *ren. δ* [*OF 1(1)*] *ren. δ* [*OF 1(2)*]

have $b: b1 = b2$ **by** *auto*

show $?case$ **unfolding** $b\ p\ d$ **by** *simp*

qed

lemma *ren-upper-time*: **assumes** *upper-time-bound f*

shows *ren.upper-time-bound f*

```

unfolding ren.upper-time-bound-def
proof (intro allI impI)
  fix w c n
  assume w: set w  $\subseteq$  ra ‘  $\Sigma$  and steps: (ren.init-config w, c)  $\in$  ren.step  $\overset{\sim}{\sim}$  n
  define v where v = map ira w
  from w have v: set v  $\subseteq$   $\Sigma$  unfolding v-def
    using inv-ren-Sigma by fastforce
  from inv-ren-steps-count[OF steps]
  have (inv.rc (ren.init-config w), inv.rc c)  $\in$  inv-ren.step  $\overset{\sim}{\sim}$  n .
  also have inv-ren.step = step using inv-ren- $\delta$  by presburger
  also have inv.rc (ren.init-config w) = init-config v unfolding v-def using v w
    by (simp add: inv-ren-init inv-ren-LE inv-ren-blank inv-ren-s)
  finally have (init-config v, inv.rc c)  $\in$  step  $\overset{\sim}{\sim}$  n .
  with assms[unfolded upper-time-bound-def] v have n  $\leq$  f (length v) by simp
  thus n  $\leq$  f (length w) unfolding v-def by auto
qed

```

end

```

lemma tm-renaming: assumes valid-tm (tm :: ('q, 'a)tm)
  and inj-on (ra :: 'a  $\Rightarrow$  'b) ( $\Gamma$ -tm tm)
  and inj-on (rq :: 'q  $\Rightarrow$  'p) (Q-tm tm)
shows  $\exists$  tm' :: ('p, 'b)tm.
  valid-tm tm'  $\wedge$ 
  Lang-tm tm' = map ra ‘ Lang-tm tm  $\wedge$ 
  (det-tm tm  $\longrightarrow$  det-tm tm')  $\wedge$ 
  ( $\forall$  f. upperb-time-tm tm f  $\longrightarrow$  upperb-time-tm tm' f)
proof (cases tm)
  case (TM Q  $\Sigma$   $\Gamma$  bl le  $\delta$  s t r)
  with assms interpret singletape-tm Q  $\Sigma$   $\Gamma$  bl le  $\delta$  s t r by auto
  interpret renaming-of-singletape-tm Q  $\Sigma$   $\Gamma$  bl le  $\delta$  s t r ra rq
    by (unfold-locales, insert assms TM, auto)
  let ?tm' = TM (rq ‘ Q) (ra ‘  $\Sigma$ ) (ra ‘  $\Gamma$ ) (ra bl) (ra le) (rd ‘  $\delta$ ) (rq s) (rq t) (rq r)
  show ?thesis
    by (rule exI[where x = ?tm'])
      (simp add: TM ren.singletape-tm-axioms ren-Lang ren-det ren-upper-time)
qed

```

end

6 Translating Multitape TMs to Singletape TMs

In this section we define the mapping from a multitape Turing machine to a singletape Turing machine. We further define soundness of the translation via several relations which establish a connection between configurations of both kinds of Turing machines.

The translation works both for deterministic and non-deterministic TMs.

Moreover, we verify a quadratic overhead in runtime.

theory *Multi-Single-TM-Translation*

imports

Multitape-TM

Singletape-TM

STM-Renaming

begin

6.1 Definition of the Translation

datatype *'a tuple-symbol* = *NO-HAT 'a* | *HAT 'a*

datatype (*'a, 'k*) *st-tape-symbol* = *ST-LE (†)* | *TUPLE 'k ⇒ 'a tuple-symbol* | *INP 'a*

datatype *'a sym-or-bullet* = *SYM 'a* | *BULLET (·)*

datatype (*'a, 'q, 'k*) *st-states* =

R₁ 'a sym-or-bullet |

R₂ |

S₀ 'q |

S 'q 'k ⇒ 'a sym-or-bullet |

S₁ 'q 'k ⇒ 'a |

E₀ 'q 'k ⇒ 'a 'k ⇒ dir |

E 'q 'k ⇒ 'a sym-or-bullet 'k ⇒ dir |

Er 'q 'k ⇒ 'a sym-or-bullet 'k ⇒ dir 'k set |

El 'q 'k ⇒ 'a sym-or-bullet 'k ⇒ dir 'k set |

Em 'q 'k ⇒ 'a sym-or-bullet 'k ⇒ dir 'k set

type-synonym (*'a, 'q, 'k*)*mt-rule* = *'q × ('k ⇒ 'a) × 'q × ('k ⇒ 'a) × ('k ⇒ dir)*

context *multitape-tm*

begin

definition *R1-Set* **where** *R1-Set* = *SYM ' Σ ∪ {·}*

definition *gamma-set* :: (*'k ⇒ 'a tuple-symbol*) *set* **where**
gamma-set = (*UNIV :: 'k set*) → *NO-HAT ' Γ ∪ HAT ' Γ*

definition *Γ'* :: (*'a, 'k*) *st-tape-symbol set* **where**
Γ' = *TUPLE ' gamma-set ∪ INP ' Σ ∪ {·}*

definition *func-set* = (*UNIV :: 'k set*) → *SYM ' Γ ∪ {·}*

definition *blank'* :: (*'a, 'k*) *st-tape-symbol* **where** *blank'* = *TUPLE (λ -. NO-HAT blank)*

definition *hatLE'* :: (*'a, 'k*) *st-tape-symbol* **where** *hatLE'* = *TUPLE (λ -. HAT LE)*

definition *encSym* :: *'a ⇒ ('a, 'k) st-tape-symbol* **where** *encSym a* = (*TUPLE (λ i. if i = 0 then NO-HAT a else NO-HAT blank)*)

definition $add\text{-}inp :: ('k \Rightarrow 'a \text{ tuple-symbol}) \Rightarrow ('k \Rightarrow 'a \text{ sym-or-bullet}) \Rightarrow ('k \Rightarrow 'a \text{ sym-or-bullet})$ **where**
 $add\text{-}inp\ inp\ inp2 = (\lambda k. \text{ case } inp\ k \text{ of } HAT\ s \Rightarrow SYM\ s \mid - \Rightarrow inp2\ k)$

definition $project\text{-}inp :: ('k \Rightarrow 'a \text{ sym-or-bullet}) \Rightarrow ('k \Rightarrow 'a)$ **where**
 $project\text{-}inp\ inp = (\lambda k. \text{ case } inp\ k \text{ of } SYM\ s \Rightarrow s)$

definition $compute\text{-}idx\text{-}set :: ('k \Rightarrow 'a \text{ tuple-symbol}) \Rightarrow ('k \Rightarrow 'a \text{ sym-or-bullet}) \Rightarrow 'k \text{ set}$ **where**
 $compute\text{-}idx\text{-}set\ tup\ ys = \{i . \text{ tup } i \in HAT\ ' \Gamma \wedge ys\ i \in SYM\ ' \Gamma\}$

definition $update\text{-}ys :: ('k \Rightarrow 'a \text{ tuple-symbol}) \Rightarrow ('k \Rightarrow 'a \text{ sym-or-bullet}) \Rightarrow ('k \Rightarrow 'a \text{ sym-or-bullet})$ **where**
 $update\text{-}ys\ tup\ ys = (\lambda k. \text{ if } k \in (compute\text{-}idx\text{-}set\ tup\ ys) \text{ then } \cdot \text{ else } ys\ k)$

definition $replace\text{-}sym :: ('k \Rightarrow 'a \text{ tuple-symbol}) \Rightarrow ('k \Rightarrow 'a \text{ sym-or-bullet}) \Rightarrow ('k \Rightarrow 'a \text{ tuple-symbol})$ **where**
 $replace\text{-}sym\ tup\ ys = (\lambda k. \text{ if } k \in (compute\text{-}idx\text{-}set\ tup\ ys) \text{ then } (\text{ case } ys\ k \text{ of } SYM\ a \Rightarrow NO\text{-}HAT\ a) \text{ else } tup\ k)$

definition $place\text{-}hats\text{-}to\text{-}dir :: dir \Rightarrow ('k \Rightarrow 'a \text{ tuple-symbol}) \Rightarrow ('k \Rightarrow dir) \Rightarrow 'k \text{ set} \Rightarrow ('k \Rightarrow 'a \text{ tuple-symbol})$ **where**
 $place\text{-}hats\text{-}to\text{-}dir\ dir\ tup\ ds\ I = (\lambda k. (\text{ case } tup\ k \text{ of } NO\text{-}HAT\ a \Rightarrow \text{ if } k \in I \wedge ds\ k = dir \text{ then } HAT\ a \text{ else } NO\text{-}HAT\ a \mid HAT\ a \Rightarrow HAT\ a))$

definition $place\text{-}hats\text{-}R :: ('k \Rightarrow 'a \text{ tuple-symbol}) \Rightarrow ('k \Rightarrow dir) \Rightarrow 'k \text{ set} \Rightarrow ('k \Rightarrow 'a \text{ tuple-symbol})$ **where**
 $place\text{-}hats\text{-}R = place\text{-}hats\text{-}to\text{-}dir\ dir.R$

definition $place\text{-}hats\text{-}M :: ('k \Rightarrow 'a \text{ tuple-symbol}) \Rightarrow ('k \Rightarrow dir) \Rightarrow 'k \text{ set} \Rightarrow ('k \Rightarrow 'a \text{ tuple-symbol})$ **where**
 $place\text{-}hats\text{-}M = place\text{-}hats\text{-}to\text{-}dir\ dir.N$

definition $place\text{-}hats\text{-}L :: ('k \Rightarrow 'a \text{ tuple-symbol}) \Rightarrow ('k \Rightarrow dir) \Rightarrow 'k \text{ set} \Rightarrow ('k \Rightarrow 'a \text{ tuple-symbol})$ **where**
 $place\text{-}hats\text{-}L = place\text{-}hats\text{-}to\text{-}dir\ dir.L$

definition $\delta' :: (('a, 'q, 'k) \text{ st-states} \times ('a, 'k) \text{ st-tape-symbol} \times ('a, 'q, 'k) \text{ st-states} \times ('a, 'k) \text{ st-tape-symbol} \times dir) \text{ set}$ **where**
 $\delta' = (\{(R_1 \cdot, \vdash, R_1 \cdot, \vdash, dir.R)\} \cup (\lambda x. (R_1 \cdot, INP\ x, R_1 (SYM\ x), hatLE', dir.R)) ' \Sigma \cup (\lambda (a,x). (R_1 (SYM\ a), INP\ x, R_1 (SYM\ x), encSym\ a, dir.R)) ' (\Sigma \times \Sigma))$

$\cup \{(R_1 \cdot, \text{blank}', R_2, \text{hatLE}', \text{dir.L})\}$
 $\cup (\lambda a. (R_1 (\text{SYM } a), \text{blank}', R_2, \text{encSym } a, \text{dir.L})) \text{ ' } \Sigma$
 $\cup (\lambda x. (R_2, x, R_2, x, \text{dir.L})) \text{ ' } (\Gamma' - \{\vdash\})$
 $\cup \{(R_2, \vdash, S_0 s, \vdash, \text{dir.N})\}$
 $\cup (\lambda q. (S_0 q, \vdash, S q (\lambda \cdot. \cdot), \vdash, \text{dir.R})) \text{ ' } (Q - \{t, r\})$
 $\cup (\lambda (q, \text{inp}, t). (S q \text{ inp}, \text{TUPLE } t, S q (\text{add-inp } t \text{ inp}), \text{TUPLE } t, \text{dir.R})) \text{ ' } (Q$
 $\times (\text{func-set} - (\text{UNIV} \rightarrow \text{SYM ' } \Gamma)) \times \text{gamma-set})$
 $\cup (\lambda (q, \text{inp}, a). (S q \text{ inp}, a, S_1 q (\text{project-inp } \text{inp}), a, \text{dir.L})) \text{ ' } (Q \times (\text{UNIV} \rightarrow$
 $\text{SYM ' } \Gamma) \times (\Gamma' - \{\vdash\}))$
 $\cup (\lambda ((q, a, q', b, d), t). (S_1 q a, t, E_0 q' b d, t, \text{dir.N})) \text{ ' } (\delta \times \Gamma')$
 $\cup (\lambda ((q, a, d), t). (E_0 q a d, t, E q (\text{SYM } o a) d, t, \text{dir.N})) \text{ ' } ((Q \times (\text{UNIV} \rightarrow$
 $\Gamma) \times \text{UNIV}) \times \Gamma')$
 $\cup (\lambda (q, d). (E q (\lambda \cdot. \cdot) d, \vdash, S_0 q, \vdash, \text{dir.N})) \text{ ' } (Q \times \text{UNIV})$
 $\cup (\lambda (q, \text{ys}, \text{ds}, t). (E q \text{ ys } \text{ds}, \text{TUPLE } t, Er q (\text{update-ys } t \text{ ys}) \text{ ds } (\text{compute-idx-set}$
 $t \text{ ys}), \text{TUPLE}(\text{replace-sym } t \text{ ys}), \text{dir.R})) \text{ ' } (Q \times \text{func-set} \times \text{UNIV} \times \text{gamma-set})$
 $\cup (\lambda (q, \text{ys}, \text{ds}, I, t). (Er q \text{ ys } \text{ds } I, \text{TUPLE } t, Em q \text{ ys } \text{ds } I, \text{TUPLE}(\text{place-hats-R}$
 $t \text{ ds } I), \text{dir.L})) \text{ ' } (Q \times \text{func-set} \times \text{UNIV} \times \text{UNIV} \times \text{gamma-set})$
 $\cup (\lambda (q, \text{ys}, \text{ds}, I, t). (Em q \text{ ys } \text{ds } I, \text{TUPLE } t, El q \text{ ys } \text{ds } I, \text{TUPLE}(\text{place-hats-M}$
 $t \text{ ds } I), \text{dir.L})) \text{ ' } (Q \times \text{func-set} \times \text{UNIV} \times \text{UNIV} \times \text{gamma-set})$
 $\cup (\lambda (q, \text{ys}, \text{ds}, I, t). (El q \text{ ys } \text{ds } I, \text{TUPLE } t, E q \text{ ys } \text{ds}, \text{TUPLE}(\text{place-hats-L}$
 $t \text{ ds } I), \text{dir.N})) \text{ ' } (Q \times \text{func-set} \times \text{UNIV} \times \text{UNIV} \times \text{gamma-set})$
 $\cup (\lambda (q, \text{ys}, \text{ds}, I). (El q \text{ ys } \text{ds } I, \vdash, E q \text{ ys } \text{ds}, \vdash, \text{dir.N})) \text{ ' } (Q \times \text{func-set} \times$
 $\text{UNIV} \times \text{Pow}(\text{UNIV}))$ — first switch into E state, so E phase is always finished in
E state

definition $Q' =$

$R_1 \text{ ' } R1\text{-Set} \cup \{R_2\} \cup$
 $S_0 \text{ ' } Q \cup (\lambda (q, \text{inp}). S q \text{ inp}) \text{ ' } (Q \times \text{func-set}) \cup (\lambda (q, a). S_1 q a) \text{ ' } (Q \times (\text{UNIV}$
 $\rightarrow \Gamma)) \cup$
 $(\lambda (q, a, d). E_0 q a d) \text{ ' } (Q \times (\text{UNIV} \rightarrow \Gamma) \times \text{UNIV}) \cup$
 $(\lambda (q, a, d). E q a d) \text{ ' } (Q \times \text{func-set} \times \text{UNIV}) \cup$
 $(\lambda (q, a, d, I). Er q a d I) \text{ ' } (Q \times \text{func-set} \times \text{UNIV} \times \text{UNIV}) \cup$
 $(\lambda (q, a, d, I). Em q a d I) \text{ ' } (Q \times \text{func-set} \times \text{UNIV} \times \text{UNIV}) \cup$
 $(\lambda (q, a, d, I). El q a d I) \text{ ' } (Q \times \text{func-set} \times \text{UNIV} \times \text{UNIV})$

