A Verified Translation of Multitape Turing Machines into Singletape Turing Machines

Christian Dalvit and René Thiemann

April 18, 2024

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

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 $t_1, \ldots, t_n$ are stored sequentially onto a single tape $t_1\# \ldots \#t_n$ 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 $\#t_{i+1}\# \ldots \#t_n$ 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 ⇒ nat ⇒ 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 ⟹ finite ((UNIV :: 'a :: finite set) → A)
  by (metis PiE-UNIV-domain finite-PiE finite-code)
```

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

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

lemma relpow-mono: fixes R :: 'a rel shows R ⊆ S ⟹ R^^n ⊆ S^^n
  by (induct n, auto)

lemma finite-infinite-inj-on: assumes A: finite (A :: 'a set) and inf: infinite
(UNIV :: 'b set)
  shows ∃ f :: 'a ⇒ 'b. 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 ⇒ '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: (∑ i< (n :: nat). i) = (n − 1) ∗ n 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: (∑ i<n. 10 + 5 ∗ i) ≤ 3 ∗ n^2 + 7 ∗ (n :: nat)
proof −
  have (∑ i<n. 10 + 5 ∗ i) = 10 ∗ n + 5 ∗ (∑ i<n. i)
    by (subst sum.distrib, auto simp: sum-distrib-left)
  also have ... ≤ 10 ∗ n + 3 ∗ ((n − 1) ∗ n)
    by (unfold gauss-sum-nat2, rule add-left-mono, cases n, auto)
  also have ... = 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 $('q,'a,'k)mttm = MTTM$
  $(Q\text{-}tm\text{: } 'q\ set)$
  $'a\ set$
  $(\Gamma\text{-}tm\text{: } 'a\ set)$
  $'a$
  $'a$
  $('q \times ('k \Rightarrow 'a) \times 'q \times ('k \Rightarrow 'a) \times ('k \Rightarrow dir))\ set$
  $'q$
  $'q$
  $'q$

datatype $('a,'q,'k)\ mt\text{-}config = Config_M$
  $(mt\text{-}state\text{: } 'q)$
  $'k \Rightarrow nat \Rightarrow 'a$
  $(mt\text{-}pos\text{: } 'k \Rightarrow nat)$

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-Γ: *finite* Γ **and**
    Σ*-sub-*Γ: $\Sigma \subseteq \Gamma$ **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**
    δ*-set*: $\delta \subseteq (Q - \{t,r\}) \times (UNIV \to \Gamma) \times Q \times (UNIV \to \Gamma) \times (UNIV \to UNIV)$ **and**
    δLE: $(q,\ a,\ q',\ a',\ d) \in \delta \Longrightarrow a\ k = LE \Longrightarrow a'\ k = LE \wedge d\ k \in \{dir.N,dir.R\}$
begin

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

shows *q* ∈ *Q a k* ∈ Γ *q'* ∈ *Q b k* ∈ Γ
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* ⇒ *bool* **where**
valid-config (Config_M *q w n*) = (*q* ∈ *Q* ∧ (∀ *k*. *range* (*w k*) ⊆ Γ) ∧ (∀ *k*. *w k 0* = *LE*))

definition *init-config* :: *'a list* ⇒ (*'a*,*'q*,*'k*)*mt-config* **where**
init-config w = (Config_M *s* (λ *k n*. *if n* = *0 then LE else if k* = *0* ∧ *n* ≤ *length w then w* ! (*n*−*1*) *else blank*) (λ -. *0*))

lemma *valid-init-config*: *set w* ⊆ Σ ⟹ *valid-config* (*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*, (λ *k*. *ts k* (*n k*)), *q'*, *a*, *dir*) ∈ δ ⟹
(Config_M *q ts n*, Config_M *q'* (λ *k*. (*ts k*)(*n k* := *a k*)) (λ *k*. *go-dir* (*dir k*) (*n k*))) ∈ *step*

lemma *valid-step*: **assumes** *step*: (α,β) ∈ *step*
and *val*: *valid-config* α
shows *valid-config* β
using *step*
proof (*cases rule*: *step.cases*)
case (*step q ts n q' a 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* . *set w* ⊆ Σ ∧ (∃ *w' n*. (*init-config w*, Config_M *t w' n*) ∈ *step*^*)}

definition *deterministic* **where**
deterministic = (∀ *q a p1 b1 d1 p2 b2 d2*. (*q*,*a*,*p1*,*b1*,*d1*) ∈ δ ⟶ (*q*,*a*,*p2*,*b2*,*d2*) ∈ δ ⟶ (*p1*,*b1*,*d1*) = (*p2*,*b2*,*d2*))

definition *upper-time-bound* :: (*nat* ⇒ *nat*) ⇒ *bool* **where**
upper-time-bound f = (∀ *w c n*. *set w* ⊆ Σ ⟶ (*init-config w*, *c*) ∈ *step*^^*n* ⟶ *n* ≤ *f* (*length w*))
end

```
fun valid-mttm :: ('q,'a,'k :: {finite,zero})mttm ⇒ bool where
  valid-mttm (MTTM Q Σ Γ bl le δ s t r) = multitape-tm Q Σ Γ bl le δ s t r

fun Lang-mttm :: ('q,'a,'k :: {finite,zero})mttm ⇒ 'a list set where
  Lang-mttm (MTTM Q Σ Γ bl le δ s t r) = multitape-tm.Lang Σ bl le δ s t

fun det-mttm :: ('q,'a,'k :: {finite,zero})mttm ⇒ bool where
  det-mttm (MTTM Q Σ Γ bl le δ s t r) = multitape-tm.deterministic δ

fun upperb-time-mttm :: ('q,'a,'k :: {finite, zero})mttm ⇒ (nat ⇒ nat) ⇒ bool
where
  upperb-time-mttm (MTTM Q Σ Γ bl le δ s t r) f = multitape-tm.upper-time-bound
Σ bl le δ 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
  (Γ-tm: 'a set)
  'a
  'a
  ('q × 'a × 'q × 'a × dir) set
  'q
  'q
  'q

datatype ('a, 'q) st-config = Config_S
  'q
  nat ⇒ 'a
  nat

locale singletape-tm =
  fixes
```

Q :: $'q$ *set* **and**
Σ :: $'a$ *set* **and**
Γ :: $'a$ *set* **and**
blank :: $'a$ **and**
LE :: $'a$ **and**
δ :: $('q \times 'a \times 'q \times 'a \times dir)$ *set* **and**
s :: $'q$ **and**
t :: $'q$ **and**
r :: $'q$
assumes
fin-Q: *finite* Q **and**
fin-Γ: *finite* Γ **and**
Σ*-sub-*Γ: $\Sigma \subseteq \Gamma$ **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**
δ*-set*: $\delta \subseteq (Q - \{t,r\}) \times \Gamma \times Q \times \Gamma \times UNIV$ **and**
δ*LE*: $(q, LE, q', a', d) \in \delta \Longrightarrow a' = LE \wedge d \in \{dir.N,dir.R\}$
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* δ*-set* **by** *auto*

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

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

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

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

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

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

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

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

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

definition *Lang* :: *′a list set* **where**
  Lang = {*w* . *set w* ⊆ Σ ∧ (∃ *w′ n*. (*init-config w*, *Configs*$_S$ *t w′ n*) ∈ *step*^*)}

definition *deterministic* **where**
  deterministic = (∀ *q a p1 b1 d1 p2 b2 d2*. (*q*,*a*,*p1*,*b1*,*d1*) ∈ δ ⟶ (*q*,*a*,*p2*,*b2*,*d2*) ∈ δ ⟶ (*p1*,*b1*,*d1*) = (*p2*,*b2*,*d2*))

definition *upper-time-bound* :: (*nat* ⇒ *nat*) ⇒ *bool* **where**
  upper-time-bound f = (∀ *w c n*. *set w* ⊆ Σ ⟶ (*init-config w*, *c*) ∈ *step*^^*n* ⟶ *n* ≤ *f* (*length w*))
end

fun *valid-tm* :: (*′q*,*′a*)*tm* ⇒ *bool* **where**
  valid-tm (*TM Q* Σ Γ *bl le* δ *s t r*) = *singletape-tm Q* Σ Γ *bl le* δ *s t r*

fun *Lang-tm* :: (*′q*,*′a*)*tm* ⇒ *′a list set* **where**
  Lang-tm (*TM Q* Σ Γ *bl le* δ *s t r*) = *singletape-tm.Lang* Σ *bl le* δ *s t*

fun *det-tm* :: (*′q*,*′a*)*tm* ⇒ *bool* **where**
  det-tm (*TM Q* Σ Γ *bl le* δ *s t r*) = *singletape-tm.deterministic* δ

fun *upperb-time-tm* :: (*′q*,*′a*)*tm* ⇒ (*nat* ⇒ *nat*) ⇒ *bool* **where**
  upperb-time-tm (*TM Q* Σ Γ *bl le* δ *s t r*) *f* = *singletape-tm.upper-time-bound* Σ *bl le* δ *s f*

context *singletape-tm*
begin

A deterministic step (in a potentially non-determistic 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-determistic, 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*) ∈ δ ⟹
    (⋀ *q1′ a1 dir1*. (*q*, *ts n*, *q1′*, *a1*, *dir1*) ∈ δ ⟹ (*q1′*,*a1*,*dir1*) = (*q′*,*a*,*dir*)) ⟹
  (*Configs*$_S$ *q ts n*, *Configs*$_S$ *q′* (*ts*(*n* := *a*)) (*go-dir dir n*)) ∈ *dstep*

