A Verified Translation of Multitape Turing Machines into Singletape Turing Machines

Christian Dalvit and René Thiemann

May 26, 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*⟩

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

lemma *fin-funcsetI*[*intro*]: *finite A* ⟹ *finite* ((*UNIV* :: *'a* :: *finite set*) → *A*)
  ⟨*proof*⟩

lemma *finite-UNIV-fun-dir*[*simp*,*intro*]: *finite* (*UNIV* :: (*'k* :: *finite* ⇒ *dir*) *set*)
  ⟨*proof*⟩

lemma *relpow-transI*: (*x*,*y*) ∈ *R*^^*n* ⟹ (*y*,*z*) ∈ *R*^^*m* ⟹ (*x*,*z*) ∈ *R*^^(*n*+*m*)

⟨*proof*⟩

lemma *relpow-mono*: **fixes** R :: *'a rel* **shows** $R \subseteq S \Longrightarrow R\hat{}\hat{}n \subseteq S\hat{}\hat{}n$
⟨*proof*⟩

lemma *finite-infinite-inj-on*: **assumes** *A*: *finite* (*A* :: *'a set*) **and** *inf*: *infinite* (*UNIV* :: *'b set*)
shows $\exists$ *f* :: *'a* ⇒ *'b*. *inj-on f A*
⟨*proof*⟩

lemma *gauss-sum-nat2*: $(\sum i< (n :: nat).\ i) = (n - 1) * n\ div\ 2$
⟨*proof*⟩

lemma *aux-sum-formula*: $(\sum i<n.\ 10 + 5 * i) \leq 3 * n\hat{}2 + 7 * (n :: nat)$
⟨*proof*⟩

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* ⇒ *'whatever* for some finite index type *'k*.

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

datatype (*'q*,*'a*,*'k*)*mttm* = *MTTM*
(*Q-tm*: *'q set*)
'a set
(Γ-*tm*: *'a set*)
'a
'a
(*'q* × (*'k* ⇒ *'a*) × *'q* × (*'k* ⇒ *'a*) × (*'k* ⇒ *dir*)) *set*
'q
'q
'q

datatype (*'a*,*'q*,*'k*) *mt-config* = $Config_M$
(*mt-state*: *'q*)
'k ⇒ *nat* ⇒ *'a*
(*mt-pos*: *'k* ⇒ *nat*)

locale *multitape-tm* =
fixes
Q :: $'q$ *set* **and**
Σ :: $'a$ *set* **and**
Γ :: $'a$ *set* **and**
blank :: $'a$ **and**
LE :: $'a$ **and**
δ :: $('q \times ('k \Rightarrow 'a) \times 'q \times ('k \Rightarrow 'a) \times ('k :: \{finite,zero\} \Rightarrow dir))$ *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*: $\delta \subseteq (Q - \{t,r\}) \times (UNIV \rightarrow \Gamma) \times Q \times (UNIV \rightarrow \Gamma) \times (UNIV \rightarrow UNIV)$ **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 $q \in Q$ $a\ k \in \Gamma$ $q' \in Q$ $b\ k \in \Gamma$
⟨*proof*⟩

lemma *fin-*Σ: *finite* Σ
⟨*proof*⟩

lemma *fin-*δ: *finite* δ
⟨*proof*⟩

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

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

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

lemma *valid-init-config*: $set\ w \subseteq \Sigma \Longrightarrow valid\text{-}config\ (init\text{-}config\ w)$
⟨*proof*⟩

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

lemma *valid-step*: **assumes** *step*: $(\alpha,\beta) \in step$
and *val*: *valid-config* α
shows *valid-config* β
⟨*proof*⟩