lemma $\text{compute-idx-range}[\text{simp}, \text{intro}]$:

assumes $tup \in \text{gamma-set}$

assumes $ys \in \text{func-set}$

shows $\text{compute-idx-set } tup \text{ ys} \in \text{UNIV}$

by *auto*

lemma $\text{update-ys-range}[\text{simp}, \text{intro}]$:

assumes $tup \in \text{gamma-set}$

assumes $ys \in \text{func-set}$

shows $\text{update-ys } tup \text{ ys} \in \text{func-set}$

by (*insert assms, fastforce simp: update-ys-def func-set-def*)

lemma $\text{replace-sym-range}[\text{simp}, \text{intro}]$:

assumes $tup \in \text{gamma-set}$
assumes $ys \in \text{func-set}$
shows $\text{replace-sym } tup \ ys \in \text{gamma-set}$
proof –
have $\forall k. (\text{if } k \in \text{compute-idx-set } tup \ ys \text{ then case } ys \ k \text{ of } \text{SYM } x \Rightarrow \text{NO-HAT } x \text{ else } tup \ k) \in \text{NO-HAT } ' \Gamma \cup \text{HAT } ' \Gamma$
by(*intro allI, insert assms, cases $k \in \text{compute-idx-set } tup \ ys$, auto simp: func-set-def compute-idx-set-def gamma-set-def replace-sym-def*)
then show *?thesis*
using *assms unfolding replace-sym-def gamma-set-def by blast*
qed

lemma *tup-hat-content:*
assumes $tup \in \text{gamma-set}$
assumes $tup \ x = \text{HAT } a$
shows $a \in \Gamma$
proof –
have $\text{range } tup \subseteq \text{NO-HAT } ' \Gamma \cup \text{HAT } ' \Gamma$
using *assms gamma-set-def by auto*
then show *?thesis*
using *assms(2)*
by (*metis UNIV-I Un-iff image-iff image-subset-iff tuple-symbol.distinct(1) tuple-symbol.inject(2)*)
qed

lemma *tup-no-hat-content:*
assumes $tup \in \text{gamma-set}$
assumes $tup \ x = \text{NO-HAT } a$
shows $a \in \Gamma$
proof –
have $\text{range } tup \subseteq \text{NO-HAT } ' \Gamma \cup \text{HAT } ' \Gamma$
using *assms gamma-set-def by auto*
then show *?thesis*
using *assms(2)*
by (*metis UNIV-I Un-iff image-iff image-subset-iff tuple-symbol.inject(1) tuple-symbol.simps(4)*)
qed

lemma *place-hats-to-dir-range[simp, intro]:*
assumes $tup \in \text{gamma-set}$
shows $\text{place-hats-to-dir } d \ tup \ ds \ I \in \text{gamma-set}$
proof –
have $\forall k. (\text{case } tup \ k \text{ of } \text{NO-HAT } a \Rightarrow \text{if } k \in I \wedge ds \ k = d \text{ then } \text{HAT } a \text{ else } \text{NO-HAT } a \mid \text{HAT } x \Rightarrow \text{HAT } x)$
 $\in \text{NO-HAT } ' \Gamma \cup \text{HAT } ' \Gamma$
proof
fix k
show (*case $tup \ k$ of $\text{NO-HAT } a \Rightarrow \text{if } k \in I \wedge ds \ k = d \text{ then } \text{HAT } a \text{ else } \text{NO-HAT } a \mid \text{HAT } x \Rightarrow \text{HAT } x$*)

$\in NO-HAT \text{ ' } \Gamma \cup HAT \text{ ' } \Gamma$
by(cases *tup k*, insert *tup-hat-content*[*OF assms(1)*] *tup-no-hat-content*[*OF assms(1)*], auto *simp*: *gamma-set-def*)
qed
then show *?thesis*
using *assms*
unfolding *place-hats-to-dir-def gamma-set-def*
by *auto*
qed

lemma *place-hats-range*[*simp,intro*]:
assumes *tup* \in *gamma-set*
shows *place-hats-R tup ds I* \in *gamma-set* **and**
place-hats-L tup ds I \in *gamma-set* **and**
place-hats-M tup ds I \in *gamma-set*
by(insert *assms*, auto *simp*: *place-hats-R-def place-hats-L-def place-hats-M-def*)

lemma *fin-R1-Set*[*intro,simp*]: *finite R1-Set*
unfolding *R1-Set-def* **using** *fin- Σ* **by** *auto*

lemma *fin-gamma-set*[*intro,simp*]: *finite gamma-set*
unfolding *gamma-set-def* **using** *fin- Γ*
by (*intro fin-funcsetI*, *auto*)

lemma *fin- Γ'* [*intro,simp*]: *finite Γ'*
unfolding *Γ' -def* **using** *fin- Σ* **by** *auto*

lemma *fin-func-set*[*simp,intro*]: *finite func-set*
unfolding *func-set-def* **using** *fin- Γ* **by** *auto*

lemma *memberships*[*simp,intro*]: $\vdash \in \Gamma'$
 $\cdot \in R1-Set$
 $x \in \Sigma \implies SYM x \in R1-Set$
 $x \in \Sigma \implies encSym x \in \Gamma'$
 $blank' \in \Gamma'$
 $hatLE' \in \Gamma'$
 $x \in \Sigma \implies INP x \in \Gamma'$
 $y \in gamma-set \implies TUPLE y \in \Gamma'$
 $(\lambda. \cdot) \in func-set$
 $f \in UNIV \rightarrow SYM \text{ ' } \Gamma \implies f \in func-set$
 $g \in UNIV \rightarrow \Gamma \implies SYM \circ g \in func-set$
 $f \in UNIV \rightarrow SYM \text{ ' } \Gamma \implies project-inp f k \in \Gamma$
unfolding *R1-Set-def Γ' -def blank'-def hatLE'-def gamma-set-def encSym-def*
func-set-def project-inp-def
using *LE blank tm funcset-mem*[of *f UNIV SYM ' Γ k*] **by** (*auto split*: *sym-or-bullet.splits*)

lemma *add-inp-func-set*[*simp,intro*]: $b \in gamma-set \implies a \in func-set \implies add-inp$
 $b a \in func-set$
unfolding *func-set-def gamma-set-def*

proof

fix x

assume $a: a \in UNIV \rightarrow SYM \text{ ' } \Gamma \cup \{\cdot\}$ **and** $b: b \in UNIV \rightarrow NO-HAT \text{ ' } \Gamma \cup HAT \text{ ' } \Gamma$

from a **have** $a: a x \in SYM \text{ ' } \Gamma \cup \{\cdot\}$ **by** *auto*

from b **have** $b: b x \in NO-HAT \text{ ' } \Gamma \cup HAT \text{ ' } \Gamma$ **by** *auto*

show $add-inp\ b\ a\ x \in SYM \text{ ' } \Gamma \cup \{\cdot\}$ **using** $a\ b$

unfolding $add-inp-def$ **by** (*cases* $b\ x$, *auto simp: gamma-set-def*)

qed

lemma $automation[simp]: \bigwedge a\ b\ A\ B. (S\ a\ b \in (\lambda x. case\ x\ of\ (x1, x2) \Rightarrow S\ x1\ x2) \text{ ' } (A \times B)) \longleftrightarrow (a \in A \wedge b \in B)$

$\bigwedge a\ b\ A\ B. (S_1\ a\ b \in (\lambda x. case\ x\ of\ (x1, x2) \Rightarrow S_1\ x1\ x2) \text{ ' } (A \times B)) \longleftrightarrow (a \in A \wedge b \in B)$

$\bigwedge a\ b\ c\ A\ B\ C. (E_0\ a\ b\ c \in (\lambda x. case\ x\ of\ (x1, x2, x3) \Rightarrow E_0\ x1\ x2\ x3) \text{ ' } (A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge c \in C)$

$\bigwedge a\ b\ c\ A\ B\ C. (E\ a\ b\ c \in (\lambda x. case\ x\ of\ (x1, x2, x3) \Rightarrow E\ x1\ x2\ x3) \text{ ' } (A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge c \in C)$

$\bigwedge a\ b\ c\ d\ A\ B\ C. (Er\ a\ b\ c\ d \in (\lambda x. case\ x\ of\ (x1, x2, x3, x4) \Rightarrow Er\ x1\ x2\ x3\ x4) \text{ ' } (A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge (c, d) \in C)$

$\bigwedge a\ b\ c\ d\ A\ B\ C. (Em\ a\ b\ c\ d \in (\lambda x. case\ x\ of\ (x1, x2, x3, x4) \Rightarrow Em\ x1\ x2\ x3\ x4) \text{ ' } (A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge (c, d) \in C)$

$\bigwedge a\ b\ c\ d\ A\ B\ C. (El\ a\ b\ c\ d \in (\lambda x. case\ x\ of\ (x1, x2, x3, x4) \Rightarrow El\ x1\ x2\ x3\ x4) \text{ ' } (A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge (c, d) \in C)$

$blank' \neq \vdash$

$\vdash \neq blank'$

$blank' \neq INP\ x$

$INP\ x \neq blank'$

by (*force simp: blank'-def*) $+$

interpretation $st: singletape-tm\ Q' (INP \text{ ' } \Sigma) \Gamma' blank' \vdash \delta' R_1 \cdot S_0\ t\ S_0\ r$

proof

show $finite\ Q'$

unfolding $Q'-def$ **using** $fin-Q\ fin-\Gamma$

by (*intro finite-UnI finite-imageI finite-cartesian-product, auto*)

show $finite\ \Gamma'$ **by** (*rule fin-\Gamma'*)

show $S_0\ t \in Q'$ **unfolding** $Q'-def$ **using** tQ **by** *auto*

show $S_0\ r \in Q'$ **unfolding** $Q'-def$ **using** rQ **by** *auto*

show $S_0\ t \neq S_0\ r$ **using** tr **by** *auto*

show $blank' \notin INP \text{ ' } \Sigma$ **unfolding** $blank'-def$ **by** *auto*

show $R_1 \cdot \in Q'$ **unfolding** $Q'-def$ **by** *auto*

show $\delta' \subseteq (Q' - \{S_0\ t, S_0\ r\}) \times \Gamma' \times Q' \times \Gamma' \times UNIV$

unfolding $\delta'-def\ Q'-def$ **using** tm

by (*auto dest: \delta*)

show $(q, \vdash, q', a', d) \in \delta' \implies a' = \vdash \wedge d \in \{dir.N, dir.R\}$ **for** $q\ q'\ a'\ d$

unfolding $\delta'-def$ **by** (*auto simp: hatLE'-def blank'-def*)

qed *auto*

lemma *valid-st: singletape-tm* $Q' (INP \text{ ' } \Sigma) \Gamma' \text{ blank}' \vdash \delta' (R_1 \cdot) (S_0 \ t) (S_0 \ r) \dots$

Determinism is preserved.

lemma *det-preservation: deterministic* \implies *st.deterministic*

unfolding *deterministic-def st.deterministic-def* **unfolding** δ' -*def*
by *auto*

6.2 Soundness of the Translation

lemma *range-mt-pos:*

$\exists i. \text{Max} (\text{range} (\text{mt-pos } \text{cm})) = \text{mt-pos } \text{cm } i$
finite ($\text{range} (\text{mt-pos } (\text{cm} :: ('a, 'q, 'k) \text{ mt-config}))$)
 $\text{range} (\text{mt-pos } \text{cm}) \neq \{\}$

proof –

show *finite* ($\text{range} (\text{mt-pos } \text{cm})$) **by** *auto*
moreover **show** $\text{range} (\text{mt-pos } \text{cm}) \neq \{\}$ **by** *auto*
ultimately **show** $\exists i. \text{Max} (\text{range} (\text{mt-pos } \text{cm})) = \text{mt-pos } \text{cm } i$
by (*meson Max-in imageE*)

qed

lemma *max-mt-pos-step: assumes* $(\text{cm}, \text{cm}') \in \text{step}$

shows $\text{Max} (\text{range} (\text{mt-pos } \text{cm}')) \leq \text{Suc} (\text{Max} (\text{range} (\text{mt-pos } \text{cm})))$

proof –

from *range-mt-pos(1)[of cm']* **obtain** i'
where $\text{max1}: \text{Max} (\text{range} (\text{mt-pos } \text{cm}')) = \text{mt-pos } \text{cm}' \ i'$ **by** *auto*
hence $\text{Max} (\text{range} (\text{mt-pos } \text{cm}')) \leq \text{mt-pos } \text{cm}' \ i'$ **by** *auto*
also **have** $\dots \leq \text{Suc} (\text{mt-pos } \text{cm } i')$ **using** *assms*

proof (*cases*)

case (*step q ts n q' a dir*)

then **show** *?thesis* **by** (*cases dir i', auto*)

qed

also **have** $\dots \leq \text{Suc} (\text{Max} (\text{range} (\text{mt-pos } \text{cm})))$ **using** *range-mt-pos[of cm]* **by**

simp

finally **show** *?thesis* .

qed

lemma *max-mt-pos-init: Max* ($\text{range} (\text{mt-pos} (\text{init-config } w))$) $= 0$

unfolding *init-config-def* **by** *auto*

lemma *INP-D: assumes* $\text{set } x \subseteq \text{INP } \text{' } \Sigma$

shows $\exists w. x = \text{map } \text{INP } w \wedge \text{set } w \subseteq \Sigma$

using *assms*

proof (*induct x*)

case (*Cons x xs*)

then **obtain** w **where** $xs = \text{map } \text{INP } w \wedge \text{set } w \subseteq \Sigma$ **by** *auto*

moreover **from** *Cons(2)* **obtain** a **where** $x = \text{INP } a$ **and** $a \in \Sigma$ **by** *auto*

ultimately **show** *?case* **by** (*intro exI[of - a # w], auto*)

qed *auto*

6.2.1 R-Phase

fun $enc :: ('a, 'q, 'k) mt-config \Rightarrow nat \Rightarrow ('a, 'k) st-tape-symbol$

where $enc (Config_M q tc p) n = TUPLE (\lambda k. \text{if } p \ k = n \text{ then } HAT (tc \ k \ n) \text{ else } NO-HAT (tc \ k \ n))$

inductive $rel-R_1 :: (('a, 'k) st-tape-symbol, ('a, 'q, 'k) st-states) st-config \Rightarrow 'a \ list \Rightarrow nat \Rightarrow bool$ **where**

$n = length \ w \Longrightarrow$

$tc' \ 0 = \vdash \Longrightarrow$

$p' \leq n \Longrightarrow$

$(\bigwedge i. i < p' \Longrightarrow enc (init-config \ w) \ i = tc' (Suc \ i)) \Longrightarrow$

$(\bigwedge i. i \geq p' \Longrightarrow tc' (Suc \ i) = (\text{if } i < n \text{ then } INP (w \ ! \ i) \text{ else } blank')) \Longrightarrow$

$(p' = 0 \Longrightarrow q' = \cdot) \Longrightarrow$

$(\bigwedge p. p' = Suc \ p \Longrightarrow q' = SYM (w \ ! \ p)) \Longrightarrow$

