Universal Turing Machine and Computability Theory in Isabelle/HOL

Jian Xu[2] Xingyuan Zhang[2] Christian Urban[1]
Sebastiaan J. C. Joosten[3]
[1]King's College London, UK
[2]PLA University of Science and Technology, China
[3]University of Twente, the Netherlands

December 14, 2021

Abstract

We formalise results from computability theory: recursive functions, undecidability of the halting problem, and the existence of a universal Turing machine. This formalisation is the AFP entry corresponding to: Mechanising Turing Machines and Computability Theory in Isabelle/HOL, ITP 2013

The AFP entry and by extension this document is largely written by Jian Xu, Xingyuan Zhang, and Christian Urban. The Universal Turing Machine is well explained in this document, starting at Figure 1. Regardless, you may want to read the original ITP article [6] instead of this pdf document corresponding to the AFP entry. If you are just interested in results about Turing Machines and Computability theory: the main book used for this formalisation is by Boolos [1].

Sebastiaan J. C. Joosten contributed mainly by making the files ready for the AFP. The need for a good formalisation of Turing Machines arose from realising that the current formalisation of saturation graphs [4] is missing a key undecidability result present in the original paper [3]. Recently, an undecidability result has been added to the AFP by Bertram Felgenhauer [2], using a definition of computably enumerable sets formalised by Michael Nedzelsky [5]. Showing the equivalence of these entirely separate notions of computability and decidability remains future work.

1 Turing Machines

theory *Turing*
imports *Main*
begin

2 Basic definitions of Turing machine

datatype *action* = *W0* | *W1* | *L* | *R* | *Nop*

datatype *cell* = *Bk* | *Oc*

type-synonym *tape* = *cell list* × *cell list*

type-synonym *state* = *nat*

type-synonym *instr* = *action* × *state*

type-synonym *tprog* = *instr list* × *nat*

type-synonym *tprog0* = *instr list*

type-synonym *config* = *state* × *tape*

fun *nth_of* **where**
nth_of xs i = (*if i* ≥ *length xs then None else Some* (*xs* ! *i*))

lemma *nth_of_map* [*simp*]:
shows *nth_of* (*map f p*) *n* = (*case* (*nth_of p n*) *of None* ⇒ *None* | *Some x* ⇒ *Some* (*f x*))
⟨*proof*⟩

fun
fetch :: *instr list* ⇒ *state* ⇒ *cell* ⇒ *instr*
where
fetch p 0 b = (*Nop*, *0*)
| *fetch p* (*Suc s*) *Bk* =
(*case nth_of p* (*2* ∗ *s*) *of*
Some i ⇒ *i*
| *None* ⇒ (*Nop*, *0*))
|*fetch p* (*Suc s*) *Oc* =
(*case nth_of p* ((*2* ∗ *s*) + *1*) *of*
Some i ⇒ *i*
| *None* ⇒ (*Nop*, *0*))

lemma *fetch_Nil* [*simp*]:
shows *fetch* [] *s b* = (*Nop*, *0*)
⟨*proof*⟩

fun
update :: *action* ⇒ *tape* ⇒ *tape*
where
update W0 (*l*, *r*) = (*l*, *Bk* # (*tl r*))
| *update W1* (*l*, *r*) = (*l*, *Oc* # (*tl r*))
| *update L* (*l*, *r*) = (*if l* = [] *then* ([], *Bk* # *r*) *else* (*tl l*, (*hd l*) # *r*))
| *update R* (*l*, *r*) = (*if r* = [] *then* (*Bk* # *l*, []) *else* ((*hd r*) # *l*, *tl r*))
| *update Nop* (*l*, *r*) = (*l*, *r*)

abbreviation
read r == *if* (*r* = []) *then Bk else hd r*

fun *step* :: *config* ⇒ *tprog* ⇒ *config*
 where
  step (*s*, *l*, *r*) (*p*, *off*) =
   (*let* (*a*, *s'*) = *fetch p* (*s* − *off*) (*read r*) *in* (*s'*, *update a* (*l*, *r*)))

abbreviation
 step0 c p $\stackrel{def}{=}$ *step c* (*p*, *0*)

fun *steps* :: *config* ⇒ *tprog* ⇒ *nat* ⇒ *config*
 where
  steps c p 0 = *c* |
  steps c p (*Suc n*) = *steps* (*step c p*) *p n*

abbreviation
 steps0 c p n $\stackrel{def}{=}$ *steps c* (*p*, *0*) *n*

lemma *step_red* [*simp*]:
 shows *steps c p* (*Suc n*) = *step* (*steps c p n*) *p*
 ⟨*proof*⟩

lemma *steps_add* [*simp*]:
 shows *steps c p* (*m* + *n*) = *steps* (*steps c p m*) *p n*
 ⟨*proof*⟩

lemma *step_0* [*simp*]:
 shows *step* (*0*, (*l*, *r*)) *p* = (*0*, (*l*, *r*))
 ⟨*proof*⟩

lemma *steps_0* [*simp*]:
 shows *steps* (*0*, (*l*, *r*)) *p n* = (*0*, (*l*, *r*))
 ⟨*proof*⟩

fun
 is_final :: *config* ⇒ *bool*
 where
  is_final (*s*, *l*, *r*) = (*s* = *0*)

lemma *is_final_eq*:
 shows *is_final* (*s*, *tp*) = (*s* = *0*)
 ⟨*proof*⟩

lemma *is_finalI* [*intro*]:
 shows *is_final* (*0*, *tp*)
 ⟨*proof*⟩

lemma *after_is_final*:
 assumes *is_final c*
 shows *is_final* (*steps c p n*)

⟨*proof*⟩

lemma *is_final*:
assumes *a*: *is_final* (*steps c p n1*)
and *b*: *n1* ≤ *n2*
shows *is_final* (*steps c p n2*)
⟨*proof*⟩

lemma *not_is_final*:
assumes *a*: ¬ *is_final* (*steps c p n1*)
and *b*: *n2* ≤ *n1*
shows ¬ *is_final* (*steps c p n2*)
⟨*proof*⟩

lemma *before_final*:
assumes *steps0* (*1*, *tp*) *A n* = (*0*, *tp'*)
shows ∃ *n'*. ¬ *is_final* (*steps0* (*1*, *tp*) *A n'*) ∧ *steps0* (*1*, *tp*) *A* (*Suc n'*) = (*0*, *tp'*)
⟨*proof*⟩

lemma *least_steps*:
assumes *steps0* (*1*, *tp*) *A n* = (*0*, *tp'*)
shows ∃ *n'*. (∀ *n''* < *n'*. ¬ *is_final* (*steps0* (*1*, *tp*) *A n''*)) ∧
(∀ *n''* ≥ *n'*. *is_final* (*steps0* (*1*, *tp*) *A n''*))
⟨*proof*⟩

abbreviation *is_even n* $\stackrel{def}{=}$ (*n*::*nat*) *mod 2* = *0*

fun
tm_wf :: *tprog* ⇒ *bool*
where
tm_wf (*p*, *off*) = (*length p* ≥ *2* ∧ *is_even* (*length p*) ∧
(∀ (*a*, *s*) ∈ *set p*. *s* ≤ *length p div 2* + *off* ∧ *s* ≥ *off*))

abbreviation
tm_wf0 p $\stackrel{def}{=}$ *tm_wf* (*p*, *0*)

abbreviation *exponent* :: *'a* ⇒ *nat* ⇒ *'a list* (_ ↑ _ [*100*, *99*] *100*)
where *x* ↑ *n* == *replicate n x*

lemma *hd_repeat_cases*:
P (*hd* (*a* ↑ *m* @ *r*)) ⟷ (*m* = *0* ⟶ *P* (*hd r*)) ∧ (∀ *nat*. *m* = *Suc nat* ⟶ *P a*)
⟨*proof*⟩

class *tape* =
fixes *tape_of* :: *'a* ⇒ *cell list* (<_> *100*)

instantiation *nat*::*tape* **begin**

definition *tape_of_nat* **where** *tape_of_nat* (*n*::*nat*) $\stackrel{def}{=}$ *Oc* ↑ (*Suc n*)
instance ⟨*proof*⟩
end

type-synonym *nat_list* = *nat list*

instantiation *list*::(*tape*) *tape* **begin**
fun *tape_of_nat_list* :: ('*a*::*tape*) *list* ⇒ *cell list*
 where
  tape_of_nat_list [] = [] |
  tape_of_nat_list [*n*] = <*n*> |
  tape_of_nat_list (*n*#*ns*) = <*n*> @ *Bk* # (*tape_of_nat_list ns*)
definition *tape_of_list* **where** *tape_of_list* $\stackrel{def}{=}$ *tape_of_nat_list*
instance ⟨*proof*⟩
end

instantiation *prod*:: (*tape*, *tape*) *tape* **begin**
fun *tape_of_nat_prod* :: ('*a*::*tape*) × ('*b*::*tape*) ⇒ *cell list*
 where *tape_of_nat_prod* (*n*, *m*) = <*n*> @ [*Bk*] @ <*m*>
definition *tape_of_prod* **where** *tape_of_prod* $\stackrel{def}{=}$ *tape_of_nat_prod*
instance ⟨*proof*⟩
end

fun
 shift :: *instr list* ⇒ *nat* ⇒ *instr list*
 where
  shift p n = (*map* (λ (*a*, *s*). (*a*, (*if s* = *0 then 0 else s* + *n*))) *p*)

fun
 adjust :: *instr list* ⇒ *nat* ⇒ *instr list*
 where
  adjust p e = *map* (λ (*a*, *s*). (*a*, *if s* = *0 then e else s*)) *p*

abbreviation
 adjust0 p $\stackrel{def}{=}$ *adjust p* (*Suc* (*length p div 2*))

lemma *length_shift* [*simp*]:
 shows *length* (*shift p n*) = *length p*
 ⟨*proof*⟩

lemma *length_adjust* [*simp*]:
 shows *length* (*adjust p n*) = *length p*
 ⟨*proof*⟩

fun
tm_comp :: *instr list* ⇒ *instr list* ⇒ *instr list* (_ |+| _ [0, 0] *100*)
where
tm_comp p1 p2 = ((*adjust0 p1*) @ (*shift p2* (*length p1 div 2*)))

lemma *tm_comp_length*:
shows *length* (*A* |+| *B*) = *length A* + *length B*
⟨*proof*⟩

lemma *tm_comp_wf*[*intro*]:
⟦*tm_wf* (*A*, *0*); *tm_wf* (*B*, *0*)⟧ ⟹ *tm_wf* (*A* |+| *B*, *0*)
⟨*proof*⟩

lemma *tm_comp_step*:
assumes *unfinal*: ¬ *is_final* (*step0 c A*)
shows *step0 c* (*A* |+| *B*) = *step0 c A*
⟨*proof*⟩

lemma *tm_comp_steps*:
assumes ¬ *is_final* (*steps0 c A n*)
shows *steps0 c* (*A* |+| *B*) *n* = *steps0 c A n*
⟨*proof*⟩

lemma *tm_comp_fetch_in_A*:
assumes *h1*: *fetch A s x* = (*a*, *0*)
and *h2*: *s* ≤ *length A div 2*
and *h3*: *s* ≠ *0*
shows *fetch* (*A* |+| *B*) *s x* = (*a*, *Suc* (*length A div 2*))
⟨*proof*⟩

lemma *tm_comp_exec_after_first*:
assumes *h1*: ¬ *is_final c*
and *h2*: *step0 c A* = (*0*, *tp*)
and *h3*: *fst c* ≤ *length A div 2*
shows *step0 c* (*A* |+| *B*) = (*Suc* (*length A div 2*), *tp*)
⟨*proof*⟩

lemma *step_in_range*:
assumes *h1*: ¬ *is_final* (*step0 c A*)
and *h2*: *tm_wf* (*A*, *0*)
shows *fst* (*step0 c A*) ≤ *length A div 2*
⟨*proof*⟩

lemma *steps_in_range*:
assumes *h1*: ¬ *is_final* (*steps0* (*1*, *tp*) *A stp*)
and *h2*: *tm_wf* (*A*, *0*)
shows *fst* (*steps0* (*1*, *tp*) *A stp*) ≤ *length A div 2*
⟨*proof*⟩

lemma *tm_comp_next*:
 assumes *a_ht*: *steps0* (*1*, *tp*) *A n* = (*0*, *tp'*)
  and *a_wf*: *tm_wf* (*A*, *0*)
 obtains *n'* **where** *steps0* (*1*, *tp*) (*A* |+| *B*) *n'* = (*Suc* (*length A div 2*), *tp'*)
⟨*proof*⟩

lemma *tm_comp_fetch_second_zero*:
 assumes *h1*: *fetch B s x* = (*a*, *0*)
  and *hs*: *tm_wf* (*A*, *0*) *s* ≠ *0*
 shows *fetch* (*A* |+| *B*) (*s* + (*length A div 2*)) *x* = (*a*, *0*)
 ⟨*proof*⟩

lemma *tm_comp_fetch_second_inst*:
 assumes *h1*: *fetch B sa x* = (*a*, *s*)
  and *hs*: *tm_wf* (*A*, *0*) *sa* ≠ *0 s* ≠ *0*
 shows *fetch* (*A* |+| *B*) (*sa* + *length A div 2*) *x* = (*a*, *s* + *length A div 2*)
 ⟨*proof*⟩

lemma *tm_comp_second*:
 assumes *a_wf*: *tm_wf* (*A*, *0*)
  and *steps*: *steps0* (*1*, *l*, *r*) *B stp* = (*s'*, *l'*, *r'*)
 shows *steps0* (*Suc* (*length A div 2*), *l*, *r*) (*A* |+| *B*) *stp*
  = (*if s'* = *0 then 0 else s'* + *length A div 2*, *l'*, *r'*)
 ⟨*proof*⟩

lemma *tm_comp_final*:
 assumes *tm_wf* (*A*, *0*)
  and *steps0* (*1*, *l*, *r*) *B stp* = (*0*, *l'*, *r'*)
 shows *steps0* (*Suc* (*length A div 2*), *l*, *r*) (*A* |+| *B*) *stp* = (*0*, *l'*, *r'*)
 ⟨*proof*⟩

end

3 Hoare Rules for TMs

theory *Turing_Hoare*
 imports *Turing*
begin

type-synonym *assert* = *tape* ⇒ *bool*

definition
 assert_imp :: *assert* ⇒ *assert* ⇒ *bool* (_ ↦ _ [*0*, *0*] *100*)
 where
  P ↦ *Q* ≡ ∀ *l r*. *P* (*l*, *r*) ⟶ *Q* (*l*, *r*)

lemma *refl_assert*[*intro*, *simp*]:
P ↦ *P*
⟨*proof*⟩

fun
holds_for :: (*tape* ⇒ *bool*) ⇒ *config* ⇒ *bool* (_ *holds′_for* _ [*100*, *99*] *100*)
where
P holds_for (*s*, *l*, *r*) = *P* (*l*, *r*)

lemma *is_final_holds*[*simp*]:
assumes *is_final c*
shows *Q holds_for* (*steps c p n*) = *Q holds_for c*
⟨*proof*⟩

definition
Hoare_halt :: *assert* ⇒ *tprog0* ⇒ *assert* ⇒ *bool* (({(*1*_)}/ (_)/ {(*1*_)}) *50*)
where
{*P*} *p* {*Q*} ≡ (∀ *tp*. *P tp* ⟶ (∃ *n*. *is_final* (*steps0* (*1*, *tp*) *p n*) ∧ *Q holds_for* (*steps0* (*1*, *tp*) *p n*)))

definition
Hoare_unhalt :: *assert* ⇒ *tprog0* ⇒ *bool* (({(*1*_)}/ (_)) ↑ *50*)
where
{*P*} *p* ↑ ≡ ∀ *tp*. *P tp* ⟶ (∀ *n* . ¬ (*is_final* (*steps0* (*1*, *tp*) *p n*)))

lemma *Hoare_haltI*:
assumes ⋀*l r*. *P* (*l*, *r*) ⟹ ∃ *n*. *is_final* (*steps0* (*1*, (*l*, *r*)) *p n*) ∧ *Q holds_for* (*steps0* (*1*, (*l*, *r*)) *p n*)
shows {*P*} *p* {*Q*}
⟨*proof*⟩

lemma *Hoare_unhaltI*:
assumes ⋀*l r n*. *P* (*l*, *r*) ⟹ ¬ *is_final* (*steps0* (*1*, (*l*, *r*)) *p n*)
shows {*P*} *p* ↑
⟨*proof*⟩

P A Q Q B S A well-formed ——————————————— P A |+| B S

lemma *Hoare_plus_halt* [*case_names A_halt B_halt A_wf*]:
assumes *A_halt* : {*P*} *A* {*Q*}
and *B_halt* : {*Q*} *B* {*S*}
and *A_wf* : *tm_wf* (*A*, *0*)
shows {*P*} *A* |+| *B* {*S*}
⟨*proof*⟩

P A Q Q B loops A well-formed ——————————————— P A |+| B loops

lemma *Hoare_plus_unhalt* [*case_names A_halt B_unhalt A_wf*]:
 assumes *A_halt*: $\{P\}\ A\ \{Q\}$
  and *B_uhalt*: $\{Q\}\ B \uparrow$
  and *A_wf* : *tm_wf* $(A, 0)$
 shows $\{P\}\ (A\ |+|\ B) \uparrow$
⟨*proof*⟩

lemma *Hoare_consequence*:
 assumes $P' \mapsto P\ \{P\}\ p\ \{Q\}\ Q \mapsto Q'$
 shows $\{P'\}\ p\ \{Q'\}$
 ⟨*proof*⟩

end

4 Undeciablity of the Halting Problem

theory *Uncomputable*
 imports *Turing_Hoare*
begin

lemma *numeral*:
 shows *2* = *Suc 1*
  and *3* = *Suc 2*
  and *4* = *Suc 3*
  and *5* = *Suc 4*
  and *6* = *Suc 5*
  and *7* = *Suc 6*
  and *8* = *Suc 7*
  and *9* = *Suc 8*
  and *10* = *Suc 9*
  and *11* = *Suc 10*
  and *12* = *Suc 11*
 ⟨*proof*⟩

lemma *gr1_conv_Suc*:*Suc 0* < *mr* ⟷ (∃ *nat*. *mr* = *Suc* (*Suc nat*)) ⟨*proof*⟩

The Copying TM, which duplicates its input.

definition
 tcopy_begin :: *instr list*
 where
  tcopy_begin $\stackrel{def}{=}$ $[(W0, 0), (R, 2), (R, 3), (R, 2),$
  $(W1, 3), (L, 4), (L, 4), (L, 0)]$

definition
 tcopy_loop :: *instr list*

where
tcopy_loop $\stackrel{def}{=}$ $[(R, 0), (R, 2), (R, 3), (W0, 2),$
$(R, 3), (R, 4), (W1, 5), (R, 4),$
$(L, 6), (L, 5), (L, 6), (L, 1)]$

definition
tcopy_end :: *instr list*
where
tcopy_end $\stackrel{def}{=}$ $[(L, 0), (R, 2), (W1, 3), (L, 4),$
$(R, 2), (R, 2), (L, 5), (W0, 4),$
$(R, 0), (L, 5)]$

definition
tcopy :: *instr list*
where
tcopy $\stackrel{def}{=}$ $(tcopy\_begin \mid+\mid tcopy\_loop) \mid+\mid tcopy\_end$

fun
inv_begin0 :: *nat* $\Rightarrow$ *tape* $\Rightarrow$ *bool* **and**
inv_begin1 :: *nat* $\Rightarrow$ *tape* $\Rightarrow$ *bool* **and**
inv_begin2 :: *nat* $\Rightarrow$ *tape* $\Rightarrow$ *bool* **and**
inv_begin3 :: *nat* $\Rightarrow$ *tape* $\Rightarrow$ *bool* **and**
inv_begin4 :: *nat* $\Rightarrow$ *tape* $\Rightarrow$ *bool*
where
$inv\_begin0\ n\ (l, r) = ((n > 1 \wedge (l, r) = (Oc \uparrow (n-2), [Oc, Oc, Bk, Oc])) \vee$
$(n = 1 \wedge (l, r) = ([], [Bk, Oc, Bk, Oc])))$
$\mid inv\_begin1\ n\ (l, r) = ((l, r) = ([], Oc \uparrow n))$
$\mid inv\_begin2\ n\ (l, r) = (\exists\ i\ j.\ i > 0 \wedge i + j = n \wedge (l, r) = (Oc \uparrow i, Oc \uparrow j))$
$\mid inv\_begin3\ n\ (l, r) = (n > 0 \wedge (l, tl\ r) = (Bk \# Oc \uparrow n, []))$
$\mid inv\_begin4\ n\ (l, r) = (n > 0 \wedge (l, r) = (Oc \uparrow n, [Bk, Oc]) \vee (l, r) = (Oc \uparrow (n-1), [Oc, Bk, Oc]))$

fun *inv_begin* :: *nat* $\Rightarrow$ *config* $\Rightarrow$ *bool*
where
$inv\_begin\ n\ (s, tp) =$
$(if\ s = 0\ then\ inv\_begin0\ n\ tp\ else$
$if\ s = 1\ then\ inv\_begin1\ n\ tp\ else$
$if\ s = 2\ then\ inv\_begin2\ n\ tp\ else$
$if\ s = 3\ then\ inv\_begin3\ n\ tp\ else$
$if\ s = 4\ then\ inv\_begin4\ n\ tp$
$else\ False)$

lemma *split_head_repeat*[*simp*]:
$Oc \# list1 = Bk \uparrow j\ @\ list2 \longleftrightarrow j = 0 \wedge Oc \# list1 = list2$
$Bk \# list1 = Oc \uparrow j\ @\ list2 \longleftrightarrow j = 0 \wedge Bk \# list1 = list2$
$Bk \uparrow j\ @\ list2 = Oc \# list1 \longleftrightarrow j = 0 \wedge Oc \# list1 = list2$

$Oc \uparrow j \mathbin{@} list2 = Bk \# list1 \longleftrightarrow j = 0 \wedge Bk \# list1 = list2$
⟨*proof*⟩

lemma *inv_begin_step_E*: $[\![0 < i; 0 < j]\!] \Longrightarrow$
$\exists ia{>}0.\ ia + j - Suc\ 0 = i + j \wedge Oc \# Oc \uparrow i = Oc \uparrow ia$
⟨*proof*⟩

lemma *inv_begin_step*:
assumes *inv_begin n cf*
and $n > 0$
shows *inv_begin n* (*step0 cf tcopy_begin*)
⟨*proof*⟩

lemma *inv_begin_steps*:
assumes *inv_begin n cf*
and $n > 0$
shows *inv_begin n* (*steps0 cf tcopy_begin stp*)
⟨*proof*⟩

lemma *begin_partial_correctness*:
assumes *is_final* (*steps0* (*1*, [], $Oc \uparrow n$) *tcopy_begin stp*)
shows $0 < n \Longrightarrow \{inv\_begin1\ n\}$ *tcopy_begin* $\{inv\_begin0\ n\}$
⟨*proof*⟩

fun *measure_begin_state* :: *config* ⇒ *nat*
where
measure_begin_state $(s, l, r) =$ (*if* $s = 0$ *then 0 else* $5 - s$)

fun *measure_begin_step* :: *config* ⇒ *nat*
where
measure_begin_step $(s, l, r) =$
(*if* $s = 2$ *then length r else*
if $s = 3$ *then* (*if* $r = [] \vee r = [Bk]$ *then 1 else 0*) *else*
if $s = 4$ *then length l*
else 0)

definition
measure_begin = *measures* [*measure_begin_state*, *measure_begin_step*]

lemma *wf_measure_begin*:
shows *wf measure_begin*
⟨*proof*⟩

lemma *measure_begin_induct* [*case_names Step*]:
$[\![\bigwedge n.\ \neg P\ (f\ n) \Longrightarrow (f\ (Suc\ n), (f\ n)) \in measure\_begin]\!] \Longrightarrow \exists n.\ P\ (f\ n)$
⟨*proof*⟩

lemma *begin_halts*:
assumes *h*: $x > 0$
shows $\exists$ *stp*. *is_final* (*steps0* (*1*, [], $Oc \uparrow x$) *tcopy_begin stp*)

⟨*proof*⟩

lemma *begin_correct*:
shows $0 < n \Longrightarrow \{inv\_begin1\ n\}\ tcopy\_begin\ \{inv\_begin0\ n\}$
⟨*proof*⟩

declare *tm_comp.simps* [*simp del*]
declare *adjust.simps*[*simp del*]
declare *shift.simps*[*simp del*]
declare *tm_wf.simps*[*simp del*]
declare *step.simps*[*simp del*]
declare *steps.simps*[*simp del*]

fun
inv_loop1_loop :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_loop1_exit :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_loop5_loop :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_loop5_exit :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_loop6_loop :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_loop6_exit :: *nat* ⇒ *tape* ⇒ *bool*
where
inv_loop1_loop n $(l, r) = (\exists\ i\ j.\ i + j + 1 = n \wedge (l, r) = (Oc{\uparrow}i, Oc \# Oc \# Bk{\uparrow}j\ @\ Oc{\uparrow}j) \wedge j > 0)$
| *inv_loop1_exit n* $(l, r) = (0 < n \wedge (l, r) = ([], Bk \# Oc \# Bk{\uparrow}n\ @\ Oc{\uparrow}n))$
| *inv_loop5_loop x* $(l, r) =$
$(\exists\ i\ j\ k\ t.\ i + j = Suc\ x \wedge i > 0 \wedge j > 0 \wedge k + t = j \wedge t > 0 \wedge (l, r) = (Oc{\uparrow}k @ Bk{\uparrow}j @ Oc{\uparrow}i, Oc{\uparrow}t))$
| *inv_loop5_exit x* $(l, r) =$
$(\exists\ i\ j.\ i + j = Suc\ x \wedge i > 0 \wedge j > 0 \wedge (l, r) = (Bk{\uparrow}(j - 1) @ Oc{\uparrow}i, Bk\ \#\ Oc{\uparrow}j))$
| *inv_loop6_loop x* $(l, r) =$
$(\exists\ i\ j\ k\ t.\ i + j = Suc\ x \wedge i > 0 \wedge k + t + 1 = j \wedge (l, r) = (Bk{\uparrow}k\ @\ Oc{\uparrow}i, Bk{\uparrow}(Suc\ t)\ @\ Oc{\uparrow}j))$
| *inv_loop6_exit x* $(l, r) =$
$(\exists\ i\ j.\ i + j = x \wedge j > 0 \wedge (l, r) = (Oc{\uparrow}i, Oc \# Bk{\uparrow}j\ @\ Oc{\uparrow}j))$

fun
inv_loop0 :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_loop1 :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_loop2 :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_loop3 :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_loop4 :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_loop5 :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_loop6 :: *nat* ⇒ *tape* ⇒ *bool*
where
inv_loop0 n $(l, r) = (0 < n \wedge (l, r) = ([Bk], Oc\ \#\ Bk{\uparrow}n\ @\ Oc{\uparrow}n))$
| *inv_loop1 n* $(l, r) =$ (*inv_loop1_loop n* $(l, r) \vee$ *inv_loop1_exit n* (l, r))
| *inv_loop2 n* $(l, r) = (\exists\ i\ j\ any.\ i + j = n \wedge n > 0 \wedge i > 0 \wedge j > 0 \wedge (l, r) = (Oc{\uparrow}i, any \# Bk{\uparrow}j @ Oc{\uparrow}j))$

| *inv_loop3 n (l, r) =*
(∃ i j k t. i + j = n ∧ i > 0 ∧ j > 0 ∧ k + t = Suc j ∧ (l, r) = (Bk↑k@Oc↑i, Bk↑t@Oc↑j))
| *inv_loop4 n (l, r) =*
(∃ i j k t. i + j = n ∧ i > 0 ∧ j > 0 ∧ k + t = j ∧ (l, r) = (Oc↑k @ Bk↑(Suc j)@Oc↑i, Oc↑t))
| *inv_loop5 n (l, r) = (inv_loop5_loop n (l, r) ∨ inv_loop5_exit n (l, r))*
| *inv_loop6 n (l, r) = (inv_loop6_loop n (l, r) ∨ inv_loop6_exit n (l, r))*

fun *inv_loop* :: *nat ⇒ config ⇒ bool*
where
inv_loop x (s, l, r) =
(if s = 0 then inv_loop0 x (l, r)
else if s = 1 then inv_loop1 x (l, r)
else if s = 2 then inv_loop2 x (l, r)
else if s = 3 then inv_loop3 x (l, r)
else if s = 4 then inv_loop4 x (l, r)
else if s = 5 then inv_loop5 x (l, r)
else if s = 6 then inv_loop6 x (l, r)
else False)

declare *inv_loop.simps*[*simp del*] *inv_loop1.simps*[*simp del*]
inv_loop2.simps[*simp del*] *inv_loop3.simps*[*simp del*]
inv_loop4.simps[*simp del*] *inv_loop5.simps*[*simp del*]
inv_loop6.simps[*simp del*]

lemma *Bk_no_Oc_repeatE*[*elim*]: *Bk # list = Oc ↑ t ⟹ RR*
⟨*proof*⟩

lemma *inv_loop3_Bk_empty_via_2*[*elim*]: *⟦0 < x; inv_loop2 x (b, [])⟧ ⟹ inv_loop3 x (Bk # b, [])*
⟨*proof*⟩

lemma *inv_loop3_Bk_empty*[*elim*]: *⟦0 < x; inv_loop3 x (b, [])⟧ ⟹ inv_loop3 x (Bk # b, [])*
⟨*proof*⟩

lemma *inv_loop5_Oc_empty_via_4*[*elim*]: *⟦0 < x; inv_loop4 x (b, [])⟧ ⟹ inv_loop5 x (b, [Oc])*
⟨*proof*⟩

lemma *inv_loop1_Bk*[*elim*]: *⟦0 < x; inv_loop1 x (b, Bk # list)⟧ ⟹ list = Oc # Bk ↑ x @ Oc ↑ x*
⟨*proof*⟩

lemma *inv_loop3_Bk_via_2*[*elim*]: *⟦0 < x; inv_loop2 x (b, Bk # list)⟧ ⟹ inv_loop3 x (Bk # b, list)*
⟨*proof*⟩

lemma *inv_loop3_Bk_move*[*elim*]: *⟦0 < x; inv_loop3 x (b, Bk # list)⟧ ⟹ inv_loop3 x (Bk # b, list)*
⟨*proof*⟩

lemma *inv_loop5_Oc_via_4_Bk*[*elim*]: *⟦0 < x; inv_loop4 x (b, Bk # list)⟧ ⟹ inv_loop5 x (b,*

Oc # list)
⟨*proof*⟩

lemma *inv_loop6_Bk_via_5*[*elim*]: ⟦*0 < x*; *inv_loop5 x* ([], *Bk # list*)⟧ ⟹ *inv_loop6 x* ([], *Bk # Bk # list*)
⟨*proof*⟩

lemma *inv_loop5_loop_no_Bk*[*simp*]: *inv_loop5_loop x* (*b*, *Bk # list*) = *False*
⟨*proof*⟩

lemma *inv_loop6_exit_no_Bk*[*simp*]: *inv_loop6_exit x* (*b*, *Bk # list*) = *False*
⟨*proof*⟩

declare *inv_loop5_loop.simps*[*simp del*] *inv_loop5_exit.simps*[*simp del*]
inv_loop6_loop.simps[*simp del*] *inv_loop6_exit.simps*[*simp del*]

lemma *inv_loop6_loopBk_via_5*[*elim*]:⟦*0 < x*; *inv_loop5_exit x* (*b*, *Bk # list*); *b* ≠ []; *hd b* = *Bk*⟧
⟹ *inv_loop6_loop x* (*tl b*, *Bk # Bk # list*)
⟨*proof*⟩

lemma *inv_loop6_loop_no_Oc_Bk*[*simp*]: *inv_loop6_loop x* (*b*, *Oc # Bk # list*) = *False*
⟨*proof*⟩

lemma *inv_loop6_exit_Oc_Bk_via_5*[*elim*]: ⟦*x > 0*; *inv_loop5_exit x* (*b*, *Bk # list*); *b* ≠ []; *hd b* = *Oc*⟧ ⟹
inv_loop6_exit x (*tl b*, *Oc # Bk # list*)
⟨*proof*⟩

lemma *inv_loop6_Bk_tail_via_5*[*elim*]: ⟦*0 < x*; *inv_loop5 x* (*b*, *Bk # list*); *b* ≠ []⟧ ⟹ *inv_loop6 x* (*tl b*, *hd b # Bk # list*)
⟨*proof*⟩

lemma *inv_loop6_loop_Bk_Bk_drop*[*elim*]: ⟦*0 < x*; *inv_loop6_loop x* (*b*, *Bk # list*); *b* ≠ []; *hd b* = *Bk*⟧
⟹ *inv_loop6_loop x* (*tl b*, *Bk # Bk # list*)
⟨*proof*⟩

lemma *inv_loop6_exit_Oc_Bk_via_loop6*[*elim*]: ⟦*0 < x*; *inv_loop6_loop x* (*b*, *Bk # list*); *b* ≠ []; *hd b* = *Oc*⟧
⟹ *inv_loop6_exit x* (*tl b*, *Oc # Bk # list*)
⟨*proof*⟩

lemma *inv_loop6_Bk_tail*[*elim*]: ⟦*0 < x*; *inv_loop6 x* (*b*, *Bk # list*); *b* ≠ []⟧ ⟹ *inv_loop6 x* (*tl b*, *hd b # Bk # list*)
⟨*proof*⟩

lemma *inv_loop2_Oc_via_1*[*elim*]: ⟦*0 < x*; *inv_loop1 x* (*b*, *Oc # list*)⟧ ⟹ *inv_loop2 x* (*Oc # b*, *list*)
⟨*proof*⟩

lemma *inv_loop2_Bk_via_Oc*[*elim*]: ⟦*0 < x*; *inv_loop2 x* (*b*, *Oc # list*)⟧ ⟹ *inv_loop2 x* (*b*, *Bk # list*)
⟨*proof*⟩

lemma *inv_loop4_Oc_via_3*[*elim*]: ⟦*0 < x*; *inv_loop3 x* (*b*, *Oc # list*)⟧ ⟹ *inv_loop4 x* (*Oc # b*, *list*)
⟨*proof*⟩

lemma *inv_loop4_Oc_move*[*elim*]:
assumes *0 < x inv_loop4 x* (*b*, *Oc # list*)
shows *inv_loop4 x* (*Oc # b*, *list*)
⟨*proof*⟩

lemma *inv_loop5_exit_no_Oc*[*simp*]: *inv_loop5_exit x* (*b*, *Oc # list*) = *False*
⟨*proof*⟩

lemma *inv_loop5_exit_Bk_Oc_via_loop*[*elim*]: ⟦*inv_loop5_loop x* (*b*, *Oc # list*); *b* ≠ []; *hd b = Bk*⟧
⟹ *inv_loop5_exit x* (*tl b*, *Bk # Oc # list*)
⟨*proof*⟩

lemma *inv_loop5_loop_Oc_Oc_drop*[*elim*]: ⟦*inv_loop5_loop x* (*b*, *Oc # list*); *b* ≠ []; *hd b* = *Oc*⟧
⟹ *inv_loop5_loop x* (*tl b*, *Oc # Oc # list*)
⟨*proof*⟩

lemma *inv_loop5_Oc_tl*[*elim*]: ⟦*inv_loop5 x* (*b*, *Oc # list*); *b* ≠ []⟧ ⟹ *inv_loop5 x* (*tl b*, *hd b # Oc # list*)
⟨*proof*⟩

lemma *inv_loop1_Bk_Oc_via_6*[*elim*]: ⟦*0 < x*; *inv_loop6 x* ([], *Oc # list*)⟧ ⟹ *inv_loop1 x* ([], *Bk # Oc # list*)
⟨*proof*⟩

lemma *inv_loop1_Oc_via_6*[*elim*]: ⟦*0 < x*; *inv_loop6 x* (*b*, *Oc # list*); *b* ≠ []⟧
⟹ *inv_loop1 x* (*tl b*, *hd b # Oc # list*)
⟨*proof*⟩

lemma *inv_loop_nonempty*[*simp*]:
inv_loop1 x (*b*, []) = *False*
inv_loop2 x ([], *b*) = *False*
inv_loop2 x (*l′*, []) = *False*
inv_loop3 x (*b*, []) = *False*
inv_loop4 x ([], *b*) = *False*
inv_loop5 x ([], *list*) = *False*
inv_loop6 x ([], *Bk # xs*) = *False*
⟨*proof*⟩

lemma *inv_loop_nonemptyE*[*elim*]:
 ⟦*inv_loop5 x* (*b*, []) ⟹ *RR inv_loop6 x* (*b*, []) ⟹ *RR*
 ⟦*inv_loop1 x* (*b*, *Bk* # *list*)⟧ ⟹ *b* = []
 ⟨*proof*⟩

lemma *inv_loop6_Bk_Bk_drop*[*elim*]: ⟦*inv_loop6 x* ([], *Bk* # *list*)⟧ ⟹ *inv_loop6 x* ([], *Bk* # *Bk* # *list*)
 ⟨*proof*⟩

lemma *inv_loop_step*:
 ⟦*inv_loop x cf*; *x* > *0*⟧ ⟹ *inv_loop x* (*step cf* (*tcopy_loop*, *0*))
 ⟨*proof*⟩

lemma *inv_loop_steps*:
 ⟦*inv_loop x cf*; *x* > *0*⟧ ⟹ *inv_loop x* (*steps cf* (*tcopy_loop*, *0*) *stp*)
 ⟨*proof*⟩

fun *loop_stage* :: *config* ⇒ *nat*
 where
  loop_stage (*s*, *l*, *r*) = (*if s* = *0 then 0*
  else (*Suc* (*length* (*takeWhile* (λ*a*. *a* = *Oc*) (*rev l* @ *r*)))))

fun *loop_state* :: *config* ⇒ *nat*
 where
  loop_state (*s*, *l*, *r*) = (*if s* = *2* ∧ *hd r* = *Oc then 0*
  else if s = *1 then 1*
  else 10 − *s*)

fun *loop_step* :: *config* ⇒ *nat*
 where
  loop_step (*s*, *l*, *r*) = (*if s* = *3 then length r*
  else if s = *4 then length r*
  else if s = *5 then length l*
  else if s = *6 then length l*
  else 0)

definition *measure_loop* :: (*config* × *config*) *set*
 where
  measure_loop = *measures* [*loop_stage*, *loop_state*, *loop_step*]

lemma *wf_measure_loop*: *wf measure_loop*
 ⟨*proof*⟩

lemma *measure_loop_induct* [*case_names Step*]:
 ⟦⋀*n*. ¬ *P* (*f n*) ⟹ (*f* (*Suc n*), (*f n*)) ∈ *measure_loop*⟧ ⟹ ∃ *n*. *P* (*f n*)
 ⟨*proof*⟩

lemma *inv_loop4_not_just_Oc*[*elim*]:
 ⟦*inv_loop4 x* (*l*′, []);
 length (*takeWhile* (λ*a*. *a* = *Oc*) (*rev l*′ @ [*Oc*])) ≠

length (*takeWhile* (λ*a*. *a* = *Oc*) (*rev l'*))⟧
⟹ *RR*
⟦*inv_loop4 x* (*l'*, *Bk* # *list*);
length (*takeWhile* (λ*a*. *a* = *Oc*) (*rev l'* @ *Oc* # *list*)) ≠
length (*takeWhile* (λ*a*. *a* = *Oc*) (*rev l'* @ *Bk* # *list*))⟧
⟹ *RR*
⟨*proof*⟩

lemma *takeWhile_replicate_append*:
P a ⟹ *takeWhile P* (*a*↑*x* @ *ys*) = *a*↑*x* @ *takeWhile P ys*
⟨*proof*⟩

lemma *takeWhile_replicate*:
P a ⟹ *takeWhile P* (*a*↑*x*) = *a*↑*x*
⟨*proof*⟩

lemma *inv_loop5_Bk_E*[*elim*]:
⟦*inv_loop5 x* (*l'*, *Bk* # *list*); *l'* ≠ []⟧;
length (*takeWhile* (λ*a*. *a* = *Oc*) (*rev* (*tl l'*) @ *hd l'* # *Bk* # *list*)) ≠
length (*takeWhile* (λ*a*. *a* = *Oc*) (*rev l'* @ *Bk* # *list*))⟧
⟹ *RR*
⟨*proof*⟩

lemma *inv_loop1_hd_Oc*[*elim*]: ⟦*inv_loop1 x* (*l'*, *Oc* # *list*)⟧ ⟹ *hd list* = *Oc*
⟨*proof*⟩

lemma *inv_loop6_not_just_Bk*[*dest*!]:
⟦*length* (*takeWhile P* (*rev* (*tl l'*) @ *hd l'* # *list*)) ≠
length (*takeWhile P* (*rev l'* @ *list*))⟧
⟹ *l'* = []
⟨*proof*⟩

lemma *inv_loop2_OcE*[*elim*]:
⟦*inv_loop2 x* (*l'*, *Oc* # *list*); *l'* ≠ []⟧ ⟹
length (*takeWhile* (λ*a*. *a* = *Oc*) (*rev l'* @ *Bk* # *list*)) <
length (*takeWhile* (λ*a*. *a* = *Oc*) (*rev l'* @ *Oc* # *list*))
⟨*proof*⟩

lemma *loop_halts*:
assumes *h*: *n* > *0 inv_loop n* (*1*, *l*, *r*)
shows ∃ *stp*. *is_final* (*steps0* (*1*, *l*, *r*) *tcopy_loop stp*)
⟨*proof*⟩

lemma *loop_correct*:
assumes *0* < *n*
shows {*inv_loop1 n*} *tcopy_loop* {*inv_loop0 n*}
⟨*proof*⟩

fun
inv_end5_loop :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_end5_exit :: *nat* ⇒ *tape* ⇒ *bool*
where
inv_end5_loop x (*l*, *r*) =
(∃ *i j*. *i* + *j* = *x* ∧ *x* > 0 ∧ *j* > 0 ∧ *l* = *Oc*↑*i* @ [*Bk*] ∧ *r* = *Oc*↑*j* @ *Bk* # *Oc*↑*x*)
| *inv_end5_exit x* (*l*, *r*) = (*x* > 0 ∧ *l* = [] ∧ *r* = *Bk* # *Oc*↑*x* @ *Bk* # *Oc*↑*x*)

fun
inv_end0 :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_end1 :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_end2 :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_end3 :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_end4 :: *nat* ⇒ *tape* ⇒ *bool* **and**
inv_end5 :: *nat* ⇒ *tape* ⇒ *bool*
where
inv_end0 n (*l*, *r*) = (*n* > 0 ∧ (*l*, *r*) = ([*Bk*], *Oc*↑*n* @ *Bk* # *Oc*↑*n*))
| *inv_end1 n* (*l*, *r*) = (*n* > 0 ∧ (*l*, *r*) = ([*Bk*], *Oc* # *Bk*↑*n* @ *Oc*↑*n*))
| *inv_end2 n* (*l*, *r*) = (∃ *i j*. *i* + *j* = *Suc n* ∧ *n* > 0 ∧ *l* = *Oc*↑*i* @ [*Bk*] ∧ *r* = *Bk*↑*j* @ *Oc*↑*n*)
| *inv_end3 n* (*l*, *r*) =
(∃ *i j*. *n* > 0 ∧ *i* + *j* = *n* ∧ *l* = *Oc*↑*i* @ [*Bk*] ∧ *r* = *Oc* # *Bk*↑*j*@ *Oc*↑*n*)
| *inv_end4 n* (*l*, *r*) = (∃ *any*. *n* > 0 ∧ *l* = *Oc*↑*n* @ [*Bk*] ∧ *r* = *any*#*Oc*↑*n*)
| *inv_end5 n* (*l*, *r*) = (*inv_end5_loop n* (*l*, *r*) ∨ *inv_end5_exit n* (*l*, *r*))

fun
inv_end :: *nat* ⇒ *config* ⇒ *bool*
where
inv_end n (*s*, *l*, *r*) = (*if s* = 0 *then inv_end0 n* (*l*, *r*)
else if s = 1 *then inv_end1 n* (*l*, *r*)
else if s = 2 *then inv_end2 n* (*l*, *r*)
else if s = 3 *then inv_end3 n* (*l*, *r*)
else if s = 4 *then inv_end4 n* (*l*, *r*)
else if s = 5 *then inv_end5 n* (*l*, *r*)
else False)

declare *inv_end*.*simps*[*simp del*] *inv_end1*.*simps*[*simp del*]
inv_end0.*simps*[*simp del*] *inv_end2*.*simps*[*simp del*]
inv_end3.*simps*[*simp del*] *inv_end4*.*simps*[*simp del*]
inv_end5.*simps*[*simp del*]

lemma *inv_end_nonempty*[*simp*]:
inv_end1 x (*b*, []) = *False*
inv_end1 x ([], *list*) = *False*
inv_end2 x (*b*, []) = *False*
inv_end3 x (*b*, []) = *False*
inv_end4 x (*b*, []) = *False*
inv_end5 x (*b*, []) = *False*

inv_end5 x ([], Oc # list) = False
⟨*proof*⟩

lemma *inv_end0_Bk_via_1*[*elim*]: ⟦*0 < x*; *inv_end1 x (b, Bk # list)*; *b ≠ []*⟧
⟹ *inv_end0 x (tl b, hd b # Bk # list)*
⟨*proof*⟩

lemma *inv_end3_Oc_via_2*[*elim*]: ⟦*0 < x*; *inv_end2 x (b, Bk # list)*⟧
⟹ *inv_end3 x (b, Oc # list)*
⟨*proof*⟩

lemma *inv_end2_Bk_via_3*[*elim*]: ⟦*0 < x*; *inv_end3 x (b, Bk # list)*⟧ ⟹ *inv_end2 x (Bk # b, list)*
⟨*proof*⟩

lemma *inv_end5_Bk_via_4*[*elim*]: ⟦*0 < x*; *inv_end4 x ([], Bk # list)*⟧ ⟹
inv_end5 x ([], Bk # Bk # list)
⟨*proof*⟩

lemma *inv_end5_Bk_tail_via_4*[*elim*]: ⟦*0 < x*; *inv_end4 x (b, Bk # list)*; *b ≠ []*⟧ ⟹
inv_end5 x (tl b, hd b # Bk # list)
⟨*proof*⟩

lemma *inv_end0_Bk_via_5*[*elim*]: ⟦*0 < x*; *inv_end5 x (b, Bk # list)*⟧ ⟹ *inv_end0 x (Bk # b, list)*
⟨*proof*⟩

lemma *inv_end2_Oc_via_1*[*elim*]: ⟦*0 < x*; *inv_end1 x (b, Oc # list)*⟧ ⟹ *inv_end2 x (Oc # b, list)*
⟨*proof*⟩

lemma *inv_end4_Bk_Oc_via_2*[*elim*]: ⟦*0 < x*; *inv_end2 x ([], Oc # list)*⟧ ⟹
inv_end4 x ([], Bk # Oc # list)
⟨*proof*⟩

lemma *inv_end4_Oc_via_2*[*elim*]: ⟦*0 < x*; *inv_end2 x (b, Oc # list)*; *b ≠ []*⟧ ⟹
inv_end4 x (tl b, hd b # Oc # list)
⟨*proof*⟩

lemma *inv_end2_Oc_via_3*[*elim*]: ⟦*0 < x*; *inv_end3 x (b, Oc # list)*⟧ ⟹ *inv_end2 x (Oc # b, list)*
⟨*proof*⟩

lemma *inv_end4_Bk_via_Oc*[*elim*]: ⟦*0 < x*; *inv_end4 x (b, Oc # list)*⟧ ⟹ *inv_end4 x (b, Bk # list)*
⟨*proof*⟩

lemma *inv_end5_Bk_drop_Oc*[*elim*]: ⟦*0 < x*; *inv_end5 x ([], Oc # list)*⟧ ⟹ *inv_end5 x ([], Bk # Oc # list)*
⟨*proof*⟩

declare *inv_end5_loop.simps*[*simp del*]
inv_end5_exit.simps[*simp del*]

lemma *inv_end5_exit_no_Oc*[*simp*]: *inv_end5_exit x* (*b*, *Oc* # *list*) = *False*
⟨*proof*⟩

lemma *inv_end5_loop_no_Bk_Oc*[*simp*]: *inv_end5_loop x* (*tl b*, *Bk* # *Oc* # *list*) = *False*
⟨*proof*⟩

lemma *inv_end5_exit_Bk_Oc_via_loop*[*elim*]:
⟦*0* < *x*; *inv_end5_loop x* (*b*, *Oc* # *list*); *b* ≠ []; *hd b* = *Bk*⟧ ⟹
inv_end5_exit x (*tl b*, *Bk* # *Oc* # *list*)
⟨*proof*⟩

lemma *inv_end5_loop_Oc_Oc_drop*[*elim*]:
⟦*0* < *x*; *inv_end5_loop x* (*b*, *Oc* # *list*); *b* ≠ []; *hd b* = *Oc*⟧ ⟹
inv_end5_loop x (*tl b*, *Oc* # *Oc* # *list*)
⟨*proof*⟩

lemma *inv_end5_Oc_tail*[*elim*]: ⟦*0* < *x*; *inv_end5 x* (*b*, *Oc* # *list*); *b* ≠ []⟧ ⟹
inv_end5 x (*tl b*, *hd b* # *Oc* # *list*)
⟨*proof*⟩

lemma *inv_end_step*:
⟦*x* > *0*; *inv_end x cf*⟧ ⟹ *inv_end x* (*step cf* (*tcopy_end*, *0*))
⟨*proof*⟩

lemma *inv_end_steps*:
⟦*x* > *0*; *inv_end x cf*⟧ ⟹ *inv_end x* (*steps cf* (*tcopy_end*, *0*) *stp*)
⟨*proof*⟩

fun *end_state* :: *config* ⇒ *nat*
where
end_state (*s*, *l*, *r*) =
(*if s* = *0 then 0*
else if s = *1 then 5*
else if s = *2* ∨ *s* = *3 then 4*
else if s = *4 then 3*
else if s = *5 then 2*
else 0)

fun *end_stage* :: *config* ⇒ *nat*
where
end_stage (*s*, *l*, *r*) =
(*if s* = *2* ∨ *s* = *3 then* (*length r*) *else 0*)

fun *end_step* :: *config* ⇒ *nat*
where
end_step (*s*, *l*, *r*) =

(*if s = 4 then (if hd r = Oc then 1 else 0)*
else if s = 5 then length l
else if s = 2 then 1
else if s = 3 then 0
else 0)

definition *end_LE* :: (*config* × *config*) *set*
where
end_LE = *measures* [*end_state*, *end_stage*, *end_step*]

lemma *wf_end_le*: *wf end_LE*
⟨*proof*⟩

lemma *halt_lemma*:
⟦*wf LE*; ∀ *n*. (¬ *P* (*f n*) ⟶ (*f* (*Suc n*), (*f n*)) ∈ *LE*)⟧ ⟹ ∃ *n*. *P* (*f n*)
⟨*proof*⟩

lemma *end_halt*:
⟦*x* > *0*; *inv_end x* (*Suc 0*, *l*, *r*)⟧ ⟹
∃ *stp*. *is_final* (*steps* (*Suc 0*, *l*, *r*) (*tcopy_end*, *0*) *stp*)
⟨*proof*⟩

lemma *end_correct*:
n > *0* ⟹ {*inv_end1 n*} *tcopy_end* {*inv_end0 n*}
⟨*proof*⟩

lemma *tm_wf_tcopy*[*intro*]:
tm_wf (*tcopy_begin*, *0*)
tm_wf (*tcopy_loop*, *0*)
tm_wf (*tcopy_end*, *0*)
⟨*proof*⟩

lemma *tcopy_correct1*:
assumes *0* < *x*
shows {*inv_begin1 x*} *tcopy* {*inv_end0 x*}
⟨*proof*⟩

abbreviation (*input*)
pre_tcopy n ≡ *def* λ*tp*. *tp* = ([]::*cell list*, *Oc* ↑ (*Suc n*))
abbreviation (*input*)
post_tcopy n ≡ *def* λ*tp*. *tp*= ([*Bk*], <(*n*, *n*::*nat*)>)

lemma *tcopy_correct*:
shows {*pre_tcopy n*} *tcopy* {*post_tcopy n*}
⟨*proof*⟩

5 The *Dithering* Turing Machine

The *Dithering* TM, when the input is *1*, it will loop forever, otherwise, it will terminate.