```
lemma dstepI: (q, a, q', b, dir) ∈ δ ⟹ ts n = a ⟹ ts' = ts(n := b) ⟹ n' =
go-dir dir n ⟹ q1 = q ⟹ q2 = q'
  ⟹ (⋀ q'' b' dir'. (q, a, q'', b', dir') ∈ δ ⟹ (q'', b', dir') = (q',b,dir))
  ⟹ (Config_S q1 ts n, Config_S q2 ts' n') ∈ dstep
  using dstep[of q ts n q' b dir] by blast

lemma dstep-step: dstep ⊆ step
proof
  fix st
  assume dstep: st ∈ dstep
  obtain s t where st: st = (s,t) by force
  have (s,t) ∈ 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 ∈ step using st by auto
qed

lemma dstep-inj: assumes (x,y) ∈ dstep
  and (x,z) ∈ 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) ∈ dstep^^n
  and (x,z) ∈ step^^m
  and ¬ (∃ u. (z,u) ∈ step)
shows ∃ k. m = n + k ∧ (y,z) ∈ step^^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') ∈ dstep and steps: (x',y) ∈ dstep^^n
by (metis relpow-Suc-E2)
  from step dstep-step have (x,x') ∈ step by auto
  with assms(3) have x ≠ z by auto
  with Suc(3) obtain mm where m: m = Suc mm by (cases m, auto)
  from Suc(3)[unfolded this] obtain x'' where step': (x,x'') ∈ step and steps':
(x'',z) ∈ step^^mm by (metis relpow-Suc-E2)
  from dstep-inj[OF step step'] have x'': x'' = x' by auto
  from Suc(1)[OF steps steps'[unfolded x'']] obtain k where mm: mm = n + k
and steps: (y, z) ∈ step ^^ k by auto
  thus ?case unfolding m mm by auto
qed auto

lemma dsteps-inj': assumes (x,y) ∈ dstep^^n
  and (x,z) ∈ step^^m
  and m ≥ n
shows ∃ k. m = n + k ∧ (y,z) ∈ step^^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') ∈ dstep and steps: (x',y) ∈ dstep^^n
by (metis relpow-Suc-E2)
  from step dstep-step have (x,x') ∈ step by auto
  with Suc obtain mm where m: m = Suc mm by (cases m, auto)
  from Suc(3)[unfolded this] obtain x'' where step': (x,x'') ∈ step and steps':
(x'',z) ∈ step^^mm by (metis relpow-Suc-E2)
  from dstep-inj[OF step step'] have x'': x'' = x' by auto
  from Suc(1)[OF steps steps'[unfolded x'']] m Suc obtain k where mm: mm =
n + k and steps: (y, z) ∈ step ^^ 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 Σ Γ blank LE δ s t r
  for Q :: 'q set and Σ :: 'a set and Γ blank LE δ s t r
    + fixes ra :: 'a ⇒ 'b
    and rq :: 'q ⇒ 'p
  assumes ra: inj-on ra Γ
    and rq: inj-on rq Q
begin

abbreviation rd where rd ≡ map-prod rq (map-prod ra (map-prod rq (map-prod
ra (λ d :: dir. d))))

sublocale ren: singletape-tm rq ‘ Q ra ‘ Σ ra ‘ Γ ra blank ra LE rd ‘ δ 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 ∉ ra ‘ Σ using ra tm LE by (simp add: inj-on-image-mem-iff)
  show ra blank ∉ ra ` Σ using ra tm blank by (simp add: inj-on-image-mem-iff)
  show rq t ≠ rq r using rq tQ rQ tr by (metis inj-on-contraD)
  show rd ` δ ⊆ (rq ` Q − {rq t, rq r}) × ra ` Γ × rq ` Q × ra ` Γ × UNIV
    using δ-set tQ rQ rq by (auto simp: inj-on-def)
  fix p1 p2 b2 d
  assume (p1, ra LE, p2, b2, d) ∈ rd ` δ
  then obtain q1 q2 a1 a2 where mem: (q1, a1, q2, a2, d) ∈ δ and
    id: p1 = rq q1 ra LE = ra a1 p2 = rq q2 b2 = ra a2 by auto
  from δ[OF mem] id(2) LE ra have a1 = LE by (simp add: inj-onD)
  with δLE[OF mem[unfolded this]]
  show b2 = ra LE ∧ d ∈ {dir.N, dir.R} unfolding id by auto
qed

fun rc :: ('a, 'q) st-config ⇒ ('b, 'p) st-config where
  rc (Config_S q tc pos) = Config_S (rq q) (ra o tc) pos

lemma ren-init: rc (init-config w) = ren.init-config (map ra w)
  unfolding init-config-def ren.init-config-def rc.simps
  by auto

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

lemma ren-steps: assumes (c,c') ∈ step^*
  shows (rc c, rc c') ∈ ren.step^*
  using assms by (induct, insert ren-step, force+)

lemma ren-steps-count: assumes (c,c') ∈ step^^n
  shows (rc c, rc c') ∈ ren.step^^n
  using assms by (induct n arbitrary: c c', insert ren-step, force+)

lemma ren-Lang-forward: assumes w ∈ Lang
  shows map ra w ∈ ren.Lang
proof −
  from assms[unfolded Lang-def, simplified]
  obtain w' n where w: set w ⊆ Σ and steps: (init-config w, Config_S t w' n) ∈
step^*
    by auto
  from ren-steps[OF steps, unfolded ren-init, unfolded rc.simps] w
  show map ra w ∈ ren.Lang unfolding ren.Lang-def by auto
qed
```

```
abbreviation ira where ira ≡ the-inv-into Γ ra
abbreviation irq where irq ≡ the-inv-into Q rq

interpretation inv: renaming-of-singletape-tm rq ‘ Q ra ‘ Σ ra ‘ Γ ra blank ra LE
rd ‘ δ 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 ‘ ra ‘ Σ = Σ 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 ‘ ra ‘ Γ = Γ 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-δ: inv.rd ‘ rd ‘ δ = δ
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-δ 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 ∈ ren.Lang
  shows ∃ w. v = map ra w ∧ w ∈ Lang
proof (intro exI conjI)
  let ?w = map ira v
  from inv.ren-Lang-forward[OF assms, unfolded inv-ren-Lang]
  show ?w ∈ Lang .
  show v = map ra ?w unfolding map-map o-def
  proof (subst map-idI)
    fix b
    assume b ∈ set v
    with assms[unfolded ren.Lang-def] Σ-sub-Γ obtain a where a: a ∈ Γ 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 ⊆ ren.Lang using ren-Lang-forward by blast
  show ren.Lang ⊆ 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 ∈ inv.rd ‘ rd ‘ δ by force
  from 1 have t2: inv.rd ?t2 ∈ inv.rd ‘ rd ‘ δ by force
  from t1 t2 have (irq q, ira a, irq p1, ira b1, d1) ∈ δ  (irq q, ira a, irq p2, ira
b2, d2) ∈ δ
    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 ⊆ ra ‘ Σ and steps: (ren.init-config w, c) ∈ ren.step ^^ n
  define v where v = map ira w
  from w have v: set v ⊆ Σ 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) ∈ inv.ren.step ^^ n .
  also have inv.ren.step = step using inv-ren-δ 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) ∈ step ^^ n .
  with assms[unfolded upper-time-bound-def] v have n ≤ f (length v) by simp
  thus n ≤ 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 ⇒ 'b) (Γ-tm tm)
  and inj-on (rq :: 'q ⇒ 'p) (Q-tm tm)
shows ∃ tm' :: ('p,'b)tm.
  valid-tm tm' ∧
  Lang-tm tm' = map ra ‘ Lang-tm tm ∧
  (det-tm tm ⟶ det-tm tm') ∧
  (∀ f. upperb-time-tm tm f ⟶ upperb-time-tm tm' f)
proof (cases tm)
  case (TM Q Σ Γ bl le δ s t r)
  with assms interpret singletape-tm Q Σ Γ bl le δ s t r by auto
  interpret renaming-of-singletape-tm Q Σ Γ bl le δ s t r ra rq
    by (unfold-locales, insert assms TM, auto)
  let ?tm' = TM (rq ‘ Q) (ra ‘ Σ) (ra ‘ Γ) (ra bl) (ra le) (rd ‘ δ) (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* ($\vdash$) | *TUPLE 'k* $\Rightarrow$ *'a tuple-symbol* | *INP 'a*
datatype *'a sym-or-bullet* = *SYM 'a* | *BULLET* ($\bullet$)

datatype (*'a*,*'q*,*'k*) *st-states* =
  R_1 *'a sym-or-bullet* |
  R_2 |
  S_0 *'q* |
  S *'q 'k* $\Rightarrow$ *'a sym-or-bullet* |
  S_1 *'q 'k* $\Rightarrow$ *'a* |
  E_0 *'q 'k* $\Rightarrow$ *'a 'k* $\Rightarrow$ *dir* |
  E *'q 'k* $\Rightarrow$ *'a sym-or-bullet 'k* $\Rightarrow$ *dir* |
  Er *'q 'k* $\Rightarrow$ *'a sym-or-bullet 'k* $\Rightarrow$ *dir 'k set* |
  El *'q 'k* $\Rightarrow$ *'a sym-or-bullet 'k* $\Rightarrow$ *dir 'k set* |
  Em *'q 'k* $\Rightarrow$ *'a sym-or-bullet 'k* $\Rightarrow$ *dir 'k set*

type-synonym (*'a*,*'q*,*'k*)*mt-rule* = *'q* $\times$ (*'k* $\Rightarrow$ *'a*) $\times$ *'q* $\times$ (*'k* $\Rightarrow$ *'a*) $\times$ (*'k* $\Rightarrow$ *dir*)

context *multitape-tm*
begin

definition *R1-Set* **where** *R1-Set* = *SYM* ‘ $\Sigma \cup \{\bullet\}$

definition *gamma-set* :: (*'k* $\Rightarrow$ *'a tuple-symbol*) *set* **where**
  gamma-set = (*UNIV* :: *'k set*) $\rightarrow$ *NO-HAT* ‘ Γ $\cup$ *HAT* ‘ Γ

definition Γ' :: (*'a*, *'k*) *st-tape-symbol set* **where**
  Γ' = *TUPLE* ‘ *gamma-set* $\cup$ *INP* ‘ $\Sigma \cup \{\vdash\}$

definition *func-set* = (*UNIV* :: *'k set*) $\rightarrow$ *SYM* ‘ $\Gamma \cup \{\bullet\}$