$rel-R_1 (Config_S (R_1 \ q') \ tc' (Suc \ p')) \ w \ p'$

lemma $rel-R_1\text{-init}$: **shows** $\exists \ cs1. (st.init-config (map \ INP \ w), \ cs1) \in st.dstep \wedge rel-R_1 \ cs1 \ w \ 0$

proof –

let $?INP = INP :: 'a \Rightarrow ('a, 'k) st-tape-symbol$

have $mem: (R_1 \cdot, \vdash, R_1 \cdot, \vdash, dir.R) \in \delta' \text{ unfolding } \delta'\text{-def \textit{by} \textit{auto}}$

let $?cs1 = Config_S (R_1 \cdot) (\lambda n. \text{if } n = 0 \text{ then } \vdash \text{ else if } n \leq length (map \ ?INP \ w) \text{ then } map \ ?INP \ w \ ! \ (n - 1) \text{ else } blank') (Suc \ 0)$

have $(st.init-config (map \ INP \ w), \ ?cs1) \in st.dstep$

unfolding $st.init-config\text{-def}$ **by** $(rule \ st.dstepI[OF \ mem], \ auto \ simp: \ \delta'\text{-def} \ blank'\text{-def})$

moreover **have** $rel-R_1 \ ?cs1 \ w \ 0$

by $(intro \ rel-R_1.intros[OF \ refl], \ auto)$

ultimately **show** $?thesis$ **by** $blast$

qed

lemma $rel-R_1\text{-}R_1$: **assumes** $rel-R_1 \ cs0 \ w \ j$

and $j < length \ w$

and $set \ w \subseteq \Sigma$

shows $\exists \ cs1. (cs0, \ cs1) \in st.dstep \wedge rel-R_1 \ cs1 \ w (Suc \ j)$

using $assms(1)$

proof $(cases \ rule: \ rel-R_1.cases)$

case $(1 \ n \ tc' \ q')$

note $cs0 = 1(1)$

from $assms$ **have** $wj: w \ ! \ j \in \Sigma$ **by** $auto$

show $?thesis$

proof $(cases \ j)$

case 0

with 1 **have** $q': q' = \cdot$ **by** $auto$

from $1(6)[of \ 0] \ 0 \ assms \ 1$ **have** $tc'1: tc' (Suc \ 0) = INP (w \ ! \ 0)$ **by** $auto$

have $mem: (R_1 \cdot, \ INP (w \ ! \ 0), \ R_1 (SYM (w \ ! \ 0)), \ hatLE', \ dir.R) \in \delta'$

unfolding $\delta'\text{-def}$

using $wj \ 0$ **by** $auto$

let $?cs1 = \text{Config}_S (R_1 (SYM (w ! 0))) (tc'(Suc 0 := \text{hatLE}')) (Suc (Suc 0))$
have $\text{enc}: \text{enc} (\text{init-config } w) 0 = \text{hatLE}'$ **unfolding** $\text{init-config-def hatLE}'\text{-def}$
by *auto*
have $(cs0, ?cs1) \in \text{st.dstep}$ **unfolding** $cs0 0$
by $(\text{intro st.dstepI}[OF \text{ mem}], \text{auto simp: } q' \text{ tc}'1 \delta'\text{-def blank}'\text{-def})$
moreover **have** $\text{rel-}R_1 ?cs1 w (Suc 0)$
by $(\text{intro rel-}R_1.\text{intros}, \text{rule } 1(2), \text{insert } 1 0 \text{ assms}(2), \text{auto simp: enc}) (\text{cases } w, \text{auto})$
ultimately **show** $?thesis$ **unfolding** 0 **by** *blast*
next
case $(Suc p)$
from $1(8)[OF \text{ Suc}]$ **have** $q': q' = SYM (w ! p)$ **by** *auto*
from $Suc \text{ assms}(2)$ **have** $p < \text{length } w$ **by** *auto*
with $\text{assms}(3)$ **have** $w ! p \in \Sigma$ **by** *auto*
with wj **have** $(w ! p, w ! j) \in \Sigma \times \Sigma$ **by** *auto*
hence $\text{mem}: (R_1 (SYM (w ! p)), INP (w ! j), R_1 (SYM (w ! j)), \text{encSym } (w ! p), \text{dir.}R) \in \delta'$ **unfolding** $\delta'\text{-def}$ **by** *auto*
have $\text{enc}: \text{enc} (\text{init-config } w) j = \text{encSym } (w ! p)$ **unfolding** Suc **using** $\langle p < \text{length } w \rangle$
by $(\text{auto simp: init-config-def encSym-def})$
from $1(6)[of j] \text{ assms } 1$ **have** $tc': tc' (Suc j) = INP (w ! j)$ **by** *auto*
let $?cs1 = \text{Config}_S (R_1 (SYM (w ! j))) (tc'(Suc j := \text{encSym } (w ! p))) (Suc (Suc j))$
have $(cs0, ?cs1) \in \text{st.dstep}$ **unfolding** $cs0$
by $(\text{rule st.dstepI}[OF \text{ mem}], \text{insert } q' \text{ tc}', \text{auto simp: } \delta'\text{-def blank}'\text{-def})$
moreover **have** $\text{rel-}R_1 ?cs1 w (Suc j)$
by $(\text{intro rel-}R_1.\text{intros}, \text{insert } 1 \text{ assms enc}, \text{auto})$
ultimately **show** $?thesis$ **by** *blast*
qed
qed

inductive $\text{rel-}R_2 :: ((a, 'k) \text{ st-tape-symbol}, (a, 'q, 'k) \text{ st-states}) \text{st-config} \Rightarrow 'a \text{ list} \Rightarrow \text{nat} \Rightarrow \text{bool}$ **where**
 $tc' 0 = \vdash \implies$
 $(\bigwedge i. \text{enc} (\text{init-config } w) i = tc' (Suc i)) \implies$
 $p \leq \text{length } w \implies$
 $\text{rel-}R_2 (\text{Config}_S R_2 tc' p) w p$

lemma $\text{rel-}R_1\text{-}R_2$: **assumes** $\text{rel-}R_1 cs0 w (\text{length } w)$
and $\text{set } w \subseteq \Sigma$
shows $\exists cs1. (cs0, cs1) \in \text{st.dstep} \wedge \text{rel-}R_2 cs1 w (\text{length } w)$
using assms
proof $(\text{cases rule: rel-}R_1.\text{cases})$
case $(1 n tc' q')$
note $cs0 = 1(1)$
have $\text{enc}: \text{enc} (\text{init-config } w) i = tc' (Suc i)$ **if** $i \neq \text{length } w$ **for** i
proof $(\text{cases } i < \text{length } w)$
case *True*

```

    thus ?thesis using 1(5)[of i] by auto
next
  case False
  with that have i: i > length w by auto
  with 1(6)[of i] 1 have tc' (Suc i) = blank' by auto
  also have ... = enc (init-config w) i using i unfolding init-config-def by
(auto simp: blank'-def)
  finally show ?thesis by simp
qed
show ?thesis
proof (cases length w)
  case 0
  with 1 have q': q' = · by auto
  from 1(6)[of 0] 0 1 have tc'1: tc' (Suc 0) = blank' by auto
  have mem: (R1 ·, blank', R2, hatLE', dir.L) ∈ δ' unfolding δ'-def
  by auto
  let ?tc = tc' (Suc 0 := hatLE')
  let ?cs1 = Configs R2 ?tc 0
  have enc0: enc (init-config w) 0 = hatLE' unfolding init-config-def hatLE'-def
  by auto
  have enc: enc (init-config w) i = ?tc (Suc i) for i using enc[of i] enc0 using
0
  by (cases i, auto)
  have (cs0, ?cs1) ∈ st.dstep unfolding cs0 0
  by (intro st.dstepI[OF mem], auto simp: q' tc'1 δ'-def blank'-def)
  moreover have rel-R2 ?cs1 w (length w) unfolding 0
  by (intro rel-R2.intros enc, insert 1 0, auto)
  ultimately show ?thesis unfolding 0 by blast
next
  case (Suc p)
  from 1(8)[OF Suc] have q': q' = SYM (w ! p) by auto
  from Suc have p < length w by auto
  with assms(2) have w ! p ∈ Σ by auto
  hence mem: (R1 (SYM (w ! p)), blank', R2, encSym (w ! p), dir.L) ∈ δ'
  unfolding δ'-def by auto
  let ?tc = tc' (Suc (length w) := encSym (w ! p))
  have encW: enc (init-config w) (length w) = encSym (w ! p) unfolding Suc
  using ⟨p < length w⟩
  by (auto simp: init-config-def encSym-def)
  from 1(6)[of length w] assms 1 have tc': tc' (Suc (length w)) = blank' by auto
  let ?cs1 = Configs R2 ?tc (length w)
  have enc: enc (init-config w) i = ?tc (Suc i) for i using enc[of i] encW by
auto
  have (cs0, ?cs1) ∈ st.dstep unfolding cs0 q'
  by (intro st.dstepI[OF mem] tc', auto simp: δ'-def blank'-def)
  moreover have rel-R2 ?cs1 w (length w)
  by (intro rel-R2.intros, insert 1 assms enc, auto)
  ultimately show ?thesis by blast
qed

```


qed

lemma *rel-R₂-R₂*: **assumes** *rel-R₂ cs0 w (Suc j)*
and *set w ⊆ Σ*
shows $\exists cs1. (cs0, cs1) \in st.dstep \wedge rel-R_2 cs1 w j$
using *assms*
proof (*cases rule: rel-R₂.cases*)
case (1 *tc'*)
note *cs0 = 1(1)*
from 1 **have** *j: j < length w by auto*
have *tc: tc' (Suc j) ∈ Γ' - {⊢}* **unfolding** 1(3)[*symmetric*] **using** *j assms(2)[unfolded set-conv-nth]* **unfolding** *init-config-def*
by (*force simp: Γ'-def gamma-set-def intro!: imageI LE blank set-mp[OF Σ-sub-Γ, of w ! (j - Suc 0)]*)
hence *mem: (R₂, tc' (Suc j), R₂, tc' (Suc j), dir.L) ∈ δ'* **unfolding** *δ'-def* **by** *auto*
let *?cs1 = Config_S R₂ tc' j*
have (*cs0, ?cs1*) $\in st.dstep$ **unfolding** *cs0* **using** *tc*
by (*intro st.dstepI[OF mem], auto simp: δ'-def blank'-def*)
moreover **have** *rel-R₂ ?cs1 w j*
by (*intro rel-R₂.intros, insert 1, auto*)
ultimately show *?thesis* **by** *blast*
qed

inductive *rel-S₀* :: (*'a, 'k* *st-tape-symbol*, (*'a, 'q, 'k* *st-states*) *st-config*) \Rightarrow (*'a, 'q, 'k*) *mt-config* \Rightarrow *bool* **where**
tc' 0 = ⊢ \Longrightarrow
 $(\bigwedge i. tc' (Suc i) = enc (Config_M q tc p) i) \Longrightarrow$
 $valid-config (Config_M q tc p) \Longrightarrow$
 $rel-S_0 (Config_S (S_0 q) tc' 0) (Config_M q tc p)$

lemma *rel-R₂-S₀*: **assumes** *rel-R₂ cs0 w 0*
and *set w ⊆ Σ*
shows $\exists cs1. (cs0, cs1) \in st.dstep \wedge rel-S_0 cs1 (init-config w)$
using *assms*
proof (*cases rule: rel-R₂.cases*)
case (1 *tc'*)
note *cs0 = 1(1)*
hence *mem: (R₂, ⊢, S₀ s, ⊢, dir.N) ∈ δ'* **unfolding** *δ'-def* **by** *auto*
let *?cs1 = Config_S (S₀ s) tc' 0*
have (*cs0, ?cs1*) $\in st.dstep$ **unfolding** *cs0*
by (*intro st.dstepI[OF mem], insert 1, auto simp: δ'-def blank'-def*)
moreover **have** *rel-S₀ ?cs1 (init-config w)* **using** *valid-init-config[OF assms(2)]*
unfolding *init-config-def*
by (*intro rel-S₀.intros, insert 1(1,2,4-), auto simp: 1(3)[symmetric] init-config-def*)
ultimately show *?thesis* **by** *blast*
qed

If we start with a proper word *w* as input on the singletape TM, then via

the R-phase one can switch to the beginning of the S-phase ($rel-S_0$) for the initial configuration.

lemma R-phase: assumes $set\ w \subseteq \Sigma$

shows $\exists\ cs. (st.init-config\ (map\ INP\ w),\ cs) \in st.dstep \sim (3 + 2 * length\ w) \wedge rel-S_0\ cs\ (init-config\ w)$

proof –

from $rel-R_1-init[of\ w]$ **obtain** $cs1\ n$ **where**

$step1: (st.init-config\ (map\ INP\ w),\ cs1) \in st.dstep$ **and**

$relR1: rel-R_1\ cs1\ w\ n$ **and**

$n0: n = 0$

by *auto*

from $relR1$

have $n + k \leq length\ w \implies \exists\ cs2. (cs1,\ cs2) \in st.dstep \sim k \wedge rel-R_1\ cs2\ w\ (n + k)$ **for** k

proof (*induction k arbitrary: cs1 n*)

case $(Suc\ k\ cs1\ n)$

hence $n < length\ w$ **by** *auto*

from $rel-R_1-R_1[OF\ Suc(3)\ this\ assms]$ **obtain** $cs3$ **where**

$step: (cs1,\ cs3) \in st.dstep$ **and** $rel: rel-R_1\ cs3\ w\ (Suc\ n)$ **by** *auto*

from $Suc.IH[OF\ -\ rel]\ Suc(2)$ **obtain** $cs2$ **where**

$steps: (cs3,\ cs2) \in st.dstep \sim k$ **and** $rel: rel-R_1\ cs2\ w\ (Suc\ n + k)$

by *auto*

from $relpow-Suc-I2[OF\ step\ steps]$ rel

show *?case* **by** *auto*

qed *auto*

from $this[of\ length\ w,\ unfolded\ n0]$

obtain $cs2$ **where** $steps2: (cs1,\ cs2) \in st.dstep \sim length\ w$ **and** $rel: rel-R_1\ cs2\ w\ (length\ w)$ **by** *auto*

from $rel-R_1-R_2[OF\ rel\ assms]$ **obtain** $cs3\ n$ **where** $step3: (cs2,\ cs3) \in st.dstep$

and $rel: rel-R_2\ cs3\ w\ n$ **and** $n: n = length\ w$

by *auto*

from rel **have** $\exists\ cs. (cs3,\ cs) \in st.dstep \sim n \wedge rel-R_2\ cs\ w\ 0$

proof (*induction n arbitrary: cs3 rule: nat-induct*)

case $(Suc\ n\ cs3)$

from $rel-R_2-R_2[OF\ Suc(2)\ assms]$ **obtain** $cs1$ **where**

$step: (cs3,\ cs1) \in st.dstep$ **and** $rel: rel-R_2\ cs1\ w\ n$ **by** *auto*

from $Suc.IH[OF\ rel]$ **obtain** cs **where** $steps: (cs1,\ cs) \in st.dstep \sim n$ **and** $rel: rel-R_2\ cs\ w\ 0$ **by** *auto*

from $relpow-Suc-I2[OF\ step\ steps]$ rel **show** *?case* **by** *auto*

qed *auto*

then obtain $cs4$ **where** $steps4: (cs3,\ cs4) \in st.dstep \sim (length\ w)$ **and** $rel: rel-R_2\ cs4\ w\ 0$ **by** (*auto simp: n*)

from $rel-R_2-S_0[OF\ rel\ assms]$ **obtain** cs **where** $step5: (cs4,\ cs) \in st.dstep$ **and**

$rel: rel-S_0\ cs\ (init-config\ w)$ **by** *auto*

from $relpow-Suc-I2[OF\ step1\ relpow-transI[OF\ steps2\ relpow-Suc-I2[OF\ step3\ relpow-Suc-I[OF\ steps4\ step5]]]]$

have $(st.init-config\ (map\ INP\ w),\ cs) \in st.dstep \sim Suc\ (length\ w + Suc\ (Suc\ (length\ w)))$.