definition *dither* :: *instr list*
where
dither $\stackrel{def}{=}$ [(*W0*, *1*), (*R*, *2*), (*L*, *1*), (*L*, *0*)]

abbreviation (*input*)
dither_halt_inv $\stackrel{def}{=}$ $\lambda tp.\ \exists k.\ tp = (Bk \uparrow k, <1::nat>)$

abbreviation (*input*)
dither_unhalt_inv $\stackrel{def}{=}$ $\lambda tp.\ \exists k.\ tp = (Bk \uparrow k, <0::nat>)$

lemma *dither_loops_aux*:
(*steps0* (*1*, *Bk* ↑ *m*, [*Oc*]) *dither stp* = (*1*, *Bk* ↑ *m*, [*Oc*])) ∨
(*steps0* (*1*, *Bk* ↑ *m*, [*Oc*]) *dither stp* = (*2*, *Oc* # *Bk* ↑ *m*, []))
⟨*proof*⟩

lemma *dither_loops*:
shows {*dither_unhalt_inv*} *dither* ↑
⟨*proof*⟩

lemma *dither_halts_aux*:
shows *steps0* (*1*, *Bk* ↑ *m*, [*Oc*, *Oc*]) *dither 2* = (*0*, *Bk* ↑ *m*, [*Oc*, *Oc*])
⟨*proof*⟩

lemma *dither_halts*:
shows {*dither_halt_inv*} *dither* {*dither_halt_inv*}
⟨*proof*⟩

6 The diagnal argument below shows the undecidability of Halting problem

halts tp x means TM *tp* terminates on input *x* and the final configuration is standard.

definition *halts* :: *tprog0* ⇒ *nat list* ⇒ *bool*
where
halts p ns $\stackrel{def}{=}$ {($\lambda tp.\ tp = ([], <ns>)$)} *p* {($\lambda tp.\ (\exists k\ n\ l.\ tp = (Bk \uparrow k,\ <n::nat> @ Bk \uparrow l))$)}

lemma *tm_wf0_tcopy*[*intro*, *simp*]: *tm_wf0 tcopy*
⟨*proof*⟩

lemma *tm_wf0_dither*[*intro*, *simp*]: *tm_wf0 dither*
⟨*proof*⟩

The following locale specifies that TM *H* can be used to solve the *Halting Problem*

and *False* is going to be derived under this locale. Therefore, the undecidability of *Halting Problem* is established.

locale *uncomputable* =

fixes *code* :: *instr list* ⇒ *nat*

and *H* :: *instr list*
assumes *h_wf*[*intro*]: *tm_wf0 H*

and *h_case*:
⋀ *M ns*. *halts M ns* ⟹ {(λ*tp*. *tp* = ([*Bk*], <(*code M*, *ns*)>))} *H* {(λ*tp*. ∃ *k*. *tp* = (*Bk* ↑ *k*, <*0*::*nat*>))}
and *nh_case*:
⋀ *M ns*. ¬ *halts M ns* ⟹ {(λ*tp*. *tp* = ([*Bk*], <(*code M*, *ns*)>))} *H* {(λ*tp*. ∃ *k*. *tp* = (*Bk* ↑ *k*, <*1*::*nat*>))}
begin

abbreviation (*input*)
pre_H_inv M ns $\stackrel{def}{=}$ λ*tp*. *tp* = ([*Bk*], <(*code M*, *ns*::*nat list*)>)

abbreviation (*input*)
post_H_halt_inv $\stackrel{def}{=}$ λ*tp*. ∃ *k*. *tp* = (*Bk* ↑ *k*, <*1*::*nat*>)

abbreviation (*input*)
post_H_unhalt_inv $\stackrel{def}{=}$ λ*tp*. ∃ *k*. *tp* = (*Bk* ↑ *k*, <*0*::*nat*>)

lemma *H_halt_inv*:
assumes ¬ *halts M ns*
shows {*pre_H_inv M ns*} *H* {*post_H_halt_inv*}
⟨*proof*⟩

lemma *H_unhalt_inv*:
assumes *halts M ns*
shows {*pre_H_inv M ns*} *H* {*post_H_unhalt_inv*}
⟨*proof*⟩

definition
tcontra $\stackrel{def}{=}$ (*tcopy* |+| *H*) |+| *dither*
abbreviation
code_tcontra $\stackrel{def}{=}$ *code tcontra*

lemma *tcontra_unhalt*:
assumes ¬ *halts tcontra* [*code tcontra*]

shows *False*
⟨*proof*⟩

lemma *tcontra_halt*:
assumes *halts tcontra* [*code tcontra*]
shows *False*
⟨*proof*⟩

False can finally derived.

lemma *false*: *False*
⟨*proof*⟩

end

declare *replicate_Suc*[*simp del*]

end

7 Mopup Turing Machine that deletes all "registers", except one

theory *Abacus_Mopup*
imports *Uncomputable*
begin

fun *mopup_a* :: *nat* ⇒ *instr list*
where
mopup_a 0 = [] |
mopup_a (*Suc n*) = *mopup_a n* @
[(*R*, 2∗*n* + 3), (*W0*, 2∗*n* + 2), (*R*, 2∗*n* + 1), (*W1*, 2∗*n* + 2)]

definition *mopup_b* :: *instr list*
where
mopup_b $\stackrel{def}{=}$ [(*R*, 2), (*R*, 1), (*L*, 5), (*W0*, 3), (*R*, 4), (*W0*, 3),
(*R*, 2), (*W0*, 3), (*L*, 5), (*L*, 6), (*R*, 0), (*L*, 6)]

fun *mopup* :: *nat* ⇒ *instr list*
where
mopup n = *mopup_a n* @ *shift mopup_b* (2∗*n*)

type-synonym *mopup_type* = *config* ⇒ *nat list* ⇒ *nat* ⇒ *cell list* ⇒ *bool*

fun *mopup_stop* :: *mopup_type*
where
mopup_stop (*s*, *l*, *r*) *lm n ires*=
(∃ *ln rn*. *l* = *Bk*↑*ln* @ *Bk* # *Bk* # *ires* ∧ *r* = <*lm* ! *n*> @ *Bk*↑*rn*)

```
fun mopup_bef_erase_a :: mopup_type
 where
  mopup_bef_erase_a (s, l, r) lm n ires=
     (∃ ln m rn. l = Bk↑ln @ Bk # Bk # ires ∧
          r = Oc↑m@ Bk # <(drop ((s + 1) div 2) lm)> @ Bk↑rn)

fun mopup_bef_erase_b :: mopup_type
 where
  mopup_bef_erase_b (s, l, r) lm n ires =
    (∃ ln m rn. l = Bk↑ln @ Bk # Bk # ires ∧ r = Bk # Oc↑m @ Bk #
                          <(drop (s div 2) lm)> @ Bk↑rn)

fun mopup_jump_over1 :: mopup_type
 where
  mopup_jump_over1 (s, l, r) lm n ires =
    (∃ ln m1 m2 rn. m1 + m2 = Suc (lm ! n) ∧
      l = Oc↑m1 @ Bk↑ln @ Bk # Bk # ires ∧
    (r = Oc↑m2 @ Bk # <(drop (Suc n) lm)> @ Bk↑rn ∨
    (r = Oc↑m2 ∧ (drop (Suc n) lm) = [])))

fun mopup_aft_erase_a :: mopup_type
 where
  mopup_aft_erase_a (s, l, r) lm n ires =
    (∃ lnl lnr rn (ml::nat list) m.
       m = Suc (lm ! n) ∧ l = Bk↑lnr @ Oc↑m @ Bk↑lnl @ Bk # Bk # ires ∧
                        (r = <ml> @ Bk↑rn))

fun mopup_aft_erase_b :: mopup_type
 where
  mopup_aft_erase_b (s, l, r) lm n ires=
 (∃ lnl lnr rn (ml::nat list) m.
    m = Suc (lm ! n) ∧
    l = Bk↑lnr @ Oc↑m @ Bk↑lnl @ Bk # Bk # ires ∧
   (r = Bk # <ml> @ Bk↑rn ∨
    r = Bk # Bk # <ml> @ Bk↑rn))

fun mopup_aft_erase_c :: mopup_type
 where
  mopup_aft_erase_c (s, l, r) lm n ires =
(∃ lnl lnr rn (ml::nat list) m.
    m = Suc (lm ! n) ∧
    l = Bk↑lnr @ Oc↑m @ Bk↑lnl @ Bk # Bk # ires ∧
   (r = <ml> @ Bk↑rn ∨ r = Bk # <ml> @ Bk↑rn))

fun mopup_left_moving :: mopup_type
 where
  mopup_left_moving (s, l, r) lm n ires =
 (∃ lnl lnr rn m.
    m = Suc (lm ! n) ∧
  ((l = Bk↑lnr @ Oc↑m @ Bk↑lnl @ Bk # Bk # ires ∧ r = Bk↑rn) ∨
```

$(l = Oc{\uparrow}(m-1)\ @\ Bk{\uparrow}lnl\ @\ Bk\ \#\ Bk\ \#\ ires \wedge r = Oc\ \#\ Bk{\uparrow}rn)))$

fun *mopup_jump_over2* :: *mopup_type*
 where
  mopup_jump_over2 (s, l, r) *lm n ires* =
  $(\exists$ *ln rn m1 m2*.
     m1 + *m2* = *Suc* $(lm\ !\ n)$
    $\wedge\ r \neq []$
    $\wedge\ (hd\ r = Oc \longrightarrow (l = Oc{\uparrow}m1\ @\ Bk{\uparrow}ln\ @\ Bk\ \#\ Bk\ \#\ ires \wedge r = Oc{\uparrow}m2\ @\ Bk{\uparrow}rn))$
    $\wedge\ (hd\ r = Bk \longrightarrow (l = Bk{\uparrow}ln\ @\ Bk\ \#\ ires \wedge r = Bk\ \#\ Oc{\uparrow}(m1{+}m2)@\ Bk{\uparrow}rn)))$

fun *mopup_inv* :: *mopup_type*
 where
  mopup_inv (s, l, r) *lm n ires* =
   (*if s* = *0 then mopup_stop* (s, l, r) *lm n ires*
    else if s $\leq$ *2*∗*n then*
        if s mod 2 = *1 then mopup_bef_erase_a* (s, l, r) *lm n ires*
          else mopup_bef_erase_b (s, l, r) *lm n ires*
      else if s = *2*∗*n* + *1 then*
        mopup_jump_over1 (s, l, r) *lm n ires*
      else if s = *2*∗*n* + *2 then mopup_aft_erase_a* (s, l, r) *lm n ires*
      else if s = *2*∗*n* + *3 then mopup_aft_erase_b* (s, l, r) *lm n ires*
      else if s = *2*∗*n* + *4 then mopup_aft_erase_c* (s, l, r) *lm n ires*
      else if s = *2*∗*n* + *5 then mopup_left_moving* (s, l, r) *lm n ires*
      else if s = *2*∗*n* + *6 then mopup_jump_over2* (s, l, r) *lm n ires*
      else False)

lemma *mop_bef_length*[*simp*]: *length* $(mopup\_a\ n)$ = *4* ∗ *n*
 ⟨*proof*⟩

lemma *mopup_a_nth*:
 $[\![q < n; x < 4]\!] \Longrightarrow mopup\_a\ n\ !\ (4 * q + x) =$
                  $mopup\_a\ (Suc\ q)\ !\ ((4 * q) + x)$
⟨*proof*⟩

lemma *fetch_bef_erase_a_o*[*simp*]:
 $[\![0 < s; s \leq 2 * n; s\ mod\ 2 = Suc\ 0]\!]$
 $\Longrightarrow (fetch\ (mopup\_a\ n\ @\ shift\ mopup\_b\ (2 * n))\ s\ Oc) = (W0, s + 1)$
 ⟨*proof*⟩

lemma *fetch_bef_erase_a_b*[*simp*]:
 $[\![0 < s; s \leq 2 * n; s\ mod\ 2 = Suc\ 0]\!]$
  $\Longrightarrow$ $(fetch\ (mopup\_a\ n\ @\ shift\ mopup\_b\ (2 * n))\ s\ Bk) = (R, s + 2)$
 ⟨*proof*⟩

lemma *fetch_bef_erase_b_b*:
 assumes *n* < *length lm 0* < *s s* ≤ *2* ∗ *n s mod 2* = *0*
 shows $(fetch\ (mopup\_a\ n\ @\ shift\ mopup\_b\ (2 * n))\ s\ Bk) = (R, s - 1)$
⟨*proof*⟩

lemma *fetch_jump_over1_o*:
fetch (*mopup_a n* @ *shift mopup_b* (2 ∗ *n*)) (*Suc* (2 ∗ *n*)) *Oc*
= (*R*, *Suc* (2 ∗ *n*))
⟨*proof*⟩

lemma *fetch_jump_over1_b*:
fetch (*mopup_a n* @ *shift mopup_b* (2 ∗ *n*)) (*Suc* (2 ∗ *n*)) *Bk*
= (*R*, *Suc* (*Suc* (2 ∗ *n*)))
⟨*proof*⟩

lemma *fetch_aft_erase_a_o*:
fetch (*mopup_a n* @ *shift mopup_b* (2 ∗ *n*)) (*Suc* (*Suc* (2 ∗ *n*))) *Oc*
= (*W0*, *Suc* (2 ∗ *n* + 2))
⟨*proof*⟩

lemma *fetch_aft_erase_a_b*:
fetch (*mopup_a n* @ *shift mopup_b* (2 ∗ *n*)) (*Suc* (*Suc* (2 ∗ *n*))) *Bk*
= (*L*, *Suc* (2 ∗ *n* + 4))
⟨*proof*⟩

lemma *fetch_aft_erase_b_b*:
fetch (*mopup_a n* @ *shift mopup_b* (2 ∗ *n*)) (2∗*n* + 3) *Bk*
= (*R*, *Suc* (2 ∗ *n* + 3))
⟨*proof*⟩

lemma *fetch_aft_erase_c_o*:
fetch (*mopup_a n* @ *shift mopup_b* (2 ∗ *n*)) (2 ∗ *n* + 4) *Oc*
= (*W0*, *Suc* (2 ∗ *n* + 2))
⟨*proof*⟩

lemma *fetch_aft_erase_c_b*:
fetch (*mopup_a n* @ *shift mopup_b* (2 ∗ *n*)) (2 ∗ *n* + 4) *Bk*
= (*R*, *Suc* (2 ∗ *n* + 1))
⟨*proof*⟩

lemma *fetch_left_moving_o*:
(*fetch* (*mopup_a n* @ *shift mopup_b* (2 ∗ *n*)) (2 ∗ *n* + 5) *Oc*)
= (*L*, 2∗*n* + 6)
⟨*proof*⟩

lemma *fetch_left_moving_b*:
(*fetch* (*mopup_a n* @ *shift mopup_b* (2 ∗ *n*)) (2 ∗ *n* + 5) *Bk*)
= (*L*, 2∗*n* + 5)
⟨*proof*⟩

lemma *fetch_jump_over2_b*:
(*fetch* (*mopup_a n* @ *shift mopup_b* (2 ∗ *n*)) (2 ∗ *n* + 6) *Bk*)
= (*R*, *0*)
⟨*proof*⟩

lemma *fetch_jump_over2_o*:
(*fetch* (*mopup_a n* @ *shift mopup_b* (2 ∗ *n*)) (2 ∗ *n* + 6) *Oc*)
= (*L*, 2∗*n* + 6)
⟨*proof*⟩

lemmas *mopupfetchs* =
fetch_bef_erase_a_o fetch_bef_erase_a_b fetch_bef_erase_b_b
fetch_jump_over1_o fetch_jump_over1_b fetch_aft_erase_a_o
fetch_aft_erase_a_b fetch_aft_erase_b_b fetch_aft_erase_c_o
fetch_aft_erase_c_b fetch_left_moving_o fetch_left_moving_b
fetch_jump_over2_b fetch_jump_over2_o

declare
mopup_jump_over2.simps[*simp del*] *mopup_left_moving.simps*[*simp del*]
mopup_aft_erase_c.simps[*simp del*] *mopup_aft_erase_b.simps*[*simp del*]
mopup_aft_erase_a.simps[*simp del*] *mopup_jump_over1.simps*[*simp del*]
mopup_bef_erase_a.simps[*simp del*] *mopup_bef_erase_b.simps*[*simp del*]
mopup_stop.simps[*simp del*]

lemma *mopup_bef_erase_b_Bk_via_a_Oc*[*simp*]:
⟦*mopup_bef_erase_a* (*s*, *l*, *Oc* # *xs*) *lm n ires*⟧ ⟹
mopup_bef_erase_b (*Suc s*, *l*, *Bk* # *xs*) *lm n ires*
⟨*proof*⟩

lemma *mopup_false1*:
⟦0 < *s*; *s* ≤ 2 ∗ *n*; *s mod* 2 = *Suc* 0; ¬ *Suc s* ≤ 2 ∗ *n*⟧
⟹ *RR*
⟨*proof*⟩

lemma *mopup_bef_erase_a_implies_two*[*simp*]:
⟦*n* < *length lm*; 0 < *s*; *s* ≤ 2 ∗ *n*; *s mod* 2 = *Suc* 0;
mopup_bef_erase_a (*s*, *l*, *Oc* # *xs*) *lm n ires*; *r* = *Oc* # *xs*⟧
⟹ (*Suc s* ≤ 2 ∗ *n* ⟶ *mopup_bef_erase_b* (*Suc s*, *l*, *Bk* # *xs*) *lm n ires*) ∧
(¬ *Suc s* ≤ 2 ∗ *n* ⟶ *mopup_jump_over1* (*Suc s*, *l*, *Bk* # *xs*) *lm n ires*)
⟨*proof*⟩

lemma *tape_of_nl_cons*: <*m* # *lm*> = (*if lm* = [] *then Oc*↑(*Suc m*)
else Oc↑(*Suc m*) @ *Bk* # <*lm*>)
⟨*proof*⟩

lemma *drop_tape_of_cons*:
⟦*Suc q* < *length lm*; *x* = *lm* ! *q*⟧ ⟹ <*drop q lm*> = *Oc* # *Oc* ↑ *x* @ *Bk* # <*drop* (*Suc q*) *lm*>
⟨*proof*⟩

lemma *erase2jumpover1*:
⟦*q* < *length list*;
∀ *rn*. <*drop q list*> ≠ *Oc* # *Oc* ↑ (*list* ! *q*) @ *Bk* # <*drop* (*Suc q*) *list*> @ *Bk* ↑ *rn*⟧
⟹ <*drop q list*> = *Oc* # *Oc* ↑ (*list* ! *q*)
⟨*proof*⟩

lemma *erase2jumpover2*:
$[\![q < \mathit{length}\ \mathit{list}; \forall\, rn.\ <\mathit{drop}\ q\ \mathit{list}> @ Bk \# Bk \uparrow n \neq$
$Oc \# Oc \uparrow (\mathit{list} ! q) @ Bk \# <\mathit{drop}\ (\mathit{Suc}\ q)\ \mathit{list}> @ Bk \uparrow rn]\!]$
$\Longrightarrow RR$
$\langle proof \rangle$

lemma *mod_ex1*: $(a\ \mathit{mod}\ 2 = \mathit{Suc}\ 0) = (\exists\ q.\ a = \mathit{Suc}\ (2 * q))$
$\langle proof \rangle$

declare *replicate_Suc*[*simp*]

lemma *mopup_bef_erase_a_2_jump_over*[*simp*]:
$[\![n < \mathit{length}\ lm; 0 < s; s\ \mathit{mod}\ 2 = \mathit{Suc}\ 0;\ s \leq 2 * n;$
$\mathit{mopup\_bef\_erase\_a}\ (s, l, Bk \# xs)\ lm\ n\ ires; \neg\ (\mathit{Suc}\ (\mathit{Suc}\ s) \leq 2 * n)]\!]$
$\Longrightarrow \mathit{mopup\_jump\_over1}\ (s', Bk \# l, xs)\ lm\ n\ ires$
$\langle proof \rangle$

lemma *Suc_Suc_div*: $[\![0 < s; s\ \mathit{mod}\ 2 = \mathit{Suc}\ 0; \mathit{Suc}\ (\mathit{Suc}\ s) \leq 2 * n]\!]$
$\Longrightarrow (\mathit{Suc}\ (\mathit{Suc}\ (s\ \mathit{div}\ 2))) \leq n\ \langle proof \rangle$

lemma *mopup_bef_erase_a_2_a*[*simp*]:
assumes $n < \mathit{length}\ lm\ 0 < s\ s\ \mathit{mod}\ 2 = \mathit{Suc}\ 0$
$\mathit{mopup\_bef\_erase\_a}\ (s, l, Bk \# xs)\ lm\ n\ ires$
$\mathit{Suc}\ (\mathit{Suc}\ s) \leq 2 * n$
shows $\mathit{mopup\_bef\_erase\_a}\ (\mathit{Suc}\ (\mathit{Suc}\ s), Bk \# l, xs)\ lm\ n\ ires$
$\langle proof \rangle$

lemma *mopup_false2*:
$[\![0 < s; s \leq 2 * n;$
$s\ \mathit{mod}\ 2 = \mathit{Suc}\ 0; \mathit{Suc}\ s \neq 2 * n;$
$\neg\ \mathit{Suc}\ (\mathit{Suc}\ s) \leq 2 * n]\!] \Longrightarrow RR$
$\langle proof \rangle$

lemma *ariths*[*simp*]: $[\![0 < s; s \leq 2 * n; s\ \mathit{mod}\ 2 \neq \mathit{Suc}\ 0]\!] \Longrightarrow$
$(s - \mathit{Suc}\ 0)\ \mathit{mod}\ 2 = \mathit{Suc}\ 0$
$[\![0 < s; s \leq 2 * n; s\ \mathit{mod}\ 2 \neq \mathit{Suc}\ 0]\!] \Longrightarrow$
$s - \mathit{Suc}\ 0 \leq 2 * n$
$[\![0 < s; s \leq 2 * n; s\ \mathit{mod}\ 2 \neq \mathit{Suc}\ 0]\!] \Longrightarrow \neg\ s \leq \mathit{Suc}\ 0$
$\langle proof \rangle$

lemma *take_suc*[*intro*]:
$\exists\, lna.\ Bk \# Bk \uparrow ln = Bk \uparrow lna$
$\langle proof \rangle$

lemma *mopup_bef_erase*[*simp*]: $\mathit{mopup\_bef\_erase\_a}\ (s, l, [])\ lm\ n\ ires \Longrightarrow$
$\mathit{mopup\_bef\_erase\_a}\ (s, l, [Bk])\ lm\ n\ ires$
$[\![n < \mathit{length}\ lm; 0 < s; s \leq 2 * n; s\ \mathit{mod}\ 2 = \mathit{Suc}\ 0; \neg\ \mathit{Suc}\ (\mathit{Suc}\ s) \leq 2 * n;$

$mopup\_bef\_erase\_a\ (s, l, [])\ lm\ n\ ires]\!]$
$\Longrightarrow mopup\_jump\_over1\ (s', Bk \# l, [])\ lm\ n\ ires$
$mopup\_bef\_erase\_b\ (s, l, Oc \# xs)\ lm\ n\ ires \Longrightarrow l \neq []$
$[\![n < length\ lm; 0 < s; s \leq 2 * n;$
$s\ mod\ 2 \neq Suc\ 0;$
$mopup\_bef\_erase\_b\ (s, l, Bk \# xs)\ lm\ n\ ires; r = Bk \# xs]\!]$
$\Longrightarrow mopup\_bef\_erase\_a\ (s - Suc\ 0, Bk \# l, xs)\ lm\ n\ ires$
$[\![mopup\_bef\_erase\_b\ (s, l, [])\ lm\ n\ ires]\!] \Longrightarrow$
$mopup\_bef\_erase\_a\ (s - Suc\ 0, Bk \# l, [])\ lm\ n\ ires$
⟨*proof*⟩

lemma *mopup_jump_over1_in_ctx*[*simp*]:
assumes $mopup\_jump\_over1\ (Suc\ (2 * n), l, Oc \# xs)\ lm\ n\ ires$
shows $mopup\_jump\_over1\ (Suc\ (2 * n), Oc \# l, xs)\ lm\ n\ ires$
⟨*proof*⟩

lemma *mopup_jump_over1_2_aft_erase_a*[*simp*]:
assumes $mopup\_jump\_over1\ (Suc\ (2 * n), l, Bk \# xs)\ lm\ n\ ires$
shows $mopup\_aft\_erase\_a\ (Suc\ (Suc\ (2 * n)), Bk \# l, xs)\ lm\ n\ ires$
⟨*proof*⟩

lemma *mopup_aft_erase_a_via_jump_over1*[*simp*]:
$[\![mopup\_jump\_over1\ (Suc\ (2 * n), l, [])\ lm\ n\ ires]\!] \Longrightarrow$
$mopup\_aft\_erase\_a\ (Suc\ (Suc\ (2 * n)), Bk \# l, [])\ lm\ n\ ires$
⟨*proof*⟩

lemma *tape_of_list_empty*[*simp*]: $<[]> = []$ ⟨*proof*⟩

lemma *mopup_aft_erase_b_via_a*[*simp*]:
assumes $mopup\_aft\_erase\_a\ (Suc\ (Suc\ (2 * n)), l, Oc \# xs)\ lm\ n\ ires$
shows $mopup\_aft\_erase\_b\ (Suc\ (Suc\ (Suc\ (2 * n))), l, Bk \# xs)\ lm\ n\ ires$
⟨*proof*⟩

lemma *mopup_left_moving_via_aft_erase_a*[*simp*]:
assumes $mopup\_aft\_erase\_a\ (Suc\ (Suc\ (2 * n)), l, Bk \# xs)\ lm\ n\ ires$
shows $mopup\_left\_moving\ (5 + 2 * n, tl\ l, hd\ l \# Bk \# xs)\ lm\ n\ ires$
⟨*proof*⟩

lemma *mopup_aft_erase_a_nonempty*[*simp*]:
$mopup\_aft\_erase\_a\ (Suc\ (Suc\ (2 * n)), l, xs)\ lm\ n\ ires \Longrightarrow l \neq []$
⟨*proof*⟩

lemma *mopup_left_moving_via_aft_erase_a_emptylst*[*simp*]:
assumes $mopup\_aft\_erase\_a\ (Suc\ (Suc\ (2 * n)), l, [])\ lm\ n\ ires$
shows $mopup\_left\_moving\ (5 + 2 * n, tl\ l, [hd\ l])\ lm\ n\ ires$
⟨*proof*⟩

lemma *mopup_aft_erase_b_no_Oc*[*simp*]: $mopup\_aft\_erase\_b\ (2 * n + 3, l, Oc \# xs)\ lm\ n\ ires$
$= False$

⟨*proof*⟩

lemma *tape_of_ex1*[*intro*]:
∃ *rna ml*. *Oc* ↑ *a* @ *Bk* ↑ *rn* = <*ml*::*nat list*> @ *Bk* ↑ *rna* ∨ *Oc* ↑ *a* @ *Bk* ↑ *rn* = *Bk* # <*ml*> @ *Bk* ↑ *rna*
⟨*proof*⟩

lemma *mopup_aft_erase_b_via_c_helper*: ∃ *rna ml*. *Oc* ↑ *a* @ *Bk* # <*list*::*nat list*> @ *Bk* ↑ *rn* =
<*ml*> @ *Bk* ↑ *rna* ∨ *Oc* ↑ *a* @ *Bk* # <*list*> @ *Bk* ↑ *rn* = *Bk* # <*ml*::*nat list*> @ *Bk* ↑ *rna*
⟨*proof*⟩

lemma *mopup_aft_erase_b_via_c*[*simp*]:
assumes *mopup_aft_erase_c* (2 ∗ *n* + 4, *l*, *Oc* # *xs*) *lm n ires*
shows *mopup_aft_erase_b* (*Suc* (*Suc* (*Suc* (2 ∗ *n*))), *l*, *Bk* # *xs*) *lm n ires*
⟨*proof*⟩

lemma *mopup_aft_erase_c_aft_erase_a*[*simp*]:
assumes *mopup_aft_erase_c* (2 ∗ *n* + 4, *l*, *Bk* # *xs*) *lm n ires*
shows *mopup_aft_erase_a* (*Suc* (*Suc* (2 ∗ *n*)), *Bk* # *l*, *xs*) *lm n ires*
⟨*proof*⟩

lemma *mopup_aft_erase_a_via_c*[*simp*]:
⟦*mopup_aft_erase_c* (2 ∗ *n* + 4, *l*, []) *lm n ires*⟧
⟹ *mopup_aft_erase_a* (*Suc* (*Suc* (2 ∗ *n*)), *Bk* # *l*, []) *lm n ires*
⟨*proof*⟩

lemma *mopup_aft_erase_b_2_aft_erase_c*[*simp*]:
assumes *mopup_aft_erase_b* (2 ∗ *n* + 3, *l*, *Bk* # *xs*) *lm n ires*
shows *mopup_aft_erase_c* (4 + 2 ∗ *n*, *Bk* # *l*, *xs*) *lm n ires*
⟨*proof*⟩

lemma *mopup_aft_erase_c_via_b*[*simp*]:
⟦*mopup_aft_erase_b* (2 ∗ *n* + 3, *l*, []) *lm n ires*⟧
⟹ *mopup_aft_erase_c* (4 + 2 ∗ *n*, *Bk* # *l*, []) *lm n ires*
⟨*proof*⟩

lemma *mopup_left_moving_nonempty*[*simp*]:
mopup_left_moving (2 ∗ *n* + 5, *l*, *Oc* # *xs*) *lm n ires* ⟹ *l* ≠ []
⟨*proof*⟩

lemma *exp_ind*: *a*↑(*Suc x*) = *a*↑*x* @ [*a*]
⟨*proof*⟩

lemma *mopup_jump_over2_via_left_moving*[*simp*]:
⟦*mopup_left_moving* (2 ∗ *n* + 5, *l*, *Oc* # *xs*) *lm n ires*⟧
⟹ *mopup_jump_over2* (2 ∗ *n* + 6, *tl l*, *hd l* # *Oc* # *xs*) *lm n ires*
⟨*proof*⟩

lemma *mopup_left_moving_nonempty_snd*[*simp*]: *mopup_left_moving* (2 ∗ *n* + 5, *l*, *xs*) *lm n*

ires $\Longrightarrow$ $l \neq []$
⟨*proof*⟩

lemma *mopup_left_moving_hd_Bk*[*simp*]:
⟦*mopup_left_moving* $(2 * n + 5, l, Bk \# xs)$ *lm n ires*⟧
$\Longrightarrow$ *mopup_left_moving* $(2 * n + 5, tl\ l, hd\ l \# Bk \# xs)$ *lm n ires*
⟨*proof*⟩

lemma *mopup_left_moving_emptylist*[*simp*]:
⟦*mopup_left_moving* $(2 * n + 5, l, [])$ *lm n ires*⟧
$\Longrightarrow$ *mopup_left_moving* $(2 * n + 5, tl\ l, [hd\ l])$ *lm n ires*
⟨*proof*⟩

lemma *mopup_jump_over2_Oc_nonempty*[*simp*]:
mopup_jump_over2 $(2 * n + 6, l, Oc \# xs)$ *lm n ires* $\Longrightarrow$ $l \neq []$
⟨*proof*⟩

lemma *mopup_jump_over2_context*[*simp*]:
⟦*mopup_jump_over2* $(2 * n + 6, l, Oc \# xs)$ *lm n ires*⟧
$\Longrightarrow$ *mopup_jump_over2* $(2 * n + 6, tl\ l, hd\ l \# Oc \# xs)$ *lm n ires*
⟨*proof*⟩

lemma *mopup_stop_via_jump_over2*[*simp*]:
⟦*mopup_jump_over2* $(2 * n + 6, l, Bk \# xs)$ *lm n ires*⟧
$\Longrightarrow$ *mopup_stop* $(0, Bk \# l, xs)$ *lm n ires*
⟨*proof*⟩

lemma *mopup_jump_over2_nonempty*[*simp*]: *mopup_jump_over2* $(2 * n + 6, l, [])$ *lm n ires* = *False*
⟨*proof*⟩

declare *fetch.simps*[*simp del*]
lemma *mod_ex2*: $(a\ mod\ (2::nat) = 0) = (\exists\ q.\ a = 2 * q)$
⟨*proof*⟩

lemma *mod_2*: *x mod 2* $= 0 \vee$ *x mod 2* $=$ *Suc 0*
⟨*proof*⟩

lemma *mopup_inv_step*:
⟦$n <$ *length lm*; *mopup_inv* (s, l, r) *lm n ires*⟧
$\Longrightarrow$ *mopup_inv* $(step\ (s, l, r)\ (mopup\_a\ n\ @\ shift\ mopup\_b\ (2 * n), 0))$ *lm n ires*
⟨*proof*⟩

declare *mopup_inv.simps*[*simp del*]
lemma *mopup_inv_steps*:
⟦$n <$ *length lm*; *mopup_inv* (s, l, r) *lm n ires*⟧ $\Longrightarrow$
mopup_inv $(steps\ (s, l, r)\ (mopup\_a\ n\ @\ shift\ mopup\_b\ (2 * n), 0)\ \ stp)$ *lm n ires*
⟨*proof*⟩

fun *abc_mopup_stage1* :: *config* ⇒ *nat* ⇒ *nat*
where
abc_mopup_stage1 (*s*, *l*, *r*) *n* =
(*if s* > 0 ∧ *s* ≤ 2∗*n then* 6
else if s = 2∗*n* + 1 *then* 4
else if s ≥ 2∗*n* + 2 ∧ *s* ≤ 2∗*n* + 4 *then* 3
else if s = 2∗*n* + 5 *then* 2
else if s = 2∗*n* + 6 *then* 1
else 0)

fun *abc_mopup_stage2* :: *config* ⇒ *nat* ⇒ *nat*
where
abc_mopup_stage2 (*s*, *l*, *r*) *n* =
(*if s* > 0 ∧ *s* ≤ 2∗*n then length r*
else if s = 2∗*n* + 1 *then length r*
else if s = 2∗*n* + 5 *then length l*
else if s = 2∗*n* + 6 *then length l*
else if s ≥ 2∗*n* + 2 ∧ *s* ≤ 2∗*n* + 4 *then length r*
else 0)

fun *abc_mopup_stage3* :: *config* ⇒ *nat* ⇒ *nat*
where
abc_mopup_stage3 (*s*, *l*, *r*) *n* =
(*if s* > 0 ∧ *s* ≤ 2∗*n then*
if hd r = *Bk then* 0
else 1
else if s = 2∗*n* + 2 *then* 1
else if s = 2∗*n* + 3 *then* 0
else if s = 2∗*n* + 4 *then* 2
else 0)

definition
abc_mopup_measure = *measures* [λ(*c*, *n*). *abc_mopup_stage1 c n*,
λ(*c*, *n*). *abc_mopup_stage2 c n*,
λ(*c*, *n*). *abc_mopup_stage3 c n*]

lemma *wf_abc_mopup_measure*:
shows *wf abc_mopup_measure*
⟨*proof*⟩

lemma *abc_mopup_measure_induct* [*case_names Step*]:
⟦⋀*n*. ¬ *P* (*f n*) ⟹ (*f* (*Suc n*), (*f n*)) ∈ *abc_mopup_measure*⟧ ⟹ ∃ *n*. *P* (*f n*)
⟨*proof*⟩

lemma *mopup_erase_nonempty*[*simp*]:
mopup_bef_erase_a (*a*, *aa*, []) *lm n ires* = *False*
mopup_bef_erase_b (*a*, *aa*, []) *lm n ires* = *False*
mopup_aft_erase_b (2 ∗ *n* + 3, *aa*, []) *lm n ires* = *False*
⟨*proof*⟩

declare *mopup_inv.simps*[*simp del*]

lemma *fetch_mopup_a_shift*[*simp*]:
 assumes $0 < q\ q \leq n$
 shows *fetch* (*mopup_a n* @ *shift mopup_b* (2 ∗ *n*)) (2∗*q*) *Bk* = (*R*, 2∗*q* − *1*)
⟨*proof*⟩

lemma *mopup_halt*:
 assumes
  less: *n* < *length lm*
  and *inv*: *mopup_inv* (*Suc 0*, *l*, *r*) *lm n ires*
  and *f*: $f = (\lambda\ stp.\ (steps\ (Suc\ 0, l, r)\ (mopup\_a\ n\ @\ shift\ mopup\_b\ (2 * n), 0)\ stp, n))$
  and *P*: $P = (\lambda\ (c, n).\ is\_final\ c)$
 shows ∃ *stp*. *P* (*f stp*)
⟨*proof*⟩

lemma *mopup_inv_start*:
 n < *length am* ⟹ *mopup_inv* (*Suc 0*, *Bk* # *Bk* # *ires*, <*am*> @ *Bk* ↑ *k*) *am n ires*
 ⟨*proof*⟩

lemma *mopup_correct*:
 assumes *less*: *n* < *length* (*am*::*nat list*)
  and *rs*: *am* ! *n* = *rs*
 shows ∃ *stp i j*. (*steps* (*Suc 0*, *Bk* # *Bk* # *ires*, <*am*> @ *Bk* ↑ *k*) (*mopup_a n* @ *shift mopup_b* (2 ∗ *n*), *0*) *stp*)
  = (*0*, *Bk*↑*i* @ *Bk* # *Bk* # *ires*, *Oc* # *Oc*↑ *rs* @ *Bk*↑*j*)
 ⟨*proof*⟩

lemma *wf_mopup*[*intro*]: *tm_wf* (*mopup n*, *0*)
 ⟨*proof*⟩

end

8 Abacus Machines

theory *Abacus*
 imports *Turing_Hoare Abacus_Mopup*
begin

declare *replicate_Suc*[*simp add*]

datatype *abc_inst* =
 Inc nat
 | *Dec nat nat*
 | *Goto nat*

type-synonym *abc_prog* = *abc_inst list*

type-synonym *abc_state* = *nat*

The memory of Abacus machine is defined as a list of contents, with every units addressed by index into the list.

type-synonym *abc_lm* = *nat list*

Fetching contents out of memory. Units not represented by list elements are considered as having content *0*.

fun *abc_lm_v* :: *abc_lm* ⇒ *nat* ⇒ *nat*
where
abc_lm_v lm n = (*if* (*n* < *length lm*) *then* (*lm*!*n*) *else 0*)

Set the content of memory unit *n* to value *v*. *am* is the Abacus memory before setting. If address *n* is outside to scope of *am*, *am* is extended so that *n* becomes in scope.

fun *abc_lm_s* :: *abc_lm* ⇒ *nat* ⇒ *nat* ⇒ *abc_lm*
where
abc_lm_s am n v = (*if* (*n* < *length am*) *then* (*am*[*n*:=*v*]) *else*
am@ (*replicate* (*n* − *length am*) *0*) @ [*v*])

The configuration of Abaucs machines consists of its current state and its current memory:

type-synonym *abc_conf* = *abc_state* × *abc_lm*

Fetch instruction out of Abacus program:

fun *abc_fetch* :: *nat* ⇒ *abc_prog* ⇒ *abc_inst option*
where
abc_fetch s p = (*if* (*s* < *length p*) *then Some* (*p* ! *s*) *else None*)

Single step execution of Abacus machine. If no instruction is feteched, configuration does not change.

fun *abc_step_l* :: *abc_conf* ⇒ *abc_inst option* ⇒ *abc_conf*
where
abc_step_l (*s*, *lm*) *a* = (*case a of*
None ⇒ (*s*, *lm*) |
Some (*Inc n*) ⇒ (*let nv* = *abc_lm_v lm n in*
(*s* + *1*, *abc_lm_s lm n* (*nv* + *1*))) |
Some (*Dec n e*) ⇒ (*let nv* = *abc_lm_v lm n in*
if (*nv* = *0*) *then* (*e*, *abc_lm_s lm n 0*)
else (*s* + *1*, *abc_lm_s lm n* (*nv* − *1*))) |
Some (*Goto n*) ⇒ (*n*, *lm*)
)

Multi-step execution of Abacus machine.

fun *abc_steps_l* :: *abc_conf* ⇒ *abc_prog* ⇒ *nat* ⇒ *abc_conf*
where

abc_steps_l (*s*, *lm*) *p* 0 = (*s*, *lm*) |
abc_steps_l (*s*, *lm*) *p* (*Suc n*) =
abc_steps_l (*abc_step_l* (*s*, *lm*) (*abc_fetch s p*)) *p n*

9 Compiling Abacus machines into Turing machines

9.1 Compiling functions

findnth n returns the TM which locates the represention of memory cell *n* on the tape and changes representation of zero on the way.

fun *findnth* :: *nat* ⇒ *instr list*
where
findnth 0 = [] |
findnth (*Suc n*) = (*findnth n* @ [(*W1*, 2 ∗ *n* + 1),
(*R*, 2 ∗ *n* + 2), (*R*, 2 ∗ *n* + 3), (*R*, 2 ∗ *n* + 2)])

tinc_b returns the TM which increments the representation of the memory cell under rw-head by one and move the representation of cells afterwards to the right accordingly.

definition *tinc_b* :: *instr list*
where
tinc_b $\stackrel{def}{=}$ [(*W1*, *1*), (*R*, *2*), (*W1*, *3*), (*R*, *2*), (*W1*, *3*), (*R*, *4*),
(*L*, *7*), (*W0*, *5*), (*R*, *6*), (*W0*, *5*), (*W1*, *3*), (*R*, *6*),
(*L*, *8*), (*L*, *7*), (*R*, *9*), (*L*, *7*), (*R*, *10*), (*W0*, *9*)]

tinc ss n returns the TM which simulates the execution of Abacus instruction *Inc n*, assuming that TM is located at location *ss* in the final TM complied from the whole Abacus program.

fun *tinc* :: *nat* ⇒ *nat* ⇒ *instr list*
where
tinc ss n = *shift* (*findnth n* @ *shift tinc_b* (2 ∗ *n*)) (*ss* − *1*)

tinc_b returns the TM which decrements the representation of the memory cell under rw-head by one and move the representation of cells afterwards to the left accordingly.

definition *tdec_b* :: *instr list*
where
tdec_b $\stackrel{def}{=}$ [(*W1*, *1*), (*R*, *2*), (*L*, *14*), (*R*, *3*), (*L*, *4*), (*R*, *3*),
(*R*, *5*), (*W0*, *4*), (*R*, *6*), (*W0*, *5*), (*L*, *7*), (*L*, *8*),
(*L*, *11*), (*W0*, *7*), (*W1*, *8*), (*R*, *9*), (*L*, *10*), (*R*, *9*),
(*R*, *5*), (*W0*, *10*), (*L*, *12*), (*L*, *11*), (*R*, *13*), (*L*, *11*),
(*R*, *17*), (*W0*, *13*), (*L*, *15*), (*L*, *14*), (*R*, *16*), (*L*, *14*),
(*R*, *0*), (*W0*, *16*)]

tdec ss n label returns the TM which simulates the execution of Abacus instruction *Dec n label*, assuming that TM is located at location *ss* in the final TM complied from the whole Abacus program.

```
fun tdec :: nat ⇒ nat ⇒ nat ⇒ instr list
 where
  tdec ss n e = shift (findnth n) (ss − 1) @ adjust (shift (shift tdec_b (2 ∗ n)) (ss − 1)) e
```

tgoto f(*label*) returns the TM simulating the execution of Abacus instruction *Goto label*, where *f*(*label*) is the corresponding location of *label* in the final TM compiled from the overall Abacus program.

```
fun tgoto :: nat ⇒ instr list
 where
  tgoto n = [(Nop, n), (Nop, n)]
```

The layout of the final TM compiled from an Abacus program is represented as a list of natural numbers, where the list element at index *n* represents the starting state of the TM simulating the execution of *n*-th instruction in the Abacus program.

```
type-synonym layout = nat list
```

length_of i is the length of the TM simulating the Abacus instruction *i*.

```
fun length_of :: abc_inst ⇒ nat
 where
  length_of i = (case i of
                 Inc n   ⇒ 2 ∗ n + 9 |
                 Dec n e ⇒ 2 ∗ n + 16 |
                 Goto n  ⇒ 1)
```

layout_of ap returns the layout of Abacus program *ap*.

```
fun layout_of :: abc_prog ⇒ layout
 where layout_of ap = map length_of ap
```

start_of layout n looks out the starting state of *n*-th TM in the finall TM.

```
fun start_of :: nat list ⇒ nat ⇒ nat
 where
  start_of ly x = (Suc (sum_list (take x ly)))
```

ci lo ss i complies Abacus instruction *i* assuming the TM of *i* starts from state *ss* within the overal layout *lo*.

```
fun ci :: layout ⇒ nat ⇒ abc_inst ⇒ instr list
 where
  ci ly ss (Inc n) = tinc ss n
| ci ly ss (Dec n e) = tdec ss n (start_of ly e)
| ci ly ss (Goto n) = tgoto (start_of ly n)
```

tpairs_of ap transfroms Abacus program *ap* pairing every instruction with its starting state.

```
fun tpairs_of :: abc_prog ⇒ (nat × abc_inst) list
 where tpairs_of ap = (zip (map (start_of (layout_of ap))
                            [0..<(length ap)]) ap)
```

tms_of ap returns the list of TMs, where every one of them simulates the corresponding Abacus intruction in *ap*.