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

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

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

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

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

definition *replace-sym* :: $('k \Rightarrow 'a$ *tuple-symbol*$) \Rightarrow ('k \Rightarrow 'a$ *sym-or-bullet*$) \Rightarrow ('k$ $\Rightarrow 'a$ *tuple-symbol*$)$ **where**

```
  replace-sym tup ys = (λ k. if k ∈ (compute-idx-set tup ys)
                               then (case ys k of SYM a ⇒ NO-HAT a)
                               else tup k)
```

definition *place-hats-to-dir* :: *dir* $\Rightarrow ('k \Rightarrow 'a$ *tuple-symbol*$) \Rightarrow ('k \Rightarrow$ *dir*$) \Rightarrow 'k$ *set* $\Rightarrow ('k \Rightarrow 'a$ *tuple-symbol*$)$ **where**

```
  place-hats-to-dir dir tup ds I = (λ k. (case tup k of
                                      NO-HAT a ⇒ if k ∈ I ∧ ds k = dir
                                                   then HAT a
                                                   else NO-HAT a
                                    | HAT a ⇒ HAT a ))
```

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

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

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

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

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

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

lemma *compute-idx-range*[*simp*,*intro*]:
assumes $\mathit{tup} \in \mathit{gamma\text{-}set}$
assumes $\mathit{ys} \in \mathit{func\text{-}set}$
shows $\mathit{compute\text{-}idx\text{-}set}\ \mathit{tup}\ \mathit{ys} \in \mathit{UNIV}$
by *auto*

lemma *update-ys-range*[*simp*,*intro*]:
assumes $\mathit{tup} \in \mathit{gamma\text{-}set}$
assumes $\mathit{ys} \in \mathit{func\text{-}set}$
shows $\mathit{update\text{-}ys}\ \mathit{tup}\ \mathit{ys} \in \mathit{func\text{-}set}$
by (*insert assms*, *fastforce simp*: *update-ys-def func-set-def*)

lemma *replace-sym-range*[*simp*,*intro*]:

assumes *tup* ∈ *gamma-set*
assumes *ys* ∈ *func-set*
shows *replace-sym tup ys* ∈ *gamma-set*
proof −
have ∀ *k*. (*if k* ∈ *compute-idx-set tup ys then case ys k of SYM x* ⇒ *NO-HAT x else tup k*) ∈ *NO-HAT* ‘ Γ ∪ *HAT* ‘ Γ
by(*intro allI*, *insert assms*, *cases k* ∈ *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* ∈ *gamma-set*
assumes *tup x* = *HAT a*
shows *a* ∈ Γ
proof −
have *range tup* ⊆ *NO-HAT* ‘ Γ ∪ *HAT* ‘ Γ
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* ∈ *gamma-set*
assumes *tup x* = *NO-HAT a*
shows *a* ∈ Γ
proof −
have *range tup* ⊆ *NO-HAT* ‘ Γ ∪ *HAT* ‘ Γ
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* ∈ *gamma-set*
shows *place-hats-to-dir d tup ds I* ∈ *gamma-set*
proof −
have ∀ *k*. (*case tup k of NO-HAT a* ⇒ *if k* ∈ *I* ∧ *ds k* = *d then HAT a else NO-HAT a* | *HAT x* ⇒ *HAT x*)
∈ *NO-HAT* ‘ Γ ∪ *HAT* ‘ Γ
proof
fix *k*
show (*case tup k of NO-HAT a* ⇒ *if k* ∈ *I* ∧ *ds k* = *d then HAT a else NO-HAT a* | *HAT x* ⇒ *HAT x*)

$\in$ *NO-HAT* ‘ Γ $\cup$ *HAT* ‘ Γ
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*
$x \in \Sigma \Longrightarrow$ *SYM* $x \in$ *R1-Set*
$x \in \Sigma \Longrightarrow$ *encSym* $x \in \Gamma'$
blank$'$ $\in \Gamma'$
hatLE$'$ $\in \Gamma'$
$x \in \Sigma \Longrightarrow$ *INP* $x \in \Gamma'$
$y \in$ *gamma-set* $\Longrightarrow$ *TUPLE* $y \in \Gamma'$
$(\lambda$-. $\cdot) \in$ *func-set*
$f \in$ *UNIV* $\rightarrow$ *SYM* ‘ $\Gamma \Longrightarrow f \in$ *func-set*
$g \in$ *UNIV* $\rightarrow \Gamma \Longrightarrow$ *SYM* $\circ\, g \in$ *func-set*
$f \in$ *UNIV* $\rightarrow$ *SYM* ‘ $\Gamma \Longrightarrow$ *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* $\Longrightarrow a \in$ *func-set* $\Longrightarrow$ *add-inp b a* $\in$ *func-set*
unfolding *func-set-def gamma-set-def*

proof
  fix x
  assume a: $a \in UNIV \to SYM$ ‘ $\Gamma \cup \{\bullet\}$ **and** b: $b \in UNIV \to NO\text{-}HAT$ ‘ $\Gamma \cup HAT$ ‘ Γ
  from a **have** a: $a\ x \in SYM$ ‘ $\Gamma \cup \{\bullet\}$ **by** *auto*
  from b **have** b: $b\ x \in NO\text{-}HAT$ ‘ $\Gamma \cup HAT$ ‘ Γ **by** *auto*
  show *add-inp* $b\ a\ x \in SYM$ ‘ $\Gamma \cup \{\bullet\}$ **using** $a\ b$
    unfolding *add-inp-def* **by** (*cases b x*, *auto simp*: *gamma-set-def*)
qed

lemma *automation*[*simp*]: $\bigwedge a\ b\ A\ B.\ (S\ a\ b \in (\lambda x.\ case\ x\ of\ (x1,\ x2) \Rightarrow S\ x1\ x2)$ ‘ $(A \times B)) \longleftrightarrow (a \in A \wedge b \in B)$
  $\bigwedge a\ b\ A\ B.\ (S_1\ a\ b \in (\lambda x.\ case\ x\ of\ (x1,\ x2) \Rightarrow S_1\ x1\ x2)$ ‘ $(A \times B)) \longleftrightarrow (a \in A \wedge b \in B)$
  $\bigwedge a\ b\ c\ A\ B\ C.\ (E_0\ a\ b\ c \in (\lambda x.\ case\ x\ of\ (x1,\ x2,\ x3) \Rightarrow E_0\ x1\ x2\ x3)$ ‘ $(A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge c \in C)$
  $\bigwedge a\ b\ c\ A\ B\ C.\ (E\ a\ b\ c \in (\lambda x.\ case\ x\ of\ (x1,\ x2,\ x3) \Rightarrow E\ x1\ x2\ x3)$ ‘ $(A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge c \in C)$
  $\bigwedge a\ b\ c\ d\ A\ B\ C.\ (Er\ a\ b\ c\ d \in (\lambda x.\ case\ x\ of\ (x1,\ x2,\ x3,\ x4) \Rightarrow Er\ x1\ x2\ x3\ x4)$ ‘ $(A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge (c,d) \in C)$
  $\bigwedge a\ b\ c\ d\ A\ B\ C.\ (Em\ a\ b\ c\ d \in (\lambda x.\ case\ x\ of\ (x1,\ x2,\ x3,\ x4) \Rightarrow Em\ x1\ x2\ x3\ x4)$ ‘ $(A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge (c,d) \in C)$
  $\bigwedge a\ b\ c\ d\ A\ B\ C.\ (El\ a\ b\ c\ d \in (\lambda x.\ case\ x\ of\ (x1,\ x2,\ x3,\ x4) \Rightarrow El\ x1\ x2\ x3\ x4)$ ‘ $(A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge (c,d) \in C)$
  $blank' \neq \vdash$
  $\vdash \neq blank'$
  $blank' \neq INP\ x$
  $INP\ x \neq blank'$
  by (*force simp*: *blank′-def*)+

interpretation *st*: *singletape-tm* Q' $(INP$ ‘ $\Sigma)$ Γ' $blank'$ $\vdash$ δ' R_1 $\bullet$ S_0 t S_0 r
proof
  show *finite* Q'
    unfolding *Q′-def* **using** *fin-Q fin-Γ*
    by (*intro finite-UnI finite-imageI finite-cartesian-product*, *auto*)
  show *finite* Γ' **by** (*rule fin-Γ′*)
  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 $blank' \notin INP$ ‘ Σ **unfolding** *blank′-def* **by** *auto*
  show $R_1\ \bullet \in Q'$ **unfolding** *Q′-def* **by** *auto*
  show $\delta' \subseteq (Q' - \{S_0\ t,\ S_0\ r\}) \times \Gamma' \times Q' \times \Gamma' \times UNIV$
    unfolding *δ′-def Q′-def* **using** *tm*
    by (*auto dest*: *δ*)
  show $(q, \vdash, q', a', d) \in \delta' \Longrightarrow a' = \vdash \wedge d \in \{dir.N,\ dir.R\}$ **for** $q\ q'\ a'\ d$
    unfolding *δ′-def* **by** (*auto simp*: *hatLE′-def blank′-def*)
qed *auto*

```
lemma valid-st: singletape-tm Q′ (INP ‘ Σ) Γ′ blank′ ⊢ δ′ (R₁ ·) (S₀ t) (S₀ r) ..
```