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

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

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

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

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

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

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

end

4 Singletape Turing Machines

theory *Singletape-TM*
imports
TM-Common
begin

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

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

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

locale *singletape-tm* =
  fixes
    $Q :: 'q\ set$ **and**
    $\Sigma :: 'a\ set$ **and**
    $\Gamma :: 'a\ set$ **and**
    $blank :: 'a$ **and**
    $LE :: 'a$ **and**
    $\delta :: ('q \times 'a \times 'q \times 'a \times dir)\ set$ **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**
    δLE: $(q,\ LE,\ q',\ a',\ d) \in \delta \Longrightarrow a' = LE \wedge d \in \{dir.N,dir.R\}$
begin

lemma δ: **assumes** $(q,a,q',b,d) \in \delta$
  shows $q \in Q\ a \in \Gamma\ q' \in Q\ b \in \Gamma$
  ⟨*proof*⟩

lemma *fin*Σ: *finite* Σ

⟨*proof*⟩

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

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

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

lemma *valid-init-config*: $set\ w \subseteq \Sigma \Longrightarrow valid\text{-}config\ (init\text{-}config\ w)$
⟨*proof*⟩

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

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

lemma *valid-step*: **assumes** *step*: $(\alpha,\beta) \in step$
and *val*: $valid\text{-}config\ \alpha$
shows $valid\text{-}config\ \beta$
⟨*proof*⟩

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

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

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

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

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

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

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

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

lemma *dstepI*: $(q,\ a,\ q',\ b,\ dir) \in \delta \Longrightarrow ts\ n = a \Longrightarrow ts' = ts(n := b) \Longrightarrow n' =$ $go\text{-}dir\ dir\ n \Longrightarrow q1 = q \Longrightarrow q2 = q'$
$\Longrightarrow (\bigwedge q''\ b'\ dir'.\ (q,\ a,\ q'',\ b',\ dir') \in \delta \Longrightarrow (q'',\ b',\ dir') = (q',b,dir))$
$\Longrightarrow (Config_S\ q1\ ts\ n,\ Config_S\ q2\ ts'\ n') \in dstep$
⟨*proof*⟩

lemma *dstep-step*: $dstep \subseteq step$
⟨*proof*⟩

lemma *dstep-inj*: **assumes** $(x,y) \in dstep$
and $(x,z) \in step$
shows $z = y$
⟨*proof*⟩

lemma *dsteps-inj*: **assumes** $(x,y) \in dstep$^^n
and $(x,z) \in step$^^m
and $\neg (\exists\ u.\ (z,u) \in step)$
shows $\exists\ k.\ m = n + k \wedge (y,z) \in step$^^$k$
⟨*proof*⟩

lemma *dsteps-inj'*: **assumes** $(x,y) \in dstep$^^n
and $(x,z) \in step$^^m
and $m \geq n$
shows $\exists\ k.\ m = n + k \wedge (y,z) \in step$^^$k$
⟨*proof*⟩
end
end

5 Renamings for Singletape Turing Machines

theory *STM-Renaming*

imports
Singletape-TM
begin

locale *renaming-of-singletape-tm* = *singletape-tm* $Q\ \Sigma\ \Gamma$ *blank LE* 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*⟩

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*)
⟨*proof*⟩

lemma *ren-step*: **assumes** (*c*,*c'*) ∈ *step*
shows (*rc c*, *rc c'*) ∈ *ren.step*
⟨*proof*⟩

lemma *ren-steps*: **assumes** (*c*,*c'*) ∈ *step*^*
shows (*rc c*, *rc c'*) ∈ *ren.step*^*
⟨*proof*⟩

lemma *ren-steps-count*: **assumes** (*c*,*c'*) ∈ *step*^^*n*
shows (*rc c*, *rc c'*) ∈ *ren.step*^^*n*
⟨*proof*⟩

lemma *ren-Lang-forward*: **assumes** *w* ∈ *Lang*
shows *map ra w* ∈ *ren.Lang*
⟨*proof*⟩