fun *tms_of* :: *abc_prog* ⇒ (*instr list*) *list*
where *tms_of ap* = *map* (λ (*n*, *tm*). *ci* (*layout_of ap*) *n tm*)
(*tpairs_of ap*)

tm_of ap returns the final TM machine compiled from Abacus program *ap*.

fun *tm_of* :: *abc_prog* ⇒ *instr list*
where *tm_of ap* = *concat* (*tms_of ap*)

lemma *length_findnth*:
length (*findnth n*) = 4 ∗ *n*
⟨*proof*⟩

lemma *ci_length* : *length* (*ci ns n ai*) *div* 2 = *length_of ai*
⟨*proof*⟩

9.2 Representation of Abacus memory by TM tapes

crsp acf tcf meams the abacus configuration *acf* is corretly represented by the TM configuration *tcf*.

fun *crsp* :: *layout* ⇒ *abc_conf* ⇒ *config* ⇒ *cell list* ⇒ *bool*
where
crsp ly (*as*, *lm*) (*s*, *l*, *r*) *inres* =
(*s* = *start_of ly as* ∧ (∃ *x*. *r* = <*lm*> @ *Bk*↑*x*) ∧
l = *Bk* # *Bk* # *inres*)

declare *crsp*.*simps*[*simp del*]

The type of invarints expressing correspondence between Abacus configuration and TM configuration.

type-synonym *inc_inv_t* = *abc_conf* ⇒ *config* ⇒ *cell list* ⇒ *bool*

declare *tms_of*.*simps*[*simp del*] *tm_of*.*simps*[*simp del*]
layout_of.*simps*[*simp del*] *abc_fetch*.*simps* [*simp del*]
tpairs_of.*simps*[*simp del*] *start_of*.*simps*[*simp del*]
ci.*simps* [*simp del*] *length_of*.*simps*[*simp del*]
layout_of.*simps*[*simp del*]

The lemmas in this section lead to the correctness of the compilation of *Inc n* instruction.

declare *abc_step_l*.*simps*[*simp del*] *abc_steps_l*.*simps*[*simp del*]
lemma *start_of_nonzero*[*simp*]: *start_of ly as* > 0 (*start_of ly as* = 0) = *False*
⟨*proof*⟩

lemma *abc_steps_l_0*: *abc_steps_l ac ap* 0 = *ac*
⟨*proof*⟩

lemma *abc_step_red*:
abc_steps_l (*as*, *am*) *ap stp* = (*bs*, *bm*) ⟹
abc_steps_l (*as*, *am*) *ap* (*Suc stp*) = *abc_step_l* (*bs*, *bm*) (*abc_fetch bs ap*)

⟨*proof*⟩

lemma *tm_shift_fetch*:
⟦*fetch A s b* = (*ac*, *ns*); *ns* ≠ *0* ⟧
⟹ *fetch* (*shift A off*) *s b* = (*ac*, *ns* + *off*)
⟨*proof*⟩

lemma *tm_shift_eq_step*:
assumes *exec*: *step* (*s*, *l*, *r*) (*A*, *0*) = (*s'*, *l'*, *r'*)
and *notfinal*: *s'* ≠ *0*
shows *step* (*s* + *off*, *l*, *r*) (*shift A off*, *off*) = (*s'* + *off*, *l'*, *r'*)
⟨*proof*⟩

declare *step.simps*[*simp del*] *steps.simps*[*simp del*] *shift.simps*[*simp del*]

lemma *tm_shift_eq_steps*:
assumes *exec*: *steps* (*s*, *l*, *r*) (*A*, *0*) *stp* = (*s'*, *l'*, *r'*)
and *notfinal*: *s'* ≠ *0*
shows *steps* (*s* + *off*, *l*, *r*) (*shift A off*, *off*) *stp* = (*s'* + *off*, *l'*, *r'*)
⟨*proof*⟩

lemma *startof_ge1*[*simp*]: *Suc 0* ≤ *start_of ly as*
⟨*proof*⟩

lemma *start_of_Suc1*: ⟦*ly* = *layout_of ap*;
abc_fetch as ap = *Some* (*Inc n*)⟧
⟹ *start_of ly* (*Suc as*) = *start_of ly as* + *2* ∗ *n* + *9*
⟨*proof*⟩

lemma *start_of_Suc2*:
⟦*ly* = *layout_of ap*;
abc_fetch as ap = *Some* (*Dec n e*)⟧ ⟹
start_of ly (*Suc as*) =
start_of ly as + *2* ∗ *n* + *16*
⟨*proof*⟩

lemma *start_of_Suc3*:
⟦*ly* = *layout_of ap*;
abc_fetch as ap = *Some* (*Goto n*)⟧ ⟹
start_of ly (*Suc as*) = *start_of ly as* + *1*
⟨*proof*⟩

lemma *length_ci_inc*:
length (*ci ly ss* (*Inc n*)) = *4*∗*n* + *18*
⟨*proof*⟩

lemma *length_ci_dec*:
length (*ci ly ss* (*Dec n e*)) = *4*∗*n* + *32*
⟨*proof*⟩

lemma *length_ci_goto*:
length (*ci ly ss* (*Goto n*)) = *2*
⟨*proof*⟩

lemma *take_Suc_last*[*elim*]: *Suc as* ≤ *length xs* ⟹
take (*Suc as*) *xs* = *take as xs* @ [*xs* ! *as*]
⟨*proof*⟩

lemma *concat_suc*: *Suc as* ≤ *length xs* ⟹
concat (*take* (*Suc as*) *xs*) = *concat* (*take as xs*) @ *xs*! *as*
⟨*proof*⟩

lemma *concat_drop_suc_iff*:
Suc n < *length tps* ⟹ *concat* (*drop* (*Suc n*) *tps*) =
tps ! *Suc n* @ *concat* (*drop* (*Suc* (*Suc n*)) *tps*)
⟨*proof*⟩

declare *append_assoc*[*simp del*]

lemma *tm_append*:
⟦*n* < *length tps*; *tp* = *tps* ! *n*⟧ ⟹
∃ *tp1 tp2*. *concat tps* = *tp1* @ *tp* @ *tp2* ∧ *tp1* =
concat (*take n tps*) ∧ *tp2* = *concat* (*drop* (*Suc n*) *tps*)
⟨*proof*⟩

declare *append_assoc*[*simp*]

lemma *length_tms_of*[*simp*]: *length* (*tms_of aprog*) = *length aprog*
⟨*proof*⟩

lemma *ci_nth*:
⟦*ly* = *layout_of aprog*;
abc_fetch as aprog = *Some ins*⟧
⟹ *ci ly* (*start_of ly as*) *ins* = *tms_of aprog* ! *as*
⟨*proof*⟩

lemma *t_split*:⟦
ly = *layout_of aprog*;
abc_fetch as aprog = *Some ins*⟧
⟹ ∃ *tp1 tp2*. *concat* (*tms_of aprog*) =
tp1 @ (*ci ly* (*start_of ly as*) *ins*) @ *tp2*
∧ *tp1* = *concat* (*take as* (*tms_of aprog*)) ∧
tp2 = *concat* (*drop* (*Suc as*) (*tms_of aprog*))
⟨*proof*⟩

lemma *div_apart*: ⟦*x mod* (*2*::*nat*) = *0*; *y mod 2* = *0*⟧
⟹ (*x* + *y*) *div 2* = *x div 2* + *y div 2*
⟨*proof*⟩

lemma *length_layout_of*[*simp*]: *length* (*layout_of aprog*) = *length aprog*

⟨*proof*⟩

lemma *length_tms_of_elem_even*[*intro*]: $n < length\ ap \Longrightarrow length\ (tms\_of\ ap\ !\ n)\ mod\ 2 = 0$
⟨*proof*⟩

lemma *compile_mod2*: $length\ (concat\ (take\ n\ (tms\_of\ ap)))\ mod\ 2 = 0$
⟨*proof*⟩

lemma *tpa_states*:
$[\![tp = concat\ (take\ as\ (tms\_of\ ap));$
$as \leq length\ ap]\!] \Longrightarrow$
$start\_of\ (layout\_of\ ap)\ as = Suc\ (length\ tp\ div\ 2)$
⟨*proof*⟩

declare *fetch.simps*[*simp*]
lemma *append_append_fetch*:
$[\![length\ tp1\ mod\ 2 = 0; length\ tp\ mod\ 2 = 0;$
$length\ tp1\ div\ 2 < a \wedge a \leq length\ tp1\ div\ 2 + length\ tp\ div\ 2]\!]$
$\Longrightarrow fetch\ (tp1\ @\ tp\ @\ tp2)\ a\ b = fetch\ tp\ (a - length\ tp1\ div\ 2)\ b$
⟨*proof*⟩

lemma *step_eq_fetch'*:
assumes *layout*: $ly = layout\_of\ ap$
and *compile*: $tp = tm\_of\ ap$
and *fetch*: $abc\_fetch\ as\ ap = Some\ ins$
and *range1*: $s \geq start\_of\ ly\ as$
and *range2*: $s < start\_of\ ly\ (Suc\ as)$
shows $fetch\ tp\ s\ b = fetch\ (ci\ ly\ (start\_of\ ly\ as)\ ins)$
$(Suc\ s - start\_of\ ly\ as)\ b$
⟨*proof*⟩

lemma *step_eq_fetch*:
assumes *layout*: $ly = layout\_of\ ap$
and *compile*: $tp = tm\_of\ ap$
and *abc_fetch*: $abc\_fetch\ as\ ap = Some\ ins$
and *fetch*: $fetch\ (ci\ ly\ (start\_of\ ly\ as)\ ins)$
$(Suc\ s - start\_of\ ly\ as)\ b = (ac, ns)$
and *notfinal*: $ns \neq 0$
shows $fetch\ tp\ s\ b = (ac, ns)$
⟨*proof*⟩

lemma *step_eq_in*:
assumes *layout*: $ly = layout\_of\ ap$
and *compile*: $tp = tm\_of\ ap$
and *fetch*: $abc\_fetch\ as\ ap = Some\ ins$
and *exec*: $step\ (s, l, r)\ (ci\ ly\ (start\_of\ ly\ as)\ ins, start\_of\ ly\ as - 1)$
$= (s', l', r')$
and *notfinal*: $s' \neq 0$
shows $step\ (s, l, r)\ (tp, 0) = (s', l', r')$

⟨*proof*⟩

lemma *steps_eq_in*:
assumes *layout*: *ly* = *layout_of ap*
and *compile*: *tp* = *tm_of ap*
and *crsp*: *crsp ly* (*as*, *lm*) (*s*, *l*, *r*) *ires*
and *fetch*: *abc_fetch as ap* = *Some ins*
and *exec*: *steps* (*s*, *l*, *r*) (*ci ly* (*start_of ly as*) *ins*, *start_of ly as* − *1*) *stp*
= (*s*′, *l*′, *r*′)
and *notfinal*: *s*′ ≠ *0*
shows *steps* (*s*, *l*, *r*) (*tp*, *0*) *stp* = (*s*′, *l*′, *r*′)
⟨*proof*⟩

lemma *tm_append_fetch_first*:
⟦*fetch A s b* = (*ac*, *ns*); *ns* ≠ *0*⟧ ⟹
fetch (*A* @ *B*) *s b* = (*ac*, *ns*)
⟨*proof*⟩

lemma *tm_append_first_step_eq*:
assumes *step* (*s*, *l*, *r*) (*A*, *off*) = (*s*′, *l*′, *r*′)
and *s*′ ≠ *0*
shows *step* (*s*, *l*, *r*) (*A* @ *B*, *off*) = (*s*′, *l*′, *r*′)
⟨*proof*⟩

lemma *tm_append_first_steps_eq*:
assumes *steps* (*s*, *l*, *r*) (*A*, *off*) *stp* = (*s*′, *l*′, *r*′)
and *s*′ ≠ *0*
shows *steps* (*s*, *l*, *r*) (*A* @ *B*, *off*) *stp* = (*s*′, *l*′, *r*′)
⟨*proof*⟩

lemma *tm_append_second_fetch_eq*:
assumes
even: *length A mod 2* = *0*
and *off*: *off* = *length A div 2*
and *fetch*: *fetch B s b* = (*ac*, *ns*)
and *notfinal*: *ns* ≠ *0*
shows *fetch* (*A* @ *shift B off*) (*s* + *off*) *b* = (*ac*, *ns* + *off*)
⟨*proof*⟩

lemma *tm_append_second_step_eq*:
assumes
exec: *step0* (*s*, *l*, *r*) *B* = (*s*′, *l*′, *r*′)
and *notfinal*: *s*′ ≠ *0*
and *off*: *off* = *length A div 2*
and *even*: *length A mod 2* = *0*
shows *step0* (*s* + *off*, *l*, *r*) (*A* @ *shift B off*) = (*s*′ + *off*, *l*′, *r*′)
⟨*proof*⟩

lemma *tm_append_second_steps_eq*:

assumes
exec: *steps* (s, l, r) $(B, 0)$ *stp* $= (s', l', r')$
and *notfinal*: $s' \neq 0$
and *off*: *off* $=$ *length A div 2*
and *even*: *length A mod 2* $= 0$
shows *steps* $(s + off, l, r)$ $(A$ @ *shift B off*$, 0)$ *stp* $= (s' + off, l', r')$
⟨*proof*⟩

lemma *tm_append_second_fetch0_eq*:
assumes
even: *length A mod 2* $= 0$
and *off*: *off* $=$ *length A div 2*
and *fetch*: *fetch B s b* $= (ac, 0)$
and *notfinal*: $s \neq 0$
shows *fetch* $(A$ @ *shift B off*$)$ $(s + off)$ $b = (ac, 0)$
⟨*proof*⟩

lemma *tm_append_second_halt_eq*:
assumes
exec: *steps* $(Suc\ 0, l, r)$ $(B, 0)$ *stp* $= (0, l', r')$
and *wf_B*: *tm_wf* $(B, 0)$
and *off*: *off* $=$ *length A div 2*
and *even*: *length A mod 2* $= 0$
shows *steps* $(Suc\ off, l, r)$ $(A$ @ *shift B off*$, 0)$ *stp* $= (0, l', r')$
⟨*proof*⟩

lemma *tm_append_steps*:
assumes
aexec: *steps* (s, l, r) $(A, 0)$ *stpa* $= (Suc\ (length\ A\ div\ 2), la, ra)$
and *bexec*: *steps* $(Suc\ 0, la, ra)$ $(B, 0)$ *stpb* $= (sb, lb, rb)$
and *notfinal*: $sb \neq 0$
and *off*: *off* $=$ *length A div 2*
and *even*: *length A mod 2* $= 0$
shows *steps* (s, l, r) $(A$ @ *shift B off*$, 0)$ $(stpa + stpb) = (sb + off, lb, rb)$
⟨*proof*⟩

9.3 Crsp of Inc

fun *at_begin_fst_bwtn* :: *inc_inv_t*
where
at_begin_fst_bwtn (as, lm) (s, l, r) *ires* $=$
$(\exists\ lm1\ tn\ rn.\ lm1 = (lm$ @ $0{\uparrow}tn) \wedge length\ lm1 = s \wedge$
$(if\ lm1 = []\ then\ l = Bk\ \#\ Bk\ \#\ ires$
$else\ l = [Bk]@<rev\ lm1>@Bk\#Bk\#ires) \wedge r = Bk{\uparrow}rn)$

fun *at_begin_fst_awtn* :: *inc_inv_t*
where
at_begin_fst_awtn (as, lm) (s, l, r) *ires* $=$
$(\exists\ lm1\ tn\ rn.\ lm1 = (lm$ @ $0{\uparrow}tn) \wedge length\ lm1 = s \wedge$

```
      (if lm1 = []  then l = Bk # Bk # ires
       else l = [Bk]@<rev lm1>@Bk#Bk#ires) ∧ r = [Oc]@Bk↑rn)

fun at_begin_norm :: inc_inv_t
 where
  at_begin_norm (as, lm) (s, l, r) ires=
   (∃ lm1 lm2 rn. lm = lm1 @ lm2 ∧ length lm1 = s ∧
    (if lm1 = [] then l = Bk # Bk # ires
     else l = Bk # <rev lm1> @ Bk # Bk # ires ) ∧ r = <lm2>@Bk↑rn)

fun in_middle :: inc_inv_t
 where
  in_middle (as, lm) (s, l, r) ires =
   (∃ lm1 lm2 tn m ml mr rn. lm @ 0↑tn = lm1 @ [m] @ lm2
    ∧ length lm1 = s ∧ m + 1 = ml + mr ∧
     ml ≠ 0 ∧ tn = s + 1 − length lm ∧
    (if lm1 = [] then l = Oc↑ml @ Bk # Bk # ires
     else l = Oc↑ml@[Bk]@<rev lm1>@
          Bk # Bk # ires) ∧ (r = Oc↑mr @ [Bk] @ <lm2>@ Bk↑rn ∨
    (lm2 = [] ∧ r = Oc↑mr))
    )

fun inv_locate_a :: inc_inv_t
 where inv_locate_a (as, lm) (s, l, r) ires =
   (at_begin_norm (as, lm) (s, l, r) ires ∨
    at_begin_fst_bwtn (as, lm) (s, l, r) ires ∨
    at_begin_fst_awtn (as, lm) (s, l, r) ires
    )

fun inv_locate_b :: inc_inv_t
 where inv_locate_b (as, lm) (s, l, r) ires =
     (in_middle (as, lm) (s, l, r)) ires

fun inv_after_write :: inc_inv_t
 where inv_after_write (as, lm) (s, l, r) ires =
       (∃ rn m lm1 lm2. lm = lm1 @ m # lm2 ∧
         (if lm1 = [] then l = Oc↑m @ Bk # Bk # ires
          else Oc # l = Oc↑Suc m@ Bk # <rev lm1> @
               Bk # Bk # ires) ∧ r = [Oc] @ <lm2> @ Bk↑rn)

fun inv_after_move :: inc_inv_t
 where inv_after_move (as, lm) (s, l, r) ires =
    (∃ rn m lm1 lm2. lm = lm1 @ m # lm2 ∧
     (if lm1 = [] then l = Oc↑Suc m @ Bk # Bk # ires
      else l = Oc↑Suc m@ Bk # <rev lm1> @ Bk # Bk # ires) ∧
     r = <lm2> @ Bk↑rn)

fun inv_after_clear :: inc_inv_t
 where inv_after_clear (as, lm) (s, l, r) ires =
    (∃ rn m lm1 lm2 r'. lm = lm1 @ m # lm2 ∧
```

```
    (if lm1 = [] then l = Oc↑Suc m @ Bk # Bk # ires
     else l = Oc↑Suc m @ Bk # <rev lm1> @ Bk # Bk # ires) ∧
     r = Bk # r' ∧ Oc # r' = <lm2> @ Bk↑rn)

fun inv_on_right_moving :: inc_inv_t
 where inv_on_right_moving (as, lm) (s, l, r) ires =
    (∃ lm1 lm2 m ml mr rn. lm = lm1 @ [m] @ lm2 ∧
       ml + mr = m ∧
      (if lm1 = [] then l = Oc↑ml @ Bk # Bk # ires
      else l = Oc↑ml  @ [Bk] @ <rev lm1> @ Bk # Bk # ires) ∧
      ((r = Oc↑mr @ [Bk] @ <lm2> @ Bk↑rn) ∨
       (r = Oc↑mr ∧ lm2 = [])))

fun inv_on_left_moving_norm :: inc_inv_t
 where inv_on_left_moving_norm (as, lm) (s, l, r) ires =
    (∃ lm1 lm2 m ml mr rn. lm = lm1 @ [m] @ lm2 ∧
        ml + mr = Suc m ∧ mr > 0 ∧ (if lm1 = [] then l = Oc↑ml @ Bk # Bk # ires
                                    else l =  Oc↑ml @ Bk # <rev lm1> @ Bk # Bk # ires)
     ∧ (r = Oc↑mr @ Bk # <lm2> @ Bk↑rn ∨
        (lm2 = [] ∧ r = Oc↑mr)))

fun inv_on_left_moving_in_middle_B:: inc_inv_t
 where inv_on_left_moving_in_middle_B (as, lm) (s, l, r) ires =
          (∃ lm1 lm2 rn. lm = lm1 @ lm2 ∧
             (if lm1 = [] then l = Bk # ires
              else l = <rev lm1> @ Bk # Bk # ires) ∧
              r = Bk # <lm2> @ Bk↑rn)

fun inv_on_left_moving :: inc_inv_t
 where inv_on_left_moving (as, lm) (s, l, r) ires =
    (inv_on_left_moving_norm  (as, lm) (s, l, r) ires ∨
     inv_on_left_moving_in_middle_B (as, lm) (s, l, r) ires)


fun inv_check_left_moving_on_leftmost :: inc_inv_t
 where inv_check_left_moving_on_leftmost (as, lm) (s, l, r) ires =
          (∃ rn. l = ires ∧ r = [Bk, Bk] @ <lm> @  Bk↑rn)

fun inv_check_left_moving_in_middle :: inc_inv_t
 where inv_check_left_moving_in_middle (as, lm) (s, l, r) ires =
         (∃ lm1 lm2 r' rn. lm = lm1 @ lm2 ∧
           (Oc # l = <rev lm1> @ Bk # Bk # ires) ∧ r = Oc # Bk # r' ∧
                r' = <lm2> @  Bk↑rn)

fun inv_check_left_moving :: inc_inv_t
 where inv_check_left_moving (as, lm) (s, l, r) ires =
        (inv_check_left_moving_on_leftmost (as, lm) (s, l, r) ires ∨
        inv_check_left_moving_in_middle (as, lm) (s, l, r) ires)

fun inv_after_left_moving :: inc_inv_t
```

where *inv_after_left_moving* (as, lm) (s, l, r) *ires*=
$(\exists\ rn.\ l = Bk \# ires \wedge r = Bk \# <lm> @\ Bk\uparrow rn)$

fun *inv_stop* :: *inc_inv_t*
where *inv_stop* (as, lm) (s, l, r) *ires*=
$(\exists\ rn.\ l = Bk \# Bk \# ires \wedge r = <lm> @\ Bk\uparrow rn)$

lemma *halt_lemma2'*:
$[\![wf\ LE;\ \forall\ n.\ ((\neg\ P\ (f\ n) \wedge Q\ (f\ n)) \longrightarrow$
$(Q\ (f\ (Suc\ n)) \wedge (f\ (Suc\ n), (f\ n)) \in LE));\ Q\ (f\ 0)]\!]$
$\Longrightarrow \exists\ n.\ P\ (f\ n)$
⟨*proof*⟩

lemma *halt_lemma2''*:
$[\![P\ (f\ n);\ \neg\ P\ (f\ (0::nat))]\!] \Longrightarrow$
$\exists\ n.\ (P\ (f\ n) \wedge (\forall\ i < n.\ \neg\ P\ (f\ i)))$
⟨*proof*⟩

lemma *halt_lemma2'''*:
$[\![\forall\ n.\ \neg\ P\ (f\ n) \wedge Q\ (f\ n) \longrightarrow Q\ (f\ (Suc\ n)) \wedge (f\ (Suc\ n), f\ n) \in LE;$
$Q\ (f\ 0);\ \forall\ i<na.\ \neg\ P\ (f\ i)]\!] \Longrightarrow Q\ (f\ na)$
⟨*proof*⟩

lemma *halt_lemma2*:
$[\![wf\ LE;$
$Q\ (f\ 0);\ \neg\ P\ (f\ 0);$
$\forall\ n.\ ((\neg\ P\ (f\ n) \wedge Q\ (f\ n)) \longrightarrow (Q\ (f\ (Suc\ n)) \wedge (f\ (Suc\ n), (f\ n)) \in LE))]\!]$
$\Longrightarrow \exists\ n.\ P\ (f\ n) \wedge Q\ (f\ n)$
⟨*proof*⟩

fun *findnth_inv* :: *layout* ⇒ *nat* ⇒ *inc_inv_t*
where
findnth_inv ly n (as, lm) (s, l, r) *ires* =
(*if s* = *0 then False*
else if s ≤ *Suc* $(2*n)$ *then*
if s mod 2 = *1 then inv_locate_a* (as, lm) $((s - 1)$ *div* $2, l, r)$ *ires*
else inv_locate_b (as, lm) $((s - 1)$ *div* $2, l, r)$ *ires*
else False)

fun *findnth_state* :: *config* ⇒ *nat* ⇒ *nat*
where
findnth_state (s, l, r) *n* = $(Suc\ (2*n) - s)$

fun *findnth_step* :: *config* ⇒ *nat* ⇒ *nat*
where
findnth_step (s, l, r) *n* =
(*if s mod 2* = *1 then*
(*if* $(r \neq [] \wedge hd\ r = Oc)$ *then 0*

else 1)
else length r)

fun *findnth_measure* :: *config* × *nat* ⇒ *nat* × *nat*
where
findnth_measure (*c*, *n*) =
(*findnth_state c n*, *findnth_step c n*)

definition *lex_pair* :: ((*nat* × *nat*) × *nat* × *nat*) *set*
where
lex_pair $\stackrel{def}{=}$ *less_than* <∗*lex*∗> *less_than*

definition *findnth_LE* :: ((*config* × *nat*) × (*config* × *nat*)) *set*
where
findnth_LE $\stackrel{def}{=}$ (*inv_image lex_pair findnth_measure*)

lemma *wf_findnth_LE*: *wf findnth_LE*
⟨*proof*⟩

declare *findnth_inv*.*simps*[*simp del*]

lemma *x_is_2n_arith*[*simp*]:
⟦*x* < *Suc* (*Suc* (2 ∗ *n*)); *Suc x mod 2* = *Suc 0*; ¬ *x* < 2 ∗ *n*⟧
⟹ *x* = 2∗*n*
⟨*proof*⟩

lemma *between_sucs*:*x* < *Suc n* ⟹ ¬ *x* < *n* ⟹ *x* = *n* ⟨*proof*⟩

lemma *fetch_findnth*[*simp*]:
⟦0 < *a*; *a* < *Suc* (2 ∗ *n*); *a mod 2* = *Suc 0*⟧ ⟹ *fetch* (*findnth n*) *a Oc* = (*R*, *Suc a*)
⟦0 < *a*; *a* < *Suc* (2 ∗ *n*); *a mod 2* ≠ *Suc 0*⟧ ⟹ *fetch* (*findnth n*) *a Oc* = (*R*, *a*)
⟦0 < *a*; *a* < *Suc* (2 ∗ *n*); *a mod 2* ≠ *Suc 0*⟧ ⟹ *fetch* (*findnth n*) *a Bk* = (*R*, *Suc a*)
⟦0 < *a*; *a* < *Suc* (2 ∗ *n*); *a mod 2* = *Suc 0*⟧ ⟹ *fetch* (*findnth n*) *a Bk* = (*W1*, *a*)
⟨*proof*⟩

declare *at_begin_norm*.*simps*[*simp del*] *at_begin_fst_bwtn*.*simps*[*simp del*]
at_begin_fst_awtn.*simps*[*simp del*] *in_middle*.*simps*[*simp del*]
abc_lm_s.*simps*[*simp del*] *abc_lm_v*.*simps*[*simp del*]
ci.*simps*[*simp del*] *inv_after_move*.*simps*[*simp del*]
inv_on_left_moving_norm.*simps*[*simp del*]
inv_on_left_moving_in_middle_B.*simps*[*simp del*]
inv_after_clear.*simps*[*simp del*]
inv_after_write.*simps*[*simp del*] *inv_on_left_moving*.*simps*[*simp del*]
inv_on_right_moving.*simps*[*simp del*]
inv_check_left_moving.*simps*[*simp del*]
inv_check_left_moving_in_middle.*simps*[*simp del*]
inv_check_left_moving_on_leftmost.*simps*[*simp del*]
inv_after_left_moving.*simps*[*simp del*]

inv_stop.simps[*simp del*] *inv_locate_a.simps*[*simp del*]
inv_locate_b.simps[*simp del*]

lemma *replicate_once*[*intro*]: $\exists\, rn.\ [Bk] = Bk \uparrow rn$
⟨*proof*⟩

lemma *at_begin_norm_Bk*[*intro*]: *at_begin_norm* (*as*, *am*) (*q*, *aaa*, []) *ires*
$\Longrightarrow$ *at_begin_norm* (*as*, *am*) (*q*, *aaa*, [*Bk*]) *ires*
⟨*proof*⟩

lemma *at_begin_fst_bwtn_Bk*[*intro*]: *at_begin_fst_bwtn* (*as*, *am*) (*q*, *aaa*, []) *ires*
$\Longrightarrow$ *at_begin_fst_bwtn* (*as*, *am*) (*q*, *aaa*, [*Bk*]) *ires*
⟨*proof*⟩

lemma *at_begin_fst_awtn_Bk*[*intro*]: *at_begin_fst_awtn* (*as*, *am*) (*q*, *aaa*, []) *ires*
$\Longrightarrow$ *at_begin_fst_awtn* (*as*, *am*) (*q*, *aaa*, [*Bk*]) *ires*
⟨*proof*⟩

lemma *inv_locate_a_Bk*[*intro*]: *inv_locate_a* (*as*, *am*) (*q*, *aaa*, []) *ires*
$\Longrightarrow$ *inv_locate_a* (*as*, *am*) (*q*, *aaa*, [*Bk*]) *ires*
⟨*proof*⟩

lemma *locate_a_2_locate_a*[*simp*]: *inv_locate_a* (*as*, *am*) (*q*, *aaa*, *Bk* # *xs*) *ires*
$\Longrightarrow$ *inv_locate_a* (*as*, *am*) (*q*, *aaa*, *Oc* # *xs*) *ires*
⟨*proof*⟩

lemma *inv_locate_a*[*simp*]: *inv_locate_a* (*as*, *am*) (*q*, *aaa*, []) *ires* $\Longrightarrow$
inv_locate_a (*as*, *am*) (*q*, *aaa*, [*Oc*]) *ires*
⟨*proof*⟩

lemma *inv_locate_b*[*simp*]: *inv_locate_b* (*as*, *am*) (*q*, *aaa*, *Oc* # *xs*) *ires*
$\Longrightarrow$ *inv_locate_b* (*as*, *am*) (*q*, *Oc* # *aaa*, *xs*) *ires*
⟨*proof*⟩

lemma *tape_nat*[*simp*]: $<[x::nat]> = Oc\uparrow(Suc\ x)$
⟨*proof*⟩

lemma *inv_locate*[*simp*]: $[\![$*inv_locate_b* (*as*, *am*) (*q*, *aaa*, *Bk* # *xs*) *ires*; $\exists\, n.\ xs = Bk\uparrow n]\!]$
$\Longrightarrow$ *inv_locate_a* (*as*, *am*) (*Suc q*, *Bk* # *aaa*, *xs*) *ires*
⟨*proof*⟩

lemma *repeat_Bk_no_Oc*[*simp*]: $(Oc \# r = Bk \uparrow rn) = False$
⟨*proof*⟩

lemma *repeat_Bk*[*simp*]: $(\exists\, rna.\ Bk \uparrow rn = Bk \# Bk \uparrow rna) \vee rn = 0$
⟨*proof*⟩

lemma *inv_locate_b_Oc_via_a*[*simp*]:
assumes *inv_locate_a* (*as*, *lm*) (*q*, *l*, *Oc* # *r*) *ires*

shows *inv_locate_b* (*as*, *lm*) (*q*, *Oc* # *l*, *r*) *ires*
⟨*proof*⟩

lemma *length_equal*: *xs* = *ys* ⟹ *length xs* = *length ys*
⟨*proof*⟩

lemma *inv_locate_a_Bk_via_b*[*simp*]: ⟦*inv_locate_b* (*as*, *am*) (*q*, *aaa*, *Bk* # *xs*) *ires*;
¬ (∃ *n*. *xs* = *Bk*↑*n*)⟧
⟹ *inv_locate_a* (*as*, *am*) (*Suc q*, *Bk* # *aaa*, *xs*) *ires*
⟨*proof*⟩

lemma *locate_b_2_a*[*intro*]:
inv_locate_b (*as*, *am*) (*q*, *aaa*, *Bk* # *xs*) *ires*
⟹ *inv_locate_a* (*as*, *am*) (*Suc q*, *Bk* # *aaa*, *xs*) *ires*
⟨*proof*⟩

lemma *inv_locate_b_Bk*[*simp*]: *inv_locate_b* (*as*, *am*) (*q*, *l*, []) *ires*
⟹ *inv_locate_b* (*as*, *am*) (*q*, *l*, [*Bk*]) *ires*
⟨*proof*⟩

lemma *div_rounding_down*[*simp*]: (*2*∗*q* − *Suc 0*) *div 2* = (*q* − *1*) (*Suc* (*2*∗*q*)) *div 2* = *q*
⟨*proof*⟩

lemma *even_plus_one_odd*[*simp*]: *x mod 2* = *0* ⟹ *Suc x mod 2* = *Suc 0*
⟨*proof*⟩

lemma *odd_plus_one_even*[*simp*]: *x mod 2* = *Suc 0* ⟹ *Suc x mod 2* = *0*
⟨*proof*⟩

lemma *locate_b_2_locate_a*[*simp*]:
⟦*q* > *0*; *inv_locate_b* (*as*, *am*) (*q* − *Suc 0*, *aaa*, *Bk* # *xs*) *ires*⟧
⟹ *inv_locate_a* (*as*, *am*) (*q*, *Bk* # *aaa*, *xs*) *ires*
⟨*proof*⟩

lemma *findnth_inv_layout_of_via_crsp*[*simp*]:
crsp (*layout_of ap*) (*as*, *lm*) (*s*, *l*, *r*) *ires*
⟹ *findnth_inv* (*layout_of ap*) *n* (*as*, *lm*) (*Suc 0*, *l*, *r*) *ires*
⟨*proof*⟩

lemma *findnth_correct_pre*:
assumes *layout*: *ly* = *layout_of ap*
and *crsp*: *crsp ly* (*as*, *lm*) (*s*, *l*, *r*) *ires*
and *not0*: *n* > *0*
and *f*: *f* = (λ *stp*. (*steps* (*Suc 0*, *l*, *r*) (*findnth n*, *0*) *stp*, *n*))
and *P*: *P* = (λ ((*s*, *l*, *r*), *n*). *s* = *Suc* (*2* ∗ *n*))

and Q: $Q = (\lambda\,((s, l, r), n).\ \mathit{findnth\_inv}\ \mathit{ly}\ n\ (\mathit{as}, \mathit{lm})\ (s, l, r)\ \mathit{ires})$
shows $\exists\ \mathit{stp}.\ P\ (\mathit{f}\ \mathit{stp}) \wedge Q\ (\mathit{f}\ \mathit{stp})$
⟨*proof*⟩

lemma *inv_locate_a_via_crsp*[*simp*]:
$\mathit{crsp}\ \mathit{ly}\ (\mathit{as}, \mathit{lm})\ (s, l, r)\ \mathit{ires} \Longrightarrow \mathit{inv\_locate\_a}\ (\mathit{as}, \mathit{lm})\ (0, l, r)\ \mathit{ires}$
⟨*proof*⟩

lemma *findnth_correct*:
assumes *layout*: $\mathit{ly} = \mathit{layout\_of}\ \mathit{ap}$
and *crsp*: $\mathit{crsp}\ \mathit{ly}\ (\mathit{as}, \mathit{lm})\ (s, l, r)\ \mathit{ires}$
shows $\exists\ \mathit{stp}\ l'\ r'.\ \mathit{steps}\ (\mathit{Suc}\ 0, l, r)\ (\mathit{findnth}\ n, 0)\ \mathit{stp} = (\mathit{Suc}\ (2 * n), l', r')$
$\wedge\ \mathit{inv\_locate\_a}\ (\mathit{as}, \mathit{lm})\ (n, l', r')\ \mathit{ires}$
⟨*proof*⟩

fun *inc_inv* :: $\mathit{nat} \Rightarrow \mathit{inc\_inv\_t}$
where
$\mathit{inc\_inv}\ n\ (\mathit{as}, \mathit{lm})\ (s, l, r)\ \mathit{ires} =$
$(\mathit{let}\ \mathit{lm}' = \mathit{abc\_lm\_s}\ \mathit{lm}\ n\ (\mathit{Suc}\ (\mathit{abc\_lm\_v}\ \mathit{lm}\ n))\ \mathit{in}$
$\mathit{if}\ s = 0\ \mathit{then}\ \mathit{False}$
$\mathit{else\ if}\ s = 1\ \mathit{then}$
$\mathit{inv\_locate\_a}\ (\mathit{as}, \mathit{lm})\ (n, l, r)\ \mathit{ires}$
$\mathit{else\ if}\ s = 2\ \mathit{then}$
$\mathit{inv\_locate\_b}\ (\mathit{as}, \mathit{lm})\ (n, l, r)\ \mathit{ires}$
$\mathit{else\ if}\ s = 3\ \mathit{then}$
$\mathit{inv\_after\_write}\ (\mathit{as}, \mathit{lm}')\ (s, l, r)\ \mathit{ires}$
$\mathit{else\ if}\ s = \mathit{Suc}\ 3\ \mathit{then}$
$\mathit{inv\_after\_move}\ (\mathit{as}, \mathit{lm}')\ (s, l, r)\ \mathit{ires}$
$\mathit{else\ if}\ s = \mathit{Suc}\ 4\ \mathit{then}$
$\mathit{inv\_after\_clear}\ (\mathit{as}, \mathit{lm}')\ (s, l, r)\ \mathit{ires}$
$\mathit{else\ if}\ s = \mathit{Suc}\ (\mathit{Suc}\ 4)\ \mathit{then}$
$\mathit{inv\_on\_right\_moving}\ (\mathit{as}, \mathit{lm}')\ (s, l, r)\ \mathit{ires}$
$\mathit{else\ if}\ s = \mathit{Suc}\ (\mathit{Suc}\ 5)\ \mathit{then}$
$\mathit{inv\_on\_left\_moving}\ (\mathit{as}, \mathit{lm}')\ (s, l, r)\ \mathit{ires}$
$\mathit{else\ if}\ s = \mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ 5))\ \mathit{then}$
$\mathit{inv\_check\_left\_moving}\ (\mathit{as}, \mathit{lm}')\ (s, l, r)\ \mathit{ires}$
$\mathit{else\ if}\ s = \mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ 5)))\ \mathit{then}$
$\mathit{inv\_after\_left\_moving}\ (\mathit{as}, \mathit{lm}')\ (s, l, r)\ \mathit{ires}$
$\mathit{else\ if}\ s = \mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ (\mathit{Suc}\ 5))))\ \mathit{then}$
$\mathit{inv\_stop}\ (\mathit{as}, \mathit{lm}')\ (s, l, r)\ \mathit{ires}$
$\mathit{else}\ \mathit{False})$

fun *abc_inc_stage1* :: $\mathit{config} \Rightarrow \mathit{nat}$
where
$\mathit{abc\_inc\_stage1}\ (s, l, r) =$
$(\mathit{if}\ s = 0\ \mathit{then}\ 0$
$\mathit{else\ if}\ s \leq 2\ \mathit{then}\ 5$
$\mathit{else\ if}\ s \leq 6\ \mathit{then}\ 4$
$\mathit{else\ if}\ s \leq 8\ \mathit{then}\ 3$

```
        else if s = 9 then 2
        else 1)

fun abc_inc_stage2 :: config ⇒ nat
 where
  abc_inc_stage2 (s, l, r) =
          (if s = 1 then 2
           else if s = 2 then 1
           else if s = 3 then length r
           else if s = 4 then length r
           else if s = 5 then length r
           else if s = 6 then
                      if r ≠ [] then length r
                      else 1
           else if s = 7 then length l
           else if s = 8 then length l
           else 0)

fun abc_inc_stage3 :: config ⇒ nat
 where
  abc_inc_stage3 (s, l, r) = (
        if s = 4 then 4
        else if s = 5 then 3
        else if s = 6 then
           if r ≠ [] ∧ hd r = Oc then 2
           else 1
        else if s = 3 then 0
        else if s = 2 then length r
        else if s = 1 then
             if (r ≠ [] ∧ hd r = Oc) then 0
             else 1
        else 10 − s)


definition inc_measure :: config ⇒ nat × nat × nat
 where
  inc_measure c =
  (abc_inc_stage1 c, abc_inc_stage2 c, abc_inc_stage3 c)

definition lex_triple ::
 ((nat × (nat × nat)) × (nat × (nat × nat))) set
 where lex_triple ≡ less_than <∗lex∗> lex_pair

definition inc_LE :: (config × config) set
 where
  inc_LE ≡ (inv_image lex_triple inc_measure)

declare inc_inv.simps[simp del]
```

lemma *wf_inc_le*[*intro*]: *wf inc_LE*
⟨*proof*⟩

lemma *inv_locate_b_2_after_write*[*simp*]:
assumes *inv_locate_b* (*as*, *am*) (*n*, *aaa*, *Bk* # *xs*) *ires*
shows *inv_after_write* (*as*, *abc_lm_s am n* (*Suc* (*abc_lm_v am n*))) (*s*, *aaa*, *Oc* # *xs*) *ires*
⟨*proof*⟩

lemma *inv_after_move_Oc_via_write*[*simp*]: *inv_after_write* (*as*, *lm*) (*x*, *l*, *Oc* # *r*) *ires*
⟹ *inv_after_move* (*as*, *lm*) (*y*, *Oc* # *l*, *r*) *ires*
⟨*proof*⟩

lemma *inv_after_write_Suc*[*simp*]: *inv_after_write* (*as*, *abc_lm_s am n* (*Suc* (*abc_lm_v am n*)
)) (*x*, *aaa*, *Bk* # *xs*) *ires* = *False*
inv_after_write (*as*, *abc_lm_s am n* (*Suc* (*abc_lm_v am n*)))
(*x*, *aaa*, []) *ires* = *False*
⟨*proof*⟩

lemma *inv_after_clear_Bk_via_Oc*[*simp*]: *inv_after_move* (*as*, *lm*) (*s*, *l*, *Oc* # *r*) *ires*
⟹ *inv_after_clear* (*as*, *lm*) (*s*′, *l*, *Bk* # *r*) *ires*
⟨*proof*⟩

lemma *inv_after_move_2_inv_on_left_moving*[*simp*]:
assumes *inv_after_move* (*as*, *lm*) (*s*, *l*, *Bk* # *r*) *ires*
shows (*l* = [] ⟶
inv_on_left_moving (*as*, *lm*) (*s*′, [], *Bk* # *Bk* # *r*) *ires*) ∧
(*l* ≠ [] ⟶
inv_on_left_moving (*as*, *lm*) (*s*′, *tl l*, *hd l* # *Bk* # *r*) *ires*)
⟨*proof*⟩

lemma *inv_after_move_2_inv_on_left_moving_B*[*simp*]:
inv_after_move (*as*, *lm*) (*s*, *l*, []) *ires*
⟹ (*l* = [] ⟶ *inv_on_left_moving* (*as*, *lm*) (*s*′, [], [*Bk*]) *ires*) ∧
(*l* ≠ [] ⟶ *inv_on_left_moving* (*as*, *lm*) (*s*′, *tl l*, [*hd l*]) *ires*)
⟨*proof*⟩

lemma *inv_after_clear_2_inv_on_right_moving*[*simp*]:
inv_after_clear (*as*, *lm*) (*x*, *l*, *Bk* # *r*) *ires*
⟹ *inv_on_right_moving* (*as*, *lm*) (*y*, *Bk* # *l*, *r*) *ires*
⟨*proof*⟩

lemma *inv_on_right_moving_Oc*[*simp*]: *inv_on_right_moving* (*as*, *lm*) (*x*, *l*, *Oc* # *r*) *ires*
⟹ *inv_on_right_moving* (*as*, *lm*) (*y*, *Oc* # *l*, *r*) *ires*
⟨*proof*⟩

lemma *inv_on_right_moving_2_inv_on_right_moving*[*simp*]:
inv_on_right_moving (*as*, *lm*) (*x*, *l*, *Bk* # *r*) *ires*
⟹ *inv_after_write* (*as*, *lm*) (*y*, *l*, *Oc* # *r*) *ires*
⟨*proof*⟩

lemma *inv_on_right_moving_singleton_Bk*[*simp*]: *inv_on_right_moving* (*as*, *lm*) (*x*, *l*, []) *ires*⟹

inv_on_right_moving (*as*, *lm*) (*y*, *l*, [*Bk*]) *ires*
⟨*proof*⟩

lemma *no_inv_on_left_moving_in_middle_B_Oc*[*simp*]: *inv_on_left_moving_in_middle_B* (*as*, *lm*)
(*s*, *l*, *Oc* # *r*) *ires* = *False*
⟨*proof*⟩

lemma *no_inv_on_left_moving_norm_Bk*[*simp*]: *inv_on_left_moving_norm* (*as*, *lm*) (*s*, *l*, *Bk* # *r*) *ires*
= *False*
⟨*proof*⟩

lemma *inv_on_left_moving_in_middle_B_Bk*[*simp*]:
⟦*inv_on_left_moving_norm* (*as*, *lm*) (*s*, *l*, *Oc* # *r*) *ires*;
hd l = *Bk*; *l* ≠ []⟧ ⟹
inv_on_left_moving_in_middle_B (*as*, *lm*) (*s*, *tl l*, *Bk* # *Oc* # *r*) *ires*
⟨*proof*⟩

lemma *inv_on_left_moving_norm_Oc_Oc*[*simp*]: ⟦*inv_on_left_moving_norm* (*as*, *lm*) (*s*, *l*, *Oc* # *r*) *ires*;
hd l = *Oc*; *l* ≠ []⟧
⟹ *inv_on_left_moving_norm* (*as*, *lm*)
(*s*, *tl l*, *Oc* # *Oc* # *r*) *ires*
⟨*proof*⟩

lemma *inv_on_left_moving_in_middle_B_Bk_Oc*[*simp*]: *inv_on_left_moving_norm* (*as*, *lm*) (*s*, [], *Oc* # *r*) *ires*
⟹ *inv_on_left_moving_in_middle_B* (*as*, *lm*) (*s*, [], *Bk* # *Oc* # *r*) *ires*
⟨*proof*⟩

lemma *inv_on_left_moving_Oc_cases*[*simp*]:*inv_on_left_moving* (*as*, *lm*) (*s*, *l*, *Oc* # *r*) *ires*
⟹ (*l* = [] ⟶ *inv_on_left_moving* (*as*, *lm*) (*s*, [], *Bk* # *Oc* # *r*) *ires*)
∧ (*l* ≠ [] ⟶ *inv_on_left_moving* (*as*, *lm*) (*s*, *tl l*, *hd l* # *Oc* # *r*) *ires*)
⟨*proof*⟩

lemma *from_on_left_moving_to_check_left_moving*[*simp*]: *inv_on_left_moving_in_middle_B* (*as*, *lm*)
(*s*, *Bk* # *list*, *Bk* # *r*) *ires*
⟹ *inv_check_left_moving_on_leftmost* (*as*, *lm*)
(*s*′, *list*, *Bk* # *Bk* # *r*) *ires*
⟨*proof*⟩

lemma *inv_check_left_moving_in_middle_no_Bk*[*simp*]:
inv_check_left_moving_in_middle (*as*, *lm*) (*s*, *l*, *Bk* # *r*) *ires*= *False*
⟨*proof*⟩

lemma *inv_check_left_moving_on_leftmost_Bk_Bk*[*simp*]:
inv_on_left_moving_in_middle_B (*as*, *lm*) (*s*, [], *Bk* # *r*) *ires*⟹
inv_check_left_moving_on_leftmost (*as*, *lm*) (*s*′, [], *Bk* # *Bk* # *r*) *ires*
⟨*proof*⟩

lemma *inv_check_left_moving_on_leftmost_no_Oc*[*simp*]: *inv_check_left_moving_on_leftmost* (*as*, *lm*)
(*s*, *list*, *Oc* # *r*) *ires*= *False*
⟨*proof*⟩

lemma *inv_check_left_moving_in_middle_Oc_Bk*[*simp*]: *inv_on_left_moving_in_middle_B* (*as*, *lm*)
(*s*, *Oc* # *list*, *Bk* # *r*) *ires*
⟹ *inv_check_left_moving_in_middle* (*as*, *lm*) (*s*′, *list*, *Oc* # *Bk* # *r*) *ires*
⟨*proof*⟩

lemma *inv_on_left_moving_2_check_left_moving*[*simp*]:
inv_on_left_moving (*as*, *lm*) (*s*, *l*, *Bk* # *r*) *ires*
⟹ (*l* = [] ⟶ *inv_check_left_moving* (*as*, *lm*) (*s*′, [], *Bk* # *Bk* # *r*) *ires*)
∧ (*l* ≠ [] ⟶
inv_check_left_moving (*as*, *lm*) (*s*′, *tl l*, *hd l* # *Bk* # *r*) *ires*)
⟨*proof*⟩

lemma *inv_on_left_moving_norm_no_empty*[*simp*]: *inv_on_left_moving_norm* (*as*, *lm*) (*s*, *l*, [])
ires = *False*
⟨*proof*⟩

lemma *inv_on_left_moving_no_empty*[*simp*]: *inv_on_left_moving* (*as*, *lm*) (*s*, *l*, []) *ires* = *False*
⟨*proof*⟩

lemma
inv_check_left_moving_in_middle_2_on_left_moving_in_middle_B[*simp*]:
assumes *inv_check_left_moving_in_middle* (*as*, *lm*) (*s*, *Bk* # *list*, *Oc* # *r*) *ires*
shows *inv_on_left_moving_in_middle_B* (*as*, *lm*) (*s*′, *list*, *Bk* # *Oc* # *r*) *ires*
⟨*proof*⟩

lemma *inv_check_left_moving_in_middle_Bk_Oc*[*simp*]:
inv_check_left_moving_in_middle (*as*, *lm*) (*s*, [], *Oc* # *r*) *ires*⟹
inv_check_left_moving_in_middle (*as*, *lm*) (*s*′, [*Bk*], *Oc* # *r*) *ires*
⟨*proof*⟩

lemma *inv_on_left_moving_norm_Oc_Oc_via_middle*[*simp*]: *inv_check_left_moving_in_middle* (*as*, *lm*)
(*s*, *Oc* # *list*, *Oc* # *r*) *ires*
⟹ *inv_on_left_moving_norm* (*as*, *lm*) (*s*′, *list*, *Oc* # *Oc* # *r*) *ires*