Determinism is preserved.

```
lemma det-preservation: deterministic ⟹ st.deterministic
  unfolding deterministic-def st.deterministic-def unfolding δ′-def
  by auto
```

6.2 Soundness of the Translation

```
lemma range-mt-pos:
  ∃ i. Max (range (mt-pos cm)) = mt-pos cm i
  finite (range (mt-pos (cm :: ('a, 'q, 'k) mt-config)))
  range (mt-pos cm) ≠ {}
proof −
  show finite (range (mt-pos cm)) by auto
  moreover show range (mt-pos cm) ≠ {} by auto
  ultimately show ∃ i. Max (range (mt-pos cm)) = mt-pos cm i
    by (meson Max-in imageE)
qed

lemma max-mt-pos-step: assumes (cm,cm′) ∈ step
  shows Max (range (mt-pos cm′)) ≤ Suc (Max (range (mt-pos cm)))
proof −
  from range-mt-pos(1)[of cm′] obtain i′
    where max1: Max (range (mt-pos cm′)) = mt-pos cm′ i′ by auto
  hence Max (range (mt-pos cm′)) ≤ mt-pos cm′ i′ by auto
  also have ... ≤ Suc (mt-pos 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 ... ≤ Suc (Max (range (mt-pos cm))) using range-mt-pos[of cm] by
simp
  finally show ?thesis .
qed

lemma max-mt-pos-init: Max (range (mt-pos (init-config w))) = 0
  unfolding init-config-def by auto

lemma INP-D: assumes set x ⊆ INP ‘ Σ
  shows ∃ w. x = map INP w ∧ set w ⊆ Σ
  using assms
proof (induct x)
  case (Cons x xs)
  then obtain w where xs = map INP w ∧ set w ⊆ Σ by auto
  moreover from Cons(2) obtain a where x = INP a and a ∈ Σ 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 ⇒ nat ⇒ ('a, 'k) st-tape-symbol
  where enc (Config_M q tc p) n = TUPLE (λ k. if p k = n then HAT (tc k n)
else NO-HAT (tc k n))

inductive rel-R_1 :: (('a, 'k) st-tape-symbol,('a, 'q, 'k) st-states)st-config ⇒ 'a list
⇒ nat ⇒ bool where
  n = length w ⟹
  tc' 0 = ⊢ ⟹
  p' ≤ n ⟹
  (⋀ i. i < p' ⟹ enc (init-config w) i = tc' (Suc i)) ⟹
  (⋀ i. i ≥ p' ⟹ tc' (Suc i) = (if i < n then INP (w ! i) else blank')) ⟹
  (p' = 0 ⟹ q' = •) ⟹
  (⋀ p. p' = Suc p ⟹ q' = SYM (w ! p)) ⟹
  rel-R_1 (Config_S (R_1 q') tc' (Suc p')) w p'


lemma rel-R_1-init: shows ∃ cs1. (st.init-config (map INP w), cs1) ∈ st.dstep ∧
rel-R_1 cs1 w 0
proof −
  let ?INP = INP :: 'a ⇒ ('a, 'k) st-tape-symbol
  have mem: (R_1 •, ⊢, R_1 •, ⊢, dir.R) ∈ δ' unfolding δ'-def by auto
  let ?cs1 = Config_S (R_1 •) (λn. if n = 0 then ⊢ else if n ≤ length (map ?INP w)
then map ?INP w ! (n − 1) else blank') (Suc 0)
  have (st.init-config (map INP w), ?cs1) ∈ st.dstep
     unfolding st.init-config-def by (rule st.dstepI[OF mem], auto simp: δ'-def
blank'-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-R_1: assumes rel-R_1 cs0 w j
  and j < length w
  and set w ⊆ Σ
shows ∃ cs1. (cs0, cs1) ∈ st.dstep ∧ 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 ∈ Σ by auto
  show ?thesis
  proof (cases j)
    case 0
    with 1 have q': q' = • 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 •, INP (w ! 0), R_1 (SYM (w ! 0)), hatLE', dir.R) ∈ δ'
unfolding δ'-def
      using wj 0 by auto
    let ?cs1 = Configs_S (R_1 (SYM (w ! 0))) (tc'(Suc 0 := hatLE')) (Suc (Suc 0))
    have enc: enc (init-config w) 0 = hatLE' unfolding init-config-def hatLE'-def
by 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-R_1 ?cs1 w (Suc 0)
     by (intro rel-R_1.intros, rule 1(2), insert 1 0 assms(2), auto simp: enc) (cases
w, 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 assms(2) have p < length w by auto
    with assms(3) have w ! p ∈ Σ by auto
    with wj have (w ! p, w ! j) ∈ Σ × Σ by auto
    hence mem: (R_1 (SYM (w ! p)), INP (w ! j), R_1 (SYM (w ! j)), encSym (w !
p), dir.R) ∈ δ' unfolding δ'-def by auto
    have enc: enc (init-config w) j = encSym (w ! p) unfolding Suc using ‹p <
length w›
      by (auto simp: init-config-def encSym-def)
    from 1(6)[of j] assms 1 have tc': tc' (Suc j) = INP (w ! j) by auto
     let ?cs1 = Configs_S (R_1 (SYM (w ! j))) (tc'(Suc j := encSym (w ! p))) (Suc
(Suc j))
    have (cs0, ?cs1) ∈ st.dstep unfolding cs0
      by (rule st.dstepI[OF mem], insert q' tc', auto simp: δ'-def blank'-def)
    moreover have rel-R_1 ?cs1 w (Suc j)
      by (intro rel-R_1.intros, insert 1 assms enc, auto)
    ultimately show ?thesis by blast
  qed
qed

inductive rel-R_2 :: (('a, 'k) st-tape-symbol,('a, 'q, 'k) st-states)st-config ⇒ 'a list
⇒ nat ⇒ bool where
  tc' 0 = ⊢ ⟹
  (⋀ i. enc (init-config w) i = tc' (Suc i)) ⟹
  p ≤ length w ⟹
  rel-R_2 (Configs_S R_2 tc' p) w p


lemma rel-R_1-R_2: assumes rel-R_1 cs0 w (length w)
  and set w ⊆ Σ
shows ∃ cs1. (cs0, cs1) ∈ st.dstep ∧ rel-R_2 cs1 w (length w)
  using assms
proof (cases rule: rel-R_1.cases)
  case (1 n tc' q')
  note cs0 = 1(1)
  have enc: enc (init-config w) i = tc' (Suc i) if i ≠ length w for i
  proof (cases i < 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 = Config_S 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 = Config_S 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-R_2-R_2*: **assumes** *rel-R_2 cs0 w (Suc j)*
  and *set w* $\subseteq \Sigma$
shows $\exists$ *cs1*. (*cs0*, *cs1*) $\in$ *st.dstep* $\wedge$ *rel-R_2 cs1 w j*
  using *assms*
proof (*cases rule*: *rel-R_2.cases*)
  case (*1 tc'*)
  note *cs0* = *1*(*1*)
  from *1* **have** *j*: *j* < *length w* **by** *auto*
  have *tc*: *tc'* (*Suc j*) $\in \Gamma' - \{ \vdash \}$ **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*: (R_2, *tc'* (*Suc j*), R_2, *tc'* (*Suc j*), *dir.L*) $\in \delta'$ **unfolding** δ'*-def* **by** *auto*
  let *?cs1* = *Config$_S$ R_2 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-R_2 ?cs1 w j*
    by (*intro rel-R_2.intros*, *insert 1*, *auto*)
  ultimately show *?thesis* **by** *blast*
qed

inductive *rel-S_0* :: (('*a*, '*k*) *st-tape-symbol*,('*a*, '*q*, '*k*) *st-states*)*st-config* $\Rightarrow$ ('*a*, '*q*, '*k*) *mt-config* $\Rightarrow$ *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*) $\Longrightarrow$
  rel-S_0 (*Config$_S$* (S_0 *q*) *tc' 0*) (*Config$_M$ q tc p*)

lemma *rel-R_2-S_0*: **assumes** *rel-R_2 cs0 w 0*
  and *set w* $\subseteq \Sigma$
shows $\exists$ *cs1*. (*cs0*, *cs1*) $\in$ *st.dstep* $\wedge$ *rel-S_0 cs1* (*init-config w*)
  using *assms*
proof (*cases rule*: *rel-R_2.cases*)
  case (*1 tc'*)
  note *cs0* = *1*(*1*)
  hence *mem*: (R_2, $\vdash$, S_0 *s*, $\vdash$, *dir.N*) $\in \delta'$ **unfolding** δ'*-def* **by** *auto*
  let *?cs1* = *Config$_S$* (S_0 *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-S_0 ?cs1* (*init-config w*) **using** *valid-init-config*[*OF assms*(*2*)] **unfolding** *init-config-def*
    by (*intro rel-S_0.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* $\subseteq$ Σ
  shows $\exists$ *cs*. (*st.init-config* (*map INP w*), *cs*) $\in$ *st.dstep*^^(*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* $\Longrightarrow$ $\exists$ *cs2*. (*cs1*, *cs2*) $\in$ *st.dstep*^^*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* ^^ *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* ^^ *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*^^*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* ^^ *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*^^(*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* ^^ *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**
  $tc'\ 0 = \vdash \Longrightarrow$
  $(\bigwedge i.\ tc'\ (Suc\ i) = enc\ (Config_M\ q\ tc\ p)\ i) \Longrightarrow$
  $valid\text{-}config\ (Config_M\ q\ tc\ p) \Longrightarrow$
  $(\bigwedge i.\ inp\ i = (if\ p\ i < p'\ then\ SYM\ (tc\ i\ (p\ i))\ else\ \bullet)) \Longrightarrow$
  $rel\text{-}S\ (Config_S\ (S\ q\ inp)\ tc'\ (Suc\ p'))\ (Config_M\ q\ tc\ p)\ p'$

lemma $rel\text{-}S_0\text{-}S$: **assumes** $rel\text{-}S_0\ cs0\ cm$
  and $mt\text{-}state\ cm \notin \{t,r\}$
shows $\exists\ cs1.\ (cs0,\ cs1) \in st.dstep \wedge rel\text{-}S\ cs1\ cm\ 0$
  using *assms*(*1*)
proof (*cases rule*: $rel\text{-}S_0.cases$)
  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*: $(S_0\ q, \vdash, S\ q\ (\lambda\text{-}.\ \bullet), \vdash, dir.R) \in \delta'$ **unfolding** δ'*-def* **by** *auto*
  let $?cs1 = Config_S\ (S\ q\ (\lambda\text{-}.\ \bullet))\ tc'\ (Suc\ 0)$
  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** $rel\text{-}S\ (Config_S\ (S\ q\ inp)\ tc'\ p')\ cm\ j$
  shows $inp \in func\text{-}set \wedge q \in Q \wedge (\exists\ t.\ tc'\ (Suc\ i) = TUPLE\ t \wedge t \in gamma\text{-}set)$

  using *assms*(*1*)
proof (*cases rule*: *rel-S.cases*)
  case (*1 tc p*)
  from *1* **have** *q*: $q \in Q$ **by** *auto*
  have *inp*: $inp \in func\text{-}set$ **unfolding** *func-set-def 1*(*6*) **using** *1*(*5*)
    by *force*
  have $\exists\ t.\ tc'\ (Suc\ i) = TUPLE\ t \wedge t \in gamma\text{-}set$ **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** $rel\text{-}S\ cs0\ cm\ p'$
  $p' \leq Max\ (range\ (mt\text{-}pos\ cm))$
shows $\exists\ cs1.\ (cs0,\ cs1) \in st.dstep \wedge rel\text{-}S\ cs1\ cm\ (Suc\ p')$
  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 × (func-set − (UNIV → SYM ‘ Γ)) × gamma-set
  let ?f = λ(q, inp, t). (S q inp, TUPLE t, S q (add-inp t inp), TUPLE t, dir.R)
  obtain i where mt-pos cm i = Max (range (mt-pos cm)) using range-mt-pos(1)[of cm] by auto
  with assms 1 have p′ ≤ p i by auto
  with 1(6)[of i] have inp i = • by auto
  hence inp: inp ∉ (UNIV → SYM ‘ Γ)
    by (metis PiE UNIV-I image-iff sym-or-bullet.distinct(1))
  with rel-S-mem[OF assms(1)[unfolded cs0], of p′] obtain t where
    (q,inp,t) ∈ ?Set and tc′: tc′ (Suc p′) = TUPLE t by auto
  hence ?f (q,inp,t) ∈ δ′ unfolding δ′-def by blast
  hence mem: (S q inp, TUPLE t, S q (add-inp t inp), TUPLE t, dir.R) ∈ δ′ by simp
  let ?cs1 = Config_S (S q (add-inp t inp)) tc′ (Suc (Suc p′))
  have (cs0,?cs1) ∈ st.dstep unfolding cs0 using inp
    by (intro st.dstepI[OF mem], auto simp: tc′ δ′-def blank′-def)
  moreover have rel-S ?cs1 cm (Suc p′) unfolding cm
  proof (intro rel-S.intros 1)
    from 1(4)[of p′, unfolded tc′]
     have t: t = (λk. if p k = p′ then HAT (tc k p′) else NO-HAT (tc k p′)) by auto
    show ⋀ k. add-inp t inp k = (if p k < Suc p′ then SYM (tc k (p k)) else •)
      unfolding add-inp-def 1 t by auto
  qed
  ultimately show ?thesis by blast
qed

inductive rel-S_1 :: (('a, 'k) st-tape-symbol,('a, 'q, 'k) st-states)st-config ⇒ ('a, 'q, 'k) mt-config ⇒ bool where
  tc′ 0 = ⊢ ⟹
  (⋀ i. tc′ (Suc i) = enc (Config_M q tc p) i) ⟹
  valid-config (Config_M q tc p) ⟹
  (⋀ i. inp i = tc i (p i)) ⟹
  (⋀ i. p i < p′) ⟹
  p′ = Suc (Max (range p)) ⟹
  rel-S_1 (Config_S (S_1 q inp) tc′ p′) (Config_M q tc p)

lemma rel-S-S_1: assumes rel-S cs0 cm p′
  p′ = Suc (Max (range (mt-pos cm)))
shows ∃ cs1. (cs0, cs1) ∈ st.dstep ∧ rel-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: p′ > Max (range (mt-pos cm)) by auto
  note cs0 = 1(1)
  note cm = 1(2)
  from p'Max range-mt-pos(2−)[of cm] have pip: p i < p' for i unfolding cm
by auto
  let ?SET = Q × func-set × gamma-set
  let ?Set = Q × (UNIV → SYM ‘ Γ) × (Γ' − { ⊢ })
  from rel-S-mem[OF assms(1)[unfolded cs0], of p'] obtain t where
    mem: (q,inp,t) ∈ ?SET and tc': tc' (Suc p') = TUPLE t by auto
  hence inp ∈ func-set by auto
  with pip have inp: inp ∈ UNIV → SYM ‘ Γ unfolding func-set-def using 1(6)
by auto
  with mem have (q,inp,TUPLE t) ∈ ?Set by auto
  hence (λ (q,inp,a). (S q inp, a, S_1 q (project-inp inp), a, dir.L)) (q,inp,TUPLE
t) ∈ δ' unfolding δ'-def by blast
  hence mem: (S q inp, TUPLE t, S_1 q (project-inp inp), TUPLE t, dir.L) ∈ δ'
by simp
  let ?cs1 = Configs_S (S_1 q (project-inp inp)) tc' p'
  have (cs0,?cs1) ∈ st.dstep unfolding cs0 using inp
    by (intro st.dstepI[OF mem], auto simp: tc' δ'-def blank'-def)
  moreover have rel-S_1 ?cs1 cm unfolding cm
  proof (intro rel-S_1.intros 1 pip)
    fix i
    from inp have inp i ∈ SYM ‘ Γ by auto
    then obtain g where inp i = SYM g and g ∈ Γ by auto
    thus project-inp inp i = tc i (p i) using 1(6)[of i] by (auto simp: project-inp-def
split: if-splits)
    show p' = Suc (Max (range p)) unfolding assms(2) unfolding cm by simp
  qed
  ultimately show ?thesis by auto
qed
```