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*
⟨*proof*⟩

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 ⟨*proof*⟩

lemma *inv-ren-Gamma*: *ira* ‘ *ra* ‘ Γ = Γ ⟨*proof*⟩

lemma *inv-ren-t*: *irq* (*rq t*) = *t* ⟨*proof*⟩
lemma *inv-ren-s*: *irq* (*rq s*) = *s* ⟨*proof*⟩
lemma *inv-ren-r*: *irq* (*rq r*) = *r* ⟨*proof*⟩
lemma *inv-ren-blank*: *ira* (*ra blank*) = *blank* ⟨*proof*⟩
lemma *inv-ren-LE*: *ira* (*ra LE*) = *LE* ⟨*proof*⟩

lemma *inv-ren-δ*: *inv.rd* ‘ *rd* ‘ δ = δ
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *ren-Lang-backward*: **assumes** *v* ∈ *ren.Lang*
  shows ∃ *w*. *v* = *map ra w* ∧ *w* ∈ *Lang*
⟨*proof*⟩

lemma *ren-Lang*: *ren.Lang* = *map ra* ‘ *Lang*
⟨*proof*⟩

lemma *ren-det*: **assumes** *deterministic*
  shows *ren.deterministic*
  ⟨*proof*⟩

lemma *ren-upper-time*: **assumes** *upper-time-bound f*
  shows *ren.upper-time-bound f*
  ⟨*proof*⟩

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*⟩

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* ‘ $\Gamma \cup$ *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*) → *SYM* ‘ Γ ∪ {•}

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

definition *add-inp* :: (*'k* ⇒ *'a tuple-symbol*) ⇒ (*'k* ⇒ *'a sym-or-bullet*) ⇒ (*'k* ⇒ *'a sym-or-bullet*) **where**
add-inp inp inp2 = (λ *k*. *case inp k of HAT s* ⇒ *SYM s* | - ⇒ *inp2 k*)

definition *project-inp* :: (*'k* ⇒ *'a sym-or-bullet*) ⇒ (*'k* ⇒ *'a*) **where**
project-inp inp = (λ *k*. *case inp k of SYM s* ⇒ *s*)

definition *compute-idx-set* :: (*'k* ⇒ *'a tuple-symbol*) ⇒ (*'k* ⇒ *'a sym-or-bullet*) ⇒ *'k set* **where**
compute-idx-set tup ys = {*i* . *tup i* ∈ *HAT* ‘ Γ ∧ *ys i* ∈ *SYM* ‘ Γ}

definition *update-ys* :: (*'k* ⇒ *'a tuple-symbol*) ⇒ (*'k* ⇒ *'a sym-or-bullet*) ⇒ (*'k* ⇒ *'a sym-or-bullet*) **where**
update-ys tup ys = (λ *k*. *if k* ∈ (*compute-idx-set tup ys*) *then* • *else ys k*)

definition *replace-sym* :: (*'k* ⇒ *'a tuple-symbol*) ⇒ (*'k* ⇒ *'a sym-or-bullet*) ⇒ (*'k* ⇒ *'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* ⇒ (*'k* ⇒ *'a tuple-symbol*) ⇒ (*'k* ⇒ *dir*) ⇒ *'k set* ⇒ (*'k* ⇒ *'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* ⇒ *'a tuple-symbol*) ⇒ (*'k* ⇒ *dir*) ⇒ *'k set* ⇒ (*'k* ⇒ *'a tuple-symbol*) **where**
place-hats-R = *place-hats-to-dir dir.R*

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

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

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

lemma *compute-idx-range*[*simp*,*intro*]:
assumes *tup* $\in$ *gamma-set*
assumes *ys* $\in$ *func-set*
shows *compute-idx-set tup ys* $\in$ *UNIV*
$\langle proof \rangle$

lemma *update-ys-range*[*simp*,*intro*]:
  assumes *tup* ∈ *gamma-set*
  assumes *ys* ∈ *func-set*
  shows *update-ys tup ys* ∈ *func-set*
  ⟨*proof*⟩

lemma *replace-sym-range*[*simp*,*intro*]:
  assumes *tup* ∈ *gamma-set*
  assumes *ys* ∈ *func-set*
  shows *replace-sym tup ys* ∈ *gamma-set*
⟨*proof*⟩

lemma *tup-hat-content*:
  assumes *tup* ∈ *gamma-set*
  assumes *tup x* = *HAT a*
  shows *a* ∈ Γ
⟨*proof*⟩