⟨*proof*⟩

lemma *inv_check_left_moving_Oc_cases*[*simp*]: *inv_check_left_moving* (*as*, *lm*) (*s*, *l*, *Oc* # *r*) *ires*
⟹ (*l* = [] ⟶ *inv_on_left_moving* (*as*, *lm*) (*s*′, [], *Bk* # *Oc* # *r*) *ires*) ∧
(*l* ≠ [] ⟶ *inv_on_left_moving* (*as*, *lm*) (*s*′, *tl l*, *hd l* # *Oc* # *r*) *ires*)
⟨*proof*⟩

lemma *inv_after_left_moving_Bk_via_check*[*simp*]: *inv_check_left_moving* (*as*, *lm*) (*s*, *l*, *Bk* # *r*) *ires*
⟹ *inv_after_left_moving* (*as*, *lm*) (*s*′, *Bk* # *l*, *r*) *ires*
⟨*proof*⟩

lemma *inv_after_left_moving_Bk_empty_via_check*[*simp*]:*inv_check_left_moving* (*as*, *lm*) (*s*, *l*, []) *ires*
⟹ *inv_after_left_moving* (*as*, *lm*) (*s*′, *Bk* # *l*, []) *ires*
⟨*proof*⟩

lemma *inv_stop_Bk_move*[*simp*]: *inv_after_left_moving* (*as*, *lm*) (*s*, *l*, *Bk* # *r*) *ires*
⟹ *inv_stop* (*as*, *lm*) (*s*′, *Bk* # *l*, *r*) *ires*
⟨*proof*⟩

lemma *inv_stop_Bk_empty*[*simp*]: *inv_after_left_moving* (*as*, *lm*) (*s*, *l*, []) *ires*
⟹ *inv_stop* (*as*, *lm*) (*s*′, *Bk* # *l*, []) *ires*
⟨*proof*⟩

lemma *inv_stop_indep_fst*[*simp*]: *inv_stop* (*as*, *lm*) (*x*, *l*, *r*) *ires* ⟹
inv_stop (*as*, *lm*) (*y*, *l*, *r*) *ires*
⟨*proof*⟩

lemma *inv_after_clear_no_Oc*[*simp*]: *inv_after_clear* (*as*, *lm*) (*s*, *aaa*, *Oc* # *xs*) *ires*= *False*
⟨*proof*⟩

lemma *inv_after_left_moving_no_Oc*[*simp*]:
inv_after_left_moving (*as*, *lm*) (*s*, *aaa*, *Oc* # *xs*) *ires* = *False*
⟨*proof*⟩

lemma *inv_after_clear_Suc_nonempty*[*simp*]:
inv_after_clear (*as*, *abc_lm_s lm n* (*Suc* (*abc_lm_v lm n*))) (*s*, *b*, []) *ires* = *False*
⟨*proof*⟩

lemma *inv_on_left_moving_Suc_nonempty*[*simp*]: *inv_on_left_moving* (*as*, *abc_lm_s lm n* (*Suc* (*abc_lm_v lm n*)))
(*s*, *b*, *Oc* # *list*) *ires* ⟹ *b* ≠ []
⟨*proof*⟩

lemma *inv_check_left_moving_Suc_nonempty*[*simp*]:
inv_check_left_moving (*as*, *abc_lm_s lm n* (*Suc* (*abc_lm_v lm n*))) (*s*, *b*, *Oc* # *list*) *ires* ⟹ *b* ≠ []
⟨*proof*⟩

lemma *tinc_correct_pre*:
assumes *layout*: *ly* = *layout_of ap*
and *inv_start*: *inv_locate_a* (*as*, *lm*) (*n*, *l*, *r*) *ires*
and *lm'*: *lm'* = *abc_lm_s lm n* (*Suc* (*abc_lm_v lm n*))
and *f*: *f* = *steps* (*Suc 0*, *l*, *r*) (*tinc_b*, *0*)
and *P*: *P* = (λ (*s*, *l*, *r*). *s* = *10*)
and *Q*: *Q* = (λ (*s*, *l*, *r*). *inc_inv n* (*as*, *lm*) (*s*, *l*, *r*) *ires*)
shows ∃ *stp*. *P* (*f stp*) ∧ *Q* (*f stp*)
⟨*proof*⟩

lemma *tinc_correct*:
assumes *layout*: *ly* = *layout_of ap*
and *inv_start*: *inv_locate_a* (*as*, *lm*) (*n*, *l*, *r*) *ires*
and *lm'*: *lm'* = *abc_lm_s lm n* (*Suc* (*abc_lm_v lm n*))
shows ∃ *stp l' r'*. *steps* (*Suc 0*, *l*, *r*) (*tinc_b*, *0*) *stp* = (*10*, *l'*, *r'*)
∧ *inv_stop* (*as*, *lm'*) (*10*, *l'*, *r'*) *ires*
⟨*proof*⟩

lemma *is_even_4*[*simp*]: (*4*::*nat*) ∗ *n mod 2* = *0*
⟨*proof*⟩

lemma *crsp_step_inc_pre*:
assumes *layout*: *ly* = *layout_of ap*
and *crsp*: *crsp ly* (*as*, *lm*) (*s*, *l*, *r*) *ires*
and *aexec*: *abc_step_l* (*as*, *lm*) (*Some* (*Inc n*)) = (*asa*, *lma*)
shows ∃ *stp k*. *steps* (*Suc 0*, *l*, *r*) (*findnth n* @ *shift tinc_b* (*2* ∗ *n*), *0*) *stp*
= (*2*∗*n* + *10*, *Bk* # *Bk* # *ires*, <*lma*> @ *Bk*↑*k*) ∧ *stp* > *0*
⟨*proof*⟩

lemma *crsp_step_inc*:
assumes *layout*: *ly* = *layout_of ap*
and *crsp*: *crsp ly* (*as*, *lm*) (*s*, *l*, *r*) *ires*
and *fetch*: *abc_fetch as ap* = *Some* (*Inc n*)
shows ∃*stp* > *0*. *crsp ly* (*abc_step_l* (*as*, *lm*) (*Some* (*Inc n*)))
(*steps* (*s*, *l*, *r*) (*ci ly* (*start_of ly as*) (*Inc n*), *start_of ly as* − *Suc 0*) *stp*) *ires*
⟨*proof*⟩

9.4 Crsp of Dec n e

type-synonym *dec_inv_t* = (*nat* ∗ *nat list*) ⇒ *config* ⇒ *cell list* ⇒ *bool*

fun *dec_first_on_right_moving* :: *nat* ⇒ *dec_inv_t*
where
dec_first_on_right_moving n (*as*, *lm*) (*s*, *l*, *r*) *ires* =
(∃ *lm1 lm2 m ml mr rn*. *lm* = *lm1* @ [*m*] @ *lm2* ∧

```
    ml + mr = Suc m ∧ length lm1 = n ∧ ml > 0 ∧ m > 0 ∧
      (if lm1 = [] then l = Oc↑ml @ Bk # Bk # ires
                   else  l = Oc↑ml @ [Bk] @ <rev lm1> @ Bk # Bk # ires) ∧
  ((r = Oc↑mr @ [Bk] @ <lm2> @ Bk↑rn) ∨ (r = Oc↑mr ∧ lm2 = [])))

fun dec_on_right_moving :: dec_inv_t
 where
  dec_on_right_moving (as, lm) (s, l, r) ires =
  (∃ lm1 lm2 m ml mr rn. lm = lm1 @ [m] @ lm2 ∧
                    ml + mr = Suc (Suc m) ∧
  (if lm1 = [] then l = Oc↑ml@ Bk # Bk # ires
          else  l = Oc↑ml @ [Bk] @ <rev lm1> @ Bk # Bk # ires) ∧
  ((r = Oc↑mr @ [Bk] @ <lm2> @ Bk↑rn) ∨ (r = Oc↑mr ∧ lm2 = [])))

fun dec_after_clear :: dec_inv_t
 where
  dec_after_clear (as, lm) (s, l, r) ires =
         (∃ lm1 lm2 m ml mr rn. lm = lm1 @ [m] @ lm2 ∧
          ml + mr = Suc m ∧ ml = Suc m ∧ r ≠ [] ∧ r ≠ [] ∧
          (if lm1 = [] then l = Oc↑ml@ Bk # Bk # ires
                       else l = Oc↑ml @ [Bk] @ <rev lm1> @ Bk # Bk # ires) ∧
          (tl r = Bk # <lm2> @ Bk↑rn ∨ tl r = [] ∧ lm2 = []))

fun dec_after_write :: dec_inv_t
 where
  dec_after_write (as, lm) (s, l, r) ires =
      (∃ lm1 lm2 m ml mr rn. lm = lm1 @ [m] @ lm2 ∧
    ml + mr = Suc m ∧ ml = Suc m ∧ lm2 ≠ [] ∧
    (if lm1 = [] then l = Bk # Oc↑ml @ Bk # Bk # ires
             else l = Bk # Oc↑ml @ [Bk] @ <rev lm1> @ Bk # Bk # ires) ∧
    tl r = <lm2> @ Bk↑rn)

fun dec_right_move :: dec_inv_t
 where
  dec_right_move (as, lm) (s, l, r) ires =
     (∃ lm1 lm2 m ml mr rn. lm = lm1 @ [m] @ lm2
       ∧ ml = Suc m ∧ mr = (0::nat) ∧
        (if lm1 = [] then l = Bk # Oc↑ml @ Bk # Bk # ires
                     else l = Bk # Oc↑ml @ [Bk] @ <rev lm1> @ Bk # Bk # ires)
      ∧ (r = Bk # <lm2> @ Bk↑rn ∨ r = [] ∧ lm2 = []))

fun dec_check_right_move :: dec_inv_t
 where
  dec_check_right_move (as, lm) (s, l, r) ires =
     (∃ lm1 lm2 m ml mr rn. lm = lm1 @ [m] @ lm2 ∧
       ml = Suc m ∧ mr = (0::nat) ∧
       (if lm1 = [] then l = Bk # Bk # Oc↑ml @ Bk # Bk # ires
                    else l = Bk # Bk # Oc↑ml @ [Bk] @ <rev lm1> @ Bk # Bk # ires) ∧
       r = <lm2> @ Bk↑rn)
```

```
fun dec_left_move :: dec_inv_t
 where
  dec_left_move (as, lm) (s, l, r) ires =
  (∃ lm1 m rn. (lm::nat list) = lm1 @ [m::nat] ∧
  rn > 0 ∧
  (if lm1 = [] then l = Bk # Oc↑Suc m @ Bk # Bk # ires
   else l = Bk # Oc↑Suc m @ Bk # <rev lm1> @ Bk # Bk # ires) ∧ r = Bk↑rn)

declare
 dec_on_right_moving.simps[simp del] dec_after_clear.simps[simp del]
 dec_after_write.simps[simp del] dec_left_move.simps[simp del]
 dec_check_right_move.simps[simp del] dec_right_move.simps[simp del]
 dec_first_on_right_moving.simps[simp del]

fun inv_locate_n_b :: inc_inv_t
 where
  inv_locate_n_b (as, lm) (s, l, r) ires=
  (∃ lm1 lm2 tn m ml mr rn. lm @ 0↑tn = lm1 @ [m] @ lm2 ∧
   length lm1 = s ∧ m + 1 = ml + mr ∧
   ml = 1 ∧ tn = s + 1 − length lm ∧
   (if lm1 = [] then l = Oc↑ml @ Bk # Bk # ires
    else l = Oc↑ml @ Bk # <rev lm1> @ Bk # Bk # ires) ∧
   (r = Oc↑mr @ [Bk] @ <lm2>@ Bk↑rn ∨ (lm2 = [] ∧ r = Oc↑mr))
 )

fun dec_inv_1 :: layout ⇒ nat ⇒ nat ⇒ dec_inv_t
 where
  dec_inv_1 ly n e (as, am) (s, l, r) ires =
       (let ss = start_of ly as in
        let am' = abc_lm_s am n (abc_lm_v am n − Suc 0) in
        let am'' = abc_lm_s am n (abc_lm_v am n) in
          if s = start_of ly e then inv_stop (as, am'') (s, l, r) ires
          else if s = ss then False
          else if s = ss + 2 ∗ n + 1 then
             inv_locate_b (as, am) (n, l, r) ires
          else if s = ss + 2 ∗ n + 13 then
             inv_on_left_moving (as, am'') (s, l, r) ires
          else if s = ss + 2 ∗ n + 14 then
             inv_check_left_moving (as, am'') (s, l, r) ires
          else if s = ss + 2 ∗ n + 15 then
             inv_after_left_moving (as, am'') (s, l, r) ires
          else False)

declare fetch.simps[simp del]


lemma x_plus_helpers:
 x + 4 = Suc (x + 3)
 x + 5 = Suc (x + 4)
 x + 6 = Suc (x + 5)
```

$x + 7 = Suc\ (x + 6)$
$x + 8 = Suc\ (x + 7)$
$x + 9 = Suc\ (x + 8)$
$x + 10 = Suc\ (x + 9)$
$x + 11 = Suc\ (x + 10)$
$x + 12 = Suc\ (x + 11)$
$x + 13 = Suc\ (x + 12)$
$14 + x = Suc\ (x + 13)$
$15 + x = Suc\ (x + 14)$
$16 + x = Suc\ (x + 15)$
$\langle proof \rangle$

lemma *fetch_Dec*[*simp*]:
$fetch\ (ci\ ly\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (Suc\ (2 * n))\ Bk = (W1,\ start\_of\ ly\ as + 2 * n)$
$fetch\ (ci\ ly\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (Suc\ (2 * n))\ Oc = (R,\ Suc\ (start\_of\ ly\ as) + 2 * n)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (Suc\ (Suc\ (2 * n)))\ Oc$
$= (R, start\_of\ ly\ as + 2*n + 2)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (Suc\ (Suc\ (2 * n)))\ Bk$
$= (L, start\_of\ ly\ as + 2*n + 13)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (Suc\ (Suc\ (Suc\ (2 * n))))\ Oc$
$= (R, start\_of\ ly\ as + 2*n + 2)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (Suc\ (Suc\ (Suc\ (2 * n))))\ Bk$
$= (L, start\_of\ ly\ as + 2*n + 3)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 4)\ Oc = (W0, start\_of\ ly\ as + 2*n + 3)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 4)\ Bk = (R, start\_of\ ly\ as + 2*n + 4)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 5)\ Bk = (R, start\_of\ ly\ as + 2*n + 5)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 6)\ Bk = (L, start\_of\ ly\ as + 2*n + 6)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 6)\ Oc = (L, start\_of\ ly\ as + 2*n + 7)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 7)\ Bk = (L, start\_of\ ly\ as + 2*n + 10)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 8)\ Bk = (W1, start\_of\ ly\ as + 2*n + 7)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 8)\ Oc = (R, start\_of\ ly\ as + 2*n + 8)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 9)\ Bk = (L, start\_of\ ly\ as + 2*n + 9)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 9)\ Oc = (R, start\_of\ ly\ as + 2*n + 8)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 10)\ Bk = (R, start\_of\ ly\ as + 2*n + 4)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 10)\ Oc = (W0, start\_of\ ly\ as + 2*n + 9)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 11)\ Oc = (L, start\_of\ ly\ as + 2*n + 10)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 11)\ Bk = (L, start\_of\ ly\ as + 2*n + 11)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 12)\ Oc = (L, start\_of\ ly\ as + 2*n + 10)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 12)\ Bk = (R, start\_of\ ly\ as + 2*n + 12)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (2 * n + 13)\ Bk = (R, start\_of\ ly\ as + 2*n + 16)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (14 + 2 * n)\ Oc = (L, start\_of\ ly\ as + 2*n + 13)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (14 + 2 * n)\ Bk = (L, start\_of\ ly\ as + 2*n + 14)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (15 + 2 * n)\ Oc = (L, start\_of\ ly\ as + 2*n + 13)$
$fetch\ (ci\ (ly)\ (start\_of\ ly\ as)\ (Dec\ n\ e))\ (15 + 2 * n)\ Bk = (R, start\_of\ ly\ as + 2*n + 15)$
$fetch\ (ci\ (ly)\ (start\_of\ (ly)\ as)\ (Dec\ n\ e))\ (16 + 2 * n)\ Bk = (R, start\_of\ (ly)\ e)$
$\langle proof \rangle$

lemma *steps_start_of_invb_inv_locate_a1*[*simp*]:
$[\![r = [] \vee hd\ r = Bk; inv\_locate\_a\ (as, lm)\ (n, l, r)\ ires]\!]$
$\Longrightarrow \exists stp\ la\ ra.$

steps (*start_of ly as* + 2 ∗ *n*, *l*, *r*) (*ci ly* (*start_of ly as*) (*Dec n e*),
start_of ly as − *Suc 0*) *stp* = (*Suc* (*start_of ly as* + 2 ∗ *n*), *la*, *ra*) ∧
inv_locate_b (*as*, *lm*) (*n*, *la*, *ra*) *ires*
⟨*proof*⟩

lemma *steps_start_of_invb_inv_locate_a2*[*simp*]:
⟦*inv_locate_a* (*as*, *lm*) (*n*, *l*, *r*) *ires*; *r* ≠ [] ∧ *hd r* ≠ *Bk*⟧
⟹ ∃ *stp la ra*.
steps (*start_of ly as* + 2 ∗ *n*, *l*, *r*) (*ci ly* (*start_of ly as*) (*Dec n e*),
start_of ly as − *Suc 0*) *stp* = (*Suc* (*start_of ly as* + 2 ∗ *n*), *la*, *ra*) ∧
inv_locate_b (*as*, *lm*) (*n*, *la*, *ra*) *ires*
⟨*proof*⟩

fun *abc_dec_1_stage1*:: *config* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
abc_dec_1_stage1 (*s*, *l*, *r*) *ss n* =
(*if s* > *ss* ∧ *s* ≤ *ss* + 2∗*n* + *1 then 4*
else if s = *ss* + 2 ∗ *n* + *13* ∨ *s* = *ss* + 2∗*n* + *14 then 3*
else if s = *ss* + 2∗*n* + *15 then 2*
else 0)

fun *abc_dec_1_stage2*:: *config* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
abc_dec_1_stage2 (*s*, *l*, *r*) *ss n* =
(*if s* ≤ *ss* + 2 ∗ *n* + *1 then* (*ss* + 2 ∗ *n* + *16* − *s*)
else if s = *ss* + 2∗*n* + *13 then length l*
else if s = *ss* + 2∗*n* + *14 then length l*
else 0)

fun *abc_dec_1_stage3* :: *config* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
abc_dec_1_stage3 (*s*, *l*, *r*) *ss n* =
(*if s* ≤ *ss* + 2∗*n* + *1 then*
if (*s* − *ss*) *mod 2* = *0 then*
if r ≠ [] ∧ *hd r* = *Oc then 0 else 1*
else length r
else if s = *ss* + 2 ∗ *n* + *13 then*
if r ≠ [] ∧ *hd r* = *Oc then 2*
else 1
else if s = *ss* + 2 ∗ *n* + *14 then*
if r ≠ [] ∧ *hd r* = *Oc then 3 else 0*
else 0)

fun *abc_dec_1_measure* :: (*config* × *nat* × *nat*) ⇒ (*nat* × *nat* × *nat*)
where
abc_dec_1_measure (*c*, *ss*, *n*) = (*abc_dec_1_stage1 c ss n*,
abc_dec_1_stage2 c ss n, *abc_dec_1_stage3 c ss n*)

definition *abc_dec_1_LE* ::
((*config* × *nat* ×

nat) × (*config* × *nat* × *nat*)) *set*
where *abc_dec_1_LE* $\stackrel{def}{=}$ (*inv_image lex_triple abc_dec_1_measure*)

lemma *wf_dec_le*: *wf abc_dec_1_LE*
⟨*proof*⟩

lemma *startof_Suc2*:
abc_fetch as ap = *Some* (*Dec n e*) ⟹
start_of (*layout_of ap*) (*Suc as*) =
start_of (*layout_of ap*) *as* + 2 ∗ *n* + 16
⟨*proof*⟩

lemma *start_of_less_2*:
start_of ly e ≤ *start_of ly* (*Suc e*)
⟨*proof*⟩

lemma *start_of_less_1*: *start_of ly e* ≤ *start_of ly* (*e* + *d*)
⟨*proof*⟩

lemma *start_of_less*:
assumes *e* < *as*
shows *start_of ly e* ≤ *start_of ly as*
⟨*proof*⟩

lemma *start_of_ge*:
assumes *fetch*: *abc_fetch as ap* = *Some* (*Dec n e*)
and *layout*: *ly* = *layout_of ap*
and *great*: *e* > *as*
shows *start_of ly e* ≥ *start_of ly as* + 2∗*n* + 16
⟨*proof*⟩

declare *dec_inv_1.simps*[*simp del*]

lemma *start_of_ineq1*[*simp*]:
⟦*abc_fetch as aprog* = *Some* (*Dec n e*); *ly* = *layout_of aprog*⟧
⟹ (*start_of ly e* ≠ *Suc* (*start_of ly as* + 2 ∗ *n*) ∧
start_of ly e ≠ *Suc* (*Suc* (*start_of ly as* + 2 ∗ *n*)) ∧
start_of ly e ≠ *start_of ly as* + 2 ∗ *n* + 3 ∧
start_of ly e ≠ *start_of ly as* + 2 ∗ *n* + 4 ∧
start_of ly e ≠ *start_of ly as* + 2 ∗ *n* + 5 ∧
start_of ly e ≠ *start_of ly as* + 2 ∗ *n* + 6 ∧
start_of ly e ≠ *start_of ly as* + 2 ∗ *n* + 7 ∧
start_of ly e ≠ *start_of ly as* + 2 ∗ *n* + 8 ∧
start_of ly e ≠ *start_of ly as* + 2 ∗ *n* + 9 ∧
start_of ly e ≠ *start_of ly as* + 2 ∗ *n* + 10 ∧
start_of ly e ≠ *start_of ly as* + 2 ∗ *n* + 11 ∧
start_of ly e ≠ *start_of ly as* + 2 ∗ *n* + 12 ∧
start_of ly e ≠ *start_of ly as* + 2 ∗ *n* + 13 ∧
start_of ly e ≠ *start_of ly as* + 2 ∗ *n* + 14 ∧

*start_of ly e ≠ start_of ly as + 2 * n + 15*)
⟨*proof*⟩

lemma *start_of_ineq2*[*simp*]: ⟦*abc_fetch as aprog = Some* (*Dec n e*); *ly = layout_of aprog*⟧
⟹ (*Suc* (*start_of ly as + 2 * n*) ≠ *start_of ly e* ∧
Suc (*Suc* (*start_of ly as + 2 * n*)) ≠ *start_of ly e* ∧
*start_of ly as + 2 * n + 3* ≠ *start_of ly e* ∧
*start_of ly as + 2 * n + 4* ≠ *start_of ly e* ∧
*start_of ly as + 2 * n + 5* ≠*start_of ly e* ∧
*start_of ly as + 2 * n + 6* ≠ *start_of ly e* ∧
*start_of ly as + 2 * n + 7* ≠ *start_of ly e* ∧
*start_of ly as + 2 * n + 8* ≠ *start_of ly e* ∧
*start_of ly as + 2 * n + 9* ≠ *start_of ly e* ∧
*start_of ly as + 2 * n + 10* ≠ *start_of ly e* ∧
*start_of ly as + 2 * n + 11* ≠ *start_of ly e* ∧
*start_of ly as + 2 * n + 12* ≠ *start_of ly e* ∧
*start_of ly as + 2 * n + 13* ≠ *start_of ly e* ∧
*start_of ly as + 2 * n + 14* ≠ *start_of ly e* ∧
*start_of ly as + 2 * n + 15* ≠ *start_of ly e*)
⟨*proof*⟩

lemma *inv_locate_b_nonempty*[*simp*]: *inv_locate_b* (*as*, *lm*) (*n*, [], []) *ires* = *False*
⟨*proof*⟩

lemma *inv_locate_b_no_Bk*[*simp*]: *inv_locate_b* (*as*, *lm*) (*n*, [], *Bk* # *list*) *ires* = *False*
⟨*proof*⟩

lemma *dec_first_on_right_moving_Oc*[*simp*]:
⟦*dec_first_on_right_moving n* (*as*, *am*) (*s*, *aaa*, *Oc* # *xs*) *ires*⟧
⟹ *dec_first_on_right_moving n* (*as*, *am*) (*s'*, *Oc* # *aaa*, *xs*) *ires*
⟨*proof*⟩

lemma *dec_first_on_right_moving_Bk_nonempty*[*simp*]:
dec_first_on_right_moving n (*as*, *am*) (*s*, *l*, *Bk* # *xs*) *ires* ⟹ *l* ≠ []
⟨*proof*⟩

lemma *replicateE*:
⟦¬ *length lm1* < *length am*;
am @ *replicate* (*length lm1* − *length am*) *0* @ [*0*::*nat*] =
lm1 @ *m* # *lm2*;
0 < *m*⟧
⟹ *RR*
⟨*proof*⟩

lemma *dec_after_clear_Bk_strip_hd*[*simp*]:
⟦*dec_first_on_right_moving n* (*as*,
abc_lm_s am n (*abc_lm_v am n*)) (*s*, *l*, *Bk* # *xs*) *ires*⟧
⟹ *dec_after_clear* (*as*, *abc_lm_s am n*
(*abc_lm_v am n* − *Suc 0*)) (*s'*, *tl l*, *hd l* # *Bk* # *xs*) *ires*
⟨*proof*⟩

lemma *dec_first_on_right_moving_dec_after_clear_cases*[*simp*]:
⟦*dec_first_on_right_moving n* (*as*,
abc_lm_s am n (*abc_lm_v am n*)) (*s*, *l*, []) *ires*⟧
⟹ (*l* = [] ⟶ *dec_after_clear* (*as*,
abc_lm_s am n (*abc_lm_v am n* − *Suc 0*)) (*s*′, [], [*Bk*]) *ires*) ∧
(*l* ≠ [] ⟶ *dec_after_clear* (*as*, *abc_lm_s am n*
(*abc_lm_v am n* − *Suc 0*)) (*s*′, *tl l*, [*hd l*]) *ires*)
⟨*proof*⟩

lemma *dec_after_clear_Bk_via_Oc*[*simp*]: ⟦*dec_after_clear* (*as*, *am*) (*s*, *l*, *Oc* # *r*) *ires*⟧
⟹ *dec_after_clear* (*as*, *am*) (*s*′, *l*, *Bk* # *r*) *ires*
⟨*proof*⟩

lemma *dec_right_move_Bk_via_clear_Bk*[*simp*]: ⟦*dec_after_clear* (*as*, *am*) (*s*, *l*, *Bk* # *r*) *ires*⟧
⟹ *dec_right_move* (*as*, *am*) (*s*′, *Bk* # *l*, *r*) *ires*
⟨*proof*⟩

lemma *dec_right_move_Bk_Bk_via_clear*[*simp*]: ⟦*dec_after_clear* (*as*, *am*) (*s*, *l*, []) *ires*⟧
⟹ *dec_right_move* (*as*, *am*) (*s*′, *Bk* # *l*, [*Bk*]) *ires*
⟨*proof*⟩

lemma *dec_right_move_no_Oc*[*simp*]:*dec_right_move* (*as*, *am*) (*s*, *l*, *Oc* # *r*) *ires* = *False*
⟨*proof*⟩

lemma *dec_right_move_2_check_right_move*[*simp*]:
⟦*dec_right_move* (*as*, *am*) (*s*, *l*, *Bk* # *r*) *ires*⟧
⟹ *dec_check_right_move* (*as*, *am*) (*s*′, *Bk* # *l*, *r*) *ires*
⟨*proof*⟩

lemma *lm_iff_empty*[*simp*]: (<*lm*::*nat list*> = []) = (*lm* = [])
⟨*proof*⟩

lemma *dec_right_move_asif_Bk_singleton*[*simp*]:
dec_right_move (*as*, *am*) (*s*, *l*, []) *ires*=
dec_right_move (*as*, *am*) (*s*, *l*, [*Bk*]) *ires*
⟨*proof*⟩

lemma *dec_check_right_move_nonempty*[*simp*]: *dec_check_right_move* (*as*, *am*) (*s*, *l*, *r*) *ires*⟹
l ≠ []
⟨*proof*⟩

lemma *dec_check_right_move_Oc_tail*[*simp*]: ⟦*dec_check_right_move* (*as*, *am*) (*s*, *l*, *Oc* # *r*)
ires⟧
⟹ *dec_after_write* (*as*, *am*) (*s*′, *tl l*, *hd l* # *Oc* # *r*) *ires*
⟨*proof*⟩

lemma *dec_left_move_Bk_tail*[*simp*]: ⟦*dec_check_right_move* (*as*, *am*) (*s*, *l*, *Bk* # *r*) *ires*⟧
⟹ *dec_left_move* (*as*, *am*) (*s*′, *tl l*, *hd l* # *Bk* # *r*) *ires*
⟨*proof*⟩

lemma *dec_left_move_tail*[*simp*]: ⟦*dec_check_right_move* (*as*, *am*) (*s*, *l*, []) *ires*⟧
⟹ *dec_left_move* (*as*, *am*) (*s'*, *tl l*, [*hd l*]) *ires*
⟨*proof*⟩

lemma *dec_left_move_no_Oc*[*simp*]: *dec_left_move* (*as*, *am*) (*s*, *aaa*, *Oc* # *xs*) *ires* = *False*
⟨*proof*⟩

lemma *dec_left_move_nonempty*[*simp*]: *dec_left_move* (*as*, *am*) (*s*, *l*, *r*) *ires*
⟹ *l* ≠ []
⟨*proof*⟩

lemma *inv_on_left_moving_in_middle_B_Oc_Bk_Bks*[*simp*]: *inv_on_left_moving_in_middle_B*
(*as*, [*m*])
(*s'*, *Oc* # *Oc*↑*m* @ *Bk* # *Bk* # *ires*, *Bk* # *Bk*↑*rn*) *ires*
⟨*proof*⟩

lemma *inv_on_left_moving_in_middle_B_Oc_Bk_Bks_rev*[*simp*]: *lm1* ≠ [] ⟹
inv_on_left_moving_in_middle_B (*as*, *lm1* @ [*m*]) (*s'*,
Oc # *Oc*↑*m* @ *Bk* # <*rev lm1*> @ *Bk* # *Bk* # *ires*, *Bk* # *Bk*↑*rn*) *ires*
⟨*proof*⟩

lemma *inv_on_left_moving_Bk_tail*[*simp*]: *dec_left_move* (*as*, *am*) (*s*, *l*, *Bk* # *r*) *ires*
⟹ *inv_on_left_moving* (*as*, *am*) (*s'*, *tl l*, *hd l* # *Bk* # *r*) *ires*
⟨*proof*⟩

lemma *inv_on_left_moving_tail*[*simp*]: *dec_left_move* (*as*, *am*) (*s*, *l*, []) *ires*
⟹ *inv_on_left_moving* (*as*, *am*) (*s'*, *tl l*, [*hd l*]) *ires*
⟨*proof*⟩

lemma *dec_on_right_moving_Oc_mv*[*simp*]: *dec_after_write* (*as*, *am*) (*s*, *l*, *Oc* # *r*) *ires*
⟹ *dec_on_right_moving* (*as*, *am*) (*s'*, *Oc* # *l*, *r*) *ires*
⟨*proof*⟩

lemma *dec_after_write_Oc_via_Bk*[*simp*]: *dec_after_write* (*as*, *am*) (*s*, *l*, *Bk* # *r*) *ires*
⟹ *dec_after_write* (*as*, *am*) (*s'*, *l*, *Oc* # *r*) *ires*
⟨*proof*⟩

lemma *dec_after_write_Oc_empty*[*simp*]: *dec_after_write* (*as*, *am*) (*s*, *aaa*, []) *ires*
⟹ *dec_after_write* (*as*, *am*) (*s'*, *aaa*, [*Oc*]) *ires*
⟨*proof*⟩

lemma *dec_on_right_moving_Oc_move*[*simp*]: *dec_on_right_moving* (*as*, *am*) (*s*, *l*, *Oc* # *r*)
ires
⟹ *dec_on_right_moving* (*as*, *am*) (*s'*, *Oc* # *l*, *r*) *ires*
⟨*proof*⟩

lemma *dec_on_right_moving_nonempty*[*simp*]: *dec_on_right_moving* (*as*, *am*) (*s*, *l*, *r*) *ires*⟹
l ≠ []

⟨*proof*⟩

lemma *dec_after_clear_Bk_tail*[*simp*]: *dec_on_right_moving* (*as*, *am*) (*s*, *l*, *Bk* # *r*) *ires*
⟹ *dec_after_clear* (*as*, *am*) (*s'*, *tl l*, *hd l* # *Bk* # *r*) *ires*
⟨*proof*⟩

lemma *dec_after_clear_tail*[*simp*]: *dec_on_right_moving* (*as*, *am*) (*s*, *l*, []) *ires*
⟹ *dec_after_clear* (*as*, *am*) (*s'*, *tl l*, [*hd l*]) *ires*
⟨*proof*⟩

lemma *dec_false_1*[*simp*]:
⟦*abc_lm_v am n* = *0*; *inv_locate_b* (*as*, *am*) (*n*, *aaa*, *Oc* # *xs*) *ires*⟧
⟹ *False*
⟨*proof*⟩

lemma *inv_on_left_moving_Bk_tl*[*simp*]:
⟦*inv_locate_b* (*as*, *am*) (*n*, *aaa*, *Bk* # *xs*) *ires*;
abc_lm_v am n = *0*⟧
⟹ *inv_on_left_moving* (*as*, *abc_lm_s am n 0*)
(*s*, *tl aaa*, *hd aaa* # *Bk* # *xs*) *ires*
⟨*proof*⟩

lemma *inv_on_left_moving_tl*[*simp*]:
⟦*abc_lm_v am n* = *0*; *inv_locate_b* (*as*, *am*) (*n*, *aaa*, []) *ires*⟧
⟹ *inv_on_left_moving* (*as*, *abc_lm_s am n 0*) (*s*, *tl aaa*, [*hd aaa*]) *ires*
⟨*proof*⟩

declare *dec_inv_1.simps*[*simp del*]

declare *inv_locate_n_b.simps* [*simp del*]

lemma *dec_first_on_right_moving_Oc_via_inv_locate_n_b*[*simp*]:
⟦*inv_locate_n_b* (*as*, *am*) (*n*, *aaa*, *Oc* # *xs*) *ires*⟧
⟹ *dec_first_on_right_moving n* (*as*, *abc_lm_s am n* (*abc_lm_v am n*))
(*s*, *Oc* # *aaa*, *xs*) *ires*
⟨*proof*⟩

lemma *inv_on_left_moving_nonempty*[*simp*]: *inv_on_left_moving* (*as*, *am*) (*s*, [], *r*) *ires*
= *False*
⟨*proof*⟩

lemma *inv_check_left_moving_startof_nonempty*[*simp*]:
inv_check_left_moving (*as*, *abc_lm_s am n 0*)
(*start_of* (*layout_of aprog*) *as* + *2* ∗ *n* + *14*, [], *Oc* # *xs*) *ires*
= *False*
⟨*proof*⟩

lemma *start_of_lessE*[*elim*]: ⟦*abc_fetch as ap* = *Some* (*Dec n e*);
start_of (*layout_of ap*) *as* < *start_of* (*layout_of ap*) *e*;

```
        start_of (layout_of ap) e ≤ Suc (start_of (layout_of ap) as + 2 * n)⟧
    ⟹ RR
 ⟨proof⟩

lemma crsp_step_dec_b_e_pre':
 assumes layout: ly = layout_of ap
  and inv_start: inv_locate_b (as, lm) (n, la, ra) ires
  and fetch: abc_fetch as ap = Some (Dec n e)
  and dec_0: abc_lm_v lm n = 0
  and f: f = (λ stp. (steps (Suc (start_of ly as) + 2 * n, la, ra) (ci ly (start_of ly as) (Dec n e),
       start_of ly as − Suc 0) stp, start_of ly as, n))
  and P: P = (λ ((s, l, r), ss, x). s = start_of ly e)
  and Q: Q = (λ ((s, l, r), ss, x). dec_inv_1 ly x e (as, lm) (s, l, r) ires)
 shows ∃ stp. P (f stp) ∧ Q (f stp)
⟨proof⟩

lemma crsp_step_dec_b_e_pre:
 assumes ly = layout_of ap
  and inv_start: inv_locate_b (as, lm) (n, la, ra) ires
  and dec_0: abc_lm_v lm n = 0
  and fetch: abc_fetch as ap = Some (Dec n e)
 shows ∃ stp lb rb.
    steps (Suc (start_of ly as) + 2 * n, la, ra) (ci ly (start_of ly as) (Dec n e),
    start_of ly as − Suc 0) stp = (start_of ly e, lb, rb) ∧
    dec_inv_1 ly n e (as, lm) (start_of ly e, lb, rb) ires
 ⟨proof⟩

lemma crsp_abc_step_via_stop[simp]:
 ⟦abc_lm_v lm n = 0;
 inv_stop (as, abc_lm_s lm n (abc_lm_v lm n)) (start_of ly e, lb, rb) ires⟧
 ⟹ crsp ly (abc_step_l (as, lm) (Some (Dec n e))) (start_of ly e, lb, rb) ires
 ⟨proof⟩

lemma crsp_step_dec_b_e:
 assumes layout: ly = layout_of ap
  and inv_start: inv_locate_a (as, lm) (n, l, r) ires
  and dec_0: abc_lm_v lm n = 0
  and fetch: abc_fetch as ap = Some (Dec n e)
 shows ∃ stp > 0. crsp ly (abc_step_l (as, lm) (Some (Dec n e)))
 (steps (start_of ly as + 2 * n, l, r) (ci ly (start_of ly as) (Dec n e), start_of ly as − Suc 0) stp)
ires
⟨proof⟩

fun dec_inv_2 :: layout ⇒ nat ⇒ nat ⇒ dec_inv_t
 where
  dec_inv_2 ly n e (as, am) (s, l, r) ires =
      (let ss = start_of ly as in
       let am' = abc_lm_s am n (abc_lm_v am n − Suc 0) in
       let am'' = abc_lm_s am n (abc_lm_v am n) in
        if s = 0 then False
```

```
      else if s = ss + 2 * n then
           inv_locate_a (as, am) (n, l, r) ires
      else if s = ss + 2 * n + 1 then
           inv_locate_n_b (as, am) (n, l, r) ires
      else if s = ss + 2 * n + 2 then
           dec_first_on_right_moving n (as, am'') (s, l, r) ires
      else if s = ss + 2 * n + 3 then
           dec_after_clear (as, am') (s, l, r) ires
      else if s = ss + 2 * n + 4 then
           dec_right_move (as, am') (s, l, r) ires
      else if s = ss + 2 * n + 5 then
           dec_check_right_move (as, am') (s, l, r) ires
      else if s = ss + 2 * n + 6 then
           dec_left_move (as, am') (s, l, r) ires
      else if s = ss + 2 * n + 7 then
           dec_after_write (as, am') (s, l, r) ires
      else if s = ss + 2 * n + 8 then
           dec_on_right_moving (as, am') (s, l, r) ires
      else if s = ss + 2 * n + 9 then
           dec_after_clear (as, am') (s, l, r) ires
      else if s = ss + 2 * n + 10 then
           inv_on_left_moving (as, am') (s, l, r) ires
      else if s = ss + 2 * n + 11 then
           inv_check_left_moving (as, am') (s, l, r) ires
      else if s = ss + 2 * n + 12 then
           inv_after_left_moving (as, am') (s, l, r) ires
      else if s = ss + 2 * n + 16 then
           inv_stop (as, am') (s, l, r) ires
      else False)

declare dec_inv_2.simps[simp del]
fun abc_dec_2_stage1 :: config ⇒ nat ⇒ nat ⇒ nat
 where
  abc_dec_2_stage1 (s, l, r) ss n =
       (if s ≤ ss + 2*n + 1 then 7
        else if s = ss + 2*n + 2 then 6
        else if s = ss + 2*n + 3 then 5
        else if s ≥ ss + 2*n + 4 ∧ s ≤ ss + 2*n + 9 then 4
        else if s = ss + 2*n + 6 then 3
        else if s = ss + 2*n + 10 ∨ s = ss + 2*n + 11 then 2
        else if s = ss + 2*n + 12 then 1
        else 0)

fun abc_dec_2_stage2 :: config ⇒ nat ⇒ nat ⇒ nat
 where
  abc_dec_2_stage2 (s, l, r) ss n =
    (if s ≤ ss + 2 * n + 1 then (ss + 2 * n + 16 − s)
     else if s = ss + 2*n + 10 then length l
     else if s = ss + 2*n + 11 then length l
     else if s = ss + 2*n + 4 then length r − 1
```

```
    else if s = ss + 2*n + 5 then length r
    else if s = ss + 2*n + 7 then length r - 1
    else if s = ss + 2*n + 8 then
        length r + length (takeWhile (λ a. a = Oc) l) - 1
    else if s = ss + 2*n + 9 then
        length r + length (takeWhile (λ a. a = Oc) l) - 1
    else 0)

fun abc_dec_2_stage3 :: config ⇒ nat ⇒ nat ⇒ nat
 where
  abc_dec_2_stage3 (s, l, r) ss n =
     (if s ≤ ss + 2*n + 1 then
       if (s - ss) mod 2 = 0 then if r ≠ [] ∧
                                  hd r = Oc then 0 else 1
       else length r
      else if s = ss + 2 * n + 10 then
        if r ≠ [] ∧ hd r = Oc then 2
        else 1
      else if s = ss + 2 * n + 11 then
        if r ≠ [] ∧ hd r = Oc then 3
        else 0
      else (ss + 2 * n + 16 - s))

fun abc_dec_2_stage4 :: config ⇒ nat ⇒ nat ⇒ nat
 where
  abc_dec_2_stage4 (s, l, r) ss n =
      (if s = ss + 2*n + 2 then length r
       else if s = ss + 2*n + 8 then length r
       else if s = ss + 2*n + 3 then
          if r ≠ [] ∧ hd r = Oc then 1
          else 0
       else if s = ss + 2*n + 7 then
          if r ≠ [] ∧ hd r = Oc then 0
          else 1
       else if s = ss + 2*n + 9 then
          if r ≠ [] ∧ hd r = Oc then 1
          else 0
       else 0)

fun abc_dec_2_measure :: (config × nat × nat) ⇒ (nat × nat × nat × nat)
 where
   abc_dec_2_measure (c, ss, n) =
 (abc_dec_2_stage1 c ss n,
 abc_dec_2_stage2 c ss n, abc_dec_2_stage3 c ss n,  abc_dec_2_stage4 c ss n)

definition lex_square::
 ((nat × nat × nat × nat) × (nat × nat × nat × nat)) set
 where lex_square ≡ less_than <*lex*> lex_triple
```

definition *abc_dec_2_LE* ::
 ((*config* × *nat* ×
 nat) × (*config* × *nat* × *nat*)) *set*
 where *abc_dec_2_LE* $\stackrel{def}{=}$ (*inv_image lex_square abc_dec_2_measure*)

lemma *wf_dec2_le*: *wf abc_dec_2_LE*
 ⟨*proof*⟩

lemma *fix_add*: *fetch ap* ((*x*::*nat*) + 2∗*n*) *b* = *fetch ap* (2∗*n* + *x*) *b*
 ⟨*proof*⟩

lemma *inv_locate_n_b_Bk_elim*[*elim*]:
 ⟦*0* < *abc_lm_v am n*; *inv_locate_n_b* (*as*, *am*) (*n*, *aaa*, *Bk* # *xs*) *ires*⟧
⟹ *RR*
 ⟨*proof*⟩

lemma *inv_locate_n_b_nonemptyE*[*elim*]:
 ⟦*0* < *abc_lm_v am n*; *inv_locate_n_b* (*as*, *am*)
 (*n*, *aaa*, []) *ires*⟧ ⟹ *RR*
 ⟨*proof*⟩

lemma *no_Ocs_dec_after_write*[*simp*]: *dec_after_write* (*as*, *am*) (*s*, *aa*, *r*) *ires*
 ⟹ *takeWhile* (λ*a*. *a* = *Oc*) *aa* = []
 ⟨*proof*⟩

lemma *fewer_Ocs_dec_on_right_moving*[*simp*]:
 ⟦*dec_on_right_moving* (*as*, *lm*) (*s*, *aa*, []) *ires*;
 length (*takeWhile* (λ*a*. *a* = *Oc*) (*tl aa*))
 ≠ *length* (*takeWhile* (λ*a*. *a* = *Oc*) *aa*) − *Suc 0*⟧
 ⟹ *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl aa*)) <
 length (*takeWhile* (λ*a*. *a* = *Oc*) *aa*) − *Suc 0*
 ⟨*proof*⟩

lemma *more_Ocs_dec_after_clear*[*simp*]:
 dec_after_clear (*as*, *abc_lm_s am n* (*abc_lm_v am n* − *Suc 0*))
 (*start_of* (*layout_of aprog*) *as* + 2 ∗ *n* + 9, *aa*, *Bk* # *xs*) *ires*
⟹ *length xs* − *Suc 0* < *length xs* +
 length (*takeWhile* (λ*a*. *a* = *Oc*) *aa*)
 ⟨*proof*⟩

lemma *more_Ocs_dec_after_clear2*[*simp*]:
 ⟦*dec_after_clear* (*as*, *abc_lm_s am n* (*abc_lm_v am n* − *Suc 0*))
 (*start_of* (*layout_of aprog*) *as* + 2 ∗ *n* + 9, *aa*, []) *ires*⟧
 ⟹ *Suc 0* < *length* (*takeWhile* (λ*a*. *a* = *Oc*) *aa*)
 ⟨*proof*⟩

lemma *inv_check_left_moving_nonemptyE*[*elim*]:
 inv_check_left_moving (*as*, *lm*) (*s*, [], *Oc* # *xs*) *ires*
⟹ *RR*