also have $Suc\ (length\ w + Suc\ (Suc\ (length\ w))) = 3 + 2 * length\ w$ **by** *simp*

finally show *?thesis* using *rel* by *auto*
qed

6.2.2 S-Phase

inductive *rel-S* :: (('a, 'k) *st-tape-symbol*, ('a, 'q, 'k) *st-states*) *st-config* \Rightarrow ('a, 'q, 'k) *mt-config* \Rightarrow *nat* \Rightarrow *bool* **where**
 $tc' 0 = \vdash \Longrightarrow$
 $(\bigwedge i. tc' (Suc i) = enc (Config_M q tc p) i) \Longrightarrow$
 $valid-config (Config_M q tc p) \Longrightarrow$
 $(\bigwedge i. inp i = (if p i < p' then SYM (tc i (p i)) else \cdot)) \Longrightarrow$
 $rel-S (Config_S (S q inp) tc' (Suc p')) (Config_M q tc p) p'$

lemma *rel-S₀-S*: **assumes** *rel-S₀ cs0 cm*

and *mt-state cm* $\notin \{t, r\}$

shows $\exists cs1. (cs0, cs1) \in st.dstep \wedge rel-S cs1 cm 0$

using *assms(1)*

proof (*cases rule: rel-S₀.cases*)

case $(1 tc' q tc p)$

note $cs0 = 1(1)$

note $cm = 1(2)$

from *assms(2) cm 1(5)* **have** *qtr*: $q \in Q - \{t, r\}$ **by** *auto*

hence *mem*: $(S_0 q, \vdash, S q (\lambda \cdot. \cdot), \vdash, dir.R) \in \delta'$ **unfolding** δ' -*def* **by** *auto*

let $?cs1 = Config_S (S q (\lambda \cdot. \cdot)) tc' (Suc 0)$

have $(cs0, ?cs1) \in st.dstep$ **unfolding** *cs0*

by (*rule st.dstepI[OF mem], insert 1, auto simp: δ' -def blank'-def*)

moreover **have** *rel-S ?cs1 cm 0* **unfolding** *cm*

by (*intro rel-S.intros 1, auto*)

ultimately show *?thesis* **by** *blast*

qed

lemma *rel-S-mem*: **assumes** *rel-S (Config_S (S q inp) tc' p') cm j*

shows $inp \in func-set \wedge q \in Q \wedge (\exists t. tc' (Suc i) = TUPLE t \wedge t \in gamma-set)$

using *assms(1)*

proof (*cases rule: rel-S.cases*)

case $(1 tc p)$

from *1* **have** *q*: $q \in Q$ **by** *auto*

have *inp*: $inp \in func-set$ **unfolding** *func-set-def 1(6)* **using** *1(5)*

by *force*

have $\exists t. tc' (Suc i) = TUPLE t \wedge t \in gamma-set$ **using** *1(5)*

unfolding *1(4)* **by** (*force simp: gamma-set-def*)

with *q inp* **show** *?thesis* **by** *auto*

qed

lemma *rel-S-S*: **assumes** *rel-S cs0 cm p'*

$p' \leq Max (range (mt-pos cm))$

shows $\exists cs1. (cs0, cs1) \in st.dstep \wedge rel-S cs1 cm (Suc p')$

using *assms(1)*

proof (*cases rule: rel-S.cases*)
case ($1\ tc' q\ tc\ p\ inp$)
note $cs0 = 1(1)$
note $cm = 1(2)$
let $?Set = Q \times (func\text{-}set - (UNIV \rightarrow SYM\ ' \Gamma)) \times gamma\text{-}set$
let $?f = \lambda(q, inp, t). (S\ q\ inp, TUPLE\ t, S\ q\ (add\text{-}inp\ t\ inp), TUPLE\ t, dir.R)$
obtain i **where** $mt\text{-}pos\ cm\ i = Max\ (range\ (mt\text{-}pos\ cm))$ **using** $range\text{-}mt\text{-}pos(1)[of\ cm]$ **by** *auto*
with *assms* 1 **have** $p' \leq p\ i$ **by** *auto*
with $1(6)[of\ i]$ **have** $inp\ i = \cdot$ **by** *auto*
hence $inp: inp \notin (UNIV \rightarrow SYM\ ' \Gamma)$
by (*metis* $PiE\ UNIV\text{-}I\ image\text{-}iff\ sym\text{-}or\text{-}bullet.distinct(1)$)
with $rel\text{-}S\text{-}mem[OF\ assms(1)[unfolded\ cs0], of\ p']$ **obtain** t **where**
 $(q, inp, t) \in ?Set$ **and** $tc': tc' (Suc\ p') = TUPLE\ t$ **by** *auto*
hence $?f\ (q, inp, t) \in \delta'$ **unfolding** $\delta'\text{-}def$ **by** *blast*
hence $mem: (S\ q\ inp, TUPLE\ t, S\ q\ (add\text{-}inp\ t\ inp), TUPLE\ t, dir.R) \in \delta'$ **by**
simp
let $?cs1 = Config_S\ (S\ q\ (add\text{-}inp\ t\ inp))\ tc'\ (Suc\ (Suc\ p'))$
have $(cs0, ?cs1) \in st.dstep$ **unfolding** $cs0$ **using** inp
by (*intro* $st.dstepI[OF\ mem], auto\ simp: tc'\ \delta'\text{-}def\ blank'\text{-}def$)
moreover **have** $rel\text{-}S\ ?cs1\ cm\ (Suc\ p')$ **unfolding** cm
proof (*intro* $rel\text{-}S.intros\ 1$)
from $1(4)[of\ p', unfolded\ tc']$
have $t: t = (\lambda k. if\ p\ k = p' then\ HAT\ (tc\ k\ p') else\ NO\text{-}HAT\ (tc\ k\ p'))$ **by**
auto
show $\bigwedge k. add\text{-}inp\ t\ inp\ k = (if\ p\ k < Suc\ p' then\ SYM\ (tc\ k\ (p\ k)) else\ \cdot)$
unfolding $add\text{-}inp\text{-}def\ 1\ t$ **by** *auto*
qed
ultimately show $?thesis$ **by** *blast*
qed

inductive $rel\text{-}S_1 :: (('a, 'k)\ st\text{-}tape\text{-}symbol, ('a, 'q, 'k)\ st\text{-}states)\ st\text{-}config \Rightarrow ('a, 'q, 'k)\ mt\text{-}config \Rightarrow bool$ **where**
 $tc'\ 0 = \vdash \Longrightarrow$
 $(\bigwedge i. tc'\ (Suc\ i) = enc\ (Config_M\ q\ tc\ p)\ i) \Longrightarrow$
 $valid\text{-}config\ (Config_M\ q\ tc\ p) \Longrightarrow$
 $(\bigwedge i. inp\ i = tc\ i\ (p\ i)) \Longrightarrow$
 $(\bigwedge i. p\ i < p') \Longrightarrow$
 $p' = Suc\ (Max\ (range\ p)) \Longrightarrow$
 $rel\text{-}S_1\ (Config_S\ (S_1\ q\ inp)\ tc'\ p')\ (Config_M\ q\ tc\ p)$

lemma $rel\text{-}S\text{-}S_1$: **assumes** $rel\text{-}S\ cs0\ cm\ p'$
 $p' = Suc\ (Max\ (range\ (mt\text{-}pos\ cm)))$
shows $\exists\ cs1. (cs0, cs1) \in st.dstep \wedge rel\text{-}S_1\ cs1\ cm$
using $assms(1)$
proof (*cases rule: rel-S.cases*)
case ($1\ tc' q\ tc\ p\ inp$)
from *assms* **have** $p' > Max\ (range\ (mt\text{-}pos\ cm))$ **by** *auto*
note $cs0 = 1(1)$

note $cm = 1(2)$
from $p' \text{Max range-}mt\text{-pos}(2-)[\text{of } cm]$ **have** $pip: p \ i < p'$ **for** i **unfolding** cm
by *auto*
let $?SET = Q \times \text{func-set} \times \text{gamma-set}$
let $?Set = Q \times (UNIV \rightarrow SYM \ ' \Gamma) \times (\Gamma' - \{ \vdash \})$
from $rel\text{-}S\text{-mem}[OF \ \text{assms}(1)[\text{unfolded } cs0], \text{ of } p']$ **obtain** t **where**
 $mem: (q, \text{inp}, t) \in ?SET$ **and** $tc': tc' (Suc \ p') = TUPLE \ t$ **by** *auto*
hence $\text{inp} \in \text{func-set}$ **by** *auto*
with pip **have** $\text{inp}: \text{inp} \in UNIV \rightarrow SYM \ ' \Gamma$ **unfolding** func-set-def **using** $1(6)$
by *auto*
with mem **have** $(q, \text{inp}, TUPLE \ t) \in ?Set$ **by** *auto*
hence $(\lambda (q, \text{inp}, a). (S \ q \ \text{inp}, a, S_1 \ q \ (\text{project-inp } \text{inp}), a, \text{dir.L})) (q, \text{inp}, TUPLE \ t) \in \delta'$ **unfolding** $\delta'\text{-def}$ **by** *blast*
hence $mem: (S \ q \ \text{inp}, TUPLE \ t, S_1 \ q \ (\text{project-inp } \text{inp}), TUPLE \ t, \text{dir.L}) \in \delta'$
by *simp*
let $?cs1 = \text{Config}_S (S_1 \ q \ (\text{project-inp } \text{inp})) \ tc' \ p'$
have $(cs0, ?cs1) \in st.dstep$ **unfolding** $cs0$ **using** inp
by $(\text{intro } st.dstepI[OF \ mem], \text{ auto } simp: tc' \ \delta'\text{-def } blank'\text{-def})$
moreover **have** $rel\text{-}S_1 \ ?cs1 \ cm$ **unfolding** cm
proof $(\text{intro } rel\text{-}S_1.\text{intros } 1 \ pip)$
fix i
from inp **have** $\text{inp } i \in SYM \ ' \Gamma$ **by** *auto*
then **obtain** g **where** $\text{inp } i = SYM \ g$ **and** $g \in \Gamma$ **by** *auto*
thus $\text{project-inp } \text{inp } i = tc \ i \ (p \ i)$ **using** $1(6)[\text{of } i]$ **by** $(\text{auto } simp: \text{project-inp-def } split: \text{if-splits})$
show $p' = Suc \ (\text{Max } (\text{range } p))$ **unfolding** $\text{assms}(2)$ **unfolding** cm **by** *simp*
qed
ultimately show $?thesis$ **by** *auto*
qed

If we start the S-phase (in $rel\text{-}S_0$), and the multitape-TM is not in a final state, then we can move to the end of the S-phase (in $rel\text{-}S_1$).

lemma *S-phase: assumes* $rel\text{-}S_0 \ cs \ cm$

and $mt\text{-state } cm \notin \{t, r\}$

shows $\exists \ cs'. (cs, cs') \in st.dstep \sim (\mathcal{B} + \text{Max } (\text{range } (mt\text{-pos } cm))) \wedge rel\text{-}S_1 \ cs' \ cm$

proof –

let $?N = \text{Max } (\text{range } (mt\text{-pos } cm))$

from $rel\text{-}S_0\text{-}S[OF \ \text{assms}]$ **obtain** $cs1 \ n$ **where**

$step1: (cs, cs1) \in st.dstep$ **and** $rel: rel\text{-}S \ cs1 \ cm \ n$ **and** $n: n = 0$

by *auto*

from rel **have** $n + k \leq Suc \ ?N \implies \exists \ cs2. (cs1, cs2) \in st.dstep \sim k \wedge rel\text{-}S \ cs2 \ cm \ (n + k)$ **for** k

proof $(\text{induction } k \ \text{arbitrary: } cs1 \ n)$

case $(Suc \ k \ cs \ n)$

hence $n \leq ?N$ **by** *auto*

from $rel\text{-}S\text{-}S[OF \ Suc(\mathcal{B}) \ \text{this}]$ **obtain** $cs1$ **where** $step: (cs, cs1) \in st.dstep$ **and** $rel: rel\text{-}S \ cs1 \ cm \ (Suc \ n)$ **by** *auto*

from Suc **have** $Suc \ n + k \leq Suc \ ?N$ **by** *auto*

from $Suc.IH[OF\ this\ rel]$ **obtain** $cs2$ **where** $steps: (cs1, cs2) \in st.dstep \overset{\sim}{\sim} k$
and $rel: rel-S\ cs2\ cm\ (n + Suc\ k)$ **by** $auto$
from $relpow-Suc-I2[OF\ step\ steps]$ rel
show $?case$ **by** $auto$
qed $auto$
from $this[of\ Suc\ ?N, unfolded\ n]$ **obtain** $cs2$ **where**
 $steps2: (cs1, cs2) \in st.dstep \overset{\sim}{\sim} (Suc\ ?N)$ **and** $rel: rel-S\ cs2\ cm\ (Suc\ ?N)$ **by**
 $auto$
from $rel-S-S_1[OF\ rel]$ **obtain** $cs3$ **where** $step3: (cs2, cs3) \in st.dstep$ **and** $rel:$
 $rel-S_1\ cs3\ cm$ **by** $auto$
from $relpow-Suc-I2[OF\ step1\ relpow-Suc-I[OF\ steps2\ step3]]$
have $(cs, cs3) \in st.dstep \overset{\sim}{\sim} Suc\ (Suc\ (Suc\ ?N))$ **by** $simp$
also **have** $Suc\ (Suc\ (Suc\ ?N)) = 3 + ?N$ **by** $simp$
finally **show** $?thesis$ **using** rel **by** $blast$
qed

6.2.3 E-Phase

context

fixes $rule :: ('a, 'q, 'k)mt-rule$

begin

inductive-set $\delta step :: ('a, 'q, 'k) mt-config\ rel$ **where**

$\delta step: rule = (q, a, q1, b, dir) \implies$

$rule \in \delta \implies$

$(\bigwedge k. ts\ k\ (n\ k) = a\ k) \implies$

$(\bigwedge k. ts'\ k = (ts\ k)(n\ k := b\ k)) \implies$

$(\bigwedge k. n'\ k = go-dir\ (dir\ k)\ (n\ k)) \implies$

$(Config_M\ q\ ts\ n, Config_M\ q1\ ts'\ n') \in \delta step$

end

lemma $step-to-\delta step: (c1, c2) \in step \implies \exists rule. (c1, c2) \in \delta step\ rule$

proof ($induct\ rule: step.induct$)

case $(step\ q\ ts\ n\ q'\ a\ dir)$

show $?case$

by $(rule\ exI, rule\ \delta step.intros[OF\ refl\ step], auto)$

qed

lemma $\delta step-to-step: (c1, c2) \in \delta step\ rule \implies (c1, c2) \in step$

proof ($induct\ rule: \delta step.induct$)

case $*$: $(\delta step\ q\ a\ q'\ b\ dir\ ts\ n\ ts'\ n')$

from $*$ **have** $a: a = (\lambda k. ts\ k\ (n\ k))$ **by** $auto$

from $*$ **have** $ts': ts' = (\lambda k. (ts\ k)(n\ k := b\ k))$ **by** $auto$

from $*$ **have** $n': n' = (\lambda k. go-dir\ (dir\ k)\ (n\ k))$ **by** $auto$

from $*$ **show** $?case$ **using** $step.intros[of\ q\ ts\ n\ q'\ b\ dir]$ **unfolding** $a\ ts'\ n'$ **by**

$auto$

qed

inductive $rel-E_0 :: (('a, 'k) st-tape-symbol, ('a, 'q, 'k) st-states)st-config$

$\Rightarrow ('a, 'q, 'k) mt-config \Rightarrow ('a, 'q, 'k) mt-config \Rightarrow ('a, 'q, 'k) mt-rule \Rightarrow bool$ **where**

$$\begin{aligned}
& tc' \ 0 = \vdash \implies \\
& (\bigwedge i. tc' (Suc \ i) = enc (Config_M \ q \ tc \ p) \ i) \implies \\
& valid-config (Config_M \ q \ tc \ p) \implies \\
& rule = (q, a, q1, b, d) \implies \\
& (Config_M \ q \ tc \ p, Config_M \ q1 \ tc1 \ p1) \in \delta step \ rule \implies \\
& (\bigwedge i. p \ i < p') \implies \\
& p' = Suc (Max (range \ p)) \implies \\
& rel-E_0 (Config_S (E_0 \ q1 \ b \ d) \ tc' \ p') (Config_M \ q \ tc \ p) (Config_M \ q1 \ tc1 \ p1) \ rule
\end{aligned}$$

For the transition between S and E phase we do not have deterministic steps. Therefore we add two lemmas: the former one is for showing that multitape can be simulated by singletape, and the latter one is for the inverse direction.