lemma *tup-no-hat-content*:
  assumes *tup* ∈ *gamma-set*
  assumes *tup x* = *NO-HAT a*
  shows *a* ∈ Γ
⟨*proof*⟩

lemma *place-hats-to-dir-range*[*simp*, *intro*]:
  assumes *tup* ∈ *gamma-set*
  shows *place-hats-to-dir d tup ds I* ∈ *gamma-set*
⟨*proof*⟩

lemma *place-hats-range*[*simp*,*intro*]:
  assumes *tup* ∈ *gamma-set*
  shows *place-hats-R tup ds I* ∈ *gamma-set* **and**
    place-hats-L tup ds I ∈ *gamma-set* **and**
    place-hats-M tup ds I ∈ *gamma-set*
  ⟨*proof*⟩

lemma *fin-R1-Set*[*intro*,*simp*]: *finite R1-Set*
  ⟨*proof*⟩

lemma *fin-gamma-set*[*intro*,*simp*]: *finite gamma-set*
  ⟨*proof*⟩

lemma *fin*-Γ′[*intro*,*simp*]: *finite* Γ′
  ⟨*proof*⟩

lemma *fin-func-set*[*simp*,*intro*]: *finite func-set*
  ⟨*proof*⟩

lemma *memberships*[*simp*,*intro*]: $\vdash \in \Gamma'$
$\bullet \in$ *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* $y \in \Gamma'$
$(\lambda\text{-}.\ \bullet) \in$ *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* $\in \Gamma$
⟨*proof*⟩

lemma *add-inp-func-set*[*simp*,*intro*]: $b \in$ *gamma-set* $\Longrightarrow a \in$ *func-set* $\Longrightarrow$ *add-inp b a* $\in$ *func-set*
⟨*proof*⟩

lemma *automation*[*simp*]: $\bigwedge a\ b\ A\ B.\ (S\ a\ b \in (\lambda x.\ case\ x\ of\ (x1,\ x2) \Rightarrow S\ x1\ x2)\ \text{`}\ (A \times B)) \longleftrightarrow (a \in A \wedge b \in B)$
$\bigwedge a\ b\ A\ B.\ (S_1\ a\ b \in (\lambda x.\ case\ x\ of\ (x1,\ x2) \Rightarrow S_1\ x1\ x2)\ \text{`}\ (A \times B)) \longleftrightarrow (a \in A \wedge b \in B)$
$\bigwedge a\ b\ c\ A\ B\ C.\ (E_0\ a\ b\ c \in (\lambda x.\ case\ x\ of\ (x1,\ x2,\ x3) \Rightarrow E_0\ x1\ x2\ x3)\ \text{`}\ (A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge c \in C)$
$\bigwedge a\ b\ c\ A\ B\ C.\ (E\ a\ b\ c \in (\lambda x.\ case\ x\ of\ (x1,\ x2,\ x3) \Rightarrow E\ x1\ x2\ x3)\ \text{`}\ (A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge c \in C)$
$\bigwedge a\ b\ c\ d\ A\ B\ C.\ (Er\ a\ b\ c\ d \in (\lambda x.\ case\ x\ of\ (x1,\ x2,\ x3,\ x4) \Rightarrow Er\ x1\ x2\ x3\ x4)\ \text{`}\ (A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge (c,d) \in C)$
$\bigwedge a\ b\ c\ d\ A\ B\ C.\ (Em\ a\ b\ c\ d \in (\lambda x.\ case\ x\ of\ (x1,\ x2,\ x3,\ x4) \Rightarrow Em\ x1\ x2\ x3\ x4)\ \text{`}\ (A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge (c,d) \in C)$
$\bigwedge a\ b\ c\ d\ A\ B\ C.\ (El\ a\ b\ c\ d \in (\lambda x.\ case\ x\ of\ (x1,\ x2,\ x3,\ x4) \Rightarrow El\ x1\ x2\ x3\ x4)\ \text{`}\ (A \times B \times C)) \longleftrightarrow (a \in A \wedge b \in B \wedge (c,d) \in C)$
blank' $\neq \vdash$
$\vdash \neq$ *blank'*
blank' $\neq$ *INP x*
INP x $\neq$ *blank'*
⟨*proof*⟩