⟨*proof*⟩

lemma *inv_locate_n_b_Oc_via_at_begin_norm*[*simp*]:
⟦*0 < abc_lm_v am n*;
at_begin_norm (*as*, *am*) (*n*, *aaa*, *Oc # xs*) *ires*⟧
⟹ *inv_locate_n_b* (*as*, *am*) (*n*, *Oc # aaa*, *xs*) *ires*
⟨*proof*⟩

lemma *inv_locate_n_b_Oc_via_at_begin_fst_awtn*[*simp*]:
⟦*0 < abc_lm_v am n*;
at_begin_fst_awtn (*as*, *am*) (*n*, *aaa*, *Oc # xs*) *ires*⟧
⟹ *inv_locate_n_b* (*as*, *am*) (*n*, *Oc # aaa*, *xs*) *ires*
⟨*proof*⟩

lemma *inv_locate_n_b_Oc_via_inv_locate_n_a*[*simp*]:
⟦*0 < abc_lm_v am n*; *inv_locate_a* (*as*, *am*) (*n*, *aaa*, *Oc # xs*) *ires*⟧
⟹ *inv_locate_n_b* (*as*, *am*) (*n*, *Oc#aaa*, *xs*) *ires*
⟨*proof*⟩

lemma *more_Oc_dec_on_right_moving*[*simp*]:
⟦*dec_on_right_moving* (*as*, *am*) (*s*, *aa*, *Bk # xs*) *ires*;
Suc (*length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl aa*)))
≠ *length* (*takeWhile* (λ*a*. *a* = *Oc*) *aa*)⟧
⟹ *Suc* (*length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl aa*)))
< *length* (*takeWhile* (λ*a*. *a* = *Oc*) *aa*)
⟨*proof*⟩

lemma *crsp_step_dec_b_suc_pre*:
assumes *layout*: *ly* = *layout_of ap*
and *crsp*: *crsp ly* (*as*, *lm*) (*s*, *l*, *r*) *ires*
and *inv_start*: *inv_locate_a* (*as*, *lm*) (*n*, *la*, *ra*) *ires*
and *fetch*: *abc_fetch as ap* = *Some* (*Dec n e*)
and *dec_suc*: *0 < abc_lm_v lm n*
and *f*: *f* = (λ *stp*. (*steps* (*start_of ly as* + *2* ∗ *n*, *la*, *ra*) (*ci ly* (*start_of ly as*) (*Dec n e*),
start_of ly as − *Suc 0*) *stp*, *start_of ly as*, *n*))
and *P*: *P* = (λ ((*s*, *l*, *r*), *ss*, *x*). *s* = *start_of ly as* + *2*∗*n* + *16*)
and *Q*: *Q* = (λ ((*s*, *l*, *r*), *ss*, *x*). *dec_inv_2 ly x e* (*as*, *lm*) (*s*, *l*, *r*) *ires*)
shows ∃ *stp*. *P* (*f stp*) ∧ *Q*(*f stp*)
⟨*proof*⟩

lemma *crsp_abc_step_l_start_of*[*simp*]:
⟦*inv_stop* (*as*, *abc_lm_s lm n* (*abc_lm_v lm n* − *Suc 0*))
(*start_of* (*layout_of ap*) *as* + *2* ∗ *n* + *16*, *a*, *b*) *ires*;
abc_lm_v lm n > *0*;
abc_fetch as ap = *Some* (*Dec n e*)⟧
⟹ *crsp* (*layout_of ap*) (*abc_step_l* (*as*, *lm*) (*Some* (*Dec n e*)))
(*start_of* (*layout_of ap*) *as* + *2* ∗ *n* + *16*, *a*, *b*) *ires*
⟨*proof*⟩

lemma *crsp_step_dec_b_suc*:

assumes *layout*: $ly = layout\_of\ ap$
and *crsp*: $crsp\ ly\ (as, lm)\ (s, l, r)\ ires$
and *inv_start*: $inv\_locate\_a\ (as, lm)\ (n, la, ra)\ ires$
and *fetch*: $abc\_fetch\ as\ ap = Some\ (Dec\ n\ e)$
and *dec_suc*: $0 < abc\_lm\_v\ lm\ n$
shows $\exists stp > 0.\ crsp\ ly\ (abc\_step\_l\ (as, lm)\ (Some\ (Dec\ n\ e)))$
$(steps\ (start\_of\ ly\ as + 2 * n, la, ra)\ (ci\ (layout\_of\ ap)$
$(start\_of\ ly\ as)\ (Dec\ n\ e), start\_of\ ly\ as - Suc\ 0)\ stp)\ ires$
⟨*proof*⟩

lemma *crsp_step_dec_b*:
assumes *layout*: $ly = layout\_of\ ap$
and *crsp*: $crsp\ ly\ (as, lm)\ (s, l, r)\ ires$
and *inv_start*: $inv\_locate\_a\ (as, lm)\ (n, la, ra)\ ires$
and *fetch*: $abc\_fetch\ as\ ap = Some\ (Dec\ n\ e)$
shows $\exists stp > 0.\ crsp\ ly\ (abc\_step\_l\ (as, lm)\ (Some\ (Dec\ n\ e)))$
$(steps\ (start\_of\ ly\ as + 2 * n, la, ra)\ (ci\ ly\ (start\_of\ ly\ as)\ (Dec\ n\ e), start\_of\ ly\ as - Suc\ 0)$
$stp)\ ires$
⟨*proof*⟩

lemma *crsp_step_dec*:
assumes *layout*: $ly = layout\_of\ ap$
and *crsp*: $crsp\ ly\ (as, lm)\ (s, l, r)\ ires$
and *fetch*: $abc\_fetch\ as\ ap = Some\ (Dec\ n\ e)$
shows $\exists stp > 0.\ crsp\ ly\ (abc\_step\_l\ (as, lm)\ (Some\ (Dec\ n\ e)))$
$(steps\ (s, l, r)\ (ci\ ly\ (start\_of\ ly\ as)\ (Dec\ n\ e), start\_of\ ly\ as - Suc\ 0)\ stp)\ ires$
⟨*proof*⟩

9.5 Crsp of Goto

lemma *crsp_step_goto*:
assumes *layout*: $ly = layout\_of\ ap$
and *crsp*: $crsp\ ly\ (as, lm)\ (s, l, r)\ ires$
shows $\exists stp>0.\ crsp\ ly\ (abc\_step\_l\ (as, lm)\ (Some\ (Goto\ n)))$
$(steps\ (s, l, r)\ (ci\ ly\ (start\_of\ ly\ as)\ (Goto\ n),$
$start\_of\ ly\ as - Suc\ 0)\ stp)\ ires$
⟨*proof*⟩

lemma *crsp_step_in*:
assumes *layout*: $ly = layout\_of\ ap$
and *compile*: $tp = tm\_of\ ap$
and *crsp*: $crsp\ ly\ (as, lm)\ (s, l, r)\ ires$
and *fetch*: $abc\_fetch\ as\ ap = Some\ ins$
shows $\exists\ stp>0.\ crsp\ ly\ (abc\_step\_l\ (as, lm)\ (Some\ ins))$
$(steps\ (s, l, r)\ (ci\ ly\ (start\_of\ ly\ as)\ ins, start\_of\ ly\ as - 1)\ stp)\ ires$
⟨*proof*⟩

lemma *crsp_step*:
assumes *layout*: $ly = layout\_of\ ap$
and *compile*: $tp = tm\_of\ ap$

and *crsp*: *crsp ly* (*as*, *lm*) (*s*, *l*, *r*) *ires*
and *fetch*: *abc_fetch as ap* = *Some ins*
shows ∃ *stp*>*0*. *crsp ly* (*abc_step_l* (*as*, *lm*) (*Some ins*))
(*steps* (*s*, *l*, *r*) (*tp*, *0*) *stp*) *ires*
⟨*proof*⟩

lemma *crsp_steps*:
assumes *layout*: *ly* = *layout_of ap*
and *compile*: *tp* = *tm_of ap*
and *crsp*: *crsp ly* (*as*, *lm*) (*s*, *l*, *r*) *ires*
shows ∃ *stp*. *crsp ly* (*abc_steps_l* (*as*, *lm*) *ap n*)
(*steps* (*s*, *l*, *r*) (*tp*, *0*) *stp*) *ires*
⟨*proof*⟩

lemma *tp_correct′*:
assumes *layout*: *ly* = *layout_of ap*
and *compile*: *tp* = *tm_of ap*
and *crsp*: *crsp ly* (*0*, *lm*) (*Suc 0*, *l*, *r*) *ires*
and *abc_halt*: *abc_steps_l* (*0*, *lm*) *ap stp* = (*length ap*, *am*)
shows ∃ *stp k*. *steps* (*Suc 0*, *l*, *r*) (*tp*, *0*) *stp* = (*start_of ly* (*length ap*), *Bk* # *Bk* # *ires*, <*am*> @ *Bk*↑*k*)
⟨*proof*⟩

The tp @ [(Nop, 0), (Nop, 0)] is nomoral turing machines, so we can use Hoare_plus when composing with Mop machine

lemma *layout_id_cons*: *layout_of* (*ap* @ [*p*]) = *layout_of ap* @ [*length_of p*]
⟨*proof*⟩

lemma *map_start_of_layout*[*simp*]:
map (*start_of* (*layout_of xs* @ [*length_of x*])) [*0*..<*length xs*] = (*map* (*start_of* (*layout_of xs*)) [*0*..<*length xs*])
⟨*proof*⟩

lemma *tpairs_id_cons*:
tpairs_of (*xs* @ [*x*]) = *tpairs_of xs* @ [(*start_of* (*layout_of* (*xs* @ [*x*])) (*length xs*), *x*)]
⟨*proof*⟩

lemma *map_length_ci*:
(*map* (*length* ∘ (λ(*xa*, *y*). *ci* (*layout_of xs* @ [*length_of x*]) *xa y*)) (*tpairs_of xs*)) =
(*map* (*length* ∘ (λ(*x*, *y*). *ci* (*layout_of xs*) *x y*)) (*tpairs_of xs*))
⟨*proof*⟩

lemma *length_tp′*[*simp*]:
⟦*ly* = *layout_of ap*; *tp* = *tm_of ap*⟧ ⟹
length tp = *2* ∗ *sum_list* (*take* (*length ap*) (*layout_of ap*))
⟨*proof*⟩

lemma *length_tp*:
⟦*ly* = *layout_of ap*; *tp* = *tm_of ap*⟧ ⟹

start_of ly (length ap) = Suc (length tp div 2)
⟨*proof*⟩

lemma *compile_correct_halt*:
assumes *layout*: *ly = layout_of ap*
and *compile*: *tp = tm_of ap*
and *crsp*: *crsp ly (0, lm) (Suc 0, l, r) ires*
and *abc_halt*: *abc_steps_l (0, lm) ap stp = (length ap, am)*
and *rs_loc*: *n < length am*
and *rs*: *abc_lm_v am n = rs*
and *off*: *off = length tp div 2*
shows ∃ *stp i j. steps (Suc 0, l, r) (tp @ shift (mopup n) off, 0) stp = (0, Bk↑i @ Bk # Bk # ires, Oc↑Suc rs @ Bk↑j)*
⟨*proof*⟩

declare *mopup.simps*[*simp del*]
lemma *abc_step_red2*:
abc_steps_l (s, lm) p (Suc n) = (let (as′, am′) = abc_steps_l (s, lm) p n in
abc_step_l (as′, am′) (abc_fetch as′ p))
⟨*proof*⟩

lemma *crsp_steps2*:
assumes
layout: *ly = layout_of ap*
and *compile*: *tp = tm_of ap*
and *crsp*: *crsp ly (0, lm) (Suc 0, l, r) ires*
and *nothalt*: *as < length ap*
and *aexec*: *abc_steps_l (0, lm) ap stp = (as, am)*
shows ∃*stpa*≥*stp*. *crsp ly (as, am) (steps (Suc 0, l, r) (tp, 0) stpa) ires*
⟨*proof*⟩

lemma *compile_correct_unhalt*:
assumes *layout*: *ly = layout_of ap*
and *compile*: *tp = tm_of ap*
and *crsp*: *crsp ly (0, lm) (1, l, r) ires*
and *off*: *off = length tp div 2*
and *abc_unhalt*: ∀ *stp*. (λ (*as*, *am*). *as < length ap*) (*abc_steps_l (0, lm) ap stp*)
shows ∀ *stp*.¬ *is_final* (*steps* (*1*, *l*, *r*) (*tp* @ *shift* (*mopup n*) *off*, *0*) *stp*)
⟨*proof*⟩

end

10 Alternative Definitions for Translating Abacus Machines to TMs

theory *Abacus_Defs*
imports *Abacus*
begin

abbreviation
layout $\stackrel{def}{=}$ *layout_of*

fun *address* :: *abc_prog* ⇒ *nat* ⇒ *nat*
where
address p x = (*Suc* (*sum_list* (*take x* (*layout p*))))

abbreviation
TMGoto $\stackrel{def}{=}$ [(*Nop*, *1*), (*Nop*, *1*)]

abbreviation
TMInc $\stackrel{def}{=}$ [(*W1*, *1*), (*R*, *2*), (*W1*, *3*), (*R*, *2*), (*W1*, *3*), (*R*, *4*),
(*L*, *7*), (*W0*, *5*), (*R*, *6*), (*W0*, *5*), (*W1*, *3*), (*R*, *6*),
(*L*, *8*), (*L*, *7*), (*R*, *9*), (*L*, *7*), (*R*, *10*), (*W0*, *9*)]

abbreviation
TMDec $\stackrel{def}{=}$ [(*W1*, *1*), (*R*, *2*), (*L*, *14*), (*R*, *3*), (*L*, *4*), (*R*, *3*),
(*R*, *5*), (*W0*, *4*), (*R*, *6*), (*W0*, *5*), (*L*, *7*), (*L*, *8*),
(*L*, *11*), (*W0*, *7*), (*W1*, *8*), (*R*, *9*), (*L*, *10*), (*R*, *9*),
(*R*, *5*), (*W0*, *10*), (*L*, *12*), (*L*, *11*), (*R*, *13*), (*L*, *11*),
(*R*, *17*), (*W0*, *13*), (*L*, *15*), (*L*, *14*), (*R*, *16*), (*L*, *14*),
(*R*, *0*), (*W0*, *16*)]

abbreviation
TMFindnth $\stackrel{def}{=}$ *findnth*

fun *compile_goto* :: *nat* ⇒ *instr list*
where
compile_goto s = *shift TMGoto* (*s* − *1*)

fun *compile_inc* :: *nat* ⇒ *nat* ⇒ *instr list*
where
compile_inc s n = (*shift* (*TMFindnth n*) (*s* − *1*)) @ (*shift* (*shift TMInc* (*2* ∗ *n*)) (*s* − *1*))

fun *compile_dec* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *instr list*
where
compile_dec s n e = (*shift* (*TMFindnth n*) (*s* − *1*)) @ (*adjust* (*shift* (*shift TMDec* (*2* ∗ *n*)) (*s* − *1*)) *e*)

fun *compile* :: *abc_prog* ⇒ *nat* ⇒ *abc_inst* ⇒ *instr list*
where
compile ap s (*Inc n*) = *compile_inc s n*
| *compile ap s* (*Dec n e*) = *compile_dec s n* (*address ap e*)
| *compile ap s* (*Goto e*) = *compile_goto* (*address ap e*)

lemma
compile ap s i = *ci* (*layout ap*) *s i*
⟨*proof*⟩

end

theory *Rec_Def*
imports *Main*
begin

datatype *recf* = *z*
| *s*
| *id nat nat*
| *Cn nat recf recf list*
| *Pr nat recf recf*
| *Mn nat recf*

definition *pred_of_nl* :: *nat list* ⇒ *nat list*
where
pred_of_nl xs = *butlast xs* @ [*last xs* − *1*]

function *rec_exec* :: *recf* ⇒ *nat list* ⇒ *nat*
where
rec_exec z xs = *0* |
rec_exec s xs = (*Suc* (*xs* ! *0*)) |
rec_exec (*id m n*) *xs* = (*xs* ! *n*) |
rec_exec (*Cn n f gs*) *xs* =
rec_exec f (*map* (λ *a*. *rec_exec a xs*) *gs*) |
rec_exec (*Pr n f g*) *xs* =
(*if last xs* = *0 then rec_exec f* (*butlast xs*)
else rec_exec g (*butlast xs* @ (*last xs* − *1*) # [*rec_exec* (*Pr n f g*) (*butlast xs* @ [*last xs* − *1*])])) |
rec_exec (*Mn n f*) *xs* = (*LEAST x*. *rec_exec f* (*xs* @ [*x*]) = *0*)
⟨*proof*⟩

termination
⟨*proof*⟩

inductive *terminate* :: *recf* ⇒ *nat list* ⇒ *bool*
where
termi_z: *terminate z* [*n*]
| *termi_s*: *terminate s* [*n*]
| *termi_id*: ⟦*n* < *m*; *length xs* = *m*⟧ ⟹ *terminate* (*id m n*) *xs*
| *termi_cn*: ⟦*terminate f* (*map* (λ*g*. *rec_exec g xs*) *gs*);
∀ *g* ∈ *set gs*. *terminate g xs*; *length xs* = *n*⟧ ⟹ *terminate* (*Cn n f gs*) *xs*
| *termi_pr*: ⟦∀ *y* < *x*. *terminate g* (*xs* @ *y* # [*rec_exec* (*Pr n f g*) (*xs* @ [*y*])]);
terminate f xs;
length xs = *n*⟧
⟹ *terminate* (*Pr n f g*) (*xs* @ [*x*])
| *termi_mn*: ⟦*length xs* = *n*; *terminate f* (*xs* @ [*r*]);
rec_exec f (*xs* @ [*r*]) = *0*;

$\forall\ i < r.\ terminate\ f\ (xs\ @\ [i]) \wedge rec\_exec\ f\ (xs\ @\ [i]) > 0] \Longrightarrow terminate\ (Mn\ n\ f)\ xs$

end

theory *Abacus_Hoare*
imports *Abacus*
begin

type-synonym *abc_assert* = *nat list* ⇒ *bool*

definition
assert_imp :: (*'a* ⇒ *bool*) ⇒ (*'a* ⇒ *bool*) ⇒ *bool* (_ ↦ _ [0, 0] 100)
where
assert_imp P Q $\stackrel{def}{=}$ ∀ *xs*. *P xs* ⟶ *Q xs*

fun *abc_holds_for* :: (*nat list* ⇒ *bool*) ⇒ (*nat* × *nat list*) ⇒ *bool* (_ *abc'_holds'_for* _ [100, 99] 100)
where
P abc_holds_for (*s*, *lm*) = *P lm*

fun *abc_final* :: (*nat* × *nat list*) ⇒ *abc_prog* ⇒ *bool*
where
abc_final (*s*, *lm*) *p* = (*s* = *length p*)

fun *abc_notfinal* :: *abc_conf* ⇒ *abc_prog* ⇒ *bool*
where
abc_notfinal (*s*, *lm*) *p* = (*s* < *length p*)

definition
abc_Hoare_halt :: *abc_assert* ⇒ *abc_prog* ⇒ *abc_assert* ⇒ *bool* (({(*1*_)}/ (_)/ {(*1*_)}) 50)
where
abc_Hoare_halt P p Q $\stackrel{def}{=}$ ∀ *lm*. *P lm* ⟶ (∃ *n*. *abc_final* (*abc_steps_l* (*0*, *lm*) *p n*) *p* ∧ *Q abc_holds_for* (*abc_steps_l* (*0*, *lm*) *p n*))

lemma *abc_Hoare_haltI*:
assumes ⋀*lm*. *P lm* ⟹ ∃ *n*. *abc_final* (*abc_steps_l* (*0*, *lm*) *p n*) *p* ∧ *Q abc_holds_for* (*abc_steps_l* (*0*, *lm*) *p n*)
shows {*P*} (*p*::*abc_prog*) {*Q*}
⟨*proof*⟩

P A Q Q B S ———————————————— P A [+] B S

definition
abc_Hoare_unhalt :: *abc_assert* ⇒ *abc_prog* ⇒ *bool* (({(*1*_)}/ (_)) ↑ 50)

where
abc_Hoare_unhalt P p $\stackrel{def}{=}$ $\forall$ *args*. *P args* $\longrightarrow$ ($\forall$ *n* .*abc_notfinal* (*abc_steps_l* (0, *args*) *p n*) *p*)

lemma *abc_Hoare_unhaltI*:
assumes $\bigwedge$*args n*. *P args* $\Longrightarrow$ *abc_notfinal* (*abc_steps_l* (0, *args*) *p n*) *p*
shows {*P*} (*p*::*abc_prog*) $\uparrow$
⟨*proof*⟩

fun *abc_inst_shift* :: *abc_inst* $\Rightarrow$ *nat* $\Rightarrow$ *abc_inst*
where
abc_inst_shift (*Inc m*) *n* = *Inc m* |
abc_inst_shift (*Dec m e*) *n* = *Dec m* (*e* + *n*) |
abc_inst_shift (*Goto m*) *n* = *Goto* (*m* + *n*)

fun *abc_shift* :: *abc_inst list* $\Rightarrow$ *nat* $\Rightarrow$ *abc_inst list*
where
abc_shift xs n = *map* (λ *x*. *abc_inst_shift x n*) *xs*

fun *abc_comp* :: *abc_inst list* $\Rightarrow$ *abc_inst list* $\Rightarrow$
abc_inst list (**infixl** [+] 99)
where
abc_comp al bl = (*let al_len* = *length al in*
al @ *abc_shift bl al_len*)

lemma *abc_comp_first_step_eq_pre*:
s < *length A*
$\Longrightarrow$ *abc_step_l* (*s*, *lm*) (*abc_fetch s* (*A* [+] *B*)) =
abc_step_l (*s*, *lm*) (*abc_fetch s A*)
⟨*proof*⟩

lemma *abc_before_final*:
⟦*abc_final* (*abc_steps_l* (0, *lm*) *p n*) *p*; *p* ≠ []⟧
$\Longrightarrow$ $\exists$ *n*′. *abc_notfinal* (*abc_steps_l* (0, *lm*) *p n*′) *p* $\wedge$
abc_final (*abc_steps_l* (0, *lm*) *p* (*Suc n*′)) *p*
⟨*proof*⟩

lemma *notfinal_Suc*:
abc_notfinal (*abc_steps_l* (0, *lm*) *A* (*Suc n*)) *A* $\Longrightarrow$
abc_notfinal (*abc_steps_l* (0, *lm*) *A n*) *A*
⟨*proof*⟩

lemma *abc_comp_frist_steps_eq_pre*:
assumes *notfinal*: *abc_notfinal* (*abc_steps_l* (0, *lm*) *A n*) *A*
and *notnull*: *A* ≠ []
shows *abc_steps_l* (0, *lm*) (*A* [+] *B*) *n* = *abc_steps_l* (0, *lm*) *A n*
⟨*proof*⟩

declare *abc_shift.simps*[*simp del*] *abc_comp.simps*[*simp del*]

lemma *halt_steps2*: $st \geq length\ A \Longrightarrow abc\_steps\_l\ (st, lm)\ A\ stp = (st, lm)$
⟨*proof*⟩

lemma *halt_steps*: $abc\_steps\_l\ (length\ A, lm)\ A\ n = (length\ A, lm)$
⟨*proof*⟩

lemma *abc_steps_add*:
$abc\_steps\_l\ (as, lm)\ ap\ (m + n) =$
$abc\_steps\_l\ (abc\_steps\_l\ (as, lm)\ ap\ m)\ ap\ n$
⟨*proof*⟩

lemma *equal_when_halt*:
assumes *exc1*: $abc\_steps\_l\ (s, lm)\ A\ na = (length\ A, lma)$
and *exc2*: $abc\_steps\_l\ (s, lm)\ A\ nb = (length\ A, lmb)$
shows $lma = lmb$
⟨*proof*⟩

lemma *abc_comp_frist_steps_halt_eq'*:
assumes *final*: $abc\_steps\_l\ (0, lm)\ A\ n = (length\ A, lm')$
and *notnull*: $A \neq []$
shows $\exists\ n'.\ abc\_steps\_l\ (0, lm)\ (A\ [+]\ B)\ n' = (length\ A, lm')$
⟨*proof*⟩

lemma *abc_exec_null*: $abc\_steps\_l\ sam\ []\ n = sam$
⟨*proof*⟩

lemma *abc_comp_frist_steps_halt_eq*:
assumes *final*: $abc\_steps\_l\ (0, lm)\ A\ n = (length\ A, lm')$
shows $\exists\ n'.\ abc\_steps\_l\ (0, lm)\ (A\ [+]\ B)\ n' = (length\ A, lm')$
⟨*proof*⟩

lemma *abc_comp_second_step_eq*:
assumes *exec*: $abc\_step\_l\ (s, lm)\ (abc\_fetch\ s\ B) = (sa, lma)$
shows $abc\_step\_l\ (s + length\ A, lm)\ (abc\_fetch\ (s + length\ A)\ (A\ [+]\ B))$
$= (sa + length\ A, lma)$
⟨*proof*⟩

lemma *abc_comp_second_steps_eq*:
assumes *exec*: $abc\_steps\_l\ (0, lm)\ B\ n = (sa, lm')$
shows $abc\_steps\_l\ (length\ A, lm)\ (A\ [+]\ B)\ n = (sa + length\ A, lm')$
⟨*proof*⟩

lemma *length_abc_comp*[*simp*, *intro*]:
$length\ (A\ [+]\ B) = length\ A + length\ B$
⟨*proof*⟩

lemma *abc_Hoare_plus_halt* :
assumes *A_halt* : $\{P\}\ (A::abc\_prog)\ \{Q\}$
and *B_halt* : $\{Q\}\ (B::abc\_prog)\ \{S\}$

shows $\{P\}$ $(A\ [+]\ B)$ $\{S\}$
⟨*proof*⟩

lemma *abc_unhalt_append_eq*:
assumes *unhalt*: $\{P\}$ $(A::abc\_prog)\uparrow$
and *P*: *P args*
shows *abc_steps_l* $(0, args)$ $(A\ [+]\ B)$ *stp* = *abc_steps_l* $(0, args)$ *A stp*
⟨*proof*⟩

lemma *abc_Hoare_plus_unhalt1*:
$\{P\}$ $(A::abc\_prog)\uparrow \Longrightarrow \{P\}$ $(A\ [+]\ B)\uparrow$
⟨*proof*⟩

lemma *notfinal_all_before*:
⟦*abc_notfinal* (*abc_steps_l* $(0, args)$ *A x*) *A*; $y \leq x$ ⟧
$\Longrightarrow$ *abc_notfinal* (*abc_steps_l* $(0, args)$ *A y*) *A*
⟨*proof*⟩

lemma *abc_Hoare_plus_unhalt2'*:
assumes *unhalt*: $\{Q\}$ $(B::abc\_prog)\uparrow$
and *halt*: $\{P\}$ $(A::abc\_prog)$ $\{Q\}$
and *notnull*: $A \neq []$
and *P*: *P args*
shows *abc_notfinal* (*abc_steps_l* $(0, args)$ $(A\ [+]\ B)$ *n*) $(A\ [+]\ B)$
⟨*proof*⟩

lemma *abc_comp_null_left*[*simp*]: $[]\ [+]\ A = A$
⟨*proof*⟩

lemma *abc_comp_null_right*[*simp*]: $A\ [+]\ [] = A$
⟨*proof*⟩

lemma *abc_Hoare_plus_unhalt2*:
⟦$\{Q\}$ $(B::abc\_prog)\uparrow$; $\{P\}$ $(A::abc\_prog)$ $\{Q\}$⟧$\Longrightarrow$ $\{P\}$ $(A\ [+]\ B)\uparrow$
⟨*proof*⟩

end

theory *Recursive*
imports *Abacus Rec_Def Abacus_Hoare*
begin

fun *addition* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *abc_prog*
where
addition m n p = [*Dec m 4*, *Inc n*, *Inc p*, *Goto 0*, *Dec p 7*, *Inc m*, *Goto 4*]

fun *mv_box* :: *nat* ⇒ *nat* ⇒ *abc_prog*
where
mv_box m n = [*Dec m 3*, *Inc n*, *Goto 0*]

The compilation of z-operator.

definition *rec_ci_z* :: *abc_inst list*
where
rec_ci_z $\stackrel{def}{=}$ [*Goto 1*]

The compilation of s-operator.

definition *rec_ci_s* :: *abc_inst list*
where
rec_ci_s $\stackrel{def}{=}$ (*addition 0 1 2* [+] [*Inc 1*])

The compilation of *id i j*-operator

fun *rec_ci_id* :: *nat* ⇒ *nat* ⇒ *abc_inst list*
where
rec_ci_id i j = *addition j i* (*i* + *1*)

fun *mv_boxes* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *abc_inst list*
where
mv_boxes ab bb 0 = [] |
mv_boxes ab bb (*Suc n*) = *mv_boxes ab bb n* [+] *mv_box* (*ab* + *n*) (*bb* + *n*)

fun *empty_boxes* :: *nat* ⇒ *abc_inst list*
where
empty_boxes 0 = [] |
empty_boxes (*Suc n*) = *empty_boxes n* [+] [*Dec n 2*, *Goto 0*]

fun *cn_merge_gs* ::
(*abc_inst list* × *nat* × *nat*) *list* ⇒ *nat* ⇒ *abc_inst list*
where
cn_merge_gs [] *p* = [] |
cn_merge_gs (*g* # *gs*) *p* =
(*let* (*gprog*, *gpara*, *gn*) = *g in*
gprog [+] *mv_box gpara p* [+] *cn_merge_gs gs* (*Suc p*))

The compiler of recursive functions, where *rec_ci recf* return (*ap*, *arity*, *fp*), where *ap* is the Abacus program, *arity* is the arity of the recursive function *recf*, *fp* is the amount of memory which is going to be used by *ap* for its execution.

fun *rec_ci* :: *recf* ⇒ *abc_inst list* × *nat* × *nat*
where
rec_ci z = (*rec_ci_z*, *1*, *2*) |
rec_ci s = (*rec_ci_s*, *1*, *3*) |
rec_ci (*id m n*) = (*rec_ci_id m n*, *m*, *m* + *2*) |
rec_ci (*Cn n f gs*) =
(*let cied_gs* = *map* (λ *g*. *rec_ci g*) *gs in*
let (*fprog*, *fpara*, *fn*) = *rec_ci f in*
let pstr = *Max* (*set* (*Suc n* # *fn* # (*map* (λ (*aprog*, *p*, *n*). *n*) *cied_gs*))) *in*
let qstr = *pstr* + *Suc* (*length gs*) *in*
(*cn_merge_gs cied_gs pstr* [+] *mv_boxes 0 qstr n* [+]
mv_boxes pstr 0 (*length gs*) [+] *fprog* [+]
mv_box fpara pstr [+] *empty_boxes* (*length gs*) [+]

```
        mv_box pstr n [+] mv_boxes qstr 0 n, n,  qstr + n)) |
  rec_ci (Pr n f g) =
     (let (fprog, fpara, fn) = rec_ci f in
      let (gprog, gpara, gn) = rec_ci g in
      let p = Max (set ([n + 3, fn, gn])) in
      let e = length gprog + 7 in
       (mv_box n p [+] fprog [+] mv_box n (Suc n) [+]
          (([Dec p e] [+] gprog [+]
            [Inc n, Dec (Suc n) 3, Goto 1]) @
              [Dec (Suc (Suc n)) 0, Inc (Suc n), Goto (length gprog + 4)]),
        Suc n, p + 1)) |
  rec_ci (Mn n f) =
     (let (fprog, fpara, fn) = rec_ci f in
      let len = length (fprog) in
       (fprog @ [Dec (Suc n) (len + 5), Dec (Suc n) (len + 3),
        Goto (len + 1), Inc n, Goto 0], n, max (Suc n) fn))

declare rec_ci.simps [simp del] rec_ci_s_def[simp del]
 rec_ci_z_def[simp del] rec_ci_id.simps[simp del]
 mv_boxes.simps[simp del]
 mv_box.simps[simp del] addition.simps[simp del]

declare abc_steps_l.simps[simp del] abc_fetch.simps[simp del]
 abc_step_l.simps[simp del]

inductive-cases terminate_pr_reverse: terminate (Pr n f g) xs

inductive-cases terminate_z_reverse[elim!]: terminate z xs

inductive-cases terminate_s_reverse[elim!]: terminate s xs

inductive-cases terminate_id_reverse[elim!]: terminate (id m n) xs

inductive-cases terminate_cn_reverse[elim!]: terminate (Cn n f gs) xs

inductive-cases terminate_mn_reverse[elim!]:terminate (Mn n f) xs

fun addition_inv :: nat × nat list ⇒ nat ⇒ nat ⇒ nat ⇒
                    nat list ⇒ bool
 where
  addition_inv (as, lm') m n p lm =
     (let sn = lm ! n in
      let sm = lm ! m in
      lm ! p = 0 ∧
        (if as = 0 then ∃ x. x ≤ lm ! m ∧ lm' = lm[m := x,
                          n := (sn + sm − x), p := (sm − x)]
        else if as = 1 then ∃ x. x < lm ! m ∧ lm' = lm[m := x,
                    n := (sn + sm − x − 1), p := (sm − x − 1)]
        else if as = 2 then ∃ x. x < lm ! m ∧ lm' = lm[m := x,
                      n := (sn + sm − x), p := (sm − x − 1)]
```

```
    else if as = 3 then ∃ x. x < lm ! m ∧ lm' = lm[m := x,
                          n := (sn + sm − x), p := (sm − x)]
    else if as = 4 then ∃ x. x ≤ lm ! m ∧ lm' = lm[m := x,
                            n := (sn + sm), p := (sm − x)]
    else if as = 5 then ∃ x. x < lm ! m ∧ lm' = lm[m := x,
                            n := (sn + sm), p := (sm − x − 1)]
    else if as = 6 then ∃ x. x < lm ! m ∧ lm' =
          lm[m := Suc x, n := (sn + sm), p := (sm − x − 1)]
    else if as = 7 then lm' = lm[m := sm, n := (sn + sm)]
    else False))

fun addition_stage1 :: nat × nat list ⇒ nat ⇒ nat ⇒ nat
 where
  addition_stage1 (as, lm) m p =
      (if as = 0 ∨ as = 1 ∨ as = 2 ∨ as = 3 then 2
       else if as = 4 ∨ as = 5 ∨ as = 6 then 1
       else 0)

fun addition_stage2 :: nat × nat list ⇒ nat ⇒  nat ⇒ nat
 where
  addition_stage2 (as, lm) m p =
        (if 0 ≤ as ∧ as ≤ 3 then lm ! m
         else if 4 ≤ as ∧ as ≤ 6 then lm ! p
         else 0)

fun addition_stage3 :: nat × nat list ⇒ nat ⇒ nat ⇒ nat
 where
  addition_stage3 (as, lm) m p =
        (if as = 1 then 4
         else if as = 2 then 3
         else if as = 3 then 2
         else if as = 0 then 1
         else if as = 5 then 2
         else if as = 6 then 1
         else if as = 4 then 0
         else 0)

fun addition_measure :: ((nat × nat list) × nat × nat) ⇒
                                  (nat × nat × nat)
 where
  addition_measure ((as, lm), m, p) =
   (addition_stage1 (as, lm) m p,
    addition_stage2 (as, lm) m p,
    addition_stage3 (as, lm) m p)

definition addition_LE :: (((nat × nat list) × nat × nat) ×
                  ((nat × nat list) × nat × nat)) set
 where addition_LE ≡ (inv_image lex_triple addition_measure)
```

lemma *wf_additon_LE*[*simp*]: *wf addition_LE*
⟨*proof*⟩

declare *addition_inv.simps*[*simp del*]

lemma *update_zero_to_zero*[*simp*]: ⟦*am* ! *n* = (*0*::*nat*); *n* < *length am*⟧ ⟹ *am*[*n* := *0*] = *am*
⟨*proof*⟩

lemma *addition_inv_init*:
⟦*m* ≠ *n*; *max m n* < *p*; *length lm* > *p*; *lm* ! *p* = *0*⟧ ⟹
addition_inv (*0*, *lm*) *m n p lm*
⟨*proof*⟩

lemma *abs_fetch*[*simp*]:
abc_fetch 0 (*addition m n p*) = *Some* (*Dec m 4*)
abc_fetch (*Suc 0*) (*addition m n p*) = *Some* (*Inc n*)
abc_fetch 2 (*addition m n p*) = *Some* (*Inc p*)
abc_fetch 3 (*addition m n p*) = *Some* (*Goto 0*)
abc_fetch 4 (*addition m n p*) = *Some* (*Dec p 7*)
abc_fetch 5 (*addition m n p*) = *Some* (*Inc m*)
abc_fetch 6 (*addition m n p*) = *Some* (*Goto 4*)
⟨*proof*⟩

lemma *exists_small_list_elem1*[*simp*]:
⟦*m* ≠ *n*; *p* < *length lm*; *lm* ! *p* = *0*; *m* < *p*; *n* < *p*; *x* ≤ *lm* ! *m*; *0* < *x*⟧
⟹ ∃ *xa*<*lm* ! *m*. *lm*[*m* := *x*, *n* := *lm* ! *n* + *lm* ! *m* − *x*,
p := *lm* ! *m* − *x*, *m* := *x* − *Suc 0*] =
lm[*m* := *xa*, *n* := *lm* ! *n* + *lm* ! *m* − *Suc xa*,
p := *lm* ! *m* − *Suc xa*]
⟨*proof*⟩

lemma *exists_small_list_elem2*[*simp*]:
⟦*m* ≠ *n*; *p* < *length lm*; *lm* ! *p* = *0*; *m* < *p*; *n* < *p*; *x* < *lm* ! *m*⟧
⟹ ∃ *xa*<*lm* ! *m*. *lm*[*m* := *x*, *n* := *lm* ! *n* + *lm* ! *m* − *Suc x*,
p := *lm* ! *m* − *Suc x*, *n* := *lm* ! *n* + *lm* ! *m* − *x*]
= *lm*[*m* := *xa*, *n* := *lm* ! *n* + *lm* ! *m* − *xa*,
p := *lm* ! *m* − *Suc xa*]
⟨*proof*⟩

lemma *exists_small_list_elem3*[*simp*]:
⟦*m* ≠ *n*; *p* < *length lm*; *lm* ! *p* = *0*; *m* < *p*; *n* < *p*; *x* < *lm* ! *m*⟧
⟹ ∃ *xa*<*lm* ! *m*. *lm*[*m* := *x*, *n* := *lm* ! *n* + *lm* ! *m* − *x*,
p := *lm* ! *m* − *Suc x*, *p* := *lm* ! *m* − *x*]
= *lm*[*m* := *xa*, *n* := *lm* ! *n* + *lm* ! *m* − *xa*,
p := *lm* ! *m* − *xa*]
⟨*proof*⟩

lemma *exists_small_list_elem4*[*simp*]:
⟦*m* ≠ *n*; *p* < *length lm*; *lm* ! *p* = (*0*::*nat*); *m* < *p*; *n* < *p*; *x* < *lm* ! *m*⟧
⟹ ∃ *xa*≤*lm* ! *m*. *lm*[*m* := *x*, *n* := *lm* ! *n* + *lm* ! *m* − *x*,

$p := lm\,!\,m - x] =$
$lm[m := xa, n := lm\,!\,n + lm\,!\,m - xa,$
$p := lm\,!\,m - xa]$
⟨*proof*⟩

lemma *exists_small_list_elem5*[*simp*]:
$[\![m \neq n; p < length\ lm; lm\,!\,p = 0; m < p; n < p;$
$x \leq lm\,!\,m; lm\,!\,m \neq x]\!]$
$\Longrightarrow \exists xa{<}lm\,!\,m.\ lm[m := x, n := lm\,!\,n + lm\,!\,m,$
$p := lm\,!\,m - x, p := lm\,!\,m - Suc\ x]$
$= lm[m := xa, n := lm\,!\,n + lm\,!\,m,$
$p := lm\,!\,m - Suc\ xa]$
⟨*proof*⟩

lemma *exists_small_list_elem6*[*simp*]:
$[\![m \neq n; p < length\ lm; lm\,!\,p = 0; m < p; n < p; x < lm\,!\,m]\!]$
$\Longrightarrow \exists xa{<}lm\,!\,m.\ lm[m := x, n := lm\,!\,n + lm\,!\,m,$
$p := lm\,!\,m - Suc\ x, m := Suc\ x]$
$= lm[m := Suc\ xa, n := lm\,!\,n + lm\,!\,m,$
$p := lm\,!\,m - Suc\ xa]$
⟨*proof*⟩

lemma *exists_small_list_elem7*[*simp*]:
$[\![m \neq n; p < length\ lm; lm\,!\,p = 0; m < p; n < p; x < lm\,!\,m]\!]$
$\Longrightarrow \exists xa{\leq}lm\,!\,m.\ lm[m := Suc\ x, n := lm\,!\,n + lm\,!\,m,$
$p := lm\,!\,m - Suc\ x]$
$= lm[m := xa, n := lm\,!\,n + lm\,!\,m, p := lm\,!\,m - xa]$
⟨*proof*⟩

lemma *abc_steps_zero*: $abc\_steps\_l\ asm\ ap\ 0 = asm$
⟨*proof*⟩

lemma *list_double_update_2*:
$lm[a := x, b := y, a := z] = lm[b := y, a{:=}z]$
⟨*proof*⟩

declare *Let_def*[*simp*]
lemma *addition_halt_lemma*:
$[\![m \neq n; max\ m\ n < p; length\ lm > p]\!] \Longrightarrow$
$\forall na.\ \neg\ (\lambda(as, lm')\ (m, p).\ as = 7)$
$(abc\_steps\_l\ (0, lm)\ (addition\ m\ n\ p)\ na)\ (m, p) \wedge$
$addition\_inv\ (abc\_steps\_l\ (0, lm)\ (addition\ m\ n\ p)\ na)\ m\ n\ p\ lm$
$\longrightarrow addition\_inv\ (abc\_steps\_l\ (0, lm)\ (addition\ m\ n\ p)$
$(Suc\ na))\ m\ n\ p\ lm$
$\wedge\ ((abc\_steps\_l\ (0, lm)\ (addition\ m\ n\ p)\ (Suc\ na), m, p),$
$abc\_steps\_l\ (0, lm)\ (addition\ m\ n\ p)\ na, m, p) \in addition\_LE$
⟨*proof*⟩

lemma *addition_correct′*:
$[\![m \neq n; max\ m\ n < p; length\ lm > p; lm\,!\,p = 0]\!] \Longrightarrow$

$\exists$ *stp*. (λ (*as*, *lm'*). *as* = 7 $\wedge$ *addition_inv* (*as*, *lm'*) *m n p lm*)
(*abc_steps_l* (*0*, *lm*) (*addition m n p*) *stp*)
⟨*proof*⟩

lemma *length_addition*[*simp*]: *length* (*addition a b c*) = *7*
⟨*proof*⟩

lemma *addition_correct*:
assumes *m* ≠ *n max m n* < *p length lm* > *p lm* ! *p* = *0*
shows {λ *a*. *a* = *lm*} (*addition m n p*) {λ *nl*. *addition_inv* (*7*, *nl*) *m n p lm*}
⟨*proof*⟩

lemma *compile_s_correct'*:
{λ*nl*. *nl* = *n* # *0* ↑ *2* @ *anything*} *addition 0* (*Suc 0*) *2* [+] [*Inc* (*Suc 0*)] {λ*nl*. *nl* = *n* # *Suc n* # *0* # *anything*}
⟨*proof*⟩

declare *rec_exec.simps*[*simp del*]

lemma *abc_comp_commute*: (*A* [+] *B*) [+] *C* = *A* [+] (*B* [+] *C*)
⟨*proof*⟩

lemma *compile_z_correct*:
⟦*rec_ci z* = (*ap*, *arity*, *fp*); *rec_exec z* [*n*] = *r*⟧ ⟹
{λ*nl*. *nl* = *n* # *0* ↑ (*fp* − *arity*) @ *anything*} *ap* {λ*nl*. *nl* = *n* # *r* # *0* ↑ (*fp* − *Suc arity*) @ *anything*}
⟨*proof*⟩

lemma *compile_s_correct*:
⟦*rec_ci s* = (*ap*, *arity*, *fp*); *rec_exec s* [*n*] = *r*⟧ ⟹
{λ*nl*. *nl* = *n* # *0* ↑ (*fp* − *arity*) @ *anything*} *ap* {λ*nl*. *nl* = *n* # *r* # *0* ↑ (*fp* − *Suc arity*) @ *anything*}
⟨*proof*⟩

lemma *compile_id_correct'*:
assumes *n* < *length args*
shows {λ*nl*. *nl* = *args* @ *0* ↑ *2* @ *anything*} *addition n* (*length args*) (*Suc* (*length args*))
{λ*nl*. *nl* = *args* @ *rec_exec* (*recf.id* (*length args*) *n*) *args* # *0* # *anything*}
⟨*proof*⟩

lemma *compile_id_correct*:
⟦*n* < *m*; *length xs* = *m*; *rec_ci* (*recf.id m n*) = (*ap*, *arity*, *fp*); *rec_exec* (*recf.id m n*) *xs* = *r*⟧
⟹ {λ*nl*. *nl* = *xs* @ *0* ↑ (*fp* − *arity*) @ *anything*} *ap* {λ*nl*. *nl* = *xs* @ *r* # *0* ↑ (*fp* − *Suc arity*) @ *anything*}
⟨*proof*⟩

lemma *cn_merge_gs_tl_app*:
cn_merge_gs (*gs* @ [*g*]) *pstr* =

```
    cn_merge_gs gs pstr [+] cn_merge_gs [g] (pstr + length gs)
 ⟨proof⟩

lemma footprint_ge:
 rec_ci a = (p, arity, fp) ⟹ arity < fp
⟨proof⟩

lemma param_pattern:
 ⟦terminate f xs; rec_ci f = (p, arity, fp)⟧ ⟹ length xs = arity
⟨proof⟩

lemma replicate_merge_anywhere:
 x↑a @ x↑b @ ys = x↑(a+b) @ ys
 ⟨proof⟩

fun mv_box_inv :: nat × nat list ⇒ nat ⇒ nat ⇒ nat list ⇒ bool
 where
  mv_box_inv (as, lm) m n initlm =
      (let plus = initlm ! m + initlm ! n in
       length initlm > max m n ∧ m ≠ n ∧
         (if as = 0 then ∃ k l. lm = initlm[m := k, n := l] ∧
             k + l = plus ∧ k ≤ initlm ! m
         else if as = 1 then ∃ k l. lm = initlm[m := k, n := l]
                  ∧ k + l + 1 = plus ∧ k < initlm ! m
         else if as = 2 then ∃ k l. lm = initlm[m := k, n := l]
                   ∧ k + l = plus ∧ k ≤ initlm ! m
         else if as = 3 then lm = initlm[m := 0, n := plus]
         else False))

fun mv_box_stage1 :: nat × nat list ⇒ nat ⇒ nat
 where
  mv_box_stage1 (as, lm) m =
       (if as = 3 then 0
        else 1)

fun mv_box_stage2 :: nat × nat list ⇒ nat ⇒ nat
 where
  mv_box_stage2 (as, lm) m = (lm ! m)

fun mv_box_stage3 :: nat × nat list ⇒ nat ⇒ nat
 where
  mv_box_stage3 (as, lm) m = (if as = 1 then 3
                        else if as = 2 then 2
                        else if as = 0 then 1
                        else 0)

fun mv_box_measure :: ((nat × nat list) × nat) ⇒ (nat × nat × nat)
 where
  mv_box_measure ((as, lm), m) =
   (mv_box_stage1 (as, lm) m, mv_box_stage2 (as, lm) m,
```

mv_box_stage3 (*as*, *lm*) *m*)

definition *lex_pair* :: ((*nat* × *nat*) × *nat* × *nat*) *set*
where
lex_pair = *less_than* <∗*lex*∗> *less_than*

definition *lex_triple* ::
((*nat* × (*nat* × *nat*)) × (*nat* × (*nat* × *nat*))) *set*
where
lex_triple $\stackrel{def}{=}$ *less_than* <∗*lex*∗> *lex_pair*

definition *mv_box_LE* ::
(((*nat* × *nat list*) × *nat*) × ((*nat* × *nat list*) × *nat*)) *set*
where
mv_box_LE $\stackrel{def}{=}$ (*inv_image lex_triple mv_box_measure*)

lemma *wf_lex_triple*: *wf lex_triple*
⟨*proof*⟩

lemma *wf_mv_box_le*[*intro*]: *wf mv_box_LE*
⟨*proof*⟩

declare *mv_box_inv.simps*[*simp del*]

lemma *mv_box_inv_init*:
⟦*m* < *length initlm*; *n* < *length initlm*; *m* ≠ *n*⟧ ⟹
mv_box_inv (*0*, *initlm*) *m n initlm*
⟨*proof*⟩

lemma *abc_fetch*[*simp*]:
abc_fetch 0 (*mv_box m n*) = *Some* (*Dec m 3*)
abc_fetch (*Suc 0*) (*mv_box m n*) = *Some* (*Inc n*)
abc_fetch 2 (*mv_box m n*) = *Some* (*Goto 0*)
abc_fetch 3 (*mv_box m n*) = *None*
⟨*proof*⟩

lemma *replicate_Suc_iff_anywhere*: *x* # *x*↑*b* @ *ys* = *x*↑(*Suc b*) @ *ys*
⟨*proof*⟩

lemma *exists_smaller_in_list0*[*simp*]:
⟦*m* ≠ *n*; *m* < *length initlm*; *n* < *length initlm*;
k + *l* = *initlm* ! *m* + *initlm* ! *n*; *k* ≤ *initlm* ! *m*; *0* < *k*⟧
⟹ ∃ *ka la*. *initlm*[*m* := *k*, *n* := *l*, *m* := *k* − *Suc 0*] =
initlm[*m* := *ka*, *n* := *la*] ∧
Suc (*ka* + *la*) = *initlm* ! *m* + *initlm* ! *n* ∧
ka < *initlm* ! *m*
⟨*proof*⟩

lemma *exists_smaller_in_list1*[*simp*]:

$[\![m \neq n; m < \mathit{length}\ \mathit{initlm}; n < \mathit{length}\ \mathit{initlm};$
$\mathit{Suc}\ (k + l) = \mathit{initlm}\ !\ m + \mathit{initlm}\ !\ n;$
$k < \mathit{initlm}\ !\ m]\!]$
$\Longrightarrow \exists\, \mathit{ka}\ \mathit{la}.\ \mathit{initlm}[m := k, n := l, n := \mathit{Suc}\ l] =$
$\mathit{initlm}[m := \mathit{ka}, n := \mathit{la}] \wedge$
$\mathit{ka} + \mathit{la} = \mathit{initlm}\ !\ m + \mathit{initlm}\ !\ n \wedge$
$\mathit{ka} \leq \mathit{initlm}\ !\ m$
⟨*proof*⟩

lemma *abc_steps_prop*[*simp*]:
$[\![\mathit{length}\ \mathit{initlm} > \mathit{max}\ m\ n; m \neq n]\!] \Longrightarrow$
$\neg\ (\lambda(\mathit{as}, \mathit{lm})\ m.\ \mathit{as} = 3)$
$(\mathit{abc\_steps\_l}\ (0, \mathit{initlm})\ (\mathit{mv\_box}\ m\ n)\ \mathit{na})\ m \wedge$
$\mathit{mv\_box\_inv}\ (\mathit{abc\_steps\_l}\ (0, \mathit{initlm})$
$(\mathit{mv\_box}\ m\ n)\ \mathit{na})\ m\ n\ \mathit{initlm} \longrightarrow$
$\mathit{mv\_box\_inv}\ (\mathit{abc\_steps\_l}\ (0, \mathit{initlm})$
$(\mathit{mv\_box}\ m\ n)\ (\mathit{Suc}\ \mathit{na}))\ m\ n\ \mathit{initlm} \wedge$
$((\mathit{abc\_steps\_l}\ (0, \mathit{initlm})\ (\mathit{mv\_box}\ m\ n)\ (\mathit{Suc}\ \mathit{na}), m),$
$\mathit{abc\_steps\_l}\ (0, \mathit{initlm})\ (\mathit{mv\_box}\ m\ n)\ \mathit{na}, m) \in \mathit{mv\_box\_LE}$
⟨*proof*⟩

lemma *mv_box_inv_halt*:
$[\![\mathit{length}\ \mathit{initlm} > \mathit{max}\ m\ n; m \neq n]\!] \Longrightarrow$
$\exists\ \mathit{stp}.\ (\lambda\ (\mathit{as}, \mathit{lm}).\ \mathit{as} = 3 \wedge$
$\mathit{mv\_box\_inv}\ (\mathit{as}, \mathit{lm})\ m\ n\ \mathit{initlm})$
$(\mathit{abc\_steps\_l}\ (0{::}\mathit{nat}, \mathit{initlm})\ (\mathit{mv\_box}\ m\ n)\ \mathit{stp})$
⟨*proof*⟩

lemma *mv_box_halt_cond*:
$[\![m \neq n; \mathit{mv\_box\_inv}\ (a, b)\ m\ n\ \mathit{lm}; a = 3]\!] \Longrightarrow$
$b = \mathit{lm}[n := \mathit{lm}\ !\ m + \mathit{lm}\ !\ n, m := 0]$
⟨*proof*⟩

lemma *mv_box_correct'*:
$[\![\mathit{length}\ \mathit{lm} > \mathit{max}\ m\ n; m \neq n]\!] \Longrightarrow$
$\exists\ \mathit{stp}.\ \mathit{abc\_steps\_l}\ (0{::}\mathit{nat}, \mathit{lm})\ (\mathit{mv\_box}\ m\ n)\ \mathit{stp}$
$= (3, (\mathit{lm}[n := (\mathit{lm}\ !\ m + \mathit{lm}\ !\ n)])[m := 0{::}\mathit{nat}])$
⟨*proof*⟩

lemma *length_mvbox*[*simp*]: $\mathit{length}\ (\mathit{mv\_box}\ m\ n) = 3$
⟨*proof*⟩

lemma *mv_box_correct*:
$[\![\mathit{length}\ \mathit{lm} > \mathit{max}\ m\ n; m{\neq}n]\!]$
$\Longrightarrow \{\lambda\ \mathit{nl}.\ \mathit{nl} = \mathit{lm}\}\ \mathit{mv\_box}\ m\ n\ \{\lambda\ \mathit{nl}.\ \mathit{nl} = \mathit{lm}[n := (\mathit{lm}\ !\ m + \mathit{lm}\ !\ n), m{:=}0]\}$
⟨*proof*⟩

declare *list_update.simps*(2)[*simp del*]

lemma *zero_case_rec_exec*[*simp*]:

$[\![length\ xs < gf; gf \leq ft; n < length\ gs]\!]$
$\Longrightarrow (rec\_exec\ (gs\ !\ n)\ xs\ \#\ 0\uparrow(ft - Suc\ (length\ xs))\ @\ map\ (\lambda i.\ rec\_exec\ i\ xs)\ (take\ n\ gs)\ @$
$0\uparrow(length\ gs - n)\ @\ 0\ \#\ 0\uparrow length\ xs\ @\ anything)$
$[ft + n - length\ xs := rec\_exec\ (gs\ !\ n)\ xs, 0 := 0] =$
$0\uparrow(ft - length\ xs)\ @\ map\ (\lambda i.\ rec\_exec\ i\ xs)\ (take\ n\ gs)\ @\ rec\_exec\ (gs\ !\ n)\ xs\ \#\ 0\uparrow(length$
$gs - Suc\ n)\ @\ 0\ \#\ 0\uparrow length\ xs\ @\ anything$
⟨*proof*⟩

lemma *compile_cn_gs_correct'*:
assumes
g_cond: $\forall g{\in}set\ (take\ n\ gs).\ terminate\ g\ xs \wedge$
$(\forall x\ xa\ xb.\ rec\_ci\ g = (x, xa, xb) \longrightarrow (\forall xc.\ \{\lambda nl.\ nl = xs\ @\ 0\uparrow(xb - xa)\ @\ xc\}\ x\ \{\lambda nl.\ nl =$
$xs\ @\ rec\_exec\ g\ xs\ \#\ 0\uparrow(xb - Suc\ xa)\ @\ xc\}))$
and *ft*: $ft = max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))$
shows
$\{\lambda nl.\ nl = xs\ @\ 0\ \#\ 0\uparrow(ft + length\ gs)\ @\ anything\}$
cn_merge_gs $(map\ rec\_ci\ (take\ n\ gs))\ ft$
$\{\lambda nl.\ nl = xs\ @\ 0\uparrow(ft - length\ xs)\ @$
$map\ (\lambda i.\ rec\_exec\ i\ xs)\ (take\ n\ gs)\ @\ 0\uparrow(length\ gs - n)\ @\ 0\uparrow Suc\ (length\ xs)\ @$
$anything\}$
⟨*proof*⟩

lemma *compile_cn_gs_correct*:
assumes
g_cond: $\forall g{\in}set\ gs.\ terminate\ g\ xs \wedge$
$(\forall x\ xa\ xb.\ rec\_ci\ g = (x, xa, xb) \longrightarrow (\forall xc.\ \{\lambda nl.\ nl = xs\ @\ 0\uparrow(xb - xa)\ @\ xc\}\ x\ \{\lambda nl.\ nl =$
$xs\ @\ rec\_exec\ g\ xs\ \#\ 0\uparrow(xb - Suc\ xa)\ @\ xc\}))$
and *ft*: $ft = max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))$
shows
$\{\lambda nl.\ nl = xs\ @\ 0\ \#\ 0\uparrow(ft + length\ gs)\ @\ anything\}$
cn_merge_gs $(map\ rec\_ci\ gs)\ ft$
$\{\lambda nl.\ nl = xs\ @\ 0\uparrow(ft - length\ xs)\ @$
$map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ 0\uparrow Suc\ (length\ xs)\ @\ anything\}$
⟨*proof*⟩

lemma *length_mvboxes*[*simp*]: $length\ (mv\_boxes\ aa\ ba\ n) = 3{*}n$
⟨*proof*⟩

lemma *exp_suc*: $a\uparrow Suc\ b = a\uparrow b\ @\ [a]$
⟨*proof*⟩

lemma *last_0*[*simp*]:
$[\![Suc\ n \leq ba - aa;\ \ length\ lm2 = Suc\ n;$
$length\ lm3 = ba - Suc\ (aa + n)]\!]$
$\Longrightarrow (last\ lm2\ \#\ lm3\ @\ butlast\ lm2\ @\ 0\ \#\ lm4)\ !\ (ba - aa) = (0{::}nat)$
⟨*proof*⟩

lemma *butlast_last*[*simp*]: $length\ lm1 = aa \Longrightarrow$
$(lm1\ @\ 0\uparrow n\ @\ last\ lm2\ \#\ lm3\ @\ butlast\ lm2\ @\ 0\ \#\ lm4)\ !\ (aa + n) = last\ lm2$
⟨*proof*⟩

lemma *arith_as_simp*[*simp*]: ⟦*Suc n* ≤ *ba* − *aa*; *aa* < *ba*⟧ ⟹
(*ba* < *Suc* (*aa* + (*ba* − *Suc* (*aa* + *n*) + *n*))) = *False*
⟨*proof*⟩

lemma *butlast_elem*[*simp*]: ⟦*Suc n* ≤ *ba* − *aa*; *aa* < *ba*; *length lm1* = *aa*;
length lm2 = *Suc n*; *length lm3* = *ba* − *Suc* (*aa* + *n*)⟧
⟹ (*lm1* @ *0*↑*n* @ *last lm2* # *lm3* @ *butlast lm2* @ *0* # *lm4*) ! (*ba* + *n*) = *0*
⟨*proof*⟩

lemma *update_butlast_eq0*[*simp*]:
⟦*Suc n* ≤ *ba* − *aa*; *aa* < *ba*; *length lm1* = *aa*; *length lm2* = *Suc n*;
length lm3 = *ba* − *Suc* (*aa* + *n*)⟧
⟹ (*lm1* @ *0*↑*n* @ *last lm2* # *lm3* @ *butlast lm2* @ (*0*::*nat*) # *lm4*)
[*ba* + *n* := *last lm2*, *aa* + *n* := *0*] =
lm1 @ *0* # *0*↑*n* @ *lm3* @ *lm2* @ *lm4*
⟨*proof*⟩

lemma *update_butlast_eq1*[*simp*]:
⟦*Suc* (*length lm1* + *n*) ≤ *ba*; *length lm2* = *Suc n*; *length lm3* = *ba* − *Suc* (*length lm1* + *n*);
¬ *ba* − *Suc* (*length lm1*) < *ba* − *Suc* (*length lm1* + *n*); ¬ *ba* + *n* − *length lm1* < *n*⟧
⟹ (*0*::*nat*) ↑ *n* @ (*last lm2* # *lm3* @ *butlast lm2* @ *0* # *lm4*)[*ba* − *length lm1* := *last lm2*,
0 := *0*] =
0 # *0* ↑ *n* @ *lm3* @ *lm2* @ *lm4*
⟨*proof*⟩

lemma *mv_boxes_correct*:
⟦*aa* + *n* ≤ *ba*; *ba* > *aa*; *length lm1* = *aa*; *length lm2* = *n*; *length lm3* = *ba* − *aa* − *n*⟧
⟹ {λ *nl*. *nl* = *lm1* @ *lm2* @ *lm3* @ *0*↑*n* @ *lm4*} (*mv_boxes aa ba n*)
{λ *nl*. *nl* = *lm1* @ *0*↑*n* @ *lm3* @ *lm2* @ *lm4*}
⟨*proof*⟩

lemma *update_butlast_eq2*[*simp*]:
⟦*Suc n* ≤ *aa* − *length lm1*; *length lm1* < *aa*;
length lm2 = *aa* − *Suc* (*length lm1* + *n*);
length lm3 = *Suc n*;
¬ *aa* − *Suc* (*length lm1*) < *aa* − *Suc* (*length lm1* + *n*);
¬ *aa* + *n* − *length lm1* < *n*⟧
⟹ *butlast lm3* @ ((*0*::*nat*) # *lm2* @ *0* ↑ *n* @ *last lm3* # *lm4*)[*0* := *last lm3*, *aa* − *length lm1*
:= *0*] = *lm3* @ *lm2* @ *0* # *0* ↑ *n* @ *lm4*
⟨*proof*⟩

lemma *mv_boxes_correct2*:
⟦*n* ≤ *aa* − *ba*;
ba < *aa*;
length (*lm1*::*nat list*) = *ba*;
length (*lm2*::*nat list*) = *aa* − *ba* − *n*;
length (*lm3*::*nat list*) = *n*⟧
⟹{λ *nl*. *nl* = *lm1* @ *0*↑*n* @ *lm2* @ *lm3* @ *lm4*}
(*mv_boxes aa ba n*)

$\{\lambda\ nl.\ nl = lm1\ @\ lm3\ @\ lm2\ @\ 0\uparrow n\ @\ lm4\}$
⟨*proof*⟩

lemma *save_paras*:
$\{\lambda nl.\ nl = xs\ @\ 0\uparrow(max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) - length\ xs)\ @$
$map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ 0\uparrow Suc\ (length\ xs)\ @\ anything\}$
$mv\_boxes\ 0\ (Suc\ (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) + length\ gs))\ (length\ xs)$
$\{\lambda nl.\ nl = 0\uparrow max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))\ @\ map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ 0\ \#\ xs\ @\ anything\}$
⟨*proof*⟩

lemma *length_le_max_insert_rec_ci*[*intro*]:
$length\ gs \le ffp \Longrightarrow length\ gs \le max\ x1\ (Max\ (insert\ ffp\ (x2\ `\ x3\ `\ set\ gs)))$
⟨*proof*⟩

lemma *restore_new_paras*:
$ffp \ge length\ gs$
$\Longrightarrow \{\lambda nl.\ nl = 0\uparrow max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))\ @\ map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ 0\ \#\ xs\ @\ anything\}$
$mv\_boxes\ (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))))\ 0\ (length\ gs)$
$\{\lambda nl.\ nl = map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ 0\uparrow max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))\ @\ 0\ \#\ xs\ @\ anything\}$
⟨*proof*⟩

lemma *le_max_insert*[*intro*]: $ffp \le max\ x0\ (Max\ (insert\ ffp\ (x1\ `\ x2\ `\ set\ gs)))$
⟨*proof*⟩

declare *max_less_iff_conj*[*simp del*]

lemma *save_rs*:
$[\![far = length\ gs;$
$ffp \le max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)));$
$far < ffp]\!]$
$\Longrightarrow \{\lambda nl.\ nl = map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @$
$rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0\uparrow max\ (Suc\ (length\ xs))$
$(Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))\ @\ xs\ @\ anything\}$
$mv\_box\ far\ (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))))$
$\{\lambda nl.\ nl = map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @$
$0\uparrow(max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) - length\ gs)\ @$
$rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0\uparrow length\ gs\ @\ xs\ @\ anything\}$
⟨*proof*⟩

lemma *length_empty_boxes*[*simp*]: $length\ (empty\_boxes\ n) = 2{*}n$
⟨*proof*⟩

lemma *empty_one_box_correct*:

$\{\lambda nl.\ nl = 0 \uparrow n\ @\ x\ \#\ lm\}\ [Dec\ n\ 2, Goto\ 0]\ \{\lambda nl.\ nl = 0\ \#\ 0 \uparrow n\ @\ lm\}$
$\langle proof \rangle$

lemma *empty_boxes_correct*:
$length\ lm \geq n \Longrightarrow$
$\{\lambda\ nl.\ nl = lm\}\ empty\_boxes\ n\ \{\lambda\ nl.\ nl = 0 \uparrow n\ @\ drop\ n\ lm\}$
$\langle proof \rangle$

lemma *insert_dominated*[*simp*]: $length\ gs \leq ffp \Longrightarrow$
$length\ gs + (max\ xs\ (Max\ (insert\ ffp\ (x1\ `\ x2\ `\ set\ gs)))) - length\ gs) =$
$max\ xs\ (Max\ (insert\ ffp\ (x1\ `\ x2\ `\ set\ gs)))$
$\langle proof \rangle$

lemma *clean_paras*:
$ffp \geq length\ gs \Longrightarrow$
$\{\lambda nl.\ nl = map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @$
$0 \uparrow (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) - length\ gs)\ @$
$rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow length\ gs\ @\ xs\ @\ anything\}$
$empty\_boxes\ (length\ gs)$
$\{\lambda nl.\ nl = 0 \uparrow max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))\ @$
$rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow length\ gs\ @\ xs\ @\ anything\}$
$\langle proof \rangle$

lemma *restore_rs*:
$\{\lambda nl.\ nl = 0 \uparrow max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))\ @$
$rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow length\ gs\ @\ xs\ @\ anything\}$
$mv\_box\ (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))))\ (length\ xs)$
$\{\lambda nl.\ nl = 0 \uparrow length\ xs\ @$
$rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#$
$0 \uparrow (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) - (length\ xs))\ @$
$0 \uparrow length\ gs\ @\ xs\ @\ anything\}$
$\langle proof \rangle$

lemma *restore_orgin_paras*:
$\{\lambda nl.\ nl = 0 \uparrow length\ xs\ @$
$rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#$
$0 \uparrow (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) - length\ xs)\ @\ 0 \uparrow length\ gs\ @\ xs\ @\ anything\}$
$mv\_boxes\ (Suc\ (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) + length\ gs))\ 0\ (length\ xs)$
$\{\lambda nl.\ nl = xs\ @\ rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow$
$(max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) + length\ gs)\ @$
$anything\}$

⟨*proof*⟩

lemma *compile_cn_correct'*:
assumes *f_ind*:
⋀ *anything r*. *rec_exec f* (*map* (λ*g*. *rec_exec g xs*) *gs*) = *rec_exec* (*Cn* (*length xs*) *f gs*) *xs* ⟹
{λ*nl*. *nl* = *map* (λ*g*. *rec_exec g xs*) *gs* @ *0* ↑ (*ffp* − *far*) @ *anything*} *fap*
{λ*nl*. *nl* = *map* (λ*g*. *rec_exec g xs*) *gs* @ *rec_exec* (*Cn* (*length xs*) *f gs*) *xs* # *0* ↑ (*ffp* − *Suc far*) @ *anything*}
and *compile*: *rec_ci f* = (*fap*, *far*, *ffp*)
and *term_f*: *terminate f* (*map* (λ*g*. *rec_exec g xs*) *gs*)
and *g_cond*: ∀ *g*∈*set gs*. *terminate g xs* ∧
(∀ *x xa xb*. *rec_ci g* = (*x*, *xa*, *xb*) ⟶
(∀ *xc*. {λ*nl*. *nl* = *xs* @ *0* ↑ (*xb* − *xa*) @ *xc*} *x* {λ*nl*. *nl* = *xs* @ *rec_exec g xs* # *0* ↑ (*xb* − *Suc xa*) @ *xc*}))
shows
{λ*nl*. *nl* = *xs* @ *0* # *0* ↑ (*max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ‘ *rec_ci* ‘ *set gs*))) + *length gs*) @ *anything*}
cn_merge_gs (*map rec_ci gs*) (*max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ‘ *rec_ci* ‘ *set gs*)))) [+]
(*mv_boxes 0* (*Suc* (*max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ‘ *rec_ci* ‘ *set gs*))) + *length gs*)) (*length xs*) [+]
(*mv_boxes* (*max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ‘ *rec_ci* ‘ *set gs*)))) *0* (*length gs*) [+]
(*fap* [+] (*mv_box far* (*max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ‘ *rec_ci* ‘ *set gs*)))) [+]
(*empty_boxes* (*length gs*) [+]
(*mv_box* (*max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ‘ *rec_ci* ‘ *set gs*)))) (*length xs*) [+]
mv_boxes (*Suc* (*max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ‘ *rec_ci* ‘ *set gs*))) + *length gs*)) *0* (*length xs*)))))))
{λ*nl*. *nl* = *xs* @ *rec_exec* (*Cn* (*length xs*) *f gs*) *xs* # *0* ↑ (*max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ‘ *rec_ci* ‘ *set gs*))) + *length gs*) @ *anything*}
⟨*proof*⟩

lemma *compile_cn_correct*:
assumes *termi_f*: *terminate f* (*map* (λ*g*. *rec_exec g xs*) *gs*)
and *f_ind*: ⋀*ap arity fp anything*.
rec_ci f = (*ap*, *arity*, *fp*)
⟹ {λ*nl*. *nl* = *map* (λ*g*. *rec_exec g xs*) *gs* @ *0* ↑ (*fp* − *arity*) @ *anything*} *ap*
{λ*nl*. *nl* = *map* (λ*g*. *rec_exec g xs*) *gs* @ *rec_exec f* (*map* (λ*g*. *rec_exec g xs*) *gs*) # *0* ↑ (*fp* − *Suc arity*) @ *anything*}
and *g_cond*:
∀ *g*∈*set gs*. *terminate g xs* ∧
(∀ *x xa xb*. *rec_ci g* = (*x*, *xa*, *xb*) ⟶ (∀ *xc*. {λ*nl*. *nl* = *xs* @ *0* ↑ (*xb* − *xa*) @ *xc*} *x* {λ*nl*. *nl* = *xs* @ *rec_exec g xs* # *0* ↑ (*xb* − *Suc xa*) @ *xc*}))
and *compile*: *rec_ci* (*Cn n f gs*) = (*ap*, *arity*, *fp*)
and *len*: *length xs* = *n*
shows {λ*nl*. *nl* = *xs* @ *0* ↑ (*fp* − *arity*) @ *anything*} *ap* {λ*nl*. *nl* = *xs* @ *rec_exec* (*Cn n f gs*)

$xs \# 0 \uparrow (fp - Suc\ arity) @ anything\}$
⟨*proof*⟩

lemma *mv_box_correct_simp*[*simp*]:
⟦*length xs* = *n*; *ft* = *max* (*n*+3) (*max fft gft*)⟧
⟹ {λ*nl*. *nl* = *xs* @ 0 # 0 ↑ (*ft* − *n*) @ *anything*} *mv_box n ft*
{λ*nl*. *nl* = *xs* @ 0 # 0 ↑ (*ft* − *n*) @ *anything*}
⟨*proof*⟩

lemma *length_under_max*[*simp*]: *length xs* < *max* (*length xs* + 3) *fft*
⟨*proof*⟩

lemma *save_init_rs*:
⟦*length xs* = *n*; *ft* = *max* (*n*+3) (*max fft gft*)⟧
⟹ {λ*nl*. *nl* = *xs* @ *rec_exec f xs* # 0 ↑ (*ft* − *n*) @ *anything*} *mv_box n* (*Suc n*)
{λ*nl*. *nl* = *xs* @ 0 # *rec_exec f xs* # 0 ↑ (*ft* − *Suc n*) @ *anything*}
⟨*proof*⟩

lemma *less_then_max_plus2*[*simp*]: *n* + (2::*nat*) < *max* (*n* + 3) *x*
⟨*proof*⟩

lemma *less_then_max_plus3*[*simp*]: *n* < *max* (*n* + (3::*nat*)) *x*
⟨*proof*⟩

lemma *mv_box_max_plus_3_correct*[*simp*]:
length xs = *n* ⟹
{λ*nl*. *nl* = *xs* @ *x* # 0 ↑ (*max* (*n* + (3::*nat*)) (*max fft gft*) − *n*) @ *anything*} *mv_box n* (*max* (*n* + 3) (*max fft gft*))
{λ*nl*. *nl* = *xs* @ 0 ↑ (*max* (*n* + 3) (*max fft gft*) − *n*) @ *x* # *anything*}
⟨*proof*⟩

lemma *max_less_suc_suc*[*simp*]: *max n* (*Suc n*) < *Suc* (*Suc* (*max* (*n* + 3) *x* + *anything* − *Suc* 0))
⟨*proof*⟩

lemma *suc_less_plus_3*[*simp*]: *Suc n* < *max* (*n* + 3) *x*
⟨*proof*⟩

lemma *mv_box_ok_suc_simp*[*simp*]:
length xs = *n*
⟹ {λ*nl*. *nl* = *xs* @ *rec_exec f xs* # 0 ↑ (*max* (*n* + 3) (*max fft gft*) − *Suc n*) @ *x* # *anything*} *mv_box n* (*Suc n*)
{λ*nl*. *nl* = *xs* @ 0 # *rec_exec f xs* # 0 ↑ (*max* (*n* + 3) (*max fft gft*) − *Suc* (*Suc n*)) @ *x* # *anything*}
⟨*proof*⟩

lemma *abc_append_frist_steps_eq_pre*:
assumes *notfinal*: *abc_notfinal* (*abc_steps_l* (0, *lm*) *A n*) *A*
and *notnull*: *A* ≠ []
shows *abc_steps_l* (0, *lm*) (*A* @ *B*) *n* = *abc_steps_l* (0, *lm*) *A n*

⟨*proof*⟩

lemma *abc_append_first_step_eq_pre*:
st < *length A*
⟹ *abc_step_l* (*st*, *lm*) (*abc_fetch st* (*A* @ *B*)) =
abc_step_l (*st*, *lm*) (*abc_fetch st A*)
⟨*proof*⟩

lemma *abc_append_frist_steps_halt_eq′*:
assumes *final*: *abc_steps_l* (*0*, *lm*) *A n* = (*length A*, *lm′*)
and *notnull*: *A* ≠ []
shows ∃ *n′*. *abc_steps_l* (*0*, *lm*) (*A* @ *B*) *n′* = (*length A*, *lm′*)
⟨*proof*⟩

lemma *abc_append_frist_steps_halt_eq*:
assumes *final*: *abc_steps_l* (*0*, *lm*) *A n* = (*length A*, *lm′*)
shows ∃ *n′*. *abc_steps_l* (*0*, *lm*) (*A* @ *B*) *n′* = (*length A*, *lm′*)
⟨*proof*⟩

lemma *suc_suc_max_simp*[*simp*]: *Suc* (*Suc* (*max* (*xs* + *3*) *fft* − *Suc* (*Suc* (*xs*))))
= *max* (*xs* + *3*) *fft* − (*xs*)
⟨*proof*⟩

lemma *contract_dec_ft_length_plus_7*[*simp*]: ⟦*ft* = *max* (*n* + *3*) (*max fft gft*); *length xs* = *n*⟧
⟹
{λ*nl*. *nl* = *xs* @ (*x* − *Suc y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*]) # *0* ↑ (*ft* − *Suc* (*Suc n*)) @ *Suc y* # *anything*}
[*Dec ft* (*length gap* + *7*)]
{λ*nl*. *nl* = *xs* @ (*x* − *Suc y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*]) # *0* ↑ (*ft* − *Suc* (*Suc n*)) @ *y* # *anything*}
⟨*proof*⟩

lemma *adjust_paras′*:
length xs = *n* ⟹ {λ*nl*. *nl* = *xs* @ *x* # *y* # *anything*} [*Inc n*] [+] [*Dec* (*Suc n*) *2*, *Goto 0*]
{λ*nl*. *nl* = *xs* @ *Suc x* # *0* # *anything*}
⟨*proof*⟩

lemma *adjust_paras*:
length xs = *n* ⟹ {λ*nl*. *nl* = *xs* @ *x* # *y* # *anything*} [*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)]
{λ*nl*. *nl* = *xs* @ *Suc x* # *0* # *anything*}
⟨*proof*⟩

lemma *rec_ci_SucSuc_n*[*simp*]: ⟦*rec_ci g* = (*gap*, *gar*, *gft*); ∀ *y*<*x*. *terminate g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])]);
length xs = *n*; *Suc y*≤*x*⟧ ⟹ *gar* = *Suc* (*Suc n*)
⟨*proof*⟩

lemma *loop_back′*:
assumes *h*: *length A* = *length gap* + *4 length xs* = *n*
and *le*: *y* ≥ *x*

shows $\exists$ *stp*. *abc_steps_l* (*length A*, *xs* @ *m* # (*y* − *x*) # *x* # *anything*) (*A* @ [*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length gap* + *4*)]) *stp*
= (*length A*, *xs* @ *m* # *y* # *0* # *anything*)
⟨*proof*⟩

lemma *loop_back*:
assumes *h*: *length A* = *length gap* + *4* *length xs* = *n*
shows $\exists$ *stp*. *abc_steps_l* (*length A*, *xs* @ *m* # *0* # *x* # *anything*) (*A* @ [*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length gap* + *4*)]) *stp*
= (*0*, *xs* @ *m* # *x* # *0* # *anything*)
⟨*proof*⟩

lemma *rec_exec_pr_0_simps*: *rec_exec* (*Pr n f g*) (*xs* @ [*0*]) = *rec_exec f xs*
⟨*proof*⟩

lemma *rec_exec_pr_Suc_simps*: *rec_exec* (*Pr n f g*) (*xs* @ [*Suc y*])
= *rec_exec g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])])
⟨*proof*⟩

lemma *suc_max_simp*[*simp*]: *Suc* (*max* (*n* + *3*) *fft* − *Suc* (*Suc* (*Suc n*))) = *max* (*n* + *3*) *fft* − *Suc* (*Suc n*)
⟨*proof*⟩

lemma *pr_loop*:
assumes *code*: *code* = ([*Dec* (*max* (*n* + *3*) (*max fft gft*)) (*length gap* + *7*)] [+] (*gap* [+] [*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)])) @
[*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length gap* + *4*)]
and *len*: *length xs* = *n*
and *g_ind*: ∀ *y*<*x*. (∀ *anything*. {λ*nl*. *nl* = *xs* @ *y* # *rec_exec* (*Pr n f g*) (*xs* @ [*y*]) # *0* ↑ (*gft* − *gar*) @ *anything*} *gap*
{λ*nl*. *nl* = *xs* @ *y* # *rec_exec* (*Pr n f g*) (*xs* @ [*y*]) # *rec_exec g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])]) # *0* ↑ (*gft* − *Suc gar*) @ *anything*})
and *compile_g*: *rec_ci g* = (*gap*, *gar*, *gft*)
and *termi_g*: ∀ *y*<*x*. *terminate g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])])
and *ft*: *ft* = *max* (*n* + *3*) (*max fft gft*)
and *less*: *Suc y* ≤ *x*
shows
∃ *stp*. *abc_steps_l* (*0*, *xs* @ (*x* − *Suc y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*]) # *0* ↑ (*ft* − *Suc* (*Suc n*)) @ *Suc y* # *anything*)
code stp = (*0*, *xs* @ (*x* − *y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *y*]) # *0* ↑ (*ft* − *Suc* (*Suc n*)) @ *y* # *anything*)
⟨*proof*⟩

lemma *abc_lm_s_simp0*[*simp*]:
length xs = *n* ⟹ *abc_lm_s* (*xs* @ *x* # *rec_exec* (*Pr n f g*) (*xs* @ [*x*]) # *0* ↑ (*max* (*n* + *3*) (*max fft gft*) − *Suc* (*Suc n*)) @ *0* # *anything*) (*max* (*n* + *3*) (*max fft gft*)) *0* =
xs @ *x* # *rec_exec* (*Pr n f g*) (*xs* @ [*x*]) # *0* ↑ (*max* (*n* + *3*) (*max fft gft*) − *Suc n*) @ *anything*
⟨*proof*⟩

lemma *index_at_zero_elem*[*simp*]:
(*xs* @ *x* # *re* # *0* ↑ (*max* (*length xs* + *3*)
(*max fft gft*) − *Suc* (*Suc* (*length xs*))) @ *0* # *anything*) !
max (*length xs* + *3*) (*max fft gft*) = *0*
⟨*proof*⟩

lemma *pr_loop_correct*:
assumes *less*: $y \leq x$
and *len*: *length xs* = *n*
and *compile_g*: *rec_ci g* = (*gap*, *gar*, *gft*)
and *termi_g*: ∀ *y*<*x*. *terminate g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])])
and *g_ind*: ∀ *y*<*x*. (∀ *anything*. {λ*nl*. *nl* = *xs* @ *y* # *rec_exec* (*Pr n f g*) (*xs* @ [*y*]) # *0* ↑ (*gft* − *gar*) @ *anything*} *gap*
{λ*nl*. *nl* = *xs* @ *y* # *rec_exec* (*Pr n f g*) (*xs* @ [*y*]) # *rec_exec g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])]) # *0* ↑ (*gft* − *Suc gar*) @ *anything*})
shows {λ*nl*. *nl* = *xs* @ (*x* − *y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *y*]) # *0* ↑ (*max* (*n* + *3*) (*max fft gft*) − *Suc* (*Suc n*)) @ *y* # *anything*}
([*Dec* (*max* (*n* + *3*) (*max fft gft*)) (*length gap* + *7*)] [+] (*gap* [+] [*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)])) @ [*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length gap* + *4*)]
{λ*nl*. *nl* = *xs* @ *x* # *rec_exec* (*Pr n f g*) (*xs* @ [*x*]) # *0* ↑ (*max* (*n* + *3*) (*max fft gft*) − *Suc n*) @ *anything*}
⟨*proof*⟩

lemma *compile_pr_correct*′:
assumes *termi_g*: ∀ *y*<*x*. *terminate g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])])
and *g_ind*:
∀ *y*<*x*. (∀ *anything*. {λ*nl*. *nl* = *xs* @ *y* # *rec_exec* (*Pr n f g*) (*xs* @ [*y*]) # *0* ↑ (*gft* − *gar*) @ *anything*} *gap*
{λ*nl*. *nl* = *xs* @ *y* # *rec_exec* (*Pr n f g*) (*xs* @ [*y*]) # *rec_exec g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])]) # *0* ↑ (*gft* − *Suc gar*) @ *anything*})
and *termi_f*: *terminate f xs*
and *f_ind*: ⋀ *anything*. {λ*nl*. *nl* = *xs* @ *0* ↑ (*fft* − *far*) @ *anything*} *fap* {λ*nl*. *nl* = *xs* @ *rec_exec f xs* # *0* ↑ (*fft* − *Suc far*) @ *anything*}
and *len*: *length xs* = *n*
and *compile1*: *rec_ci f* = (*fap*, *far*, *fft*)
and *compile2*: *rec_ci g* = (*gap*, *gar*, *gft*)
shows
{λ*nl*. *nl* = *xs* @ *x* # *0* ↑ (*max* (*n* + *3*) (*max fft gft*) − *n*) @ *anything*}
mv_box n (*max* (*n* + *3*) (*max fft gft*)) [+]
(*fap* [+] (*mv_box n* (*Suc n*) [+]
([*Dec* (*max* (*n* + *3*) (*max fft gft*)) (*length gap* + *7*)] [+] (*gap* [+] [*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)]) @
[*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length gap* + *4*)])))
{λ*nl*. *nl* = *xs* @ *x* # *rec_exec* (*Pr n f g*) (*xs* @ [*x*]) # *0* ↑ (*max* (*n* + *3*) (*max fft gft*) − *Suc n*) @ *anything*}
⟨*proof*⟩

lemma *compile_pr_correct*:
assumes *g_ind*: ∀ *y*<*x*. *terminate g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])]) ∧
(∀ *x xa xb*. *rec_ci g* = (*x*, *xa*, *xb*) ⟶

(∀ xc. {λnl. nl = xs @ y # rec_exec (Pr n f g) (xs @ [y]) # 0 ↑ (xb − xa) @ xc} x
{λnl. nl = xs @ y # rec_exec (Pr n f g) (xs @ [y]) # rec_exec g (xs @ [y, rec_exec (Pr n f g) (xs @ [y])]) # 0 ↑ (xb − Suc xa) @ xc}))
and *termi_f*: *terminate f xs*
and *f_ind*:
⋀*ap arity fp anything*.
rec_ci f = (*ap*, *arity*, *fp*) ⟹ {λnl. nl = xs @ 0 ↑ (fp − arity) @ anything} ap {λnl. nl = xs @ rec_exec f xs # 0 ↑ (fp − Suc arity) @ anything}
and *len*: *length xs* = *n*
and *compile*: *rec_ci* (*Pr n f g*) = (*ap*, *arity*, *fp*)
shows {λnl. nl = xs @ x # 0 ↑ (fp − arity) @ anything} ap {λnl. nl = xs @ x # rec_exec (Pr n f g) (xs @ [x]) # 0 ↑ (fp − Suc arity) @ anything}
⟨*proof*⟩

fun *mn_ind_inv* ::
nat × *nat list* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat list* ⇒ *nat list* ⇒ *bool*
where
mn_ind_inv (*as*, *lm'*) *ss x rsx suf_lm lm* =
(*if as* = *ss* then *lm'* = *lm* @ *x* # *rsx* # *suf_lm*
else if as = *ss* + 1 *then*
∃ *y*. (*lm'* = *lm* @ *x* # *y* # *suf_lm*) ∧ *y* ≤ *rsx*
else if as = *ss* + 2 *then*
∃ *y*. (*lm'* = *lm* @ *x* # *y* # *suf_lm*) ∧ *y* ≤ *rsx*
else if as = *ss* + 3 *then lm'* = *lm* @ *x* # 0 # *suf_lm*
else if as = *ss* + 4 *then lm'* = *lm* @ *Suc x* # 0 # *suf_lm*
else if as = 0 *then lm'* = *lm* @ *Suc x* # 0 # *suf_lm*
else False
)

fun *mn_stage1* :: *nat* × *nat list* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
mn_stage1 (*as*, *lm*) *ss n* =
(*if as* = 0 *then* 0
else if as = *ss* + 4 *then* 1
else if as = *ss* + 3 *then* 2
else if as = *ss* + 2 ∨ *as* = *ss* + 1 *then* 3
else if as = *ss then* 4
else 0
)

fun *mn_stage2* :: *nat* × *nat list* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
mn_stage2 (*as*, *lm*) *ss n* =
(*if as* = *ss* + 1 ∨ *as* = *ss* + 2 *then* (*lm* ! (*Suc n*))
else 0)

fun *mn_stage3* :: *nat* × *nat list* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
mn_stage3 (*as*, *lm*) *ss n* = (*if as* = *ss* + 2 *then* 1 *else* 0)

fun *mn_measure* :: $((nat \times nat\ list) \times nat \times nat) \Rightarrow$
$(nat \times nat \times nat)$
where
mn_measure $((as, lm), ss, n) =$
$(mn\_stage1\ (as, lm)\ ss\ n, mn\_stage2\ (as, lm)\ ss\ n,$
$mn\_stage3\ (as, lm)\ ss\ n)$

definition *mn_LE* :: $(((nat \times nat\ list) \times nat \times nat) \times$
$((nat \times nat\ list) \times nat \times nat))\ set$
where *mn_LE* $\stackrel{def}{=}$ $(inv\_image\ lex\_triple\ mn\_measure)$

lemma *wf_mn_le*[*intro*]: *wf mn_LE*
⟨*proof*⟩

declare *mn_ind_inv.simps*[*simp del*]

lemma *put_in_tape_small_enough0*[*simp*]:
$0 < rsx \Longrightarrow$
$\exists y.\ (xs\ @\ x\ \#\ rsx\ \#\ anything)[Suc\ (length\ xs) := rsx - Suc\ 0] = xs\ @\ x\ \#\ y\ \#\ anything \wedge y \leq rsx$
⟨*proof*⟩

lemma *put_in_tape_small_enough1*[*simp*]:
$[\![y \leq rsx; 0 < y]\!]$
$\Longrightarrow \exists ya.\ (xs\ @\ x\ \#\ y\ \#\ anything)[Suc\ (length\ xs) := y - Suc\ 0] = xs\ @\ x\ \#\ ya\ \#\ anything \wedge ya \leq rsx$
⟨*proof*⟩

lemma *abc_comp_null*[*simp*]: $(A\ [+]\ B = []) = (A = [] \wedge B = [])$
⟨*proof*⟩

lemma *rec_ci_not_null*[*simp*]: $(rec\_ci\ f \neq ([], a, b))$
⟨*proof*⟩

lemma *mn_correct*:
assumes *compile*: *rec_ci rf* $= (fap, far, fft)$
and *ge*: $0 < rsx$
and *len*: *length xs* $=$ *arity*
and *B*: $B = [Dec\ (Suc\ arity)\ (length\ fap + 5), Dec\ (Suc\ arity)\ (length\ fap + 3), Goto\ (Suc\ (length\ fap)), Inc\ arity, Goto\ 0]$
and *f*: $f = (\lambda\ stp.\ (abc\_steps\_l\ (length\ fap, xs\ @\ x\ \#\ rsx\ \#\ anything)\ (fap\ @\ B)\ stp, (length\ fap), arity))$
and *P*: $P = (\lambda\ ((as, lm), ss, arity).\ as = 0)$
and *Q*: $Q = (\lambda\ ((as, lm), ss, arity).\ mn\_ind\_inv\ (as, lm)\ (length\ fap)\ x\ rsx\ anything\ xs)$
shows $\exists\ stp.\ P\ (f\ stp) \wedge Q\ (f\ stp)$
⟨*proof*⟩

lemma *abc_Hoare_haltE*:
$\{\lambda\ nl.\ nl = lm1\}\ p\ \{\lambda\ nl.\ nl = lm2\}$
$\Longrightarrow \exists\ stp.\ abc\_steps\_l\ (0, lm1)\ p\ stp = (length\ p, lm2)$
⟨*proof*⟩

lemma *mn_loop*:
assumes *B*: $B = [Dec\ (Suc\ arity)\ (length\ fap + 5), Dec\ (Suc\ arity)\ (length\ fap + 3), Goto\ (Suc\ (length\ fap)), Inc\ arity, Goto\ 0]$
and *ft*: $ft = max\ (Suc\ arity)\ fft$
and *len*: $length\ xs = arity$
and *far*: $far = Suc\ arity$
and *ind*: $(\forall xc.\ (\{\lambda nl.\ nl = xs\ @\ x\ \#\ 0 \uparrow (fft - far)\ @\ xc\}\ fap$
$\{\lambda nl.\ nl = xs\ @\ x\ \#\ rec\_exec\ f\ (xs\ @\ [x])\ \#\ 0 \uparrow (fft - Suc\ far)\ @\ xc\}))$
and *exec_less*: $rec\_exec\ f\ (xs\ @\ [x]) > 0$
and *compile*: $rec\_ci\ f = (fap, far, fft)$
shows $\exists\ stp > 0.\ abc\_steps\_l\ (0, xs\ @\ x\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)\ (fap\ @\ B)\ stp =$
$(0, xs\ @\ Suc\ x\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)$
⟨*proof*⟩

lemma *mn_loop_correct'*:
assumes *B*: $B = [Dec\ (Suc\ arity)\ (length\ fap + 5), Dec\ (Suc\ arity)\ (length\ fap + 3), Goto\ (Suc\ (length\ fap)), Inc\ arity, Goto\ 0]$
and *ft*: $ft = max\ (Suc\ arity)\ fft$
and *len*: $length\ xs = arity$
and *ind_all*: $\forall i \leq x.\ (\forall xc.\ (\{\lambda nl.\ nl = xs\ @\ i\ \#\ 0 \uparrow (fft - far)\ @\ xc\}\ fap$
$\{\lambda nl.\ nl = xs\ @\ i\ \#\ rec\_exec\ f\ (xs\ @\ [i])\ \#\ 0 \uparrow (fft - Suc\ far)\ @\ xc\}))$
and *exec_ge*: $\forall\ i \leq x.\ rec\_exec\ f\ (xs\ @\ [i]) > 0$
and *compile*: $rec\_ci\ f = (fap, far, fft)$
and *far*: $far = Suc\ arity$
shows $\exists\ stp > x.\ abc\_steps\_l\ (0, xs\ @\ 0\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)\ (fap\ @\ B)\ stp =$
$(0, xs\ @\ Suc\ x\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)$
⟨*proof*⟩

lemma *mn_loop_correct*:
assumes *B*: $B = [Dec\ (Suc\ arity)\ (length\ fap + 5), Dec\ (Suc\ arity)\ (length\ fap + 3), Goto\ (Suc\ (length\ fap)), Inc\ arity, Goto\ 0]$
and *ft*: $ft = max\ (Suc\ arity)\ fft$
and *len*: $length\ xs = arity$
and *ind_all*: $\forall i \leq x.\ (\forall xc.\ (\{\lambda nl.\ nl = xs\ @\ i\ \#\ 0 \uparrow (fft - far)\ @\ xc\}\ fap$
$\{\lambda nl.\ nl = xs\ @\ i\ \#\ rec\_exec\ f\ (xs\ @\ [i])\ \#\ 0 \uparrow (fft - Suc\ far)\ @\ xc\}))$
and *exec_ge*: $\forall\ i \leq x.\ rec\_exec\ f\ (xs\ @\ [i]) > 0$
and *compile*: $rec\_ci\ f = (fap, far, fft)$
and *far*: $far = Suc\ arity$
shows $\exists\ stp.\ abc\_steps\_l\ (0, xs\ @\ 0\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)\ (fap\ @\ B)\ stp =$
$(0, xs\ @\ Suc\ x\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)$
⟨*proof*⟩

lemma *compile_mn_correct'*:
assumes *B*: $B = [Dec\ (Suc\ arity)\ (length\ fap + 5), Dec\ (Suc\ arity)\ (length\ fap + 3), Goto\ (Suc\ (length\ fap)), Inc\ arity, Goto\ 0]$

and *ft*: $ft = max\ (Suc\ arity)\ fft$
and *len*: $length\ xs = arity$
and *termi_f*: $terminate\ f\ (xs\ @\ [r])$
and *f_ind*: $\bigwedge anything.\ \{\lambda nl.\ nl = xs\ @\ r\ \#\ 0 \uparrow (fft - far)\ @\ anything\}\ fap$
$\{\lambda nl.\ nl = xs\ @\ r\ \#\ 0\ \#\ 0 \uparrow (fft - Suc\ far)\ @\ anything\}$
and *ind_all*: $\forall i < r.\ (\forall xc.\ (\{\lambda nl.\ nl = xs\ @\ i\ \#\ 0 \uparrow (fft - far)\ @\ xc\}\ fap$
$\{\lambda nl.\ nl = xs\ @\ i\ \#\ rec\_exec\ f\ (xs\ @\ [i])\ \#\ 0 \uparrow (fft - Suc\ far)\ @\ xc\}))$
and *exec_less*: $\forall\ i<r.\ rec\_exec\ f\ (xs\ @\ [i]) > 0$
and *exec*: $rec\_exec\ f\ (xs\ @\ [r]) = 0$
and *compile*: $rec\_ci\ f = (fap, far, fft)$
shows $\{\lambda nl.\ nl = xs\ @\ 0 \uparrow (max\ (Suc\ arity)\ fft - arity)\ @\ anything\}$
$fap\ @\ B$
$\{\lambda nl.\ nl = xs\ @\ rec\_exec\ (Mn\ arity\ f)\ xs\ \#\ 0 \uparrow (max\ (Suc\ arity)\ fft - Suc\ arity)\ @\ anything\}$
⟨*proof*⟩

lemma *compile_mn_correct*:
assumes *len*: $length\ xs = n$
and *termi_f*: $terminate\ f\ (xs\ @\ [r])$
and *f_ind*: $\bigwedge ap\ arity\ fp\ anything.\ rec\_ci\ f = (ap, arity, fp) \Longrightarrow$
$\{\lambda nl.\ nl = xs\ @\ r\ \#\ 0 \uparrow (fp - arity)\ @\ anything\}\ ap\ \{\lambda nl.\ nl = xs\ @\ r\ \#\ 0\ \#\ 0 \uparrow (fp - Suc\ arity)\ @\ anything\}$
and *exec*: $rec\_exec\ f\ (xs\ @\ [r]) = 0$
and *ind_all*:
$\forall i<r.\ terminate\ f\ (xs\ @\ [i]) \wedge$
$(\forall x\ xa\ xb.\ rec\_ci\ f = (x, xa, xb) \longrightarrow$
$(\forall xc.\ \{\lambda nl.\ nl = xs\ @\ i\ \#\ 0 \uparrow (xb - xa)\ @\ xc\}\ x\ \{\lambda nl.\ nl = xs\ @\ i\ \#\ rec\_exec\ f\ (xs\ @\ [i])\ \#\ 0 \uparrow (xb - Suc\ xa)\ @\ xc\})) \wedge$
$0 < rec\_exec\ f\ (xs\ @\ [i])$
and *compile*: $rec\_ci\ (Mn\ n\ f) = (ap, arity, fp)$
shows $\{\lambda nl.\ nl = xs\ @\ 0 \uparrow (fp - arity)\ @\ anything\}\ ap$
$\{\lambda nl.\ nl = xs\ @\ rec\_exec\ (Mn\ n\ f)\ xs\ \#\ 0 \uparrow (fp - Suc\ arity)\ @\ anything\}$
⟨*proof*⟩