lemma *rel-S₁-E₀-step*: **assumes** *rel-S₁ cs cm*

and $(cm, cm1) \in step$

shows $\exists rule \ cs1. (cs, cs1) \in st.step \wedge rel-E_0 \ cs1 \ cm \ cm1 \ rule$

proof –

from *step-to- δ step*[*OF assms*(2)] **obtain** *rule* **where** *rstep*: $(cm, cm1) \in \delta step$

rule **by** *auto*

show *?thesis* **using** *assms*(1)

proof (*cases rule*: *rel-S₁.cases*)

case $(1 \ tc' \ q \ tc \ p \ inp \ p')$

note $cs = 1(1)$

note $cm = 1(2)$

have $tc': tc' \ p' \in \Gamma'$ **unfolding** *1(8,4)* *enc.simps* Γ' -*def* *gamma-set-def* **using** *1(5)*

by (*force intro!*: *imageI*)

show *?thesis* **using** *rstep*[*unfolded cm*]

proof (*cases rule*: *δ step.cases*)

case $2: (\delta step \ a \ q1 \ b \ dir \ ts' \ n')$

note $rule = 2(2)$

note $cm1 = 2(1)$

have $(\lambda(q, a, q', b, d), t). (S_1 \ q \ a, t, E_0 \ q' \ b \ d, t, dir.N)) (rule, tc' \ p') \in \delta'$

unfolding δ' -*def* **using** *tc' 2(3)* **by** *blast*

hence *mem*: $(S_1 \ q \ a, tc' \ p', E_0 \ q1 \ b \ dir, tc' \ p', dir.N) \in \delta'$ **by** (*auto simp*:

rule)

have *inp-a*: $inp = a$ **using** *2(4)*[*folded 1(6)*] **by** *auto*

let *?cs1* = *Config_S* $(E_0 \ q1 \ b \ dir) \ tc' \ p'$

have *step*: $(cs, ?cs1) \in st.step$ **unfolding** *cs*

by (*intro st.stepI*[*OF mem*], *insert inp-a*, *auto*)

moreover **have** *rel-E₀* *?cs1 cm cm1 rule* **unfolding** *cm cm1*

by (*intro rel-E₀.intros*[*OF - - - rule*], *insert 1 2 assms rstep*, *auto*)

ultimately **show** *?thesis* **by** *blast*

qed

qed

qed

lemma *rel-S₁-E₀-st-step*: **assumes** *rel-S₁ cs cm*

and $(cs, cs1) \in st.step$

shows \exists *cm1 rule*. $(cm, cm1) \in step \wedge rel-E_0 cs1 cm cm1 rule$
using *assms(1)*
proof (*cases rule: rel-S₁.cases*)
case $(1 tc' q tc p inp p')$
note $cs = 1(1)$
note $cm = 1(2)$
have $tc': tc' p' \in \Gamma'$ **unfolding** $1(8,4)$ *enc.simps* Γ' -*def gamma-set-def* **using**
 $1(5)$
by (*force intro!: imageI*)
show *?thesis using assms(2)[unfolded cs]*
proof (*cases rule: st.step.cases*)
case $2: (step qq bb ddir)$
from $2(2)[unfolded \delta'$ -*def]
have $(S_1 q inp, tc' p', qq, bb, ddir) \in (\lambda((q, a, q', b, d), t). (S_1 q a, t, E_0 q' b$
 $d, t, dir.N)) '(\delta \times \Gamma')$ **by** *auto*
then obtain $q' b dir$ **where** $mem: (q, inp, q', b, dir) \in \delta$ **and** $qq: qq = E_0 q' b$
 dir **and** $ddir: ddir = dir.N$
and $bb: bb = tc' p'$ **by** *auto*
hence $cs1: cs1 = Config_S (E_0 q' b dir) tc' p'$ **unfolding** $2 qq ddir bb$ **by** *auto*
let $?rule = (q, inp, q', b, dir)$
let $?cm1 = Config_M q' (\lambda k. (tc k)(p k := b k)) (\lambda k. go-dir (dir k) (p k))$
have $\delta step: (cm, ?cm1) \in \delta step ?rule$ **unfolding** *cm*
by (*intro* $\delta step.intros[OF refl mem]$, *auto simp: 1(6)*)
from $\delta step$ -*to-step*[*OF this*]
have $(cm, ?cm1) \in step$.
moreover have $rel-E_0 cs1 cm ?cm1 ?rule$ **unfolding** *cm cs1*
by (*intro rel-E₀.intros* $\delta step[unfolded cm]$, *insert 1 2, auto*)
ultimately show *?thesis by blast*
qed
qed*

fun *enc2* :: $('a, 'q, 'k) mt-config \Rightarrow ('a, 'q, 'k) mt-config \Rightarrow nat \Rightarrow nat \Rightarrow ('a, 'k)$
st-tape-symbol
where *enc2* $(Config_M q tc p) (Config_M q1 tc1 p1) p' n = TUPLE (\lambda k. if p k$
 $< p'$
then if $p k = n$ *then* *HAT* $(tc k n)$ *else* *NO-HAT* $(tc k n)$
else if $p1 k = n$ *then* *HAT* $(tc1 k n)$ *else* *NO-HAT* $(tc1 k n)$)

inductive *rel-E* :: $((('a, 'k) st-tape-symbol, ('a, 'q, 'k) st-states) st-config$
 $\Rightarrow ('a, 'q, 'k) mt-config \Rightarrow ('a, 'q, 'k) mt-config \Rightarrow ('a, 'q, 'k) mt-rule \Rightarrow nat \Rightarrow$
bool where
 $tc' 0 = \vdash \Longrightarrow$
 $(\bigwedge i. tc' (Suc i) = enc2 (Config_M q tc p) (Config_M q1 tc1 p1) p' i) \Longrightarrow$
 $valid-config (Config_M q tc p) \Longrightarrow$
 $rule = (q, a, q1, b, d) \Longrightarrow$
 $(Config_M q tc p, Config_M q1 tc1 p1) \in \delta step rule \Longrightarrow$
 $bo = (\lambda k. if p k < p' then SYM (b k) else \cdot) \Longrightarrow$
 $rel-E (Config_S (E q1 bo d) tc' p') (Config_M q tc p) (Config_M q1 tc1 p1) rule p'$

lemma *rel-E₀-E*: **assumes** *rel-E₀ cs cm cm1 rule*
shows $\exists cs1. (cs, cs1) \in st.dstep \wedge rel-E\ cs1\ cm\ cm1\ rule\ (Suc\ (Max\ (range\ (mt-pos\ cm))))$
using *assms(1)*
proof (*cases rule: rel-E₀.cases*)
case (*1 tc' q tc p a q1 b d tc1 p1 p'*)
note *cs = 1(1)*
note *cm = 1(2)*
note *cm1 = 1(3)*
note *rule = 1(7)*
let *?rule = (q, a, q1, b, d)*
have *tc': tc' p' ∈ Γ'* **unfolding** *1(10,5) enc.simps Γ'-def gamma-set-def* **using** *1(6)*
by (*force intro!: imageI*)
have *rmem: ?rule ∈ δ* **using** *1(8,7)* **by** (*cases rule: δstep.cases, auto*)
note *elem = δ[OF this]*
have $((q1, b, d), tc' p') \in (Q \times (UNIV \rightarrow \Gamma) \times UNIV) \times \Gamma'$
using *elem tc' by auto*
hence $(\lambda((q, a, d), t). (E_0\ q\ a\ d,\ t,\ E\ q\ (SYM \circ a)\ d,\ t,\ dir.N)) ((q1, b, d), tc' p') \in \delta'$
unfolding *δ'-def by blast*
hence *mem: (E₀ q1 b d, tc' p', E q1 (SYM ∘ b) d, tc' p', dir.N) ∈ δ'* **by** *simp*
have *Max: Suc (Max (range (mt-pos cm))) = p'* **unfolding** *cm using 1 by auto*
let *?cs1 = Config_S (E q1 (SYM ∘ b) d) tc' p'*
have $(cs, ?cs1) \in st.dstep$ **unfolding** *cs*
by (*intro st.dstepI[OF mem] refl, auto simp: δ'-def*)
moreover **have** *rel-E ?cs1 cm cm1 rule (Suc (Max (range (mt-pos cm))))*
unfolding *Max unfolding cm cm1 rule*
by (*intro rel-E.intros, insert 1, auto*)
ultimately show *?thesis by blast*
qed

lemma *rel-E-S₀*: **assumes** *rel-E cs cm cm1 rule 0*
shows $\exists cs1. (cs, cs1) \in st.dstep \wedge rel-S_0\ cs1\ cm1$
using *assms(1)*
proof (*cases rule: rel-E.cases*)
case (*1 tc' q tc p q1 tc1 p1 a b d bo*)
note *cs = 1(1)*
note *cm = 1(2)*
note *cm1 = 1(3)*
from *valid-step[OF δstep-to-step[OF 1(8)] 1(6)]*
have *valid: valid-config (Config_M q1 tc1 p1) .*
from *valid* **have** *q1: q1 ∈ Q by auto*
hence *mem: (E q1 (λ -. ·) d, ⊢, S₀ q1, ⊢, dir.N) ∈ δ'* **unfolding** *δ'-def by auto*
let *?cs1 = Config_S (S₀ q1) tc' 0*
have $(cs, ?cs1) \in st.dstep$ **unfolding** *cs*
by (*intro st.dstepI[OF mem], insert 1, auto simp: δ'-def*)
moreover **have** *rel-S₀ ?cs1 cm1 unfolding cm1*
by (*intro rel-S₀.intros valid, insert 1, auto*)

ultimately show *?thesis* **by** *blast*
qed

lemma *dsteps-to-steps*: $a \in st.dstep \overset{\sim}{\sim} n \implies a \in st.step \overset{\sim}{\sim} n$
using *relpow-mono*[*OF st.dstep-step*] **by** *auto*

lemma *δ' -mem*: **assumes** $tup \in A$
and $f \cdot A \subseteq \delta'$
shows $f \cdot tup \in \delta'$
using *assms* **by** *auto*

lemma *rel-E-E*: **assumes** *rel-E cs cm cm1 rule (Suc p')*
shows $\exists cs1. (cs, cs1) \in st.dstep \overset{\sim}{\sim} 4 \wedge rel-E \ cs1 \ cm \ cm1 \ rule \ p'$
using *assms*

proof(*cases rule: rel-E.cases*)

case $(1 \ tc' \ q \ tc \ p \ q1 \ tc1 \ p1 \ a \ b \ d \ bo)$

let $?rule = (q, a, q1, b, d)$

have *valid-next*: *valid-config*(*Config_M q1 tc1 p1*)

using $1(8,6)$ *δ step-to-step valid-step* **by** *blast*

have *trans*: $(\lambda(q, ys, ds, t). (E \ q \ ys \ ds, \ TUPLE \ t, \ Er \ q \ (update-ys \ t \ ys) \ ds$
(compute-idx-set t ys),

$TUPLE \ (replace-sym \ t \ ys), \ dir.R)) \cdot (Q \times \ func-set \times \ UNIV \times$
gamma-set) $\subseteq \delta'$

$(\lambda(q, ys, ds, I, t). (Er \ q \ ys \ ds \ I, \ TUPLE \ t, \ Em \ q \ ys \ ds \ I, \ TUPLE \ (place-hats-R$
t ds I), dir.L))

$\cdot (Q \times \ func-set \times \ UNIV \times \ UNIV \times \ gamma-set) \subseteq \delta'$

$(\lambda(q, ys, ds, I, t). (Em \ q \ ys \ ds \ I, \ TUPLE \ t, \ El \ q \ ys \ ds \ I, \ TUPLE \ (place-hats-M$
t ds I), dir.L))

$\cdot (Q \times \ func-set \times \ UNIV \times \ UNIV \times \ gamma-set) \subseteq \delta'$

$(\lambda(q, ys, ds, I). (El \ q \ ys \ ds \ I, \vdash, \ E \ q \ ys \ ds, \vdash, \ dir.N))$

$\cdot (Q \times \ func-set \times \ UNIV \times \ Pow \ UNIV) \subseteq \delta'$

$(\lambda(q, ys, ds, I, t). (El \ q \ ys \ ds \ I, \ TUPLE \ t, \ E \ q \ ys \ ds, \ TUPLE \ (place-hats-L \ t \ ds$
I), dir.N))

$\cdot (Q \times \ func-set \times \ UNIV \times \ UNIV \times \ gamma-set) \subseteq \delta'$

unfolding *δ' -def*

by(*blast, blast, blast, blast, blast*)

obtain *tup where* $tup: tc' (Suc \ p') = TUPLE \ tup$

unfolding $1(5)$ *Γ' -def gamma-set-def enc2.simps* **by** *auto*

have *tup-mem*: $tup \in gamma-set$

using *tup* $1(6)$ *valid-next* **unfolding** *gamma-set-def* $1(5)$ *enc2.simps valid-config.simps*

by (*smt* (*z3*) *Pi-iff UnI1 UnI2 image-iff range-subsetD st-tape-symbol.simps*(1))

have *bo-set*: $bo \in func-set$

using $1 \ \delta(4)$ *δ step.simps* **unfolding** *func-set-def* **by** *fastforce*

have *rmem*: $?rule \in \delta$ **using** $1(8,7)$ **by** (*cases rule: δ step.cases, auto*)

note *elem* = δ [*OF this*]

let *?rep-tup* = $TUPLE \ (replace-sym \ tup \ bo)$

let *?tc1* = $(tc' (Suc \ p') := ?rep-tup)$

let *?I* = *compute-idx-set tup bo*

let $?ys' = \text{update-ys } \text{tup } \text{bo}$
let $?er = \text{Er } q1 \ ?ys' \ d \ ?I$
let $?cs1 = \text{Configs } ?er \ ?tc1 \ (\text{Suc}(\text{Suc } p'))$
have $(q1, \text{bo}, d, \text{tup}) \in Q \times \text{func-set} \times \text{UNIV} \times \text{gamma-set}$ **using** $\text{elem } \text{tup-mem}$
bo-set **by** blast
hence $\text{mem}: (E \ q1 \ \text{bo } d, \ \text{tc}'(\text{Suc } p'), \ ?er, \ ?rep\text{-tup}, \ \text{dir}.R) \in \delta'$
using $\delta'\text{-mem}[\text{OF} - \text{trans}(1)]$ **unfolding** $\text{split } \text{tup}$ **by** fast
note $\text{dstep-help} = \text{st.dstep}[\text{of } E \ q1 \ \text{bo } d \ \text{tc}'(\text{Suc } p') \ ?er \ ?rep\text{-tup} \ \text{dir}.R]$
have $\text{to-r}: (\text{Configs } (E \ q1 \ \text{bo } d) \ \text{tc}'(\text{Suc } p'), \ ?cs1) \in \text{st.dstep}$
using $\text{dstep-help}[\text{OF } \text{mem}]$ **unfolding** $\delta'\text{-def } \text{go-dir.simps } \text{tup}$ **by** fast

obtain tup2 **where** $\text{tup2}: \text{tc}'(\text{Suc}(\text{Suc } p')) = \text{TUPLE } \text{tup2}$
unfolding $1(5)$ $\Gamma'\text{-def } \text{gamma-set-def } \text{enc2.simps}$ **by** auto
have $\text{tup2-mem}: \text{tup2} \in \text{gamma-set}$
using $\text{tup2 } 1(6)$ valid-next
unfolding $\text{gamma-set-def } 1(5)$ $\text{enc2.simps } \text{valid-config.simps}$
by $(\text{smt } (z3) \ \text{Pi-iff } \text{UnI1 } \text{UnI2 } \text{imageI } \text{range-subsetD } \text{st-tape-symbol.inject}(1))$
let $?em = \text{Em } q1 \ ?ys' \ d \ ?I$
let $?tc2 = (\text{tc}'(\text{Suc}(\text{Suc } p')) := \text{TUPLE } (\text{place-hats-R } \text{tup2 } d \ ?I))$
let $?cs2 = \text{Configs } ?em \ ?tc2 \ (\text{Suc } p')$
have $(q1, ?ys', d, ?I, \text{tup2}) \in Q \times \text{func-set} \times \text{UNIV} \times \text{UNIV} \times \text{gamma-set}$
using $\text{update-ys-range}[\text{OF } \text{tup-mem } \text{bo-set}]$ $\text{elem } \text{tup2-mem}$ **by** blast
hence $\text{mem2}: (?er, \ ?tc1(\text{Suc}(\text{Suc } p')), \ \text{Em } q1 \ ?ys' \ d \ ?I, \ \text{TUPLE } (\text{place-hats-R } \text{tup2 } d \ ?I), \ \text{dir}.L) \in \delta'$
using $\delta'\text{-mem}[\text{OF} - \text{trans}(2)]$ tup2 **unfolding** split **by** auto
note $\text{dstep-help2} = \text{st.dstep}[\text{of } ?er \ ?tc1 \ \text{Suc}(\text{Suc } p') \ ?em \ \text{TUPLE } (\text{place-hats-R } \text{tup2 } d \ ?I) \ \text{dir}.L]$
have $\text{to-m}: (?cs1, \ ?cs2) \in \text{st.dstep}$
using $\text{dstep-help2}[\text{OF } \text{mem2}]$ tup2 **unfolding** $\delta'\text{-def } \text{go-dir.simps}$ **by** fastforce