If we start the S-phase (in *rel-*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-*S_1).

```
lemma S-phase: assumes rel-S_0 cs cm
  and mt-state cm ∉ {t, r}
shows ∃ cs'. (cs, cs') ∈ st.dstep^^(3 + Max (range (mt-pos cm))) ∧ rel-S_1 cs'
cm
proof −
  let ?N = Max (range (mt-pos cm))
  from rel-S_0-S[OF assms] obtain cs1 n where
    step1: (cs, cs1) ∈ st.dstep and rel: rel-S cs1 cm n and n: n = 0
    by auto
  from rel have n + k ≤ Suc ?N ⟹ ∃ cs2. (cs1, cs2) ∈ st.dstep ^^ k ∧ rel-S
cs2 cm (n + k) for k
  proof (induction k arbitrary: cs1 n)
    case (Suc k cs n)
    hence n ≤ ?N by auto
    from rel-S-S[OF Suc(3) this] obtain cs1 where step: (cs, cs1) ∈ st.dstep and
rel: rel-S cs1 cm (Suc n) by auto
    from Suc have Suc n + k ≤ Suc ?N by auto
```

from *Suc.IH*[*OF this rel*] **obtain** *cs2* **where** *steps*: (*cs1*, *cs2*) ∈ *st.dstep* ^^ *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*) ∈ *st.dstep* ^^ (*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*) ∈ *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*) ∈ *st.dstep* ^^ *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 *δstep* :: (*'a*, *'q*, *'k*) *mt-config rel* **where**
δstep: *rule* = (*q*, *a*, *q1*, *b*, *dir*) ⟹
rule ∈ *δ* ⟹
(⋀ *k*. *ts k* (*n k*) = *a k*) ⟹
(⋀ *k*. *ts' k* = (*ts k*)(*n k* := *b k*)) ⟹
(⋀ *k*. *n' k* = *go-dir* (*dir k*) (*n k*)) ⟹
(*Config*$_M$ *q ts n*, *Config*$_M$ *q1 ts' n'*) ∈ *δstep*
end

lemma *step-to-δstep*: (*c1*,*c2*) ∈ *step* ⟹ ∃ *rule*. (*c1*,*c2*) ∈ *δstep rule*
proof (*induct rule*: *step.induct*)
case (*step q ts n q' a dir*)
show *?case*
by (*rule exI*, *rule δstep.intros*[*OF refl step*], *auto*)
qed

lemma *δstep-to-step*: (*c1*,*c2*) ∈ *δstep rule* ⟹ (*c1*,*c2*) ∈ *step*
proof (*induct rule*: *δstep.induct*)
case ∗: (*δstep q a q' b dir ts n ts' n'*)
from ∗ **have** *a*: *a* = (λ *k*. *ts k* (*n k*)) **by** *auto*
from ∗ **have** *ts'*: *ts'* = (λ *k*. (*ts k*)(*n k* := *b k*)) **by** *auto*
from ∗ **have** *n'*: *n'* = (λ *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*
⇒ (*'a*, *'q*, *'k*) *mt-config* ⇒ (*'a*, *'q*, *'k*) *mt-config* ⇒ (*'a*,*'q*,*'k*)*mt-rule* ⇒ *bool* **where**