lemma *recursive_compile_correct*:
$[\![terminate\ recf\ args;\ rec\_ci\ recf = (ap, arity, fp)]\!]$
$\Longrightarrow \{\lambda\ nl.\ nl = args\ @\ 0\uparrow(fp - arity)\ @\ anything\}\ ap$
$\{\lambda\ nl.\ nl = args@\ rec\_exec\ recf\ args\ \#\ 0\uparrow(fp - Suc\ arity)\ @\ anything\}$
⟨*proof*⟩

definition *dummy_abc* :: $nat \Rightarrow abc\_inst\ list$
where
$dummy\_abc\ k = [Inc\ k, Dec\ k\ 0, Goto\ 3]$

definition *abc_list_crsp*:: $nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
$abc\_list\_crsp\ xs\ ys = (\exists\ n.\ xs = ys\ @\ 0\uparrow n \vee ys = xs\ @\ 0\uparrow n)$

lemma *abc_list_crsp_simp1*[*intro*]: $abc\_list\_crsp\ (lm\ @\ 0\uparrow m)\ lm$
⟨*proof*⟩

lemma *abc_list_crsp_lm_v*:
abc_list_crsp lma lmb $\Longrightarrow$ *abc_lm_v lma n* = *abc_lm_v lmb n*
⟨*proof*⟩

lemma *abc_list_crsp_elim*:
⟦*abc_list_crsp lma lmb*; ∃ *n*. *lma* = *lmb* @ *0*↑*n* ∨ *lmb* = *lma* @ *0*↑*n* $\Longrightarrow$ *P* ⟧ $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *abc_list_crsp_simp*[*simp*]:
⟦*abc_list_crsp lma lmb*; *m* < *length lma*; *m* < *length lmb*⟧ $\Longrightarrow$
abc_list_crsp (*lma*[*m* := *n*]) (*lmb*[*m* := *n*])
⟨*proof*⟩

lemma *abc_list_crsp_simp2*[*simp*]:
⟦*abc_list_crsp lma lmb*; *m* < *length lma*; ¬ *m* < *length lmb*⟧ $\Longrightarrow$
abc_list_crsp (*lma*[*m* := *n*]) (*lmb* @ *0* ↑ (*m* − *length lmb*) @ [*n*])
⟨*proof*⟩

lemma *abc_list_crsp_simp3*[*simp*]:
⟦*abc_list_crsp lma lmb*; ¬ *m* < *length lma*; *m* < *length lmb*⟧ $\Longrightarrow$
abc_list_crsp (*lma* @ *0* ↑ (*m* − *length lma*) @ [*n*]) (*lmb*[*m* := *n*])
⟨*proof*⟩

lemma *abc_list_crsp_simp4*[*simp*]: ⟦*abc_list_crsp lma lmb*; ¬ *m* < *length lma*; ¬ *m* < *length lmb*⟧ $\Longrightarrow$
abc_list_crsp (*lma* @ *0* ↑ (*m* − *length lma*) @ [*n*]) (*lmb* @ *0* ↑ (*m* − *length lmb*) @ [*n*])
⟨*proof*⟩

lemma *abc_list_crsp_lm_s*:
abc_list_crsp lma lmb $\Longrightarrow$
abc_list_crsp (*abc_lm_s lma m n*) (*abc_lm_s lmb m n*)
⟨*proof*⟩

lemma *abc_list_crsp_step*:
⟦*abc_list_crsp lma lmb*; *abc_step_l* (*aa*, *lma*) *i* = (*a*, *lma*′);
abc_step_l (*aa*, *lmb*) *i* = (*a*′, *lmb*′)⟧
$\Longrightarrow$ *a*′ = *a* ∧ *abc_list_crsp lma*′ *lmb*′
⟨*proof*⟩

lemma *abc_list_crsp_steps*:
⟦*abc_steps_l* (*0*, *lm* @ *0*↑*m*) *aprog stp* = (*a*, *lm*′); *aprog* ≠ []⟧
$\Longrightarrow$ ∃ *lma*. *abc_steps_l* (*0*, *lm*) *aprog stp* = (*a*, *lma*) ∧
abc_list_crsp lm′ *lma*
⟨*proof*⟩

lemma *list_crsp_simp2*: *abc_list_crsp* (*lm1* @ *0*↑*n*) *lm2* $\Longrightarrow$ *abc_list_crsp lm1 lm2*
⟨*proof*⟩

lemma *recursive_compile_correct_norm′*:
⟦*rec_ci f* = (*ap*, *arity*, *ft*);
terminate f args⟧
⟹ ∃ *stp rl*. (*abc_steps_l* (0, *args*) *ap stp*) = (*length ap*, *rl*) ∧ *abc_list_crsp* (*args* @ [*rec_exec f args*]) *rl*
⟨*proof*⟩

lemma *find_exponent_rec_exec*[*simp*]:
assumes *a*: *args* @ [*rec_exec f args*] = *lm* @ 0 ↑ *n*
and *b*: *length args* < *length lm*
shows ∃ *m*. *lm* = *args* @ *rec_exec f args* # 0 ↑ *m*
⟨*proof*⟩

lemma *find_exponent_complex*[*simp*]:
⟦*args* @ [*rec_exec f args*] = *lm* @ 0 ↑ *n*; ¬ *length args* < *length lm*⟧
⟹ ∃ *m*. (*lm* @ 0 ↑ (*length args* − *length lm*) @ [*Suc* 0])[*length args* := 0] =
args @ *rec_exec f args* # 0 ↑ *m*
⟨*proof*⟩

lemma *compile_append_dummy_correct*:
assumes *compile*: *rec_ci f* = (*ap*, *ary*, *fp*)
and *termi*: *terminate f args*
shows {λ *nl*. *nl* = *args*} (*ap* [+] *dummy_abc* (*length args*)) {λ *nl*. (∃ *m*. *nl* = *args* @ *rec_exec f args* # 0↑*m*)}
⟨*proof*⟩

lemma *cn_merge_gs_split*:
⟦*i* < *length gs*; *rec_ci* (*gs*!*i*) = (*ga*, *gb*, *gc*)⟧ ⟹
cn_merge_gs (*map rec_ci gs*) *p* = *cn_merge_gs* (*map rec_ci* (*take i gs*)) *p* [+] (*ga* [+]
mv_box gb (*p* + *i*)) [+] *cn_merge_gs* (*map rec_ci* (*drop* (*Suc i*) *gs*)) (*p* + *Suc i*)
⟨*proof*⟩

lemma *cn_unhalt_case*:
assumes *compile1*: *rec_ci* (*Cn n f gs*) = (*ap*, *ar*, *ft*) ∧ *length args* = *ar*
and *g*: *i* < *length gs*
and *compile2*: *rec_ci* (*gs*!*i*) = (*gap*, *gar*, *gft*) ∧ *gar* = *length args*
and *g_unhalt*: ⋀ *anything*. {λ *nl*. *nl* = *args* @ 0↑(*gft* − *gar*) @ *anything*} *gap* ↑
and *g_ind*: ⋀ *apj arj ftj j anything*. ⟦*j* < *i*; *rec_ci* (*gs*!*j*) = (*apj*, *arj*, *ftj*)⟧
⟹ {λ *nl*. *nl* = *args* @ 0↑(*ftj* − *arj*) @ *anything*} *apj* {λ *nl*. *nl* = *args* @ *rec_exec* (*gs*!*j*) *args* # 0↑(*ftj* − *Suc arj*) @ *anything*}
and *all_termi*: ∀ *j*<*i*. *terminate* (*gs*!*j*) *args*
shows {λ *nl*. *nl* = *args* @ 0↑(*ft* − *ar*) @ *anything*} *ap* ↑
⟨*proof*⟩

lemma *mn_unhalt_case′*:
assumes *compile*: *rec_ci f* = (*a*, *b*, *c*)
and *all_termi*: ∀ *i*. *terminate f* (*args* @ [*i*]) ∧ 0 < *rec_exec f* (*args* @ [*i*])
and *B*: *B* = [*Dec* (*Suc* (*length args*)) (*length a* + 5), *Dec* (*Suc* (*length args*)) (*length a* + 3),

Goto (*Suc* (*length a*)), *Inc* (*length args*), *Goto 0*]
shows {λ*nl*. *nl* = *args* @ *0* ↑ (*max* (*Suc* (*length args*)) *c* − *length args*) @ *anything*}
a @ *B* ↑
⟨*proof*⟩

lemma *mn_unhalt_case*:
assumes *compile*: *rec_ci* (*Mn n f*) = (*ap*, *ar*, *ft*) ∧ *length args* = *ar*
and *all_term*: ∀ *i*. *terminate f* (*args* @ [*i*]) ∧ *rec_exec f* (*args* @ [*i*]) > *0*
shows {λ *nl*. *nl* = *args* @ *0*↑(*ft* − *ar*) @ *anything*} *ap* ↑
⟨*proof*⟩

fun *tm_of_rec* :: *recf* ⇒ *instr list*
where *tm_of_rec recf* = (*let* (*ap*, *k*, *fp*) = *rec_ci recf in*
let tp = *tm_of* (*ap* [+] *dummy_abc k*) *in*
tp @ (*shift* (*mopup k*) (*length tp div 2*)))

lemma *recursive_compile_to_tm_correct1*:
assumes *compile*: *rec_ci recf* = (*ap*, *ary*, *fp*)
and *termi*: *terminate recf args*
and *tp*: *tp* = *tm_of* (*ap* [+] *dummy_abc* (*length args*))
shows ∃ *stp m l*. *steps0* (*Suc 0*, *Bk* # *Bk* # *ires*, <*args*> @ *Bk*↑*rn*)
(*tp* @ *shift* (*mopup* (*length args*)) (*length tp div 2*)) *stp* = (*0*, *Bk*↑*m* @ *Bk* # *Bk* # *ires*, *Oc*↑*Suc*
(*rec_exec recf args*) @ *Bk*↑*l*)
⟨*proof*⟩

lemma *recursive_compile_to_tm_correct2*:
assumes *termi*: *terminate recf args*
shows ∃ *stp m l*. *steps0* (*Suc 0*, [*Bk*, *Bk*], <*args*>) (*tm_of_rec recf*) *stp* =
(*0*, *Bk*↑*Suc* (*Suc m*), *Oc*↑*Suc* (*rec_exec recf args*) @ *Bk*↑*l*)
⟨*proof*⟩

lemma *recursive_compile_to_tm_correct3*:
assumes *termi*: *terminate recf args*
shows {λ *tp*. *tp* =([*Bk*, *Bk*], <*args*>)} (*tm_of_rec recf*)
{λ *tp*. ∃ *k l*. *tp* = (*Bk*↑ *k*, <*rec_exec recf args*> @ *Bk* ↑ *l*)}
⟨*proof*⟩

lemma *list_all_suc_many*[*simp*]:
list_all (λ(*acn*, *s*). *s* ≤ *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*2* ∗ *n*)))))) *xs* ⟹
list_all (λ(*acn*, *s*). *s* ≤ *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*2* ∗ *n*)))))))) *xs*
⟨*proof*⟩

lemma *shift_append*: *shift* (*xs* @ *ys*) *n* = *shift xs n* @ *shift ys n*
⟨*proof*⟩

lemma *length_shift_mopup*[*simp*]: *length* (*shift* (*mopup n*) *ss*) = *4* ∗ *n* + *12*
⟨*proof*⟩

lemma *length_tm_even*[*intro*]: *length* (*tm_of ap*) *mod 2* = *0*

⟨*proof*⟩

lemma *tms_of_at_index*[*simp*]: *k* < *length ap* ⟹ *tms_of ap* ! *k* =
ci (*layout_of ap*) (*start_of* (*layout_of ap*) *k*) (*ap* ! *k*)
⟨*proof*⟩

lemma *start_of_suc_inc*:
⟦*k* < *length ap*; *ap* ! *k* = *Inc n*⟧ ⟹ *start_of* (*layout_of ap*) (*Suc k*) =
start_of (*layout_of ap*) *k* + 2 ∗ *n* + 9
⟨*proof*⟩

lemma *start_of_suc_dec*:
⟦*k* < *length ap*; *ap* ! *k* = (*Dec n e*)⟧ ⟹ *start_of* (*layout_of ap*) (*Suc k*) =
start_of (*layout_of ap*) *k* + 2 ∗ *n* + 16
⟨*proof*⟩

lemma *inc_state_all_le*:
⟦*k* < *length ap*; *ap* ! *k* = *Inc n*;
(*a*, *b*) ∈ *set* (*shift* (*shift tinc_b* (2 ∗ *n*))
(*start_of* (*layout_of ap*) *k* − *Suc 0*))⟧
⟹ *b* ≤ *start_of* (*layout_of ap*) (*length ap*)
⟨*proof*⟩

lemma *findnth_le*[*elim*]:
(*a*, *b*) ∈ *set* (*shift* (*findnth n*) (*start_of* (*layout_of ap*) *k* − *Suc 0*))
⟹ *b* ≤ *Suc* (*start_of* (*layout_of ap*) *k* + 2 ∗ *n*)
⟨*proof*⟩

lemma *findnth_state_all_le1*:
⟦*k* < *length ap*; *ap* ! *k* = *Inc n*;
(*a*, *b*) ∈
set (*shift* (*findnth n*) (*start_of* (*layout_of ap*) *k* − *Suc 0*))⟧
⟹ *b* ≤ *start_of* (*layout_of ap*) (*length ap*)
⟨*proof*⟩

lemma *start_of_eq*: *length ap* < *as* ⟹ *start_of* (*layout_of ap*) *as* = *start_of* (*layout_of ap*)
(*length ap*)
⟨*proof*⟩

lemma *start_of_all_le*: *start_of* (*layout_of ap*) *as* ≤ *start_of* (*layout_of ap*) (*length ap*)
⟨*proof*⟩

lemma *findnth_state_all_le2*:
⟦*k* < *length ap*;
ap ! *k* = *Dec n e*;
(*a*, *b*) ∈ *set* (*shift* (*findnth n*) (*start_of* (*layout_of ap*) *k* − *Suc 0*))⟧
⟹ *b* ≤ *start_of* (*layout_of ap*) (*length ap*)
⟨*proof*⟩

lemma *dec_state_all_le*[*simp*]:

$[\![k < \mathit{length}\ \mathit{ap};\ \mathit{ap}\ !\ k = \mathit{Dec}\ n\ e;$
$(a, b) \in \mathit{set}\ (\mathit{shift}\ (\mathit{shift}\ \mathit{tdec\_b}\ (2 * n))$
$(\mathit{start\_of}\ (\mathit{layout\_of}\ \mathit{ap})\ k - \mathit{Suc}\ 0))]\!]$
$\Longrightarrow b \leq \mathit{start\_of}\ (\mathit{layout\_of}\ \mathit{ap})\ (\mathit{length}\ \mathit{ap})$
⟨*proof*⟩

lemma *tms_any_less*:
$[\![k < \mathit{length}\ \mathit{ap};\ (a, b) \in \mathit{set}\ (\mathit{tms\_of}\ \mathit{ap}\ !\ k)]\!] \Longrightarrow$
$b \leq \mathit{start\_of}\ (\mathit{layout\_of}\ \mathit{ap})\ (\mathit{length}\ \mathit{ap})$
⟨*proof*⟩

lemma *concat_in*: $i < \mathit{length}\ (\mathit{concat}\ \mathit{xs}) \Longrightarrow$
$\exists k < \mathit{length}\ \mathit{xs}.\ \mathit{concat}\ \mathit{xs}\ !\ i \in \mathit{set}\ (\mathit{xs}\ !\ k)$
⟨*proof*⟩

declare *length_concat*[*simp*]

lemma *in_tms*: $i < \mathit{length}\ (\mathit{tm\_of}\ \mathit{ap}) \Longrightarrow \exists\ k < \mathit{length}\ \mathit{ap}.\ (\mathit{tm\_of}\ \mathit{ap}\ !\ i) \in \mathit{set}\ (\mathit{tms\_of}\ \mathit{ap}\ !\ k)$
⟨*proof*⟩

lemma *all_le_start_of*: *list_all* $(\lambda(\mathit{acn}, s).$
$s \leq \mathit{start\_of}\ (\mathit{layout\_of}\ \mathit{ap})\ (\mathit{length}\ \mathit{ap}))\ (\mathit{tm\_of}\ \mathit{ap})$
⟨*proof*⟩

lemma *length_ci*:
$[\![k < \mathit{length}\ \mathit{ap};\ \mathit{length}\ (\mathit{ci}\ \mathit{ly}\ y\ (\mathit{ap}\ !\ k)) = 2 * \mathit{qa}]\!]$
$\Longrightarrow \mathit{layout\_of}\ \mathit{ap}\ !\ k = \mathit{qa}$
⟨*proof*⟩

lemma *ci_even*[*intro*]: $\mathit{length}\ (\mathit{ci}\ \mathit{ly}\ y\ i)\ \mathit{mod}\ 2 = 0$
⟨*proof*⟩

lemma *sum_list_ci_even*[*intro*]: $\mathit{sum\_list}\ (\mathit{map}\ (\mathit{length} \circ (\lambda(x, y).\ \mathit{ci}\ \mathit{ly}\ x\ y))\ \mathit{zs})\ \mathit{mod}\ 2 = 0$
⟨*proof*⟩

lemma *zip_pre*:
$(\mathit{length}\ \mathit{ys}) \leq \mathit{length}\ \mathit{ap} \Longrightarrow$
$\mathit{zip}\ \mathit{ys}\ \mathit{ap} = \mathit{zip}\ \mathit{ys}\ (\mathit{take}\ (\mathit{length}\ \mathit{ys})\ (\mathit{ap}::'a\ \mathit{list}))$
⟨*proof*⟩

lemma *length_start_of_tm*: $\mathit{start\_of}\ (\mathit{layout\_of}\ \mathit{ap})\ (\mathit{length}\ \mathit{ap}) = \mathit{Suc}\ (\mathit{length}\ (\mathit{tm\_of}\ \mathit{ap})\ \ \mathit{div}\ 2)$
⟨*proof*⟩

lemma *list_all_add_6E*[*elim*]: *list_all* $(\lambda(\mathit{acn}, s).\ s \leq \mathit{Suc}\ q)\ \mathit{xs}$
$\Longrightarrow \mathit{list\_all}\ (\lambda(\mathit{acn}, s).\ s \leq q + (2 * n + 6))\ \mathit{xs}$
⟨*proof*⟩

lemma *mopup_b_12*[*simp*]: $\mathit{length}\ \mathit{mopup\_b} = 12$
⟨*proof*⟩

lemma *mp_up_all_le*: *list_all* $(\lambda(acn, s).\ s \leq q + (2 * n + 6))$
$[(R, Suc\ (Suc\ (2 * n + q))), (R, Suc\ (2 * n + q)),$
$(L, 5 + 2 * n + q), (W0, Suc\ (Suc\ (Suc\ (2 * n + q)))), (R, 4 + 2 * n + q),$
$(W0, Suc\ (Suc\ (Suc\ (2 * n + q)))), (R, Suc\ (Suc\ (2 * n + q))),$
$(W0, Suc\ (Suc\ (Suc\ (2 * n + q)))), (L, 5 + 2 * n + q),$
$(L, 6 + 2 * n + q), (R, 0),\ (L, 6 + 2 * n + q)]$
⟨*proof*⟩

lemma *mopup_le6*[*simp*]: $(a, b) \in set\ (mopup\_a\ n) \Longrightarrow b \leq 2 * n + 6$
⟨*proof*⟩

lemma *shift_le2*[*simp*]: $(a, b) \in set\ (shift\ (mopup\ n)\ x)$
$\Longrightarrow b \leq (2 * x + length\ (mopup\ n))\ div\ 2$
⟨*proof*⟩

lemma *mopup_ge2*[*intro*]: $2 \leq x + length\ (mopup\ n)$
⟨*proof*⟩

lemma *mopup_even*[*intro*]: $(2 * x + length\ (mopup\ n))\ mod\ 2 = 0$
⟨*proof*⟩

lemma *mopup_div_2*[*simp*]: $b \leq Suc\ x$
$\Longrightarrow b \leq (2 * x + length\ (mopup\ n))\ div\ 2$
⟨*proof*⟩

lemma *wf_tm_from_abacus*: **assumes** $tp = tm\_of\ ap$
shows $tm\_wf0\ (tp\ @\ shift\ (mopup\ n)\ (length\ tp\ div\ 2))$
⟨*proof*⟩

lemma *wf_tm_from_recf*:
assumes *compile*: $tp = tm\_of\_rec\ recf$
shows $tm\_wf0\ tp$
⟨*proof*⟩

end

theory *Recs*
imports *Main*
HOL−Library.Nat_Bijection
HOL−Library.Discrete
begin

A more streamlined and cleaned-up version of Recursive Functions following
A Course in Formal Languages, Automata and Groups I. M. Chiswell
and
Lecture on Undecidability Michael M. Wolf

declare *One_nat_def*[*simp del*]

lemma *if_zero_one* [*simp*]:
$(\mathit{if}\ P\ \mathit{then}\ 1\ \mathit{else}\ 0) = (0::\mathit{nat}) \longleftrightarrow \neg P$
$(0::\mathit{nat}) < (\mathit{if}\ P\ \mathit{then}\ 1\ \mathit{else}\ 0) = P$
$(\mathit{if}\ P\ \mathit{then}\ 0\ \mathit{else}\ 1) = (\mathit{if}\ \neg P\ \mathit{then}\ 1\ \mathit{else}\ (0::\mathit{nat}))$
⟨*proof*⟩

lemma *nth*:
$(x \# xs) ! 0 = x$
$(x \# y \# xs) ! 1 = y$
$(x \# y \# z \# xs) ! 2 = z$
$(x \# y \# z \# u \# xs) ! 3 = u$
⟨*proof*⟩

11 Some auxiliary lemmas about $\sum$ and $\prod$

lemma *setprod_atMost_Suc*[*simp*]:
$(\prod i \leq \mathit{Suc}\ n. f\ i) = (\prod i \leq n. f\ i) * f(\mathit{Suc}\ n)$
⟨*proof*⟩

lemma *setprod_lessThan_Suc*[*simp*]:
$(\prod i < \mathit{Suc}\ n. f\ i) = (\prod i < n. f\ i) * f\ n$
⟨*proof*⟩

lemma *setsum_add_nat_ivl2*: $n \leq p \implies$
$\mathit{sum}\ f\ \{..<n\} + \mathit{sum}\ f\ \{n..p\} = \mathit{sum}\ f\ \{..p::\mathit{nat}\}$
⟨*proof*⟩

lemma *setsum_eq_zero* [*simp*]:
fixes $f::\mathit{nat} \Rightarrow \mathit{nat}$
shows $(\sum i < n. f\ i) = 0 \longleftrightarrow (\forall i < n. f\ i = 0)$
$(\sum i \leq n. f\ i) = 0 \longleftrightarrow (\forall i \leq n. f\ i = 0)$
⟨*proof*⟩

lemma *setprod_eq_zero* [*simp*]:
fixes $f::\mathit{nat} \Rightarrow \mathit{nat}$
shows $(\prod i < n. f\ i) = 0 \longleftrightarrow (\exists i < n. f\ i = 0)$
$(\prod i \leq n. f\ i) = 0 \longleftrightarrow (\exists i \leq n. f\ i = 0)$
⟨*proof*⟩

lemma *setsum_one_less*:
fixes $n::\mathit{nat}$
assumes $\forall i < n. f\ i \leq 1$
shows $(\sum i < n. f\ i) \leq n$
⟨*proof*⟩

lemma *setsum_one_le*:
fixes $n::\mathit{nat}$
assumes $\forall i \leq n. f\ i \leq 1$
shows $(\sum i \leq n. f\ i) \leq \mathit{Suc}\ n$

⟨*proof*⟩

lemma *setsum_eq_one_le*:
fixes *n*::*nat*
assumes $\forall\, i \leq n.\, f\, i = 1$
shows $(\sum i \leq n.\, f\, i) = Suc\ n$
⟨*proof*⟩

lemma *setsum_least_eq*:
fixes *f*::*nat* ⇒ *nat*
assumes *h0*: $p \leq n$
assumes *h1*: $\forall\, i \in \{..<p\}.\, f\, i = 1$
assumes *h2*: $\forall\, i \in \{p..n\}.\, f\, i = 0$
shows $(\sum i \leq n.\, f\, i) = p$
⟨*proof*⟩

lemma *nat_mult_le_one*:
fixes *m n*::*nat*
assumes $m \leq 1\ n \leq 1$
shows $m * n \leq 1$
⟨*proof*⟩

lemma *setprod_one_le*:
fixes *f*::*nat* ⇒ *nat*
assumes $\forall\, i \leq n.\, f\, i \leq 1$
shows $(\prod i \leq n.\, f\, i) \leq 1$
⟨*proof*⟩

lemma *setprod_greater_zero*:
fixes *f*::*nat* ⇒ *nat*
assumes $\forall\, i \leq n.\, f\, i \geq 0$
shows $(\prod i \leq n.\, f\, i) \geq 0$
⟨*proof*⟩

lemma *setprod_eq_one*:
fixes *f*::*nat* ⇒ *nat*
assumes $\forall\, i \leq n.\, f\, i = Suc\ 0$
shows $(\prod i \leq n.\, f\, i) = Suc\ 0$
⟨*proof*⟩

lemma *setsum_cut_off_less*:
fixes *f*::*nat* ⇒ *nat*
assumes *h1*: $m \leq n$
and *h2*: $\forall\, i \in \{m..<n\}.\, f\, i = 0$
shows $(\sum i < n.\, f\, i) = (\sum i < m.\, f\, i)$
⟨*proof*⟩

lemma *setsum_cut_off_le*:
fixes *f*::*nat* ⇒ *nat*
assumes *h1*: $m \leq n$

and $h2$: $\forall i \in \{m..n\}.\, f\, i = 0$
shows $(\sum i \leq n.\, f\, i) = (\sum i < m.\, f\, i)$
⟨*proof*⟩

lemma *setprod_one* [*simp*]:
fixes n::*nat*
shows $(\prod i < n.\ Suc\ 0) = Suc\ 0$
$(\prod i \leq n.\ Suc\ 0) = Suc\ 0$
⟨*proof*⟩

12 Recursive Functions

datatype *recf* = *Z*
| *S*
| *Id nat nat*
| *Cn nat recf recf list*
| *Pr nat recf recf*
| *Mn nat recf*

fun *arity* :: *recf* ⇒ *nat*
where
arity Z = *1*
| *arity S* = *1*
| *arity* (*Id m n*) = *m*
| *arity* (*Cn n f gs*) = *n*
| *arity* (*Pr n f g*) = *Suc n*
| *arity* (*Mn n f*) = *n*

Abbreviations for calculating the arity of the constructors

abbreviation
CN f gs $\stackrel{def}{=}$ *Cn* (*arity* (*hd gs*)) *f gs*

abbreviation
PR f g $\stackrel{def}{=}$ *Pr* (*arity f*) *f g*

abbreviation
MN f $\stackrel{def}{=}$ *Mn* (*arity f* − *1*) *f*

the evaluation function and termination relation

fun *rec_eval* :: *recf* ⇒ *nat list* ⇒ *nat*
where
rec_eval Z xs = *0*
| *rec_eval S xs* = *Suc* (*xs* ! *0*)
| *rec_eval* (*Id m n*) *xs* = *xs* ! *n*
| *rec_eval* (*Cn n f gs*) *xs* = *rec_eval f* (*map* (λ*x*. *rec_eval x xs*) *gs*)
| *rec_eval* (*Pr n f g*) (*0* # *xs*) = *rec_eval f xs*
| *rec_eval* (*Pr n f g*) (*Suc x* # *xs*) =
rec_eval g (*x* # (*rec_eval* (*Pr n f g*) (*x* # *xs*)) # *xs*)

| *rec_eval* (*Mn n f*) *xs* = (*LEAST x*. *rec_eval f* (*x* # *xs*) = *0*)

inductive
terminates :: *recf* ⇒ *nat list* ⇒ *bool*
where
termi_z: *terminates Z* [*n*]
| *termi_s*: *terminates S* [*n*]
| *termi_id*: ⟦*n* < *m*; *length xs* = *m*⟧ ⟹ *terminates* (*Id m n*) *xs*
| *termi_cn*: ⟦*terminates f* (*map* (λ*g*. *rec_eval g xs*) *gs*);
∀ *g* ∈ *set gs*. *terminates g xs*; *length xs* = *n*⟧ ⟹ *terminates* (*Cn n f gs*) *xs*
| *termi_pr*: ⟦∀ *y* < *x*. *terminates g* (*y* # (*rec_eval* (*Pr n f g*) (*y* # *xs*) # *xs*));
terminates f xs;
length xs = *n*⟧
⟹ *terminates* (*Pr n f g*) (*x* # *xs*)
| *termi_mn*: ⟦*length xs* = *n*; *terminates f* (*r* # *xs*);
rec_eval f (*r* # *xs*) = *0*;
∀ *i* < *r*. *terminates f* (*i* # *xs*) ∧ *rec_eval f* (*i* # *xs*) > *0*⟧ ⟹ *terminates* (*Mn n f*) *xs*

13 Arithmetic Functions

constn n is the recursive function which computes natural number *n*.

fun *constn* :: *nat* ⇒ *recf*
where
constn 0 = *Z* |
constn (*Suc n*) = *CN S* [*constn n*]

definition
rec_swap f = *CN f* [*Id 2 1*, *Id 2 0*]

definition
rec_add = *PR* (*Id 1 0*) (*CN S* [*Id 3 1*])

definition
rec_mult = *PR Z* (*CN rec_add* [*Id 3 1*, *Id 3 2*])

definition
rec_power = *rec_swap* (*PR* (*constn 1*) (*CN rec_mult* [*Id 3 1*, *Id 3 2*]))

definition
rec_fact_aux = *PR* (*constn 1*) (*CN rec_mult* [*CN S* [*Id 3 0*], *Id 3 1*])

definition
rec_fact = *CN rec_fact_aux* [*Id 1 0*, *Id 1 0*]

definition
rec_predecessor = *CN* (*PR Z* (*Id 3 0*)) [*Id 1 0*, *Id 1 0*]

definition

rec_minus = *rec_swap* (*PR* (*Id 1 0*) (*CN rec_predecessor* [*Id 3 1*]))

lemma *constn_lemma* [*simp*]:
rec_eval (*constn n*) *xs* = *n*
⟨*proof*⟩

lemma *swap_lemma* [*simp*]:
rec_eval (*rec_swap f*) [*x*, *y*] = *rec_eval f* [*y*, *x*]
⟨*proof*⟩

lemma *add_lemma* [*simp*]:
rec_eval rec_add [*x*, *y*] = *x* + *y*
⟨*proof*⟩

lemma *mult_lemma* [*simp*]:
rec_eval rec_mult [*x*, *y*] = *x* ∗ *y*
⟨*proof*⟩

lemma *power_lemma* [*simp*]:
rec_eval rec_power [*x*, *y*] = *x* ^ *y*
⟨*proof*⟩

lemma *fact_aux_lemma* [*simp*]:
rec_eval rec_fact_aux [*x*, *y*] = *fact x*
⟨*proof*⟩

lemma *fact_lemma* [*simp*]:
rec_eval rec_fact [*x*] = *fact x*
⟨*proof*⟩

lemma *pred_lemma* [*simp*]:
rec_eval rec_predecessor [*x*] = *x* − *1*
⟨*proof*⟩

lemma *minus_lemma* [*simp*]:
rec_eval rec_minus [*x*, *y*] = *x* − *y*
⟨*proof*⟩

14 Logical functions

The *sign* function returns 1 when the input argument is greater than *0*.

definition
rec_sign = *CN rec_minus* [*constn 1*, *CN rec_minus* [*constn 1*, *Id 1 0*]]

definition
rec_not = *CN rec_minus* [*constn 1*, *Id 1 0*]

rec_eq compares two arguments: returns *1* if they are equal; *0* otherwise.

definition

rec_eq = *CN rec_minus* [*CN* (*constn 1*) [*Id 2 0*], *CN rec_add* [*rec_minus*, *rec_swap rec_minus*]]

definition
rec_noteq = *CN rec_not* [*rec_eq*]

definition
rec_conj = *CN rec_sign* [*rec_mult*]

definition
rec_disj = *CN rec_sign* [*rec_add*]

definition
rec_imp = *CN rec_disj* [*CN rec_not* [*Id 2 0*], *Id 2 1*]

rec_ifz [*z*, *x*, *y*] returns x if z is zero, y otherwise; *rec_if* [*z*, *x*, *y*] returns x if z is *not* zero, y otherwise

definition
rec_ifz = *PR* (*Id 2 0*) (*Id 4 3*)

definition
rec_if = *CN rec_ifz* [*CN rec_not* [*Id 3 0*], *Id 3 1*, *Id 3 2*]

lemma *sign_lemma* [*simp*]:
rec_eval rec_sign [*x*] = (*if x* = *0 then 0 else 1*)
⟨*proof*⟩

lemma *not_lemma* [*simp*]:
rec_eval rec_not [*x*] = (*if x* = *0 then 1 else 0*)
⟨*proof*⟩

lemma *eq_lemma* [*simp*]:
rec_eval rec_eq [*x*, *y*] = (*if x* = *y then 1 else 0*)
⟨*proof*⟩

lemma *noteq_lemma* [*simp*]:
rec_eval rec_noteq [*x*, *y*] = (*if x* ≠ *y then 1 else 0*)
⟨*proof*⟩

lemma *conj_lemma* [*simp*]:
rec_eval rec_conj [*x*, *y*] = (*if x* = *0* ∨ *y* = *0 then 0 else 1*)
⟨*proof*⟩

lemma *disj_lemma* [*simp*]:
rec_eval rec_disj [*x*, *y*] = (*if x* = *0* ∧ *y* = *0 then 0 else 1*)
⟨*proof*⟩

lemma *imp_lemma* [*simp*]:
rec_eval rec_imp [*x*, *y*] = (*if* 0 < *x* ∧ *y* = *0 then 0 else 1*)
⟨*proof*⟩

lemma *ifz_lemma* [*simp*]:
rec_eval rec_ifz [*z*, *x*, *y*] = (*if z = 0 then x else y*)
⟨*proof*⟩

lemma *if_lemma* [*simp*]:
rec_eval rec_if [*z*, *x*, *y*] = (*if 0 < z then x else y*)
⟨*proof*⟩

15 Less and Le Relations

rec_less compares two arguments and returns *1* if the first is less than the second; otherwise returns *0*.

definition
rec_less = *CN rec_sign* [*rec_swap rec_minus*]

definition
rec_le = *CN rec_disj* [*rec_less*, *rec_eq*]

lemma *less_lemma* [*simp*]:
rec_eval rec_less [*x*, *y*] = (*if x < y then 1 else 0*)
⟨*proof*⟩

lemma *le_lemma* [*simp*]:
rec_eval rec_le [*x*, *y*] = (*if* (*x* ≤ *y*) *then 1 else 0*)
⟨*proof*⟩

16 Summation and Product Functions

definition
rec_sigma1 f = *PR* (*CN f* [*CN Z* [*Id 1 0*], *Id 1 0*])
(*CN rec_add* [*Id 3 1*, *CN f* [*CN S* [*Id 3 0*], *Id 3 2*]])

definition
rec_sigma2 f = *PR* (*CN f* [*CN Z* [*Id 2 0*], *Id 2 0*, *Id 2 1*])
(*CN rec_add* [*Id 4 1*, *CN f* [*CN S* [*Id 4 0*], *Id 4 2*, *Id 4 3*]])

definition
rec_accum1 f = *PR* (*CN f* [*CN Z* [*Id 1 0*], *Id 1 0*])
(*CN rec_mult* [*Id 3 1*, *CN f* [*CN S* [*Id 3 0*], *Id 3 2*]])

definition
rec_accum2 f = *PR* (*CN f* [*CN Z* [*Id 2 0*], *Id 2 0*, *Id 2 1*])
(*CN rec_mult* [*Id 4 1*, *CN f* [*CN S* [*Id 4 0*], *Id 4 2*, *Id 4 3*]])

definition
rec_accum3 f = *PR* (*CN f* [*CN Z* [*Id 3 0*], *Id 3 0*, *Id 3 1*, *Id 3 2*])
(*CN rec_mult* [*Id 5 1*, *CN f* [*CN S* [*Id 5 0*], *Id 5 2*, *Id 5 3*, *Id 5 4*]])

lemma *sigma1_lemma* [*simp*]:
shows *rec_eval* (*rec_sigma1 f*) [*x*, *y*] = ($\sum$ *z* ≤ *x*. *rec_eval f* [*z*, *y*])
⟨*proof*⟩

lemma *sigma2_lemma* [*simp*]:
shows *rec_eval* (*rec_sigma2 f*) [*x*, *y1*, *y2*] = ($\sum$ *z* ≤ *x*. *rec_eval f* [*z*, *y1*, *y2*])
⟨*proof*⟩

lemma *accum1_lemma* [*simp*]:
shows *rec_eval* (*rec_accum1 f*) [*x*, *y*] = ($\prod$ *z* ≤ *x*. *rec_eval f* [*z*, *y*])
⟨*proof*⟩

lemma *accum2_lemma* [*simp*]:
shows *rec_eval* (*rec_accum2 f*) [*x*, *y1*, *y2*] = ($\prod$ *z* ≤ *x*. *rec_eval f* [*z*, *y1*, *y2*])
⟨*proof*⟩

lemma *accum3_lemma* [*simp*]:
shows *rec_eval* (*rec_accum3 f*) [*x*, *y1*, *y2*, *y3*] = ($\prod$ *z* ≤ *x*. (*rec_eval f*) [*z*, *y1*, *y2*, *y3*])
⟨*proof*⟩

17 Bounded Quantifiers

definition
rec_all1 f = *CN rec_sign* [*rec_accum1 f*]

definition
rec_all2 f = *CN rec_sign* [*rec_accum2 f*]

definition
rec_all3 f = *CN rec_sign* [*rec_accum3 f*]

definition
rec_all1_less f = (*let cond1* = *CN rec_eq* [*Id 3 0*, *Id 3 1*] *in*
let cond2 = *CN f* [*Id 3 0*, *Id 3 2*]
in CN (*rec_all2* (*CN rec_disj* [*cond1*, *cond2*])) [*Id 2 0*, *Id 2 0*, *Id 2 1*])

definition
rec_all2_less f = (*let cond1* = *CN rec_eq* [*Id 4 0*, *Id 4 1*] *in*
let cond2 = *CN f* [*Id 4 0*, *Id 4 2*, *Id 4 3*] *in*
CN (*rec_all3* (*CN rec_disj* [*cond1*, *cond2*])) [*Id 3 0*, *Id 3 0*, *Id 3 1*, *Id 3 2*])

definition
rec_ex1 f = *CN rec_sign* [*rec_sigma1 f*]

definition
rec_ex2 f = *CN rec_sign* [*rec_sigma2 f*]

lemma *ex1_lemma* [*simp*]:
rec_eval (*rec_ex1 f*) [*x*, *y*] = (*if* (∃ *z* ≤ *x*. 0 < *rec_eval f* [*z*, *y*]) *then 1 else 0*)
⟨*proof*⟩

lemma *ex2_lemma* [*simp*]:
rec_eval (*rec_ex2 f*) [*x*, *y1*, *y2*] = (*if* (∃ *z* ≤ *x*. 0 < *rec_eval f* [*z*, *y1*, *y2*]) *then 1 else 0*)
⟨*proof*⟩

lemma *all1_lemma* [*simp*]:
rec_eval (*rec_all1 f*) [*x*, *y*] = (*if* (∀ *z* ≤ *x*. 0 < *rec_eval f* [*z*, *y*]) *then 1 else 0*)
⟨*proof*⟩

lemma *all2_lemma* [*simp*]:
rec_eval (*rec_all2 f*) [*x*, *y1*, *y2*] = (*if* (∀ *z* ≤ *x*. 0 < *rec_eval f* [*z*, *y1*, *y2*]) *then 1 else 0*)
⟨*proof*⟩

lemma *all3_lemma* [*simp*]:
rec_eval (*rec_all3 f*) [*x*, *y1*, *y2*, *y3*] = (*if* (∀ *z* ≤ *x*. 0 < *rec_eval f* [*z*, *y1*, *y2*, *y3*]) *then 1 else 0*)
⟨*proof*⟩

lemma *all1_less_lemma* [*simp*]:
rec_eval (*rec_all1_less f*) [*x*, *y*] = (*if* (∀ *z* < *x*. 0 < *rec_eval f* [*z*, *y*]) *then 1 else 0*)
⟨*proof*⟩

lemma *all2_less_lemma* [*simp*]:
rec_eval (*rec_all2_less f*) [*x*, *y1*, *y2*] = (*if* (∀ *z* < *x*. 0 < *rec_eval f* [*z*, *y1*, *y2*]) *then 1 else 0*)
⟨*proof*⟩

18 Quotients

definition
rec_quo = (*let lhs* = *CN S* [*Id 3 0*] *in*
let rhs = *CN rec_mult* [*Id 3 2*, *CN S* [*Id 3 1*]] *in*
let cond = *CN rec_eq* [*lhs*, *rhs*] *in*
let if_stmt = *CN rec_if* [*cond*, *CN S* [*Id 3 1*], *Id 3 1*]
in PR Z if_stmt)

fun *Quo* **where**
Quo x 0 = *0*
| *Quo x* (*Suc y*) = (*if* (*Suc y* = *x* ∗ (*Suc* (*Quo x y*))) *then Suc* (*Quo x y*) *else Quo x y*)

lemma *Quo0*:
shows *Quo 0 y* = *0*
⟨*proof*⟩

lemma *Quo1*:
x ∗ (*Quo x y*) ≤ *y*
⟨*proof*⟩

lemma *Quo2*:
$b * (Quo\ b\ a) + a\ mod\ b = a$
⟨*proof*⟩

lemma *Quo3*:
$n * (Quo\ n\ m) = m - m\ mod\ n$
⟨*proof*⟩

lemma *Quo4*:
assumes *h*: $0 < x$
shows $y < x + x * Quo\ x\ y$
⟨*proof*⟩

lemma *Quo_div*:
shows $Quo\ x\ y = y\ div\ x$
⟨*proof*⟩

lemma *Quo_rec_quo*:
shows $rec\_eval\ rec\_quo\ [y, x] = Quo\ x\ y$
⟨*proof*⟩

lemma *quo_lemma* [*simp*]:
shows $rec\_eval\ rec\_quo\ [y, x] = y\ div\ x$
⟨*proof*⟩

19 Iteration

definition
$rec\_iter\ f = PR\ (Id\ 1\ 0)\ (CN\ f\ [Id\ 3\ 1])$

fun *Iter* **where**
$Iter\ f\ 0 = id$
| $Iter\ f\ (Suc\ n) = f \circ (Iter\ f\ n)$

lemma *Iter_comm*:
$(Iter\ f\ n)\ (f\ x) = f\ ((Iter\ f\ n)\ x)$
⟨*proof*⟩

lemma *iter_lemma* [*simp*]:
$rec\_eval\ (rec\_iter\ f)\ [n, x] = Iter\ (\lambda x.\ rec\_eval\ f\ [x])\ n\ x$
⟨*proof*⟩

20 Bounded Maximisation

fun *BMax_rec* **where**
$BMax\_rec\ R\ 0 = 0$
| $BMax\_rec\ R\ (Suc\ n) = (if\ R\ (Suc\ n)\ then\ (Suc\ n)\ else\ BMax\_rec\ R\ n)$

definition
BMax_set :: $(nat \Rightarrow bool) \Rightarrow nat \Rightarrow nat$
where
$BMax\_set\ R\ x = Max\ (\{z.\ z \leq x \wedge R\ z\} \cup \{0\})$

lemma *BMax_rec_eq1*:
$BMax\_rec\ R\ x = (GREATEST\ z.\ (R\ z \wedge z \leq x) \vee z = 0)$
⟨*proof*⟩

lemma *BMax_rec_eq2*:
$BMax\_rec\ R\ x = Max\ (\{z.\ z \leq x \wedge R\ z\} \cup \{0\})$
⟨*proof*⟩

lemma *BMax_rec_eq3*:
$BMax\_rec\ R\ x = Max\ (Set.filter\ (\lambda z.\ R\ z)\ \{..x\} \cup \{0\})$
⟨*proof*⟩

definition
$rec\_max1\ f = PR\ Z\ (CN\ rec\_ifz\ [CN\ f\ [CN\ S\ [Id\ 3\ 0], Id\ 3\ 2], CN\ S\ [Id\ 3\ 0], Id\ 3\ 1])$

lemma *max1_lemma* [*simp*]:
$rec\_eval\ (rec\_max1\ f)\ [x, y] = BMax\_rec\ (\lambda u.\ rec\_eval\ f\ [u, y] = 0)\ x$
⟨*proof*⟩

definition
$rec\_max2\ f = PR\ Z\ (CN\ rec\_ifz\ [CN\ f\ [CN\ S\ [Id\ 4\ 0], Id\ 4\ 2, Id\ 4\ 3], CN\ S\ [Id\ 4\ 0], Id\ 4\ 1])$

lemma *max2_lemma* [*simp*]:
$rec\_eval\ (rec\_max2\ f)\ [x, y1, y2] = BMax\_rec\ (\lambda u.\ rec\_eval\ f\ [u, y1, y2] = 0)\ x$
⟨*proof*⟩

21 Encodings using Cantor's pairing function

We use Cantor's pairing function from Nat-Bijection. However, we need to prove that the formulation of the decoding function there is recursive. For this we first prove that we can extract the maximal triangle number using *prod_decode*.

abbreviation *Max_triangle_aux* **where**
$Max\_triangle\_aux\ k\ z \stackrel{def}{=} fst\ (prod\_decode\_aux\ k\ z) + snd\ (prod\_decode\_aux\ k\ z)$

abbreviation *Max_triangle* **where**
$Max\_triangle\ z \stackrel{def}{=} Max\_triangle\_aux\ 0\ z$

abbreviation
$pdec1\ z \stackrel{def}{=} fst\ (prod\_decode\ z)$

abbreviation

pdec2 z $\stackrel{def}{=}$ *snd* (*prod_decode z*)

abbreviation
penc m n $\stackrel{def}{=}$ *prod_encode* (*m*, *n*)

lemma *fst_prod_decode*:
pdec1 z = *z* − *triangle* (*Max_triangle z*)
⟨*proof*⟩

lemma *snd_prod_decode*:
pdec2 z = *Max_triangle z* − *pdec1 z*
⟨*proof*⟩

lemma *le_triangle*:
m ≤ *triangle* (*n* + *m*)
⟨*proof*⟩

lemma *Max_triangle_triangle_le*:
triangle (*Max_triangle z*) ≤ *z*
⟨*proof*⟩

lemma *Max_triangle_le*:
Max_triangle z ≤ *z*
⟨*proof*⟩

lemma *w_aux*:
Max_triangle (*triangle k* + *m*) = *Max_triangle_aux k m*
⟨*proof*⟩

lemma *y_aux*: *y* ≤ *Max_triangle_aux y k*
⟨*proof*⟩

lemma *Max_triangle_greatest*:
Max_triangle z = (*GREATEST k*. (*triangle k* ≤ *z* ∧ *k* ≤ *z*) ∨ *k* = *0*)
⟨*proof*⟩

definition
rec_triangle = *CN rec_quo* [*CN rec_mult* [*Id 1 0*, *S*], *constn 2*]

definition
rec_max_triangle =
(*let cond* = *CN rec_not* [*CN rec_le* [*CN rec_triangle* [*Id 2 0*], *Id 2 1*]] *in*
CN (*rec_max1 cond*) [*Id 1 0*, *Id 1 0*])

lemma *triangle_lemma* [*simp*]:
rec_eval rec_triangle [*x*] = *triangle x*
⟨*proof*⟩

lemma *max_triangle_lemma* [*simp*]:
rec_eval rec_max_triangle [*x*] = *Max_triangle x*
⟨*proof*⟩

Encodings for Products

definition
rec_penc = *CN rec_add* [*CN rec_triangle* [*CN rec_add* [*Id 2 0*, *Id 2 1*]], *Id 2 0*]

definition
rec_pdec1 = *CN rec_minus* [*Id 1 0*, *CN rec_triangle* [*CN rec_max_triangle* [*Id 1 0*]]]

definition
rec_pdec2 = *CN rec_minus* [*CN rec_max_triangle* [*Id 1 0*], *CN rec_pdec1* [*Id 1 0*]]

lemma *pdec1_lemma* [*simp*]:
rec_eval rec_pdec1 [*z*] = *pdec1 z*
⟨*proof*⟩

lemma *pdec2_lemma* [*simp*]:
rec_eval rec_pdec2 [*z*] = *pdec2 z*
⟨*proof*⟩

lemma *penc_lemma* [*simp*]:
rec_eval rec_penc [*m*, *n*] = *penc m n*
⟨*proof*⟩

Encodings of Lists

fun
lenc :: *nat list* ⇒ *nat*
where
lenc [] = *0*
| *lenc* (*x* # *xs*) = *penc* (*Suc x*) (*lenc xs*)

fun
ldec :: *nat* ⇒ *nat* ⇒ *nat*
where
ldec z 0 = (*pdec1 z*) − *1*
| *ldec z* (*Suc n*) = *ldec* (*pdec2 z*) *n*

lemma *pdec_zero_simps* [*simp*]:
pdec1 0 = *0*
pdec2 0 = *0*
⟨*proof*⟩

lemma *ldec_zero*:
ldec 0 n = *0*
⟨*proof*⟩

lemma *list_encode_inverse*:

ldec (*lenc xs*) *n* = (*if n* < *length xs then xs* ! *n else 0*)
⟨*proof*⟩

lemma *lenc_length_le*:
length xs ≤ *lenc xs*
⟨*proof*⟩

Membership for the List Encoding

fun *inside* :: *nat* ⇒ *nat* ⇒ *bool* **where**
inside z 0 = (*0* < *z*)
| *inside z* (*Suc n*) = *inside* (*pdec2 z*) *n*

definition *enclen* :: *nat* ⇒ *nat* **where**
enclen z = *BMax_rec* (λ*x*. *inside z* (*x* − *1*)) *z*

lemma *inside_False* [*simp*]:
inside 0 n = *False*
⟨*proof*⟩

lemma *inside_length* [*simp*]:
inside (*lenc xs*) *s* = (*s* < *length xs*)
⟨*proof*⟩

Length of Encoded Lists

lemma *enclen_length* [*simp*]:
enclen (*lenc xs*) = *length xs*
⟨*proof*⟩

lemma *enclen_penc* [*simp*]:
enclen (*penc* (*Suc x*) (*lenc xs*)) = *Suc* (*enclen* (*lenc xs*))
⟨*proof*⟩

lemma *enclen_zero* [*simp*]:
enclen 0 = *0*
⟨*proof*⟩

Recursive Definitions for List Encodings

fun
rec_lenc :: *recf list* ⇒ *recf*
where
rec_lenc [] = *Z*
| *rec_lenc* (*f* # *fs*) = *CN rec_penc* [*CN S* [*f*], *rec_lenc fs*]

definition
rec_ldec = *CN rec_predecessor* [*CN rec_pdec1* [*rec_swap* (*rec_iter rec_pdec2*)]]

definition
rec_inside = *CN rec_less* [*Z*, *rec_swap* (*rec_iter rec_pdec2*)]

definition

rec_enclen = *CN* (*rec_max1* (*CN rec_not* [*CN rec_inside* [*Id 2 1*, *CN rec_predecessor* [*Id 2 0*]]])) [*Id 1 0*, *Id 1 0*]

lemma *ldec_iter*:
ldec z n = *pdec1* (*Iter pdec2 n z*) − *1*
⟨*proof*⟩

lemma *inside_iter*:
inside z n = (*0* < *Iter pdec2 n z*)
⟨*proof*⟩

lemma *lenc_lemma* [*simp*]:
rec_eval (*rec_lenc fs*) *xs* = *lenc* (*map* (λ*f*. *rec_eval f xs*) *fs*)
⟨*proof*⟩

lemma *ldec_lemma* [*simp*]:
rec_eval rec_ldec [*z*, *n*] = *ldec z n*
⟨*proof*⟩

lemma *inside_lemma* [*simp*]:
rec_eval rec_inside [*z*, *n*] = (*if inside z n then 1 else 0*)
⟨*proof*⟩

lemma *enclen_lemma* [*simp*]:
rec_eval rec_enclen [*z*] = *enclen z*
⟨*proof*⟩

end

22 Construction of a Universal Function

theory *UF*
imports *Rec_Def HOL.GCD Abacus*
begin

This theory file constructs the Universal Function *rec_F*, which is the UTM defined in terms of recursive functions. This *rec_F* is essentially an interpreter of Turing Machines. Once the correctness of *rec_F* is established, UTM can easil be obtained by compling *rec_F* into the corresponding Turing Machine.