have $(q1, ?ys', d, ?I, \text{replace-sym } \text{tup } \text{bo}) \in Q \times \text{func-set} \times \text{UNIV} \times \text{UNIV} \times \text{gamma-set}$
using $\text{update-ys-range}[\text{OF } \text{tup-mem } \text{bo-set}]$ $\text{replace-sym-range}[\text{OF } \text{tup-mem } \text{bo-set}]$ $\text{elem}(3)$ **by** blast
hence $\text{mem3}: (?em, \ ?tc2(\text{Suc } p'), \ \text{El } q1 \ ?ys' \ d \ ?I, \ \text{TUPLE } (\text{place-hats-M } (\text{replace-sym } \text{tup } \text{bo}) \ d \ ?I), \ \text{dir}.L) \in \delta'$
using $\delta'\text{-mem}[\text{OF} - \text{trans}(3)]$ **by** auto
note $\text{dstep-help3} = \text{st.dstep}[\text{of } ?em \ ?tc2 \ \text{Suc } p' \ \text{El } q1 \ ?ys' \ d \ ?I \ \text{TUPLE } (\text{place-hats-M } (\text{replace-sym } \text{tup } \text{bo}) \ d \ ?I) \ \text{dir}.L]$
let $?el = \text{El } q1 \ (\text{update-ys } \text{tup } \text{bo}) \ d \ (\text{compute-idx-set } \text{tup } \text{bo})$
let $?tc3 = \text{tc}'(\text{Suc } p') := \text{TUPLE } (\text{replace-sym } \text{tup } \text{bo}),$
 $\text{Suc } (\text{Suc } p') := \text{TUPLE } (\text{place-hats-R } \text{tup2 } d \ (\text{compute-idx-set } \text{tup } \text{bo}),$
 $\text{Suc } p' := \text{TUPLE } (\text{place-hats-M } (\text{replace-sym } \text{tup } \text{bo}) \ d \ (\text{compute-idx-set } \text{tup } \text{bo}))$
let $?cs3 = \text{Configs } ?el \ ?tc3 \ p'$
have $\text{to-l}: (?cs2, \ ?cs3) \in \text{st.dstep}$
using $\text{dstep-help3}[\text{OF } \text{mem3}]$ **unfolding** $\delta'\text{-def } \text{go-dir.simps}$ **by** fastforce

have $steps3: (cs, ?cs3) \in st.dstep \overset{\sim}{\sim} 3$ **unfolding** *numeral-3-eq-3* **using** *to-l to-m to-r 1(1)* **by** *auto*

have $tc1-def: \bigwedge k. tc1\ k = (tc\ k)(p\ k := b\ k)$

using *1(8) unfolding 1(7) by (simp add: $\delta step.simps\ old.prod.inject$)*

have $p1-def: \bigwedge k. p1\ k = go-dir\ (d\ k)\ (p\ k)$

using *1(8) unfolding 1(7) by (simp add: $\delta step.simps\ old.prod.inject$)*

have $not-I-mem-current-pos: \forall k'. k' \notin ?I \longrightarrow p\ k' \neq p'$

by(*intro allI impI, insert tup elem 1(6), fastforce simp: 1 compute-idx-set-def*)

have $I-mem-current-pos: \forall k. k \in ?I \longrightarrow p\ k = p'$

using *compute-idx-set-def 1(5,9) tup by (simp, fastforce)*

have $I-mem-eq-cur-pos: \forall k. k \in ?I \longleftrightarrow p\ k = p'$

using *I-mem-current-pos not-I-mem-current-pos by auto*

show *?thesis*

proof(*cases p'*)

case *p-zero: 0*

hence $tc3-tup: ?tc3\ p' = \vdash$ **using** *1(4) by simp*

have $(q1, ?ys', d, ?I) \in Q \times func-set \times UNIV \times UNIV$

using *update-ys-range[OF tup-mem bo-set] replace-sym-range[OF tup-mem bo-set] elem(3) by blast*

hence $mem4: (El\ q1\ ?ys'\ d\ ?I, ?tc3\ p', E\ q1\ ?ys'\ d, \vdash, dir.N) \in \delta'$

using *δ' -mem[OF - trans(4)] unfolding tc3-tup by auto*

let $?tc4 = ?tc3(p' := \vdash)$

let $?cs4 = Config_S\ (E\ q1\ ?ys'\ d)\ ?tc4\ p'$

note $dstep-help4 = st.dstep[of\ El\ q1\ ?ys'\ d\ ?I\ ?tc3\ p'\ E\ q1\ ?ys'\ d\ \vdash\ dir.N]$

have $(?cs3, ?cs4) \in st.dstep \overset{\sim}{\sim} 1$

using *dstep-help4[OF mem4] unfolding δ' -def go-dir.simps relpow-1 tc3-tup*

by *fastforce*

hence $steps: (cs, ?cs4) \in st.dstep \overset{\sim}{\sim} 4$

using *relpow-transI[OF steps3] by fastforce*

note *intro-helper = rel-E.intros[of ?tc4 q tc p q1 tc1 p1 p' rule a b d update-ys tup bo]*

have $valid-subs: (\forall i. (tc'(Suc\ p' := TUPLE\ (replace-sym\ tup\ bo),$

$Suc\ (Suc\ p') := TUPLE\ (place-hats-R\ tup2\ d\ (compute-idx-set\ tup\ bo)),$

$Suc\ p' := TUPLE\ (place-hats-M\ (replace-sym\ tup\ bo)\ d\ (compute-idx-set$

$tup\ bo)),$

$p' := \vdash))\ (Suc\ i) = enc2\ (Config_M\ q\ tc\ p)\ (Config_M\ q1\ tc1\ p1)\ p'\ i)$

proof

fix *i*

consider $(zer)\ i = 0 \mid (suc)\ i = Suc\ 0 \mid (ge-one)\ i > Suc\ 0$ **by** *linarith*

then show $(tc'(Suc\ p' := TUPLE\ (replace-sym\ tup\ bo),$

$Suc\ (Suc\ p') := TUPLE\ (place-hats-R\ tup2\ d\ (compute-idx-set\ tup\ bo)),$

$Suc\ p' := TUPLE\ (place-hats-M\ (replace-sym\ tup\ bo)\ d\ (compute-idx-set$

$tup\ bo)),$

$p' := \vdash))\ (Suc\ i) = enc2\ (Config_M\ q\ tc\ p)\ (Config_M\ q1\ tc1\ p1)\ p'\ i)$

proof(*cases*)

case *zer*

have *(case replace-sym tup bo k of*

```

      NO-HAT a  $\Rightarrow$  if  $k \in \text{compute-idx-set tup bo} \wedge d k = \text{dir.N}$  then HAT a else
NO-HAT a
      | HAT x  $\Rightarrow$  HAT x) = (if  $p1 k = 0$  then HAT (tc1 k 0) else NO-HAT (tc1
k 0)) for k
      proof(cases  $k \in ?I$ )
      case  $k\text{-in-}I$ : True
      then obtain x where rep-no-hat: replace-sym tup bo k = NO-HAT x
      unfolding replace-sym-def compute-idx-set-def by auto
      have  $pk\text{-zero}$ :  $p k = 0$ 
      using  $k\text{-in-}I$  less-Suc0  $p\text{-zero}$  unfolding compute-idx-set-def 1(9) by
fastforce
      show ?thesis
      proof(cases  $d k = \text{dir.N}$ )
      case False
      moreover have  $tc k (p k) = LE$ 
      using 1(6)  $pk\text{-zero}$  1(8) by auto
      moreover have  $tc k (p k) = a k$ 
      using 1(7,8)  $pk\text{-zero}$  unfolding  $\delta\text{step.simps}$  by blast
      ultimately have  $d k = \text{dir.R}$ 
      using  $\delta LE[OF rmem]$  by auto
      then show ?thesis
      by(insert  $k\text{-in-}I$   $p1\text{-def}$   $tc1\text{-def}$ , simp, insert rep-no-hat, auto simp:
replace-sym-def 1(9)  $p\text{-zero}$   $pk\text{-zero}$ )
      qed(insert  $k\text{-in-}I$  rep-no-hat  $pk\text{-zero}$  1(9), auto simp: replace-sym-def
tc1-def  $p1\text{-def}$ )
      next
      case False
      show ?thesis
      using tup False 1(5)  $p\text{-zero}$  not-I-mem-current-pos  $p1\text{-def}$   $tc1\text{-def}$ 
      unfolding  $p\text{-zero}$  replace-sym-def by fastforce
      qed
      then show ?thesis
      unfolding zer  $p\text{-zero}$  place-hats-M-def place-hats-to-dir-def by simp
next
case suc
have (case tup2 k of
      NO-HAT a  $\Rightarrow$  if  $k \in \text{compute-idx-set tup bo} \wedge d k = \text{dir.R}$  then HAT a else
NO-HAT a
      | HAT x  $\Rightarrow$  HAT x) = (if  $p1 k = \text{Suc } 0$  then HAT (tc1 k (Suc 0)) else
NO-HAT (tc1 k (Suc 0))) for k
      using tup2  $p\text{-zero}$  elem(4) 1(5,6,9)  $gr\text{-zeroI}$  tm(4) tup
      by (auto simp: compute-idx-set-def tc1-def  $p1\text{-def}$ , smt (verit, best) diff-0-eq-0
go-dir.elims not-gr0)
      then show ?thesis unfolding suc  $p\text{-zero}$  place-hats-R-def place-hats-to-dir-def
by simp
next
case  $ge\text{-one}$ 
have (if  $p k = 0$  then if  $p k = i$  then HAT (tc k i) else NO-HAT (tc k i)
      else if  $p1 k = i$  then HAT (tc1 k i) else NO-HAT (tc1 k i)) = (if  $p1 k = i$ 

```

```

then HAT (tc1 k i) else NO-HAT (tc1 k i) for k
  using insert ge-one p1-def tc1-def
  by (smt (verit, ccfv-threshold) diff-0-eq-0 fun-upd-other go-dir.elims
less-irrefl-nat less-nat-zero-code)
  then show ?thesis using ge-one p-zero 1(5)[of i] enc2.simps by simp
qed
qed
have only-hats:  $\bigwedge k. p\ k = 0 \longrightarrow \text{tup}\ k \in \text{HAT}\ \Gamma$ 
  using tup unfolding p-zero 1(5)[of 0] enc2.simps by (simp, meson imageI
tup-hat-content tup-mem)
  have (if  $k \in \text{compute-idx-set}\ \text{tup}\ \text{bo}$  then  $\cdot$  else  $\text{bo}\ k$ ) =  $\cdot$  for k
  using only-hats elem(4) 1(9) by (auto simp: compute-idx-set-def p-zero)
  hence replaced-all:  $\text{update-ys}\ \text{tup}\ \text{bo} = (\lambda k. \text{if}\ p\ k < p'$  then  $\text{SYM}\ (b\ k)$  else  $\cdot)$ 
  unfolding update-ys-def p-zero by auto
  have invs:  $\text{rel-E}\ ?cs4\ \text{cm}\ \text{cm1}\ \text{rule}\ p'$ 
  using valid-subs p-zero intro-helper[OF - - 1(6) 1(7) 1(8) replaced-all]
1(2,3,7) by auto
  then show ?thesis
  by(insert steps invs, intro exI conjI, auto)
next
case (Suc nat)
obtain tup3 where tc3-p':  $?tc3\ p' = \text{TUPLE}\ \text{tup3}$  and
  tup3-def:  $\text{tup3} = (\lambda k. \text{if}\ p\ k < \text{Suc}\ (\text{Suc}\ \text{nat})$  then  $\text{if}\ p\ k = \text{nat}$  then  $\text{HAT}\ (tc\ k\ \text{nat})$  else  $\text{NO-HAT}\ (tc\ k\ \text{nat})$ 
  else  $\text{if}\ p1\ k = \text{nat}$  then  $\text{HAT}\ (tc1\ k\ \text{nat})$  else  $\text{NO-HAT}\ (tc1\ k\ \text{nat}))$ 
  using 1(5)[of nat] unfolding Suc enc2.simps by simp
  have tup3-mem:  $\text{tup3} \in \text{gamma-set}$ 
  using tup3-def 1(6) valid-next unfolding gamma-set-def 1(5) enc2.simps
valid-config.simps
  by (smt (z3) Pi-I UnI1 UnI2 imageI range-subsetD st-tape-symbol.inject(1))
  let ?a4 =  $\text{TUPLE}\ (\text{place-hats-L}\ \text{tup3}\ d\ (\text{compute-idx-set}\ \text{tup}\ \text{bo}))$ 
  let ?tc4 =  $?tc3(p' := ?a4)$ 
  let ?cs4 =  $\text{Config}_S\ (E\ q1\ ?ys'\ d)\ ?tc4\ p'$ 
  have step-mem:  $(q1, ?ys', d, ?I, \text{tup3}) \in Q \times \text{func-set} \times \text{UNIV} \times \text{UNIV} \times$ 
gamma-set
  using update-ys-range[OF tup-mem bo-set] replace-sym-range[OF tup-mem
bo-set] elem(3) tup3-mem by blast
  have mem5:  $(E\ q1\ ?ys'\ d\ ?I, ?tc3\ p', E\ q1\ ?ys'\ d, \text{TUPLE}\ (\text{place-hats-L}\ \text{tup3}\ d\ (\text{compute-idx-set}\ \text{tup}\ \text{bo})), \text{dir.N}) \in \delta'$ 
  using  $\delta'$ -mem[OF step-mem trans(5)] unfolding split tc3-p' .
  note dstep-help5 =  $\text{st.dstep}[of\ E\ q1\ ?ys'\ d\ ?I\ ?tc3\ p'\ E\ q1\ ?ys'\ d\ ?a4\ \text{dir.N}]$ 
  note intros-helper2 =  $\text{rel-E.intros}[of\ ?tc4\ q\ tc\ p\ q1\ tc1\ p1\ p'\ \text{rule}\ a\ b\ d\ \text{update-ys}\ \text{tup}\ \text{bo}]$ 
  have correct-shift:  $(tc'(Suc\ p' := \text{TUPLE}\ (\text{replace-sym}\ \text{tup}\ \text{bo}),$ 
 $\text{Suc}\ (\text{Suc}\ p') := \text{TUPLE}\ (\text{place-hats-R}\ \text{tup2}\ d\ (\text{compute-idx-set}\ \text{tup}\ \text{bo})),$ 
 $\text{Suc}\ p' := \text{TUPLE}\ (\text{place-hats-M}\ (\text{replace-sym}\ \text{tup}\ \text{bo})\ d\ (\text{compute-idx-set}\ \text{tup}\ \text{bo})),$ 
 $p' := \text{TUPLE}\ (\text{place-hats-L}\ \text{tup3}\ d\ (\text{compute-idx-set}\ \text{tup}\ \text{bo}))))$ 
 $(\text{Suc}\ i) = \text{enc2}\ (\text{Config}_M\ q\ tc\ p)\ (\text{Config}_M\ q1\ tc1\ p1)\ p'\ i$  for i