interpretation *st*: *singletape-tm* Q' (*INP* ' Σ) Γ' *blank'* $\vdash \delta'\ R_1 \bullet S_0\ t\ S_0\ r$
⟨*proof*⟩

lemma *valid-st*: *singletape-tm* Q' (*INP* ' Σ) Γ' *blank'* $\vdash \delta'\ (R_1 \bullet)\ (S_0\ t)\ (S_0\ r)$
⟨*proof*⟩

Determinism is preserved.

lemma *det-preservation*: *deterministic* $\Longrightarrow$ *st.deterministic*
⟨*proof*⟩

6.2 Soundness of the Translation

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

lemma *max-mt-pos-step*: **assumes** *(cm,cm')* $\in$ *step*
shows *Max (range (mt-pos cm'))* $\leq$ *Suc (Max (range (mt-pos cm)))*
⟨*proof*⟩

lemma *max-mt-pos-init*: *Max (range (mt-pos (init-config w))) = 0*
⟨*proof*⟩

lemma *INP-D*: **assumes** *set x* $\subseteq$ *INP* ' Σ
shows $\exists$ *w. x = map INP w* $\wedge$ *set w* $\subseteq$ Σ
⟨*proof*⟩

6.2.1 R-Phase

fun *enc* :: *('a, 'q, 'k) mt-config* $\Rightarrow$ *nat* $\Rightarrow$ *('a, 'k) st-tape-symbol*
where *enc* (*Config*$_M$ *q tc p) n = TUPLE* (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* $\Rightarrow$ *'a list* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
n = length w $\Longrightarrow$
tc' 0 = $\vdash$ $\Longrightarrow$
p' $\leq$ *n* $\Longrightarrow$
$(\bigwedge$ *i. i < p'* $\Longrightarrow$ *enc (init-config w) i = tc' (Suc i))* $\Longrightarrow$
$(\bigwedge$ *i. i* $\geq$ *p'* $\Longrightarrow$ *tc' (Suc i) = (if i < n then INP (w ! i) else blank'))* $\Longrightarrow$
(p' = 0 $\Longrightarrow$ *q' =* $\bullet$*)* $\Longrightarrow$
$(\bigwedge$ *p. p' = Suc p* $\Longrightarrow$ *q' = SYM (w ! p))* $\Longrightarrow$
rel-R_1 (*Config*$_S$ (R_1 *q') tc' (Suc p')) w p'*

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

lemma *rel-*R_1*-*R_1: **assumes** *rel-*R_1 *cs0 w j*
and *j < length w*
and *set w* $\subseteq$ Σ
shows $\exists$ *cs1. (cs0, cs1)* $\in$ *st.dstep* $\wedge$ *rel-*R_1 *cs1 w (Suc j)*
⟨*proof*⟩

inductive *rel-*R_2 :: *(('a, 'k) st-tape-symbol,('a, 'q, 'k) st-states)st-config* $\Rightarrow$ *'a list* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**
tc' 0 = $\vdash$ $\Longrightarrow$

$(\bigwedge i.\ enc\ (init\text{-}config\ w)\ i = tc'\ (Suc\ i)) \Longrightarrow$
$p \leq length\ w \Longrightarrow$
$rel\text{-}R_2\ (Config_S\ R_2\ tc'\ p)\ w\ p$

lemma $rel\text{-}R_1\text{-}R_2$: **assumes** $rel\text{-}R_1\ cs0\ w\ (length\ w)$
and $set\ w \subseteq \Sigma$
shows $\exists\ cs1.\ (cs0,\ cs1) \in st.dstep \wedge rel\text{-}R_2\ cs1\ w\ (length\ w)$
⟨*proof*⟩

lemma $rel\text{-}R_2\text{-}R_2$: **assumes** $rel\text{-}R_2\ cs0\ w\ (Suc\ j)$
and $set\ w \subseteq \Sigma$
shows $\exists\ cs1.\ (cs0,\ cs1) \in st.dstep \wedge rel\text{-}R_2\ cs1\ w\ j$
⟨*proof*⟩