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

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 $rel\text{-}S_1\text{-}E_0\text{-}step$: **assumes** $rel\text{-}S_1\ cs\ cm$
  and $(cm,cm1) \in step$
shows $\exists\ rule\ cs1.\ (cs,\ cs1) \in st.step \wedge rel\text{-}E_0\ cs1\ cm\ cm1\ rule$
proof −
  from $step\text{-}to\text{-}\delta step[OF\ assms(2)]$ **obtain** *rule* **where** *rstep*: $(cm,\ cm1) \in \delta step$ *rule* **by** *auto*
  show *?thesis* **using** *assms(1)*
  proof (*cases rule*: $rel\text{-}S_1.cases$)
    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*: $\delta step.cases$)
      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 $?cs1 = Configs_S\ (E_0\ q1\ b\ dir)\ tc'\ p'$
      have *step*: $(cs,\ ?cs1) \in st.step$ **unfolding** *cs*
        by (*intro st.stepI*[*OF mem*], *insert inp-a*, *auto*)
      moreover have $rel\text{-}E_0\ ?cs1\ cm\ cm1\ rule$ **unfolding** *cm cm1*
        by (*intro* $rel\text{-}E_0.intros$[*OF - - - rule*], *insert 1 2 assms rstep*, *auto*)
      ultimately show *?thesis* **by** *blast*
    qed
  qed
qed

lemma $rel\text{-}S_1\text{-}E_0\text{-}st\text{-}step$: **assumes** $rel\text{-}S_1\ cs\ cm$
  and $(cs,cs1) \in st.step$

shows ∃ *cm1 rule*. (*cm*, *cm1*) ∈ *step* ∧ *rel-*E_0 *cs1 cm cm1 rule*
  using *assms*(*1*)
proof (*cases rule*: *rel-*S_1*.cases*)
  case (*1 tc′ q tc p inp p′*)
  note *cs* = *1*(*1*)
  note *cm* = *1*(*2*)
  have *tc′*: *tc′ p′* ∈ Γ′ **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* δ′*-def*]
    have (S_1 *q inp*, *tc′ p′*, *qq*, *bb*, *ddir*) ∈ (λ((*q*, *a*, *q′*, *b*, *d*), *t*). (S_1 *q a*, *t*, E_0 *q′ b d*, *t*, *dir.N*)) ‘ (δ × Γ′) **by** *auto*
    then obtain *q′ b dir* **where** *mem*: (*q*,*inp*,*q′*,*b*,*dir*) ∈ δ **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′* (λ *k*. (*tc k*)(*p k* := *b k*)) (λ *k*. *go-dir* (*dir k*) (*p k*))
    have δ*step*: (*cm*, *?cm1*) ∈ δ*step ?rule* **unfolding** *cm*
      by (*intro* δ*step.intros*[*OF refl mem*], *auto simp*: *1*(*6*))
    from δ*step-to-step*[*OF this*]
    have (*cm*, *?cm1*) ∈ *step* **.**
    moreover have *rel-*E_0 *cs1 cm ?cm1 ?rule* **unfolding** *cm cs1*
      by (*intro rel-*E_0*.intros* δ*step*[*unfolded cm*], *insert 1 2*, *auto*)
    ultimately show *?thesis* **by** *blast*
  qed
qed

fun *enc2* :: (*′a*, *′q*, *′k*) *mt-config* ⇒ (*′a*, *′q*, *′k*) *mt-config* ⇒ *nat* ⇒ *nat* ⇒ (*′a*, *′k*) *st-tape-symbol*
  where *enc2* (*Config*$_M$ *q tc p*) (*Config*$_M$ *q1 tc1 p1*) *p′ n* = *TUPLE* (λ *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*
  ⇒ (*′a*, *′q*, *′k*) *mt-config* ⇒ (*′a*, *′q*, *′k*) *mt-config* ⇒ (*′a*,*′q*,*′k*)*mt-rule* ⇒ *nat* ⇒ *bool* **where**
  tc′ 0 = ⊢ ⟹
  (⋀ *i*. *tc′* (*Suc i*) = *enc2* (*Config*$_M$ *q tc p*) (*Config*$_M$ *q1 tc1 p1*) *p′ i*) ⟹
  valid-config (*Config*$_M$ *q tc p*) ⟹
  rule = (*q*,*a*,*q1*,*b*,*d*) ⟹
  (*Config*$_M$ *q tc p*, *Config*$_M$ *q1 tc1 p1*) ∈ δ*step rule* ⟹
  bo = (λ *k*. *if p k* < *p′ then SYM* (*b k*) *else* •) ⟹
  rel-E (*Config*$_S$ (*E q1 bo d*) *tc′ p′*) (*Config*$_M$ *q tc p*) (*Config*$_M$ *q1 tc1 p1*) *rule p′*

lemma rel-E_0-E: **assumes** rel-E_0 cs cm cm1 rule
  shows ∃ cs1. (cs, cs1) ∈ st.dstep ∧ rel-E cs1 cm cm1 rule (Suc (Max (range (mt-pos cm))))
  using assms(1)
proof (cases rule: rel-E_0.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′) ∈ (Q × (UNIV → Γ) × UNIV) × Γ′
    using elem tc′ **by** auto
  hence (λ((q, a, d), t). (E_0 q a d, t, E q (SYM ∘ a) d, t, dir.N)) ((q1,b,d), tc′ p′) ∈ δ′
    unfolding δ′-def **by** blast
  hence mem: (E_0 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 = $Configs_S$ (E q1 (SYM ∘ b) d) tc′ p′
  have (cs, ?cs1) ∈ 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-S_0: **assumes** rel-E cs cm cm1 rule 0
  shows ∃ cs1. (cs,cs1) ∈ st.dstep ∧ 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, ⊢, S_0 q1, ⊢, dir.N) ∈ δ′ **unfolding** δ′-def **by** auto
  let ?cs1 = $Configs_S$ (S_0 q1) tc′ 0
  have (cs, ?cs1) ∈ st.dstep **unfolding** cs
    by (intro st.dstepI[OF mem], insert 1, auto simp: δ′-def)
  moreover have rel-S_0 ?cs1 cm1 **unfolding** cm1
    by (intro rel-S_0.intros valid, insert 1, auto)

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

lemma *dsteps-to-steps*: $a \in st.dstep$ ^^ $n \Longrightarrow a \in st.step$ ^^ n
using *relpow-mono*[*OF st.dstep-step*] **by** *auto*

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

lemma *rel-E-E*: **assumes** *rel-E cs cm cm1 rule* (*Suc p*′)
shows ∃ *cs1*. (*cs*,*cs1*) ∈ *st.dstep*^^*4* ∧ *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*) *δstep-to-step valid-step* **by** *blast*
have *trans*: (λ(*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*)) ‘ (*Q* × *func-set* × *UNIV* × *gamma-set*) ⊆ δ′
(λ(*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*))
‘ (*Q* × *func-set* × *UNIV* × *UNIV* × *gamma-set*) ⊆ δ′
(λ(*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*))
‘ (*Q* × *func-set* × *UNIV* × *UNIV* × *gamma-set*) ⊆ δ′
(λ(*q*, *ys*, *ds*, *I*). (*El q ys ds I*, ⊢, *E q ys ds*, ⊢, *dir.N*))
‘ (*Q* × *func-set* × *UNIV* × *Pow UNIV*) ⊆ δ′
(λ (*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*))
‘ (*Q* × *func-set* × *UNIV* × *UNIV* × *gamma-set*) ⊆ δ′
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* ∈ *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* ∈ *func-set*
using *1* δ(*4*) *δstep.simps* **unfolding** *func-set-def* **by** *fastforce*
have *rmem*: *?rule* ∈ δ **using** *1*(*8*,*7*) **by** (*cases rule*: *δstep.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' = update-ys tup bo
  let ?er = Er q1 ?ys' d ?I
  let ?cs1 = Configs_S ?er ?tc1 (Suc(Suc p'))
  have (q1,bo,d,tup) ∈ Q × func-set × UNIV × gamma-set using elem tup-mem
bo-set by blast
  hence mem: (E q1 bo d, tc' (Suc p'), ?er, ?rep-tup, dir.R) ∈ δ'
    using δ'-mem[OF - trans(1)] unfolding split tup by fast
  note dstep-help = st.dstep[of E q1 bo d tc' (Suc p') ?er ?rep-tup dir.R]
  have to-r: (Configs_S (E q1 bo d) tc' (Suc p'), ?cs1) ∈ st.dstep
    using dstep-help[OF mem] unfolding δ'-def go-dir.simps tup by fast