```

proof –

consider *(one)* $Suc\ i = Suc\ nat$
| *(two)* $Suc\ i = Suc(Suc\ nat)$
| *(three)* $Suc\ i = Suc(Suc(Suc\ nat))$
| *(else)* $Suc\ i \notin \{Suc\ nat, Suc(Suc\ nat), Suc(Suc(Suc\ nat))\}$ **by** *blast*

then show *?thesis*

proof cases

case one
have *(case tup3 k of*
 $NO-HAT\ a \Rightarrow if\ k \in compute-idx-set\ tup\ bo \wedge d\ k = dir.L\ then\ HAT\ a$
else NO-HAT a
| $HAT\ x \Rightarrow HAT\ x) = (if\ p\ k < Suc\ nat\ then\ if\ p\ k = i\ then\ HAT\ (tc\ k\ i)$
else NO-HAT (tc k i)
 $else\ if\ p1\ k = i\ then\ HAT\ (tc1\ k\ i)\ else\ NO-HAT\ (tc1\ k\ i))$ **for** k
using *one I-mem-eq-cur-pos Suc nat-less-le*
by *(cases d k, auto simp: tc1-def p1-def tup3-def compute-idx-set-def)*
then show *?thesis*
using one unfolding $Suc\ place-hats-L-def\ place-hats-to-dir-def$ **by** *auto*

next

case two
have *(case replace-sym tup bo k of*
 $NO-HAT\ a \Rightarrow if\ k \in compute-idx-set\ tup\ bo \wedge d\ k = dir.N\ then\ HAT$
a else NO-HAT a
| $HAT\ x \Rightarrow HAT\ x) = (if\ p\ k < Suc\ nat\ then\ if\ p\ k = i\ then\ HAT$
 $(tc\ k\ i)\ else\ NO-HAT\ (tc\ k\ i)$
 $else\ if\ p1\ k = i\ then\ HAT\ (tc1\ k\ i)\ else\ NO-HAT\ (tc1\ k\ i))$ **for** k
using *two 1(5,9) Suc tup elem(4) not-I-mem-current-pos*
by *(cases d k, auto simp: replace-sym-def compute-idx-set-def p1-def tc1-def)*
then show *?thesis*
unfolding $two\ Suc\ enc2.simps\ place-hats-M-def\ place-hats-to-dir-def$ **by**
simp

next

case three
have *(case tup2 k of*
 $NO-HAT\ a \Rightarrow if\ k \in compute-idx-set\ tup\ bo \wedge d\ k = dir.R\ then\ HAT$
a else NO-HAT a
| $HAT\ x \Rightarrow HAT\ x) = (if\ p\ k < Suc\ nat\ then\ if\ p\ k = i\ then\ HAT$
 $(tc\ k\ i)\ else\ NO-HAT\ (tc\ k\ i)$
 $else\ if\ p1\ k = i\ then\ HAT\ (tc1\ k\ i)\ else\ NO-HAT\ (tc1\ k\ i))$ **for** k
using *three p1-def tc1-def compute-idx-set-def tup2 Suc 1(9) elem(4)*
I-mem-eq-cur-pos less-SucE
unfolding $1(5)\ enc2.simps$ **by** *(cases d k, auto)*
then show *?thesis*
unfolding $three\ Suc\ enc2.simps\ place-hats-R-def\ place-hats-to-dir-def$ **by**
simp

next

case else
have *(if p k < Suc (Suc nat) then if p k = i then HAT (tc k i) else NO-HAT*
 $(tc\ k\ i)$

```

    else if p1 k = i then HAT (tc1 k i) else NO-HAT (tc1 k i) =
    (if p k < Suc nat then if p k = i then HAT (tc k i) else NO-HAT (tc k i)
    else if p1 k = i then HAT (tc1 k i) else NO-HAT (tc1 k i)) for k
    unfolding 1(5) enc2.simps Suc
    using else p1-def tc1-def
    by (cases d k) auto
  then show ?thesis
    using else Suc 1(5) enc2.simps by auto
qed
qed
have correct-replace: update-ys tup bo = (λk. if p k < p' then SYM (b k) else
.)
  by(insert I-mem-eq-cur-pos 1(9), auto simp: Suc update-ys-def)
  have step: (?cs3, ?cs4) ∈ st.dstep  $\overset{\sim}{\sim}$  1
    using mem5 dstep-help5[OF mem5]
    unfolding δ'-def go-dir.simps relpow-1 Suc by fastforce
  hence (cs, ?cs4) ∈ st.dstep  $\overset{\sim}{\sim}$  4
    using relpow-transI[OF steps3 step] by simp
  moreover have rel-E ?cs4 cm cm1 rule p'
    using Suc 1 intros-helper2[OF - - 1(6) 1(7) 1(8) correct-replace] correct-shift
  by simp
  ultimately show ?thesis by blast
qed
qed

lemma E-phase: assumes rel-E0 cs cm cm1 rule
  shows ∃ cs'. (cs, cs') ∈ st.dstep  $\overset{\sim}{\sim}$  (6 + 4 * Max (range (mt-pos cm))) ∧ rel-S0
  cs' cm1
proof -
  from rel-E0-E[OF assms] obtain n cs1 where step1: (cs, cs1) ∈ st.dstep
    and n: n = Suc (Max (range (mt-pos cm)))
    and rel: rel-E cs1 cm cm1 rule n
  by auto
  from rel have ∃ cs2. (cs1, cs2) ∈ st.dstep  $\overset{\sim}{\sim}$  (4 * n) ∧ rel-E cs2 cm cm1 rule 0
  proof (induction n arbitrary: cs1 rule: nat-induct)
    case (Suc n)
    from rel-E-E[OF Suc.prem] obtain cs' where
      step4: (cs1, cs') ∈ st.dstep  $\overset{\sim}{\sim}$  4 and rel: rel-E cs' cm cm1 rule n by auto
    from Suc.IH[OF rel] obtain cs2 where
      steps: (cs', cs2) ∈ st.dstep  $\overset{\sim}{\sim}$  (4 * n) and rel: rel-E cs2 cm cm1 rule 0
    by auto
    from relpow-transI[OF step4 steps] rel show ?case by auto
  qed auto
  then obtain cs2 where steps2: (cs1, cs2) ∈ st.dstep  $\overset{\sim}{\sim}$  (4 * n) and rel: rel-E
  cs2 cm cm1 rule 0 by auto
  from rel-E-S0[OF rel] obtain cs' where step3: (cs2, cs') ∈ st.dstep and rel:
  rel-S0 cs' cm1 by auto
  from relpow-Suc-I2[OF step1 relpow-Suc-I[OF steps2 step3]]
  have (cs, cs') ∈ st.dstep  $\overset{\sim}{\sim}$  Suc (Suc (4 * n)) by simp

```


also have $Suc (Suc (4 * n)) = 6 + 4 * Max (range (mt-pos cm))$ **unfolding** n
by simp
finally show $?thesis$ **using** rel **by** $auto$
qed

6.2.4 Simulation of multitape TM by singletape TM

lemma *step-simulation*: **assumes** $rel-S_0 cs cm$
and $(cm, cm') \in step$
shows $\exists cs'. (cs, cs') \in st.step \rightsquigarrow (10 + 5 * Max (range (mt-pos cm))) \wedge rel-S_0 cs' cm'$
proof –
let $?n = Max (range (mt-pos cm))$
from $assms(2)$ **have** $mt-state cm \notin \{t, r\}$ **using** $\delta-set$
by $(cases, auto)$
from $S-phase[OF assms(1) this]$ **obtain** $cs1$ **where**
 $steps1: (cs, cs1) \in st.dstep \rightsquigarrow (3 + ?n)$ **and** $rel: rel-S_1 cs1 cm$
by $auto$
from $rel-S_1-E_0-step[OF rel assms(2)]$ **obtain** $r cs2$ **where**
 $step2: (cs1, cs2) \in st.step$ **and** $rel: rel-E_0 cs2 cm cm' r$
by $auto$
from $E-phase[OF rel]$ **obtain** cs' **where**
 $steps3: (cs2, cs') \in st.dstep \rightsquigarrow (6 + 4 * ?n)$ **and** $rel: rel-S_0 cs' cm'$
by $auto$
from $relpow-transI[OF dsteps-to-steps[OF steps1] relpow-Suc-I2[OF step2 dsteps-to-steps[OF steps3]]]$
have $(cs, cs') \in st.step \rightsquigarrow ((3 + ?n) + Suc (6 + 4 * ?n))$ **by** $simp$
also have $((3 + ?n) + Suc (6 + 4 * ?n)) = 10 + 5 * ?n$ **by** $simp$
finally show $?thesis$ **using** rel **by** $auto$
qed

lemma *steps-simulation-main*: **assumes** $rel-S_0 cs cm$
and $Max (range (mt-pos cm)) \leq N$
and $(cm, cm') \in step \rightsquigarrow n$
shows $\exists m cs'. (cs, cs') \in st.step \rightsquigarrow m \wedge rel-S_0 cs' cm' \wedge m \leq sum (\lambda i. 10 + 5 * (N + i)) \{..< n\} \wedge Max (range (mt-pos cm')) \leq N + n$
using $assms(3,1,2)$
proof (*induct n arbitrary: cm' N*)
case 0
show $?case$
by $(intro exI[of - 0] exI[of - cs], insert 0, auto)$
next
case $(Suc n cm' N)$
from $Suc(2)$ **obtain** cm'' **where** $(cm, cm'') \in step \rightsquigarrow n$ **and** $step: (cm'', cm') \in step$ **by** $auto$
from $Suc(1)[OF this(1) Suc(3-4)]$ **obtain** $m cs''$ **where**
 $steps: (cs, cs'') \in st.step \rightsquigarrow m$ **and** $m: m \leq (\sum i < n. 10 + 5 * (N + i))$ **and**
 $rel: rel-S_0 cs'' cm''$ **and** $max: Max (range (mt-pos cm'')) \leq N + n$
by $auto$

from *step-simulation*[*OF rel step*] **obtain** *cs'* **where**
steps2: $(cs'', cs') \in st.step \sim (10 + 5 * Max (range (mt-pos cm'')))$ **and** *rel*:
rel-S₀ cs' cm' **by** *auto*
let *?m* = $m + (10 + 5 * Max (range (mt-pos cm'')))$
from *relpow-transI*[*OF steps steps2*]
have *steps*: $(cs, cs') \in st.step \sim ?m$ **by** *auto*
show *?case*
proof (*intro exI conjI, rule steps, rule rel*)
from *max-mt-pos-step*[*OF step*] *max*
show $Max (range (mt-pos cm')) \leq N + Suc n$ **by** *linarith*
have *id*: $\{..<Suc n\} = insert n \{..<n\}$ **by** *auto*
have $?m \leq m + (10 + 5 * (N + n))$ **using** *max* **by** *presburger*
also have $\dots \leq (\sum i < Suc n. 10 + 5 * (N + i))$ **using** *m unfolding id* **by**
auto
finally show $?m \leq (\sum i < Suc n. 10 + 5 * (N + i))$ **by** *auto*
qed
qed

lemma *steps-simulation-rel-S₀*: **assumes** *rel-S₀ cs (init-config w)*
and $(init-config w, cm') \in step \sim n$
shows $\exists m cs'. (cs, cs') \in st.step \sim m \wedge rel-S_0 cs' cm' \wedge m \leq 3 * n^2 + 7 * n$
proof –
from *steps-simulation-main*[*OF assms(1) - assms(2), unfolded max-mt-pos-init, OF le-refl*]
obtain *m cs'* **where** *steps*: $(cs, cs') \in st.step \sim m$ **and** *rel*: *rel-S₀ cs' cm'* **and**
m: $m \leq (\sum i < n. 10 + 5 * i)$
by *auto*
have $m \leq (\sum i < n. 10 + 5 * i)$ **by** *fact*
also have $\dots \leq 3 * n^2 + 7 * n$ **using** *aux-sum-formula* .
finally show *?thesis* **using** *steps rel* **by** *auto*
qed

lemma *simulation-with-complexity*: **assumes** *w*: $set w \subseteq \Sigma$
and *steps*: $(init-config w, Config_M q mtape p) \in step \sim n$
shows $\exists stape k. (st.init-config (map INP w), Config_S (S_0 q) stape 0) \in st.step \sim k$
 $\wedge k \leq 2 * length w + 3 * n^2 + 7 * n + 3$
proof –
let *?INP* = *INP* :: '*a* \Rightarrow ('*a*, '*k*) *st-tape-symbol*
let *?initm* = *init-config w*
define *x* **where** *x* = *map ?INP w*
from *R-phase*[*OF w, folded x-def*] **obtain** *cs* **where**
steps1: $(st.init-config x, cs) \in st.dstep \sim (3 + 2 * length w)$ **and** *rel*: *rel-S₀*
cs ?initm
by *auto*
from *steps-simulation-rel-S₀*[*of - w, OF rel steps*] **obtain** *k' cs'* **where**
steps2: $(cs, cs') \in st.step \sim k'$ **and**
rel: *rel-S₀ cs' (Config_M q mtape p)* **and**
k': $k' \leq 3 * n^2 + 7 * n$ **by** *auto*
let *?k* = $3 + 2 * length w + k'$

```

from relpow-transI[OF dsteps-to-steps[OF steps1] steps2]
have steps: (st.init-config x, cs') ∈ st.step  $\widehat{\sim}$ ?k .
from rel obtain stape where cs': cs' = ConfigS (S0 q) stape 0
  by (cases, auto)
show ?thesis
  by (intro exI[of - stape] exI[of - ?k], insert steps cs' k', auto simp: x-def)
qed

```

lemma simulation: map INP ' Lang ⊆ st.Lang

proof

```

fix x :: ('a, 'k) st-tape-symbol list
assume x ∈ map INP ' Lang
then obtain w where mem: w ∈ Lang and x: x = map INP w by auto
define cm where cm = init-config w
from mem[unfolded Lang-def] obtain w' n where w: set w ⊆ Σ and steps: (cm,
ConfigM t w' n) ∈ step  $\widehat{\sim}$ * by (auto simp: cm-def)
from rtrancl-imp-relpow[OF steps] obtain num where steps: (cm, ConfigM t w'
n) ∈ step  $\widehat{\sim}$ num by auto
from simulation-with-complexity[OF w, folded cm-def, OF steps] obtain stape
where steps: (st.init-config (map INP w), ConfigS (S0 t) stape 0) ∈ st.step  $\widehat{\sim}$ *
using relpow-imp-rtrancl by blast
show x ∈ st.Lang using steps w unfolding x st.Lang-def by auto
qed

```

6.2.5 Simulation of singletape TM by multitape TM

lemma rev-simulation: st.Lang ⊆ map INP ' Lang

proof

```

fix x :: ('a, 'k) st-tape-symbol list
let ?INP = INP :: 'a ⇒ ('a, 'k) st-tape-symbol
assume x ∈ st.Lang
from this[unfolded st.Lang-def] obtain ts p where x: set x ⊆ ?INP ' Σ
and steps: (st.init-config x, ConfigS (S0 t) ts p) ∈ st.step  $\widehat{\sim}$ * by force
let ?NF = ConfigS (S0 t) ts p
have NF: ¬ (∃ c. (?NF, c) ∈ st.step)
proof
  assume ∃ c. (?NF, c) ∈ st.step
  then obtain c where (?NF, c) ∈ st.step by auto
  thus False
  by (cases rule: st.step.cases, insert st.δ-set, auto)
qed
from INP-D[OF x] obtain w where x: x = map ?INP w and w: set w ⊆ Σ by
auto
define cm where cm = init-config w
from R-phase[OF w, folded x] obtain cs where
  dsteps: (st.init-config x, cs) ∈ st.dstep  $\widehat{\sim}$  (3 + 2 * length w) and rel: rel-S0
cs cm
by (auto simp: cm-def)