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

lemma $rel\text{-}R_2\text{-}S_0$: **assumes** $rel\text{-}R_2\ cs0\ w\ 0$
and $set\ w \subseteq \Sigma$
shows $\exists\ cs1.\ (cs0,\ cs1) \in st.dstep \wedge rel\text{-}S_0\ cs1\ (init\text{-}config\ w)$
⟨*proof*⟩

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 ($rel\text{-}S_0$) for the initial configuration.

lemma *R-phase*: **assumes** $set\ w \subseteq \Sigma$
shows $\exists\ cs.\ (st.init\text{-}config\ (map\ INP\ w),\ cs) \in st.dstep^{\frown\frown}(3 + 2 * length\ w) \wedge rel\text{-}S_0\ cs\ (init\text{-}config\ w)$
⟨*proof*⟩

6.2.2 S-Phase

inductive *rel-S* :: $(('a,\ 'k)\ st\text{-}tape\text{-}symbol,('a,\ 'q,\ 'k)\ st\text{-}states)st\text{-}config \Rightarrow ('a,\ 'q,\ 'k)\ mt\text{-}config \Rightarrow nat \Rightarrow bool$ **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 $\exists\ cs1.\ (cs0,\ cs1) \in st.dstep \wedge rel\text{-}S\ cs1\ cm\ 0$

⟨*proof*⟩

lemma *rel-S-mem*: **assumes** *rel-S* $(Config_S\ (S\ q\ inp)\ tc'\ p')\ cm\ j$
shows $inp \in func\text{-}set \wedge q \in Q \wedge (\exists\ t.\ tc'\ (Suc\ i) = TUPLE\ t \wedge t \in gamma\text{-}set)$

⟨*proof*⟩

lemma *rel-S-S*: **assumes** *rel-S cs0 cm p'*
$p' \leq Max\ (range\ (mt\text{-}pos\ cm))$
shows $\exists\ cs1.\ (cs0,\ cs1) \in st.dstep \wedge rel\text{-}S\ cs1\ cm\ (Suc\ p')$
⟨*proof*⟩

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

lemma $rel\text{-}S\text{-}S_1$: **assumes** *rel-S cs0 cm p'*
$p' = Suc\ (Max\ (range\ (mt\text{-}pos\ cm)))$
shows $\exists\ cs1.\ (cs0,\ cs1) \in st.dstep \wedge rel\text{-}S_1\ cs1\ cm$
⟨*proof*⟩

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

lemma *S-phase*: **assumes** $rel\text{-}S_0\ cs\ cm$
and $mt\text{-}state\ cm \notin \{t,\ r\}$
shows $\exists\ cs'.\ (cs,\ cs') \in st.dstep^{\frown\frown}(3 + Max\ (range\ (mt\text{-}pos\ cm))) \wedge rel\text{-}S_1\ cs'\ cm$
⟨*proof*⟩

6.2.3 E-Phase

context
fixes *rule* :: $('a,'q,'k)mt\text{-}rule$
begin
inductive-set $\delta step$ :: $('a,\ 'q,\ 'k)\ mt\text{-}config\ rel$ **where**
$\delta step$: $rule = (q,\ a,\ q1,\ b,\ dir) \Longrightarrow$
$rule \in \delta \Longrightarrow$
$(\bigwedge\ k.\ ts\ k\ (n\ k) = a\ k) \Longrightarrow$
$(\bigwedge\ k.\ ts'\ k = (ts\ k)(n\ k := b\ k)) \Longrightarrow$
$(\bigwedge\ k.\ n'\ k = go\text{-}dir\ (dir\ k)\ (n\ k)) \Longrightarrow$
$(Config_M\ q\ ts\ n,\ Config_M\ q1\ ts'\ n') \in \delta step$
end

lemma *step-to-δstep*: $(c1,c2) \in step \Longrightarrow \exists\ rule.\ (c1,c2) \in \delta step\ rule$
⟨*proof*⟩

lemma *δstep-to-step*: $(c1,c2) \in \delta step\ rule \Longrightarrow (c1,c2) \in step$
⟨*proof*⟩