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


  have (q1,?ys',d,?I,replace-sym tup bo) ∈ Q × func-set × UNIV × UNIV ×
gamma-set
      using update-ys-range[OF tup-mem bo-set] replace-sym-range[OF tup-mem
bo-set] elem(3) by blast
   hence mem3: (?em, ?tc2 (Suc p'), El q1 ?ys' d ?I, TUPLE (place-hats-M
(replace-sym tup bo) d ?I), dir.L) ∈ δ'
    using δ'-mem[OF - trans(3)] by auto
  note dstep-help3 = st.dstep[of ?em ?tc2 Suc p' El q1 ?ys' d ?I TUPLE (place-hats-M
(replace-sym tup bo) d ?I) dir.L]
  let ?el = El q1 (update-ys tup bo) d (compute-idx-set tup bo)
  let ?tc3 = 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)))
  let ?cs3 = Configs_S ?el ?tc3 p'
  have to-l: (?cs2, ?cs3) ∈ st.dstep
    using dstep-help3[OF mem3] unfolding δ'-def go-dir.simps by fastforce
```

have *steps3*: (*cs*, *?cs3*) ∈ *st.dstep* ^^ *3* **unfolding** *numeral-3-eq-3* **using** *to-l to-m to-r 1*(*1*) **by** *auto*

have *tc1-def*: ⋀ *k*. *tc1 k* = (*tc k*)(*p k* := *b k*)
  using *1*(*8*) **unfolding** *1*(*7*) **by** (*simp add*: *δstep.simps old.prod.inject*)
have *p1-def*: ⋀ *k*. *p1 k* = *go-dir* (*d k*) (*p k*)
  using *1*(*8*) **unfolding** *1*(*7*) **by** (*simp add*: *δstep.simps old.prod.inject*)
have *not-I-mem-current-pos*: ∀ *k'*. *k'* ∉ *?I* ⟶ *p k'* ≠ *p'*
  by(*intro allI impI*, *insert tup elem 1*(*6*), *fastforce simp*: *1 compute-idx-set-def*)
have *I-mem-current-pos*: ∀ *k*. *k* ∈ *?I* ⟶ *p k* = *p'*
  using *compute-idx-set-def 1*(*5*,*9*) *tup* **by**(*simp*, *fastforce*)
have *I-mem-eq-cur-pos*: ∀ *k*. *k* ∈ *?I* ⟷ *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'* = ⊢ **using** *1*(*4*) **by** *simp*
  have (*q1*,*?ys'*,*d*,*?I*) ∈ *Q* × *func-set* × *UNIV* × *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*, ⊢, *dir.N*) ∈ *δ'*
    using *δ'-mem*[*OF - trans*(*4*)] **unfolding** *tc3-tup* **by** *auto*
  let *?tc4* = *?tc3*(*p'* := ⊢)
  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* ⊢ *dir.N*]
  have (*?cs3*, *?cs4*) ∈ *st.dstep* ^^ *1*
    using *dstep-help4*[*OF mem4*] **unfolding** *δ'-def go-dir.simps relpow-1 tc3-tup* **by** *fastforce*
  hence *steps*: (*cs*, *?cs4*) ∈ *st.dstep* ^^ *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*: (∀ *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' := ⊢)) (*Suc i*) = *enc2* (*Config*$_M$ *q tc p*) (*Config*$_M$ *q1 tc1 p1*) *p' i*)
  proof
    fix *i*
    consider (*zer*) *i* = *0* | (*suc*) *i* = *Suc 0* | (*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' := ⊢)) (*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 ⇒ if k ∈ compute-idx-set tup bo ∧ d k = dir.N then HAT a else
NO-HAT a
        | HAT x ⇒ HAT x) = (if p1 k = 0 then HAT (tc1 k 0) else NO-HAT (tc1
k 0)) for k
        proof(cases k ∈ ?I)
          case k-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-zero: p k = 0
              using k-in-I less-Suc0 p-zero unfolding compute-idx-set-def 1(9) by
fastforce
          show ?thesis
          proof(cases d k = dir.N)
            case False
            moreover have tc k (p k) = LE
              using 1(6) pk-zero 1(8) by auto
            moreover have  tc k (p k) = a k
              using 1(7,8) pk-zero unfolding δstep.simps by blast
            ultimately have d k = dir.R
              using δLE[OF rmem] by auto
            then show ?thesis
                by(insert k-in-I p1-def tc1-def, simp, insert rep-no-hat, auto simp:
replace-sym-def 1(9) p-zero pk-zero)
           qed(insert k-in-I rep-no-hat pk-zero 1(9), auto simp:  replace-sym-def
tc1-def p1-def)
        next
          case False
          show ?thesis
            using tup False 1(5) p-zero not-I-mem-current-pos p1-def tc1-def
            unfolding p-zero replace-sym-def by fastforce
        qed
        then show ?thesis
          unfolding zer p-zero place-hats-M-def place-hats-to-dir-def by simp
      next
        case suc
        have (case tup2 k of
        NO-HAT a ⇒ if k ∈ compute-idx-set tup bo ∧ d k = dir.R then HAT a else
NO-HAT a
         | HAT x ⇒ HAT x) = (if p1 k = Suc 0 then HAT (tc1 k (Suc 0)) else
NO-HAT (tc1 k (Suc 0))) for k
          using tup2 p-zero elem(4) 1(5,6,9) gr-zeroI tm(4) tup
       by (auto simp: compute-idx-set-def tc1-def p1-def, smt (verit, best) diff-0-eq-0
go-dir.elims not-gr0)
      then show ?thesis unfolding suc p-zero place-hats-R-def place-hats-to-dir-def
by simp
      next
        case ge-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: ⋀ k. p k = 0 ⟶ tup k ∈ HAT ` Γ
      using tup unfolding p-zero 1(5)[of 0] enc2.simps by (simp, meson imageI
tup-hat-content tup-mem)
    have (if k ∈ compute-idx-set tup bo then • else bo k) = • for k
      using only-hats elem(4) 1(9) by (auto simp: compute-idx-set-def p-zero)
    hence replaced-all: update-ys tup bo = (λk. if p k < p' then SYM (b k) else •)
      unfolding update-ys-def p-zero by auto
    have invs: rel-E ?cs4 cm cm1 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' = TUPLE tup3 and
      tup3-def: tup3 = (λk. if p k < Suc (Suc nat) then if p k = nat then HAT (tc
k nat) else NO-HAT (tc k nat)
        else if p1 k = nat then HAT (tc1 k nat) else NO-HAT (tc1 k nat))
      using 1(5)[of nat] unfolding Suc enc2.simps by simp
    have tup3-mem: tup3 ∈ 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 = TUPLE (place-hats-L tup3 d (compute-idx-set tup bo))
    let ?tc4 = ?tc3(p' := ?a4)
    let ?cs4 = Config_S (E q1 ?ys' d) ?tc4 p'
     have step-mem:(q1, ?ys', d, ?I, tup3) ∈ Q × func-set × UNIV × UNIV ×
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: (El q1 ?ys' d ?I, ?tc3 p', E q1 ?ys' d, TUPLE (place-hats-L tup3
d (compute-idx-set tup bo)), dir.N) ∈ δ'
      using δ'-mem[OF step-mem trans(5)] unfolding split tc3-p' .
    note dstep-help5 = st.dstep[of El q1 ?ys' d ?I ?tc3 p' E q1 ?ys' d ?a4 dir.N]
   note intros-helper2 = rel-E.intros[of ?tc4 q tc p q1 tc1 p1 p' rule a b d update-ys
tup bo]
    have correct-shift: (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' := TUPLE (place-hats-L tup3 d (compute-idx-set tup bo))))
        (Suc i) = enc2 (Config_M q tc p) (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 ∉ {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 ⇒ if k ∈ compute-idx-set tup bo ∧ d k = dir.L then HAT a
else NO-HAT a
          | HAT x ⇒ 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 ⇒ if k ∈ compute-idx-set tup bo ∧ d k = dir.N then HAT
a else NO-HAT a
                | HAT x ⇒ 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 ⇒ if k ∈ compute-idx-set tup bo ∧ d k = dir.R then HAT
a else NO-HAT a
                | HAT x ⇒ 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 ^^ 1
      using mem5 dstep-help5[OF mem5]
      unfolding δ'-def go-dir.simps relpow-1 Suc by fastforce
    hence (cs, ?cs4) ∈ st.dstep ^^ 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 ^^ (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^^(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.prems] obtain cs' where
      step4: (cs1, cs') ∈ st.dstep ^^ 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 ^^ (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^^(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 ^^ 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′*) ∈ *step*
shows ∃ *cs′*. (*cs*,*cs′*) ∈ *st.step* ^^ (*10* + *5* ∗ *Max* (*range* (*mt-pos cm*))) ∧ *rel-*S_0
cs′ cm′
proof −
let *?n* = *Max* (*range* (*mt-pos cm*))
from *assms*(*2*) **have** *mt-state cm* ∉ {*t*, *r*} **using** *δ-set*
by (*cases*, *auto*)
from *S-phase*[*OF assms*(*1*) *this*] **obtain** *cs1* **where**
steps1: (*cs*, *cs1*) ∈ *st.dstep* ^^ (*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*) ∈ *st.step* **and** *rel*: *rel-*E_0 *cs2 cm cm′ r*
by *auto*
from *E-phase*[*OF rel*] **obtain** *cs′* **where**
steps3: (*cs2*, *cs′*) ∈ *st.dstep* ^^ (*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′*) ∈ *st.step* ^^ ((*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*)) ≤ *N*
and (*cm*, *cm′*) ∈ *step*^^*n*
shows ∃ *m cs′*. (*cs*,*cs′*) ∈ *st.step*^^*m* ∧ *rel-*S_0 *cs′ cm′* ∧ *m* ≤ *sum* (*λ i*. *10* + *5*
∗ (*N* + *i*)) {..< *n*} ∧ *Max* (*range* (*mt-pos cm′*)) ≤ *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″*) ∈ *step*^^*n* **and** *step*: (*cm″*, *cm′*) ∈
step **by** *auto*
from *Suc*(*1*)[*OF this*(*1*) *Suc*(*3*−*4*)] **obtain** *m cs″* **where**
steps: (*cs*, *cs″*) ∈ *st.step* ^^ *m* **and** *m*: *m* ≤ ($\sum$ *i* < *n*. *10* + *5* ∗ (*N* + *i*)) **and**
rel: *rel-*S_0 *cs″ cm″* **and** *max*: *Max* (*range* (*mt-pos cm″*)) ≤ *N* + *n*
by *auto*