```

from *steps* **obtain** k **where** $(st.init-config\ x, ?NF) \in st.step \hat{\sim} k$ **using** *rtrancl-power*
by *blast*
from *st.dsteps-inj*[*OF dsteps this NF*] **obtain** n **where** $ssteps: (cs, ?NF) \in st.step \hat{\sim} n$ **by** *auto*
from *rel ssteps* **have** $\exists cm'. (cm, cm') \in step \hat{*} \wedge rel-S_0\ ?NF\ cm'$
proof (*induct n arbitrary: cs cm rule: less-induct*)
case (*less n cs cm*)
note $rel = less(2)$
note $steps = less(3)$
show *?case*
proof (*cases mt-state cm* $\in \{t, r\}$)
case *True*
with *rel* **obtain** $ts' p' q$ **where** $cs: cs = Config_S(S_0\ q)\ ts' p'$ **and** $q: q \in \{t, r\}$
by (*cases, auto*)
have $NF: False$ **if** $(cs, cs') \in st.step$ **for** cs' **using** *that unfolding cs using q*
by (*cases rule: st.step.cases, insert st. δ -set, auto*)
have $cs = ?NF$
proof (*cases n*)
case *0*
thus *?thesis using steps by auto*
next
case (*Suc m*)
from *NF relpow-Suc-E2*[*OF steps[unfolded this]*] **show** *?thesis by auto*
qed
thus *?thesis using rel by auto*
next
case *False*
define N **where** $N = Max(range(mt-pos\ cm))$
from *S-phase*[*OF rel False*] **obtain** $cs1$ **where** $dsteps: (cs, cs1) \in st.dstep \hat{\sim} (3 + N)$ **and** $rel: rel-S_1\ cs1\ cm$ **by** (*auto simp: N-def*)
from *st.dsteps-inj*[*OF dsteps steps NF*] **obtain** $k1$ **where** $n: n = 3 + N + k1$ **and** $steps: (cs1, ?NF) \in st.step \hat{\sim} k1$ **by** *auto*
from *rel* **have** $cs1 \neq ?NF$ **by** (*cases, auto*)
then **obtain** k **where** $k1: k1 = Suc\ k$ **using** *steps by (cases k1, auto)*
from *relpow-Suc-E2*[*OF steps[unfolded this]*] **obtain** $cs2$ **where** $step: (cs1, cs2) \in st.step$ **and** $steps: (cs2, ?NF) \in st.step \hat{\sim} k$ **by** *auto*
from *rel-S₁-E₀-st-step*[*OF rel step*] **obtain** $cm1$ **rule** **where** $mstep: (cm, cm1) \in step$ **and** $rel: rel-E_0\ cs2\ cm\ cm1$ **rule** **by** *auto*
from *E-phase*[*OF rel*] **obtain** $cs3$ **where** $dsteps: (cs2, cs3) \in st.dstep \hat{\sim} (6 + 4 * N)$ **and** $rel: rel-S_0\ cs3\ cm1$ **by** (*auto simp: N-def*)
from *st.dsteps-inj*[*OF dsteps steps NF*] **obtain** m **where** $k: k = 6 + 4 * N + m$ **and** $steps: (cs3, Config_S(S_0\ t)\ ts\ p) \in st.step \hat{\sim} m$
by *auto*
have $m < n$ **unfolding** $n\ k1\ k$ **by** *auto*
from *less(1)*[*OF this rel steps*] **obtain** cm' **where** $msteps: (cm1, cm') \in step \hat{*}$ **and** $rel: rel-S_0\ ?NF\ cm'$ **by** *auto*
from *mstep msteps* **have** $(cm, cm') \in step \hat{*}$ **by** *auto*

```

    with rel show ?thesis by auto
  qed
  qed
  then obtain cm' where msteps: (cm, cm') ∈ step∞* and rel: rel-S0 ?NF cm'
  by auto
  from rel obtain tc p where cm': cm' = ConfigM t tc p by (cases, auto)
  from msteps have w ∈ Lang unfolding cm-def cm' Lang-def using w by auto
  thus x ∈ map INP ' Lang unfolding x by auto
  qed

lemma rev-simulation-complexity: assumes w: set w ⊆ Σ
  and steps: (st.init-config (map INP w), cs) ∈ st.step∞n
  and n: n ≥ 2 * length w + 3 * k2 + 7 * k + 3
  shows ∃ cm. (init-config w, cm) ∈ step∞k
  proof -
    let ?INP = INP :: 'a ⇒ ('a, 'k) st-tape-symbol
    define cm1 where cm1 = init-config w
    define x where x = map ?INP w
    from R-phase[OF w, folded x-def] obtain cs1 where
      steps1: (st.init-config x, cs1) ∈ st.dstep∞(3 + 2 * length w) and rel1: rel-S0
    cs1 cm1
      by (auto simp: cm1-def)
    from st.dsteps-inj'[OF steps1 steps[folded x-def]] obtain n1 where
      nn1: n = 3 + 2 * length w + n1 and
      ssteps1: (cs1, cs) ∈ st.step∞n1
      using n by auto
    define M where M = (0 :: nat)
    have r: Max (range (mt-pos cm1)) ≤ M unfolding M-def cm1-def by (simp
    add: max-mt-pos-init)
    from nn1 n have n1 ≥ 3 * k2 + 7 * k by auto
    hence n1: n1 ≥ sum (λ i. 10 + 5 * (M + i)) {..< k} unfolding M-def
      using aux-sum-formula[of k] by simp
    from ssteps1 rel1 n1 r show ?thesis unfolding cm1-def[symmetric]
    proof (induct k arbitrary: cs1 cm1 M n1)
      case (Suc k cs1 cm1 M n1)
      let ?M = Max (range (mt-pos cm1))
      define n where n = (∑ i<k. 10 + 5 * (Suc M + i))
      from Suc(4) have n1 ≥ n + 10 + M * 5 unfolding n-def
        by (simp add: algebra-simps sum.distrib)
      with Suc(5) have n1: n1 ≥ n + 10 + 5 * ?M by linarith
      show ?case
      proof (cases mt-state cm1 ∈ {t, r})
        case False
        from S-phase[OF Suc(3) False]
        obtain cs2 where steps2: (cs1, cs2) ∈ st.dstep∞(3 + ?M) and rel2: rel-S1
      cs2 cm1 by auto
      from st.dsteps-inj'[OF steps2 Suc(2)] n1 obtain n2 where
        n12: n1 = 3 + ?M + n2 and
        steps2: (cs2, cs) ∈ st.step∞n2

```

by auto
from $n12$ $n1$ **obtain** $n3$ **where** $n23$: $n2 = \text{Suc } n3$ **by** (*cases* $n2$, *auto*)
from $steps2$ **obtain** $cs3$ **where** $step$: $(cs2, cs3) \in st.step$ **and** $steps3$: $(cs3, cs) \in st.step \rightsquigarrow n3$ **unfolding** $n23$ **by** (*metis relpow-Suc-E2*)
from $rel-S_1-E_0-st-step[OF rel2 step]$ **obtain** $cm2$ **rule** **where** $step$: $(cm1, cm2) \in step$ **and** $rel3$: $rel-E_0 cs3 cm1 cm2$ **rule** **by** *auto*
let $?M2 = \text{Max } (\text{range } (mt\text{-pos } cm2))$
from $n1$ $n12$ $n23$ **have** $n3$: $6 + 4 * ?M \leq n3$ **by** *auto*
from $E\text{-phase}[OF rel3]$ **obtain** $cs4$ **where** $dsteps$: $(cs3, cs4) \in st.dstep \rightsquigarrow (6 + 4 * ?M)$ **and** $rel4$: $rel-S_0 cs4 cm2$ **by** *auto*
from $st.dsteps\text{-inj}'[OF dsteps steps3 n3]$ **obtain** $n4$ **where** $n34$: $n3 = 6 + 4 * ?M + n4$
and $steps$: $(cs4, cs) \in st.step \rightsquigarrow n4$ **by** *auto*
from $max\text{-mt-pos-step}[OF step] \text{Suc}(5)$
have $M2$: $?M2 \leq \text{Suc } M$ **by** *linarith*
have $n4$: $n \leq n4$ **using** $n34$ $n12$ $n23$ $n1$ **by** *simp*
from $\text{Suc}(1)[OF steps rel4 n4[\text{unfolded } n\text{-def}] M2]$ **obtain** cm **where** $(cm2, cm) \in step \rightsquigarrow k ..$
with $step$ **have** $(cm1, cm) \in step \rightsquigarrow (\text{Suc } k)$ **by** (*metis relpow-Suc-I2*)
thus $?thesis ..$
next
case *True*
from $n1$ **obtain** $n2$ **where** $n1 = \text{Suc } n2$ **by** (*cases* $n1$, *auto*)
from $relpow\text{-Suc-E2}[OF \text{Suc}(2)[\text{unfolded } this]]$
obtain $cs2$ **where** $step$: $(cs1, cs2) \in st.step$ **by** *auto*
from $rel-S_0.cases[OF \text{Suc}(3)]$ **obtain** $tc' q tc p$ **where**
 $cs1$: $cs1 = \text{Config}_S (S_0 q) tc' 0$ **and**
 $cm1$: $cm1 = \text{Config}_M q tc p$
by *metis*
with *True* **have** q : $q \in \{t, r\}$ **by** *auto*
from $st.step.cases[OF step]$ **obtain** $ts q' a dir$ **where** $rule$: $(S_0 q, ts, q', a, dir) \in \delta'$
unfolding $cs1$ **by** *fastforce*
with q **have** *False* **unfolding** $\delta'\text{-def}$ **by** *auto*
thus $?thesis$ **by** *simp*
qed
qed *auto*
qed

6.2.6 Main Results

theorem *language-equivalence*: $st.Lang = \text{map } INP \text{ ' } Lang$
using *simulation rev-simulation* **by** *auto*

theorem *upper-time-bound-quadratic-increase*: **assumes** *upper-time-bound f*
shows *st.upper-time-bound* $(\lambda n. 3 * (f n)^2 + 13 * f n + 2 * n + 12)$
unfolding *st.upper-time-bound-def*
proof (*intro allI impI, rule ccontr*)
fix $ww c n$

```

assume set  $ww \subseteq INP \text{ ' } \Sigma$  and steps:  $(st.init\text{-}config\ w, c) \in st.step \widehat{\sim} n$ 
and  $bnd: \neg n \leq 3 * (f\ (length\ ww))^2 + 13 * (f\ (length\ ww)) + 2 * length\ ww$ 
+ 12
from  $INP\text{-}D[OF\ this(1)]$  obtain  $w$  where  $w: set\ w \subseteq \Sigma$  and  $ww: ww = map$ 
 $INP\ w$  by auto
let  $?lw = length\ w$ 
from  $bnd$  have  $n \geq 2 * ?lw + 3 * (f\ ?lw + 1)^2 + 7 * (f\ ?lw + 1) + 3$ 
by (auto simp: ww power2-eq-square)
from  $rev\text{-}simulation\text{-}complexity[OF\ w\ steps[unfolded\ ww]\ this]$ 
obtain  $cm$  where  $(init\text{-}config\ w, cm) \in step \widehat{\sim} (f\ ?lw + 1)$  by auto
from  $assms[unfolded\ upper\text{-}time\text{-}bound\text{-}def, rule\text{-}format, OF\ w\ this]$  show False
by simp
qed
end

```

6.3 Main Results with Proper Renamings

By using the renaming capabilities we can get rid of the *map INP* in the language equivalence theorem. We just assume that there will always be enough symbols for the renaming, i.e., an infinite supply of fresh names is available.

theorem *multitape-to-singletape*: **assumes** *valid-mttm* $(mttm :: ('p, 'a, 'k :: \{finite, zero\})mttm)$

```

and infinite  $(UNIV :: 'q\ set)$ 
and infinite  $(UNIV :: 'a\ set)$ 
shows  $\exists tm :: ('q, 'a)tm. valid\text{-}tm\ tm \wedge$ 
 $Lang\text{-}mttm\ mttm = Lang\text{-}tm\ tm \wedge$ 
 $(det\text{-}mttm\ mttm \longrightarrow det\text{-}tm\ tm) \wedge$ 
 $(upperb\text{-}time\text{-}mttm\ mttm\ f \longrightarrow upperb\text{-}time\text{-}tm\ tm\ (\lambda\ n. 3 * (f\ n)^2 + 13 * f\ n$ 
+  $2 * n + 12))$ 
proof (cases mttm)
let  $?INP = INP :: 'a \Rightarrow ('a, 'k)\ st\text{-}tape\text{-}symbol$ 
case  $(MTTM\ Q\ \Sigma\ \Gamma\ bl\ le\ \delta\ s\ t\ r)$ 
from  $assms(1)[unfolded\ this]$ 
interpret multitape-tm  $Q\ \Sigma\ \Gamma\ bl\ le\ \delta\ s\ t\ r$  by simp
let  $?TM1 = TM\ Q' ( ?INP \text{ ' } \Sigma)\ \Gamma'\ blank' \vdash \delta' (R_1 \cdot) (S_0\ t)\ (S_0\ r)$ 
have valid: valid-tm ?TM1 unfolding valid-tm.simps using valid-st .
interpret st: singletape-tm  $Q' ?INP \text{ ' } \Sigma\ \Gamma'\ blank' \vdash \delta' R_1 \cdot S_0\ t\ S_0\ r$  using
valid-st .
from language-equivalence
have id: Lang-tm ?TM1 = map INP ' Lang-mttm mttm unfolding MTTM by
auto
have finite Q' using st.fin-Q .
with  $assms(2)$  have  $\exists tq :: (('a, 'p, 'k)\ st\text{-}states \Rightarrow 'q). inj\text{-}on\ tq\ Q'$  by (metis
finite-infinite-inj-on)
then obtain  $tq :: - \Rightarrow 'q$  where inj-on tq Q' by blast
hence  $tq: inj\text{-}on\ tq\ (Q\text{-}tm\ ?TM1)$  by simp

```

```

from st.fin- $\Gamma$  have finG': finite  $\Gamma'$  .
from fin- $\Sigma$  assms( $\beta$ ) have infinite (UNIV -  $\Sigma$ ) by auto
then obtain B :: 'a set where B: finite B card B = card  $\Gamma'$  B  $\subseteq$  UNIV -  $\Sigma$ 
  by (meson infinite-arbitrarily-large)
from st.fin $\Sigma$  have finS': finite (?INP ' $\Sigma$ ) .
from finG' B obtain ta' where bij: bij-betw ta'  $\Gamma'$  B
  by (metis bij-betw-iff-card)
define ta where ta x = (if x  $\in$  ?INP ' $\Sigma$  then (case x of INP y  $\Rightarrow$  y) else ta' x)
for x
  have ta: inj-on ta ( $\Gamma$ -tm ?TM1) using bij B( $\beta$ )
    by (auto simp: bij-betw-def inj-on-def ta-def split: st-tape-symbol.splits)
  obtain tm' :: ('q, 'a)tm where valid: valid-tm tm' and lang: Lang-tm tm' = map
ta ' map INP ' $\Sigma$  Lang-mttm mttm
    and det: st.deterministic  $\Longrightarrow$  det-tm tm'
    and upper:  $\bigwedge$  f. st.upper-time-bound f  $\Longrightarrow$  upperb-time-tm tm' f
    using tm-renaming[OF valid ta tq, unfolded id] by auto
  note lang also have map ta ' map INP ' $\Sigma$  Lang-mttm mttm = ( $\lambda$  w. w) ' Lang-mttm mttm
    unfolding image-comp o-def map-map
  proof (rule image-cong[OF refl])
    fix w
    assume w  $\in$  Lang-mttm mttm
    hence w: set w  $\subseteq$   $\Sigma$  unfolding Lang-def MTTM Lang-mttm.simps by auto
    show map ( $\lambda$ x. ta (INP x)) w = w
      by (intro map-idI, insert w, auto simp: ta-def)
  qed
finally have lang: Lang-tm tm' = Lang-mttm mttm by simp
{
  assume det-mttm mttm
  hence deterministic unfolding MTTM by simp
  from det-preservation[OF this] have st.deterministic by auto
  from det[OF this] have det-tm tm' .
} note det = this
{
  assume upperb-time-mttm mttm f
  hence upper-time-bound f unfolding MTTM by simp
  from upper[OF upper-time-bound-quadratic-increase[OF this]]
  have upperb-time-tm tm' ( $\lambda$ n.  $3 * (f n)^2 + 13 * f n + 2 * n + 12$ ) .
} note upper = this
from valid lang det upper show ?thesis by blast
qed

end

```


References

- [1] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.
- [2] J. E. Hopcroft. *Introduction to automata theory, languages, and computation*. 3. edition, 2014.
- [3] M. Sipser. *Introduction to the theory of computation*. 2. edition, 2006.