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**

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

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-S*$_1$*-E*$_0$*-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*⟩

lemma *rel-S*$_1$*-E*$_0$*-st-step*: **assumes** $rel\text{-}S_1\ cs\ cm$
and $(cs,cs1) \in st.step$
shows $\exists\ cm1\ rule.\ (cm,\ cm1) \in step \wedge rel\text{-}E_0\ cs1\ cm\ cm1\ rule$
⟨*proof*⟩

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

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**
$tc'\ 0 = \vdash \Longrightarrow$
$(\bigwedge i.\ tc'\ (Suc\ i) = enc2\ (Config_M\ q\ tc\ p)\ (Config_M\ q1\ tc1\ p1)\ p'\ i) \Longrightarrow$
$valid\text{-}config\ (Config_M\ q\ tc\ p) \Longrightarrow$
$rule = (q,a,q1,b,d) \Longrightarrow$
$(Config_M\ q\ tc\ p,\ Config_M\ q1\ tc1\ p1) \in \delta step\ rule \Longrightarrow$
$bo = (\lambda\ k.\ if\ p\ k < p'\ then\ SYM\ (b\ k)\ else\ \bullet) \Longrightarrow$
$rel\text{-}E\ (Config_S\ (E\ q1\ bo\ d)\ tc'\ p')\ (Config_M\ q\ tc\ p)\ (Config_M\ q1\ tc1\ p1)\ rule\ p'$

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*))))
⟨*proof*⟩

lemma *rel-E-*S_0: **assumes** *rel-E cs cm cm1 rule 0*
shows ∃ *cs1*. (*cs*,*cs1*) ∈ *st.dstep* ∧ *rel-*S_0 *cs1 cm1*
⟨*proof*⟩

lemma *dsteps-to-steps*: *a* ∈ *st.dstep* ^^ *n* ⟹ *a* ∈ *st.step* ^^ *n*
⟨*proof*⟩

lemma *δ′-mem*: **assumes** *tup* ∈ *A*
and *f* ‘ *A* ⊆ *δ′*
shows *f tup* ∈ *δ′*
⟨*proof*⟩

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′*
⟨*proof*⟩

lemma *E-phase*: **assumes** *rel-*E_0 *cs cm cm1 rule*
shows ∃ *cs′*. (*cs*,*cs′*) ∈ *st.dstep* ^^ (*6* + *4* ∗ *Max* (*range* (*mt-pos cm*))) ∧ *rel-*S_0 *cs′ cm1*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

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*⟩

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*

$\wedge$ *k* $\leq$ *2* $*$ *length w* $+$ *3* $*$ *n*^*2* $+$ *7* $*$ *n* $+$ *3*
⟨*proof*⟩

lemma *simulation*: *map INP ‘ Lang* $\subseteq$ *st.Lang*
⟨*proof*⟩

6.2.5 Simulation of singletape TM by multitape TM

lemma *rev-simulation*: *st.Lang* $\subseteq$ *map INP ‘ Lang*
⟨*proof*⟩

lemma *rev-simulation-complexity*: **assumes** *w*: *set w* $\subseteq \Sigma$
  and *steps*: (*st.init-config* (*map INP w*), *cs*) $\in$ *st.step*^^*n*
  and *n*: *n* $\geq$ *2* $*$ *length w* $+$ *3* $*$ *k*^*2* $+$ *7* $*$ *k* $+$ *3*
shows $\exists$ *cm*. (*init-config w*, *cm*) $\in$ *step*^^*k*
⟨*proof*⟩

6.2.6 Main Results

theorem *language-equivalence*: *st.Lang* $=$ *map INP ‘ Lang*
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩
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*⟩

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.