```
  from step-simulation[OF rel step] obtain cs' where
    steps2: (cs'', cs') ∈ st.step ^^ (10 + 5 ∗ Max (range (mt-pos cm''))) and rel:
rel-S_0 cs' cm' by auto
  let ?m = m + (10 + 5 ∗ Max (range (mt-pos cm'')))
  from relpow-transI[OF steps steps2]
  have steps: (cs, cs') ∈ st.step ^^ ?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')) ≤ N + Suc n by linarith
    have id: {..<Suc n} = insert n {..< n} by auto
    have ?m ≤ m + (10 + 5 ∗ (N + n)) using max by presburger
    also have ... ≤ (∑ i < Suc n. 10 + 5 ∗ (N + i)) using m unfolding id by
auto
    finally show ?m ≤ (∑ i < Suc n. 10 + 5 ∗ (N + i)) by auto
  qed
qed

lemma steps-simulation-rel-S_0: assumes rel-S_0 cs (init-config w)
  and (init-config w, cm') ∈ step^^n
shows ∃ m cs'. (cs,cs') ∈ st.step^^m ∧ rel-S_0 cs' cm' ∧ m ≤ 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') ∈ st.step ^^ m and rel: rel-S_0 cs' cm' and
m: m ≤ (∑ i<n. 10 + 5 ∗ i)
    by auto
  have m ≤ (∑ i<n. 10 + 5 ∗ i) by fact
  also have ... ≤ 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 ⊆ Σ
  and steps: (init-config w, Config_M q mtape p) ∈ step^^n
shows ∃ stape k. (st.init-config (map INP w), Config_S (S_0 q) stape 0) ∈ st.step^^k
∧ k ≤ 2 ∗ length w + 3 ∗ n^2 + 7 ∗ n + 3
proof −
  let ?INP = INP :: 'a ⇒ ('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) ∈ st.dstep ^^ (3 + 2 ∗ length w) and rel: rel-S_0
cs ?initm
    by auto
  from steps-simulation-rel-S_0[of - w, OF rel steps] obtain k' cs' where
    steps2: (cs, cs') ∈ st.step^^k' and
    rel: rel-S_0 cs' (Config_M q mtape p) and
    k': k' ≤ 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 ^^ ?k .
  from rel obtain stape where cs': cs' = Config_S (S_0 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,
Config_M t w' n) ∈ step^* by (auto simp: cm-def)
  from rtrancl-imp-relpow[OF steps] obtain num where steps: (cm, Config_M t w'
n) ∈ step^^num by auto
  from simulation-with-complexity[OF w, folded cm-def, OF steps] obtain stape
    where steps: (st.init-config (map INP w), Config_S (S_0 t) stape 0) ∈ st.step^*
    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, Config_S (S_0 t) ts p) ∈ st.step^* by force
  let ?NF = Config_S (S_0 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 ^^ (3 + 2 * length w) and rel: rel-S_0
cs cm
    by (auto simp: cm-def)
```

```
 from steps obtain k where (st.init-config x, ?NF) ∈ st.step^^k using rtrancl-power
by blast
  from st.dsteps-inj[OF dsteps this NF] obtain n where ssteps: (cs, ?NF) ∈
st.step ^^ n by auto
 from rel ssteps have ∃ cm'. (cm,cm') ∈ step^* ∧ 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 ∈ {t, r})
     case True
      with rel obtain ts' p' q where cs: cs = Configs_S (S_0 q) ts' p' and q: q ∈
{t,r}
       by (cases, auto)
     have NF: False if (cs, cs') ∈ 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) ∈ st.dstep
^^ (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) ∈ st.step ^^ k1 by auto
     from rel have cs1 ≠ ?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) ∈ st.step and steps: (cs2,?NF) ∈ st.step ^^ k by auto
    from rel-S_1-E_0-st-step[OF rel step] obtain cm1 rule where mstep: (cm, cm1)
∈ step and rel: rel-E_0 cs2 cm cm1 rule by auto
     from E-phase[OF rel] obtain cs3 where dsteps: (cs2, cs3) ∈ st.dstep ^^ (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, Configs_S (S_0 t) ts p) ∈ st.step ^^ 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') ∈
step^* and rel: rel-S_0 ?NF cm' by auto
     from mstep msteps have (cm, cm') ∈ step^* by auto
```

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

6.2.6 Main Results

```
theorem language-equivalence: st.Lang = map INP ` Lang
  using simulation rev-simulation by auto

theorem upper-time-bound-quadratic-increase: assumes upper-time-bound f
  shows st.upper-time-bound (λ 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* ' Σ **and** *steps*: (*st.init-config ww*, *c*) $\in$ *st.step* ^^ *n*
and *bnd*: $\neg$ *n* $\leq$ *3* $*$ (*f* (*length ww*))^2 + *13* $*$ (*f* (*length ww*)) + *2* $*$ *length ww* + *12*
from *INP-D*[*OF this*(*1*)] **obtain** *w* **where** *w*: *set w* $\subseteq$ Σ **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-simulation-complexity*[*OF w steps*[*unfolded ww*] *this*]
obtain *cm* **where** (*init-config w*, *cm*) $\in$ *step* ^^ (*f ?lw* + *1*) **by** *auto*
from *assms*[*unfolded upper-time-bound-def*, *rule-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-tm tm* $\wedge$
Lang-mttm mttm = *Lang-tm tm* $\wedge$
(*det-mttm mttm* $\longrightarrow$ *det-tm tm*) $\wedge$
(*upperb-time-mttm mttm f* $\longrightarrow$ *upperb-time-tm tm* (λ *n*. *3* $*$ (*f n*)^*2* + *13* $*$ *f n* + *2* $*$ *n* + *12*))
proof (*cases mttm*)
let *?INP* = *INP* :: '*a* $\Rightarrow$ ('*a*, '*k*) *st-tape-symbol*
case (*MTTM Q* Σ Γ *bl le* δ *s t r*)
from *assms*(*1*)[*unfolded this*]
interpret *multitape-tm Q* Σ Γ *bl le* δ *s t r* **by** *simp*
let *?TM1* = *TM Q'* (*?INP* ' Σ) Γ' *blank'* $\vdash$ δ' (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* ' Σ Γ' *blank'* $\vdash$ δ' 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-states* $\Rightarrow$ '*q*). *inj-on tq Q'* **by** (*metis finite-infinite-inj-on*)
then obtain *tq* :: - $\Rightarrow$ '*q* **where** *inj-on tq Q'* **by** *blast*
hence *tq*: *inj-on tq* (*Q-tm ?TM1*) **by** *simp*

```
  from st.fin-Γ have finG': finite Γ' .
  from fin-Σ assms(3) have infinite (UNIV − Σ) by auto
  then obtain B :: 'a set where B: finite B card B = card Γ' B ⊆ UNIV − Σ
    by (meson infinite-arbitrarily-large)
  from st.finΣ have finS': finite (?INP ` Σ) .
  from finG' B obtain ta' where bij: bij-betw ta' Γ' B
    by (metis bij-betw-iff-card)
  define ta where ta x = (if x ∈ ?INP ` Σ then (case x of INP y ⇒ y) else ta' x)
for x
  have ta: inj-on ta (Γ-tm ?TM1) using bij B(3)
    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 ` Lang-mttm mttm
    and det: st.deterministic ⟹ det-tm tm'
    and upper: ⋀ f. st.upper-time-bound f ⟹ upperb-time-tm tm' f
    using tm-renaming[OF valid ta tq, unfolded id] by auto
   note lang also have map ta ` map INP ` Lang-mttm mttm = (λ w. w) `
Lang-mttm mttm
    unfolding image-comp o-def map-map
  proof (rule image-cong[OF refl])
    fix w
    assume w ∈ Lang-mttm mttm
    hence w: set w ⊆ Σ unfolding Lang-def MTTM Lang-mttm.simps by auto
    show map (λx. 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' (λn. 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.