23 Universal Function

23.1 The construction of component functions

The recursive function used to do arithmetic addition.

definition *rec_add* :: *recf*
where

rec_add $\stackrel{def}{=}$ *Pr 1* (*id 1 0*) (*Cn 3 s* [*id 3 2*])

The recursive function used to do arithmetic multiplication.

definition *rec_mult* :: *recf*
where
rec_mult = *Pr 1 z* (*Cn 3 rec_add* [*id 3 0*, *id 3 2*])

The recursive function used to do arithmetic precede.

definition *rec_pred* :: *recf*
where
rec_pred = *Cn 1* (*Pr 1 z* (*id 3 1*)) [*id 1 0*, *id 1 0*]

The recursive function used to do arithmetic subtraction.

definition *rec_minus* :: *recf*
where
rec_minus = *Pr 1* (*id 1 0*) (*Cn 3 rec_pred* [*id 3 2*])

constn n is the recursive function which computes nature number *n*.

fun *constn* :: *nat* ⇒ *recf*
where
constn 0 = *z* |
constn (*Suc n*) = *Cn 1 s* [*constn n*]

Sign function, which returns 1 when the input argument is greater than *0*.

definition *rec_sg* :: *recf*
where
rec_sg = *Cn 1 rec_minus* [*constn 1*,
Cn 1 rec_minus [*constn 1*, *id 1 0*]]

rec_less compares its two arguments, returns *1* if the first is less than the second; otherwise returns *0*.

definition *rec_less* :: *recf*
where
rec_less = *Cn 2 rec_sg* [*Cn 2 rec_minus* [*id 2 1*, *id 2 0*]]

rec_not inverse its argument: returns *1* when the argument is *0*; returns *0* otherwise.

definition *rec_not* :: *recf*
where
rec_not = *Cn 1 rec_minus* [*constn 1*, *id 1 0*]

rec_eq compares its two arguments: returns *1* if they are equal; return *0* otherwise.

definition *rec_eq* :: *recf*
where
rec_eq = *Cn 2 rec_minus* [*Cn 2* (*constn 1*) [*id 2 0*],
Cn 2 rec_add [*Cn 2 rec_minus* [*id 2 0*, *id 2 1*],
Cn 2 rec_minus [*id 2 1*, *id 2 0*]]]

rec_conj computes the conjunction of its two arguments, returns *1* if both of them are non-zero; returns *0* otherwise.

definition *rec_conj* :: *recf*
 where
  rec_conj = *Cn 2 rec_sg* [*Cn 2 rec_mult* [*id 2 0*, *id 2 1*]]

rec_disj computes the disjunction of its two arguments, returns *0* if both of them are zero; returns *0* otherwise.

definition *rec_disj* :: *recf*
 where
  rec_disj = *Cn 2 rec_sg* [*Cn 2 rec_add* [*id 2 0*, *id 2 1*]]

Computes the arity of recursive function.

fun *arity* :: *recf* ⇒ *nat*
 where
  arity z = *1*
| *arity s* = *1*
| *arity* (*id m n*) = *m*
| *arity* (*Cn n f gs*) = *n*
| *arity* (*Pr n f g*) = *Suc n*
| *arity* (*Mn n f*) = *n*

get_fstn_args n (*Suc k*) returns [*id n 0*, *id n 1*, *id n 2*, . . . , *id n k*], the effect of which is to take out the first *Suc k* arguments out of the *n* input arguments.

fun *get_fstn_args* :: *nat* ⇒ *nat* ⇒ *recf list*
 where
  get_fstn_args n 0 = []
| *get_fstn_args n* (*Suc y*) = *get_fstn_args n y* @ [*id n y*]

rec_sigma f returns the recursive functions which sums up the results of f:

$$(rec\_sigma f)(x, y) = f(x, 0) + f(x, 1) + \cdots + f(x, y)$$

fun *rec_sigma* :: *recf* ⇒ *recf*
 where
  rec_sigma rf =
   (*let vl* = *arity rf in*
    Pr (*vl* − *1*) (*Cn* (*vl* − *1*) *rf* (*get_fstn_args* (*vl* − *1*) (*vl* − *1*) @
        [*Cn* (*vl* − *1*) (*constn 0*) [*id* (*vl* − *1*) *0*]]))
     (*Cn* (*Suc vl*) *rec_add* [*id* (*Suc vl*) *vl*,
        Cn (*Suc vl*) *rf* (*get_fstn_args* (*Suc vl*) (*vl* − *1*)
         @ [*Cn* (*Suc vl*) *s* [*id* (*Suc vl*) (*vl* − *1*)]])]))

rec_exec is the interpreter function for reursive functions. The function is defined such that it always returns meaningful results for primitive recursive functions.

declare *rec_exec.simps*[*simp del*] *constn.simps*[*simp del*]

Correctness of *rec_add*.

lemma *add_lemma*: ⋀ *x y*. *rec_exec rec_add* [*x*, *y*] = *x* + *y*
⟨*proof*⟩

Correctness of *rec_mult*.

lemma *mult_lemma*: $\bigwedge x\ y.$ *rec_exec rec_mult* $[x, y] = x * y$
⟨*proof*⟩

Correctness of *rec_pred*.

lemma *pred_lemma*: $\bigwedge x.$ *rec_exec rec_pred* $[x] =\ x - 1$
⟨*proof*⟩

Correctness of *rec_minus*.

lemma *minus_lemma*: $\bigwedge x\ y.$ *rec_exec rec_minus* $[x, y] = x - y$
⟨*proof*⟩

Correctness of *rec_sg*.

lemma *sg_lemma*: $\bigwedge x.$ *rec_exec rec_sg* $[x] =$ (*if* $x = 0$ *then 0 else 1*)
⟨*proof*⟩

Correctness of *constn*.

lemma *constn_lemma*: *rec_exec* (*constn n*) $[x] = n$
⟨*proof*⟩

Correctness of *rec_less*.

lemma *less_lemma*: $\bigwedge x\ y.$ *rec_exec rec_less* $[x, y] =$
(*if* $x < y$ *then 1 else 0*)
⟨*proof*⟩

Correctness of *rec_not*.

lemma *not_lemma*:
$\bigwedge x.$ *rec_exec rec_not* $[x] =$ (*if* $x = 0$ *then 1 else 0*)
⟨*proof*⟩

Correctness of *rec_eq*.

lemma *eq_lemma*: $\bigwedge x\ y.$ *rec_exec rec_eq* $[x, y] =$ (*if* $x = y$ *then 1 else 0*)
⟨*proof*⟩

Correctness of *rec_conj*.

lemma *conj_lemma*: $\bigwedge x\ y.$ *rec_exec rec_conj* $[x, y] =$ (*if* $x = 0 \vee y = 0$ *then 0*
else 1)
⟨*proof*⟩

Correctness of *rec_disj*.

lemma *disj_lemma*: $\bigwedge x\ y.$ *rec_exec rec_disj* $[x, y] =$ (*if* $x = 0 \wedge y = 0$ *then 0*
else 1)
⟨*proof*⟩

primrec recf n is true iff *recf* is a primitive recursive function with arity *n*.

inductive *primerec* :: *recf* ⇒ *nat* ⇒ *bool*
where
prime_z[*intro*]: *primerec z* (*Suc 0*) |
prime_s[*intro*]: *primerec s* (*Suc 0*) |

prime_id[*intro*!]: ⟦$n < m$⟧ ⟹ *primerec* (*id m n*) *m* |
prime_cn[*intro*!]: ⟦*primerec f k*; *length gs* = *k*;
∀ *i* < *length gs*. *primerec* (*gs* ! *i*) *m*; *m* = *n*⟧
⟹ *primerec* (*Cn n f gs*) *m* |
prime_pr[*intro*!]: ⟦*primerec f n*;
primerec g (*Suc* (*Suc n*)); *m* = *Suc n*⟧
⟹ *primerec* (*Pr n f g*) *m*

inductive-cases *prime_cn_reverse'*[*elim*]: *primerec* (*Cn n f gs*) *n*
inductive-cases *prime_mn_reverse*: *primerec* (*Mn n f*) *m*
inductive-cases *prime_z_reverse*[*elim*]: *primerec z n*
inductive-cases *prime_s_reverse*[*elim*]: *primerec s n*
inductive-cases *prime_id_reverse*[*elim*]: *primerec* (*id m n*) *k*
inductive-cases *prime_cn_reverse*[*elim*]: *primerec* (*Cn n f gs*) *m*
inductive-cases *prime_pr_reverse*[*elim*]: *primerec* (*Pr n f g*) *m*

declare *mult_lemma*[*simp*] *add_lemma*[*simp*] *pred_lemma*[*simp*]
minus_lemma[*simp*] *sg_lemma*[*simp*] *constn_lemma*[*simp*]
less_lemma[*simp*] *not_lemma*[*simp*] *eq_lemma*[*simp*]
conj_lemma[*simp*] *disj_lemma*[*simp*]

Sigma is the logical specification of the recursive function *rec_sigma*.

function *Sigma* :: (*nat list* ⇒ *nat*) ⇒ *nat list* ⇒ *nat*
where
Sigma g xs = (*if last xs* = *0 then g xs*
else (*Sigma g* (*butlast xs* @ [*last xs* − *1*]) +
g xs))
⟨*proof*⟩
termination
⟨*proof*⟩

declare *rec_exec.simps*[*simp del*] *get_fstn_args.simps*[*simp del*]
arity.simps[*simp del*] *Sigma.simps*[*simp del*]
rec_sigma.simps[*simp del*]

lemma *rec_pr_Suc_simp_rewrite*:
rec_exec (*Pr n f g*) (*xs* @ [*Suc x*]) =
rec_exec g (*xs* @ [*x*] @
[*rec_exec* (*Pr n f g*) (*xs* @ [*x*])])
⟨*proof*⟩

lemma *Sigma_0_simp_rewrite*:
Sigma f (*xs* @ [*0*]) = *f* (*xs* @ [*0*])
⟨*proof*⟩

lemma *Sigma_Suc_simp_rewrite*:
Sigma f (*xs* @ [*Suc x*]) = *Sigma f* (*xs* @ [*x*]) + *f* (*xs* @ [*Suc x*])
⟨*proof*⟩

lemma *append_access_1*[*simp*]: (*xs* @ *ys*) ! (*Suc* (*length xs*)) = *ys* ! *1*

⟨*proof*⟩

lemma *get_fstn_args_take*: ⟦*length xs* = *m*; *n* ≤ *m*⟧ ⟹
map (λ*f*. *rec_exec f xs*) (*get_fstn_args m n*)= *take n xs*
⟨*proof*⟩

lemma *arity_primerec*[*simp*]: *primerec f n* ⟹ *arity f* = *n*
⟨*proof*⟩

lemma *rec_sigma_Suc_simp_rewrite*:
primerec f (*Suc* (*length xs*))
⟹ *rec_exec* (*rec_sigma f*) (*xs* @ [*Suc x*]) =
rec_exec (*rec_sigma f*) (*xs* @ [*x*]) + *rec_exec f* (*xs* @ [*Suc x*])
⟨*proof*⟩

The correctness of *rec_sigma* with respect to its specification.

lemma *sigma_lemma*:
primerec rg (*Suc* (*length xs*))
⟹ *rec_exec* (*rec_sigma rg*) (*xs* @ [*x*]) = *Sigma* (*rec_exec rg*) (*xs* @ [*x*])
⟨*proof*⟩

rec_accum f (*x1*, *x2*, . . . , *xn*, *k*) = *f*(*x1*, *x2*, . . . , *xn*, 0) ∗ *f*(*x1*, *x2*, . . . , *xn*, 1) ∗ . . .
f(*x1*, *x2*, . . . , *xn*, *k*)

fun *rec_accum* :: *recf* ⇒ *recf*
where
rec_accum rf =
(*let vl* = *arity rf in*
Pr (*vl* − 1) (*Cn* (*vl* − 1) *rf* (*get_fstn_args* (*vl* − 1) (*vl* − 1) @
[*Cn* (*vl* − 1) (*constn* 0) [*id* (*vl* − 1) 0]]))
(*Cn* (*Suc vl*) *rec_mult* [*id* (*Suc vl*) (*vl*),
Cn (*Suc vl*) *rf* (*get_fstn_args* (*Suc vl*) (*vl* − 1)
@ [*Cn* (*Suc vl*) *s* [*id* (*Suc vl*) (*vl* − 1)]])]))

Accum is the formal specification of *rec_accum*.

function *Accum* :: (*nat list* ⇒ *nat*) ⇒ *nat list* ⇒ *nat*
where
Accum f xs = (*if last xs* = 0 *then f xs*
else (*Accum f* (*butlast xs* @ [*last xs* − 1]) ∗
f xs))
⟨*proof*⟩
termination
⟨*proof*⟩

lemma *rec_accum_Suc_simp_rewrite*:
primerec f (*Suc* (*length xs*))
⟹ *rec_exec* (*rec_accum f*) (*xs* @ [*Suc x*]) =
rec_exec (*rec_accum f*) (*xs* @ [*x*]) ∗ *rec_exec f* (*xs* @ [*Suc x*])
⟨*proof*⟩

The correctness of *rec_accum* with respect to its specification.

lemma *accum_lemma* :
primerec rg (*Suc* (*length xs*))
$\Longrightarrow$ *rec_exec* (*rec_accum rg*) (*xs* @ [*x*]) = *Accum* (*rec_exec rg*) (*xs* @ [*x*])
⟨*proof*⟩

declare *rec_accum.simps* [*simp del*]

rec_all t f (*x1*, *x2*, . . . , *xn*) computes the charactrization function of the following FOL formula: $(\forall\ x \leq t(x1, x2, \ldots, xn).\ (f(x1, x2, \ldots, xn, x) > 0))$

fun *rec_all* :: *recf* ⇒ *recf* ⇒ *recf*
where
rec_all rt rf =
(*let vl* = *arity rf in*
Cn (*vl* − *1*) *rec_sg* [*Cn* (*vl* − *1*) (*rec_accum rf*)
(*get_fstn_args* (*vl* − *1*) (*vl* − *1*) @ [*rt*])])

lemma *rec_accum_ex*:
assumes *primerec rf* (*Suc* (*length xs*))
shows (*rec_exec* (*rec_accum rf*) (*xs* @ [*x*]) = *0*) =
(∃ *t* ≤ *x*. *rec_exec rf* (*xs* @ [*t*]) = *0*)
⟨*proof*⟩

The correctness of *rec_all*.

lemma *all_lemma*:
⟦*primerec rf* (*Suc* (*length xs*));
primerec rt (*length xs*)⟧
⟹ *rec_exec* (*rec_all rt rf*) *xs* = (*if* (∀ *x* ≤ (*rec_exec rt xs*). *0* < *rec_exec rf* (*xs* @ [*x*])) *then* *1*
else 0)
⟨*proof*⟩

rec_ex t f (*x1*, *x2*, . . . , *xn*) computes the charactrization function of the following FOL formula: $(\exists\ x \leq t(x1, x2, \ldots, xn).\ (f(x1, x2, \ldots, xn, x) > 0))$

fun *rec_ex* :: *recf* ⇒ *recf* ⇒ *recf*
where
rec_ex rt rf =
(*let vl* = *arity rf in*
Cn (*vl* − *1*) *rec_sg* [*Cn* (*vl* − *1*) (*rec_sigma rf*)
(*get_fstn_args* (*vl* − *1*) (*vl* − *1*) @ [*rt*])])

lemma *rec_sigma_ex*:
assumes *primerec rf* (*Suc* (*length xs*))
shows (*rec_exec* (*rec_sigma rf*) (*xs* @ [*x*]) = *0*) =
(∀ *t* ≤ *x*. *rec_exec rf* (*xs* @ [*t*]) = *0*)
⟨*proof*⟩

The correctness of *ex_lemma*.

lemma *ex_lemma*:
⟦*primerec rf* (*Suc* (*length xs*));

```
  primerec rt (length xs)⟧
⟹ (rec_exec (rec_ex rt rf) xs =
   (if (∃ x ≤ (rec_exec rt xs). 0 <rec_exec rf (xs @ [x])) then 1
    else 0))
 ⟨proof⟩
```

Definition of $Min[R]$ on page 77 of Boolos's book.

```
fun Minr :: (nat list ⇒ bool) ⇒ nat list ⇒ nat ⇒ nat
 where Minr Rr xs w = (let setx = {y | y. (y ≤ w) ∧ Rr (xs @ [y])} in
                 if (setx = {}) then (Suc w)
                              else (Min setx))

declare Minr.simps[simp del] rec_all.simps[simp del]
```

The following is a set of auxilliary lemmas about *Minr*.

```
lemma Minr_range: Minr Rr xs w ≤ w ∨ Minr Rr xs w = Suc w
 ⟨proof⟩

lemma expand_conj_in_set: {x. x ≤ Suc w ∧ Rr (xs @ [x])}
   = (if Rr (xs @ [Suc w]) then insert (Suc w)
                                {x. x ≤ w ∧ Rr (xs @ [x])}
    else {x. x ≤ w ∧ Rr (xs @ [x])})
 ⟨proof⟩

lemma Minr_strip_Suc[simp]: Minr Rr xs w ≤ w ⟹ Minr Rr xs (Suc w) = Minr Rr xs w
 ⟨proof⟩

lemma x_empty_set[simp]: ∀ x≤w. ¬ Rr (xs @ [x]) ⟹
                         {x. x ≤ w ∧ Rr (xs @ [x])} = {}
 ⟨proof⟩

lemma Minr_is_Suc[simp]: ⟦Minr Rr xs w = Suc w; Rr (xs @ [Suc w])⟧ ⟹
                          Minr Rr xs (Suc w) = Suc w
 ⟨proof⟩

lemma Minr_is_Suc_Suc[simp]: ⟦Minr Rr xs w = Suc w; ¬ Rr (xs @ [Suc w])⟧ ⟹
                            Minr Rr xs (Suc w) = Suc (Suc w)
 ⟨proof⟩

lemma Minr_Suc_simp:
 Minr Rr xs (Suc w) =
    (if Minr Rr xs w ≤ w then Minr Rr xs w
     else if (Rr (xs @ [Suc w])) then (Suc w)
     else Suc (Suc w))
 ⟨proof⟩
```

rec_Minr is the recursive function used to implement *Minr*: if *Rr* is implemented by a recursive function *recf*, then *rec_Minr recf* is the recursive function used to implement *Minr Rr*

```
fun rec_Minr :: recf ⇒ recf
```

where
rec_Minr rf =
(*let vl* = *arity rf*
in let rq = *rec_all* (*id vl* (*vl* − *1*)) (*Cn* (*Suc vl*)
rec_not [*Cn* (*Suc vl*) *rf*
(*get_fstn_args* (*Suc vl*) (*vl* − *1*) @
[*id* (*Suc vl*) (*vl*)])])
in rec_sigma rq)

lemma *length_getpren_params*[*simp*]: *length* (*get_fstn_args m n*) = *n*
⟨*proof*⟩

lemma *length_app*:
(*length* (*get_fstn_args* (*arity rf* − *Suc 0*)
(*arity rf* − *Suc 0*)
@ [*Cn* (*arity rf* − *Suc 0*) (*constn 0*)
[*recf.id* (*arity rf* − *Suc 0*) *0*]]))
= (*Suc* (*arity rf* − *Suc 0*))
⟨*proof*⟩

lemma *primerec_accum*: *primerec* (*rec_accum rf*) *n* ⟹ *primerec rf n*
⟨*proof*⟩

lemma *primerec_all*: *primerec* (*rec_all rt rf*) *n* ⟹
primerec rt n ∧ *primerec rf* (*Suc n*)
⟨*proof*⟩

declare *numeral_3_eq_3*[*simp*]

lemma *primerec_rec_pred_1*[*intro*]: *primerec rec_pred* (*Suc 0*)
⟨*proof*⟩

lemma *primerec_rec_minus_2*[*intro*]: *primerec rec_minus* (*Suc* (*Suc 0*))
⟨*proof*⟩

lemma *primerec_constn_1*[*intro*]: *primerec* (*constn n*) (*Suc 0*)
⟨*proof*⟩

lemma *primerec_rec_sg_1*[*intro*]: *primerec rec_sg* (*Suc 0*)
⟨*proof*⟩

lemma *primerec_getpren*[*elim*]: ⟦*i* < *n*; *n* ≤ *m*⟧ ⟹ *primerec* (*get_fstn_args m n* ! *i*) *m*
⟨*proof*⟩

lemma *primerec_rec_add_2*[*intro*]: *primerec rec_add* (*Suc* (*Suc 0*))
⟨*proof*⟩

lemma *primerec_rec_mult_2*[*intro*]:*primerec rec_mult* (*Suc* (*Suc 0*))
⟨*proof*⟩

lemma *primerec_ge_2_elim*[*elim*]: ⟦*primerec rf n*; *n* ≥ *Suc* (*Suc 0*)⟧ ⟹
primerec (*rec_accum rf*) *n*
⟨*proof*⟩

lemma *primerec_all_iff*:
⟦*primerec rt n*; *primerec rf* (*Suc n*); *n* > *0*⟧ ⟹
primerec (*rec_all rt rf*) *n*
⟨*proof*⟩

lemma *primerec_rec_not_1*[*intro*]: *primerec rec_not* (*Suc 0*)
⟨*proof*⟩

lemma *Min_false1*[*simp*]: ⟦¬ *Min* {*uu*. *uu* ≤ *w* ∧ *0* < *rec_exec rf* (*xs* @ [*uu*])} ≤ *w*;
x ≤ *w*; *0* < *rec_exec rf* (*xs* @ [*x*])⟧
⟹ *False*
⟨*proof*⟩

lemma *sigma_minr_lemma*:
assumes *prrf*: *primerec rf* (*Suc* (*length xs*))
shows *UF.Sigma* (*rec_exec* (*rec_all* (*recf.id* (*Suc* (*length xs*)) (*length xs*))
(*Cn* (*Suc* (*Suc* (*length xs*))) *rec_not*
[*Cn* (*Suc* (*Suc* (*length xs*))) *rf* (*get_fstn_args* (*Suc* (*Suc* (*length xs*)))
(*length xs*) @ [*recf.id* (*Suc* (*Suc* (*length xs*))) (*Suc* (*length xs*))]])))
(*xs* @ [*w*]) =
Minr (λ*args*. *0* < *rec_exec rf args*) *xs w*
⟨*proof*⟩

The correctness of *rec_Minr*.

lemma *Minr_lemma*:
⟦*primerec rf* (*Suc* (*length xs*))⟧
⟹ *rec_exec* (*rec_Minr rf*) (*xs* @ [*w*]) =
Minr (λ *args*. (*0* < *rec_exec rf args*)) *xs w*
⟨*proof*⟩

rec_le is the comparasion function which compares its two arguments, testing whether the first is less or equal to the second.

definition *rec_le* :: *recf*
where
rec_le = *Cn* (*Suc* (*Suc 0*)) *rec_disj* [*rec_less*, *rec_eq*]

The correctness of *rec_le*.

lemma *le_lemma*:
⋀*x y*. *rec_exec rec_le* [*x*, *y*] = (*if* (*x* ≤ *y*) *then 1 else 0*)
⟨*proof*⟩

Definition of *Max*[*Rr*] on page 77 of Boolos's book.

fun *Maxr* :: (*nat list* ⇒ *bool*) ⇒ *nat list* ⇒ *nat* ⇒ *nat*
where
Maxr Rr xs w = (*let setx* = {*y*. *y* ≤ *w* ∧ *Rr* (*xs* @[*y*])} *in*

if setx = {} then 0
else Max setx)

rec_maxr is the recursive function used to implementation *Maxr*.

```
fun rec_maxr :: recf ⇒ recf
 where
  rec_maxr rr = (let vl = arity rr in
          let rt = id (Suc vl) (vl − 1) in
          let rf1 = Cn (Suc (Suc vl)) rec_le
           [id (Suc (Suc vl))
            ((Suc vl)), id (Suc (Suc vl)) (vl)] in
          let rf2 = Cn (Suc (Suc vl)) rec_not
             [Cn (Suc (Suc vl))
                 rr (get_fstn_args (Suc (Suc vl))
                  (vl − 1) @
                   [id (Suc (Suc vl)) ((Suc vl))])] in
          let rf = Cn (Suc (Suc vl)) rec_disj [rf1, rf2] in
          let Qf = Cn (Suc vl) rec_not [rec_all rt rf]
          in Cn vl (rec_sigma Qf) (get_fstn_args vl vl @
                                   [id vl (vl − 1)]))

declare rec_maxr.simps[simp del] Maxr.simps[simp del]
declare le_lemma[simp]

declare numeral_2_eq_2[simp]

lemma primerec_rec_disj_2[intro]: primerec rec_disj (Suc (Suc 0))
 ⟨proof⟩

lemma primerec_rec_less_2[intro]: primerec rec_less (Suc (Suc 0))
 ⟨proof⟩

lemma primerec_rec_eq_2[intro]: primerec rec_eq (Suc (Suc 0))
 ⟨proof⟩

lemma primerec_rec_le_2[intro]: primerec rec_le (Suc (Suc 0))
 ⟨proof⟩

lemma Sigma_0: ∀ i ≤ n. (f (xs @ [i]) = 0) ⟹
                  Sigma f (xs @ [n]) = 0
 ⟨proof⟩

lemma Sigma_Suc[elim]: ∀ k<Suc w. f (xs @ [k]) = Suc 0
    ⟹ Sigma f (xs @ [w]) = Suc w
 ⟨proof⟩

lemma Sigma_max_point: ⟦∀ k < ma. f (xs @ [k]) = 1;
    ∀ k ≥ ma. f (xs @ [k]) = 0; ma ≤ w⟧
  ⟹ Sigma f (xs @ [w]) = ma
 ⟨proof⟩
```

lemma *Sigma_Max_lemma*:
assumes *prrf*: *primerec rf* (*Suc* (*length xs*))
shows *UF.Sigma* (*rec_exec* (*Cn* (*Suc* (*Suc* (*length xs*))) *rec_not*
[*rec_all* (*recf.id* (*Suc* (*Suc* (*length xs*))) (*length xs*))
(*Cn* (*Suc* (*Suc* (*Suc* (*length xs*)))) *rec_disj*
[*Cn* (*Suc* (*Suc* (*Suc* (*length xs*)))) *rec_le*
[*recf.id* (*Suc* (*Suc* (*Suc* (*length xs*)))) (*Suc* (*Suc* (*length xs*))),
recf.id (*Suc* (*Suc* (*Suc* (*length xs*)))) (*Suc* (*length xs*))],
Cn (*Suc* (*Suc* (*Suc* (*length xs*)))) *rec_not*
[*Cn* (*Suc* (*Suc* (*Suc* (*length xs*)))) *rf*
(*get_fstn_args* (*Suc* (*Suc* (*Suc* (*length xs*)))) (*length xs*) @
[*recf.id* (*Suc* (*Suc* (*Suc* (*length xs*)))) (*Suc* (*Suc* (*length xs*)))])])])])
((*xs* @ [*w*]) @ [*w*]) =
Maxr (λ*args*. *0* < *rec_exec rf args*) *xs w*
⟨*proof*⟩

The correctness of *rec_maxr*.

lemma *Maxr_lemma*:
assumes *h*: *primerec rf* (*Suc* (*length xs*))
shows *rec_exec* (*rec_maxr rf*) (*xs* @ [*w*]) =
Maxr (λ *args*. *0* < *rec_exec rf args*) *xs w*
⟨*proof*⟩

quo is the formal specification of division.

fun *quo* :: *nat list* ⇒ *nat*
where
quo [*x*, *y*] = (*let Rr* =
(λ *zs*. ((*zs* ! (*Suc 0*) ∗ *zs* ! (*Suc* (*Suc 0*))
≤ *zs* ! *0*) ∧ *zs* ! *Suc 0* ≠ (*0*::*nat*)))
in Maxr Rr [*x*, *y*] *x*)

declare *quo.simps*[*simp del*]

The following lemmas shows more directly the menaing of *quo*:

lemma *quo_is_div*: *y* > *0* ⟹ *quo* [*x*, *y*] = *x div y*
⟨*proof*⟩

lemma *quo_zero*[*intro*]: *quo* [*x*, *0*] = *0*
⟨*proof*⟩

lemma *quo_div*: *quo* [*x*, *y*] = *x div y*
⟨*proof*⟩

rec_noteq is the recursive function testing whether its two arguments are not equal.

definition *rec_noteq*:: *recf*
where
rec_noteq = *Cn* (*Suc* (*Suc 0*)) *rec_not* [*Cn* (*Suc* (*Suc 0*))
rec_eq [*id* (*Suc* (*Suc 0*)) (*0*), *id* (*Suc* (*Suc 0*))

$((Suc\ 0))]]$

The correctness of *rec_noteq*.

lemma *noteq_lemma*:
$\bigwedge x\ y.\ rec\_exec\ rec\_noteq\ [x, y] =$
$(if\ x \neq y\ then\ 1\ else\ 0)$
$\langle proof \rangle$

declare *noteq_lemma*[*simp*]

rec_quo is the recursive function used to implement *quo*

definition *rec_quo* :: *recf*
where
$rec\_quo = (let\ rR = Cn\ (Suc\ (Suc\ (Suc\ 0)))\ rec\_conj$
$[Cn\ (Suc\ (Suc\ (Suc\ 0)))\ rec\_le$
$[Cn\ (Suc\ (Suc\ (Suc\ 0)))\ rec\_mult$
$[id\ (Suc\ (Suc\ (Suc\ 0)))\ (Suc\ 0),$
$id\ (Suc\ (Suc\ (Suc\ 0)))\ ((Suc\ (Suc\ 0)))],$
$id\ (Suc\ (Suc\ (Suc\ 0)))\ (0)],$
$Cn\ (Suc\ (Suc\ (Suc\ 0)))\ rec\_noteq$
$[id\ (Suc\ (Suc\ (Suc\ 0)))\ (Suc\ (0)),$
$Cn\ (Suc\ (Suc\ (Suc\ 0)))\ (constn\ 0)$
$[id\ (Suc\ (Suc\ (Suc\ 0)))\ (0)]]]$
$in\ Cn\ (Suc\ (Suc\ 0))\ (rec\_maxr\ rR))\ [id\ (Suc\ (Suc\ 0))$
$(0),id\ (Suc\ (Suc\ 0))\ (Suc\ (0)),$
$id\ (Suc\ (Suc\ 0))\ (0)]$

lemma *primerec_rec_conj_2*[*intro*]: *primerec rec_conj* $(Suc\ (Suc\ 0))$
$\langle proof \rangle$

lemma *primerec_rec_noteq_2*[*intro*]: *primerec rec_noteq* $(Suc\ (Suc\ 0))$
$\langle proof \rangle$

lemma *quo_lemma1*: *rec_exec rec_quo* $[x, y] = quo\ [x, y]$
$\langle proof \rangle$

The correctness of *quo*.

lemma *quo_lemma2*: *rec_exec rec_quo* $[x, y] = x\ div\ y$
$\langle proof \rangle$

rec_mod is the recursive function used to implement the reminder function.

definition *rec_mod* :: *recf*
where
$rec\_mod = Cn\ (Suc\ (Suc\ 0))\ rec\_minus\ [id\ (Suc\ (Suc\ 0))\ (0),$
$Cn\ (Suc\ (Suc\ 0))\ rec\_mult\ [rec\_quo, id\ (Suc\ (Suc\ 0))$
$(Suc\ (0))]]$

The correctness of *rec_mod*:

lemma *mod_lemma*: $\bigwedge x\ y.\ rec\_exec\ rec\_mod\ [x, y] = (x\ mod\ y)$

⟨*proof*⟩

lemmas for embranch function

type-synonym *ftype* = *nat list* ⇒ *nat*
type-synonym *rtype* = *nat list* ⇒ *bool*

The specifation of the mutli-way branching statement on page 79 of Boolos's book.

fun *Embranch* :: (*ftype* ∗ *rtype*) *list* ⇒ *nat list* ⇒ *nat*
where
Embranch [] *xs* = *0* |
Embranch (*gc* # *gcs*) *xs* = (
let (*g*, *c*) = *gc in*
if c xs then g xs else Embranch gcs xs)

fun *rec_embranch′* :: (*recf* ∗ *recf*) *list* ⇒ *nat* ⇒ *recf*
where
rec_embranch′ [] *vl* = *Cn vl z* [*id vl* (*vl* − *1*)] |
rec_embranch′ ((*rg*, *rc*) # *rgcs*) *vl* = *Cn vl rec_add*
[*Cn vl rec_mult* [*rg*, *rc*], *rec_embranch′ rgcs vl*]

rec_embrach is the recursive function used to implement *Embranch*.

fun *rec_embranch* :: (*recf* ∗ *recf*) *list* ⇒ *recf*
where
rec_embranch ((*rg*, *rc*) # *rgcs*) =
(*let vl* = *arity rg in*
rec_embranch′ ((*rg*, *rc*) # *rgcs*) *vl*)

declare *Embranch.simps*[*simp del*] *rec_embranch.simps*[*simp del*]

lemma *embranch_all0*:
⟦∀ *j* < *length rcs*. *rec_exec* (*rcs* ! *j*) *xs* = *0*;
length rgs = *length rcs*;
rcs ≠ [];
list_all (λ *rf*. *primerec rf* (*length xs*)) (*rgs* @ *rcs*)⟧ ⟹
rec_exec (*rec_embranch* (*zip rgs rcs*)) *xs* = *0*
⟨*proof*⟩

lemma *embranch_exec_0*: ⟦*rec_exec aa xs* = *0*; *zip rgs list* ≠ [];
list_all (λ *rf*. *primerec rf* (*length xs*)) ([*a*, *aa*] @ *rgs* @ *list*)⟧
⟹ *rec_exec* (*rec_embranch* ((*a*, *aa*) # *zip rgs list*)) *xs*
= *rec_exec* (*rec_embranch* (*zip rgs list*)) *xs*
⟨*proof*⟩

lemma *zip_null_iff*: ⟦*length xs* = *k*; *length ys* = *k*; *zip xs ys* = []⟧ ⟹ *xs* = [] ∧ *ys* = []
⟨*proof*⟩

lemma *zip_null_gr*: ⟦*length xs* = *k*; *length ys* = *k*; *zip xs ys* ≠ []⟧ ⟹ *0* < *k*
⟨*proof*⟩

lemma *Embranch_0*:
 ⟦*length rgs* = *k*; *length rcs* = *k*; *k* > *0*;
 ∀ *j* < *k*. *rec_exec* (*rcs* ! *j*) *xs* = *0*⟧ ⟹
 Embranch (*zip* (*map rec_exec rgs*) (*map* (λ*r args*. *0* < *rec_exec r args*) *rcs*)) *xs* = *0*
⟨*proof*⟩

The correctness of *rec_embranch*.

lemma *embranch_lemma*:
 assumes *branch_num*:
  length rgs = *n length rcs* = *n n* > *0*
  and *partition*:
  (∃ *i* < *n*. (*rec_exec* (*rcs* ! *i*) *xs* = *1* ∧ (∀ *j* < *n*. *j* ≠ *i* ⟶
                 rec_exec (*rcs* ! *j*) *xs* = *0*)))
  and *prime_all*: *list_all* (λ *rf*. *primerec rf* (*length xs*)) (*rgs* @ *rcs*)
 shows *rec_exec* (*rec_embranch* (*zip rgs rcs*)) *xs* =
          Embranch (*zip* (*map rec_exec rgs*)
            (*map* (λ *r args*. *0* < *rec_exec r args*) *rcs*)) *xs*
⟨*proof*⟩

prime n means *n* is a prime number.

fun *Prime* :: *nat* ⇒ *bool*
 where
  Prime x = (*1* < *x* ∧ (∀ *u* < *x*. (∀ *v* < *x*. *u* ∗ *v* ≠ *x*)))

declare *Prime.simps* [*simp del*]

lemma *primerec_all1*:
 primerec (*rec_all rt rf*) *n* ⟹ *primerec rt n*
 ⟨*proof*⟩

lemma *primerec_all2*: *primerec* (*rec_all rt rf*) *n* ⟹
 primerec rf (*Suc n*)
 ⟨*proof*⟩

rec_prime is the recursive function used to implement *Prime*.

definition *rec_prime* :: *recf*
 where
  rec_prime = *Cn* (*Suc 0*) *rec_conj*
 [*Cn* (*Suc 0*) *rec_less* [*constn 1*, *id* (*Suc 0*) (*0*)],
     rec_all (*Cn 1 rec_minus* [*id 1 0*, *constn 1*])
    (*rec_all* (*Cn 2 rec_minus* [*id 2 0*, *Cn 2* (*constn 1*)
 [*id 2 0*]]) (*Cn 3 rec_noteq*
    [*Cn 3 rec_mult* [*id 3 1*, *id 3 2*], *id 3 0*]))]

declare *numeral_2_eq_2*[*simp del*] *numeral_3_eq_3*[*simp del*]

lemma *exec_tmp*:
 rec_exec (*rec_all* (*Cn 2 rec_minus* [*recf.id 2 0*, *Cn 2* (*constn* (*Suc 0*)) [*recf.id 2 0*]])
 (*Cn 3 rec_noteq* [*Cn 3 rec_mult* [*recf.id 3* (*Suc 0*), *recf.id 3 2*], *recf.id 3 0*])) [*x*, *k*] =
 ((*if* (∀ *w*≤*rec_exec* (*Cn 2 rec_minus* [*recf.id 2 0*, *Cn 2* (*constn* (*Suc 0*)) [*recf.id 2 0*]]) ([*x*, *k*]).

$0 < rec\_exec\ (Cn\ 3\ rec\_noteq\ [Cn\ 3\ rec\_mult\ [recf.id\ 3\ (Suc\ 0), recf.id\ 3\ 2], recf.id\ 3\ 0])$
$([x, k] @ [w]))\ then\ 1\ else\ 0))$
$\langle proof \rangle$

The correctness of *Prime*.

lemma *prime_lemma*: $rec\_exec\ rec\_prime\ [x] = (if\ Prime\ x\ then\ 1\ else\ 0)$
$\langle proof \rangle$

definition *rec_dummyfac* :: *recf*
where
$rec\_dummyfac = Pr\ 1\ (constn\ 1)$
$(Cn\ 3\ rec\_mult\ [id\ 3\ 2, Cn\ 3\ s\ [id\ 3\ 1]])$

The recursive function used to implment factorization.

definition *rec_fac* :: *recf*
where
$rec\_fac = Cn\ 1\ rec\_dummyfac\ [id\ 1\ 0, id\ 1\ 0]$

Formal specification of factorization.

fun *fac* :: $nat \Rightarrow nat$ (_! [100] 99)
where
$fac\ 0 = 1\ |$
$fac\ (Suc\ x) = (Suc\ x) * fac\ x$

lemma *fac_dummy*: $rec\_exec\ rec\_dummyfac\ [x, y] = y\ !$
$\langle proof \rangle$

The correctness of *rec_fac*.

lemma *fac_lemma*: $rec\_exec\ rec\_fac\ [x] = x!$
$\langle proof \rangle$

declare *fac.simps*[*simp del*]

Np x returns the first prime number after *x*.

fun *Np* ::$nat \Rightarrow nat$
where
$Np\ x = Min\ \{y.\ y \leq Suc\ (x!) \wedge x < y \wedge Prime\ y\}$

declare *Np.simps*[*simp del*] *rec_Minr.simps*[*simp del*]

rec_np is the recursive function used to implement *Np*.

definition *rec_np* :: *recf*
where
$rec\_np = (let\ Rr = Cn\ 2\ rec\_conj\ [Cn\ 2\ rec\_less\ [id\ 2\ 0, id\ 2\ 1],$
$Cn\ 2\ rec\_prime\ [id\ 2\ 1]]$
$in\ Cn\ 1\ (rec\_Minr\ Rr)\ [id\ 1\ 0, Cn\ 1\ s\ [rec\_fac]])$

lemma *n_le_fact*[*simp*]: $n < Suc\ (n!)$
$\langle proof \rangle$

lemma *divsor_ex*:
$[\![\neg\ Prime\ x; x > Suc\ 0]\!] \Longrightarrow (\exists\ u > Suc\ 0.\ (\exists\ v > Suc\ 0.\ u * v = x))$
⟨*proof*⟩

lemma *divsor_prime_ex*: $[\![\neg\ Prime\ x; x > Suc\ 0]\!] \Longrightarrow$
$\exists\ p.\ Prime\ p \wedge p\ dvd\ x$
⟨*proof*⟩

lemma *fact_pos*[*intro*]: $0 < n!$
⟨*proof*⟩

lemma *fac_Suc*: $Suc\ n! = (Suc\ n) * (n!)$ ⟨*proof*⟩

lemma *fac_dvd*: $[\![0 < q; q \leq n]\!] \Longrightarrow q\ dvd\ n!$
⟨*proof*⟩

lemma *fac_dvd2*: $[\![Suc\ 0 < q; q\ dvd\ n!; q \leq n]\!] \Longrightarrow \neg\ q\ dvd\ Suc\ (n!)$
⟨*proof*⟩

lemma *prime_ex*: $\exists\ p.\ n < p \wedge p \leq Suc\ (n!) \wedge Prime\ p$
⟨*proof*⟩

lemma *Suc_Suc_induct*[*elim*!]: $[\![i < Suc\ (Suc\ 0);$
$primerec\ (ys\ !\ 0)\ n; primerec\ (ys\ !\ 1)\ n]\!] \Longrightarrow primerec\ (ys\ !\ i)\ n$
⟨*proof*⟩

lemma *primerec_rec_prime_1*[*intro*]: *primerec rec_prime* (*Suc 0*)
⟨*proof*⟩

The correctness of *rec_np*.

lemma *np_lemma*: *rec_exec rec_np* $[x] = Np\ x$
⟨*proof*⟩

rec_power is the recursive function used to implement power function.

definition *rec_power* :: *recf*
where
rec_power = *Pr 1* (*constn 1*) (*Cn 3 rec_mult* [*id 3 0*, *id 3 2*])

The correctness of *rec_power*.

lemma *power_lemma*: *rec_exec rec_power* $[x, y] = x\hat{}y$
⟨*proof*⟩

Pi k returns the *k*-th prime number.

fun *Pi* :: *nat* ⇒ *nat*
where
Pi 0 = *2* |
Pi (*Suc x*) = *Np* (*Pi x*)

definition *rec_dummy_pi* :: *recf*
 where
  rec_dummy_pi = *Pr 1* (*constn 2*) (*Cn 3 rec_np* [*id 3 2*])

rec_pi is the recursive function used to implement *Pi*.

definition *rec_pi* :: *recf*
 where
  rec_pi = *Cn 1 rec_dummy_pi* [*id 1 0*, *id 1 0*]

lemma *pi_dummy_lemma*: *rec_exec rec_dummy_pi* [*x*, *y*] = *Pi y*
 ⟨*proof*⟩

The correctness of *rec_pi*.

lemma *pi_lemma*: *rec_exec rec_pi* [*x*] = *Pi x*
 ⟨*proof*⟩

fun *loR* :: *nat list* ⇒ *bool*
 where
  loR [*x*, *y*, *u*] = (*x mod* (*y*^*u*) = *0*)

declare *loR.simps*[*simp del*]

Lo specifies the *lo* function given on page 79 of Boolos's book. It is one of the two notions of integeral logarithmetic operation on that page. The other is *lg*.

fun *lo* :: *nat* ⇒ *nat* ⇒ *nat*
 where
  lo x y = (*if x* > *1* ∧ *y* > *1* ∧ {*u*. *loR* [*x*, *y*, *u*]} ≠ {} *then Max* {*u*. *loR* [*x*, *y*, *u*]}
                                else 0)

declare *lo.simps*[*simp del*]

lemma *primerec_sigma*[*intro*!]:
 ⟦*n* > *Suc 0*; *primerec rf n*⟧ ⟹
 primerec (*rec_sigma rf*) *n*
 ⟨*proof*⟩

lemma *primerec_rec_maxr*[*intro*!]: ⟦*primerec rf n*; *n* > *0*⟧ ⟹ *primerec* (*rec_maxr rf*) *n*
 ⟨*proof*⟩

lemma *Suc_Suc_Suc_induct*[*elim*!]:
 ⟦*i* < *Suc* (*Suc* (*Suc* (*0*::*nat*))); *primerec* (*ys* ! *0*) *n*;
 primerec (*ys* ! *1*) *n*;
 primerec (*ys* ! *2*) *n*⟧ ⟹ *primerec* (*ys* ! *i*) *n*
 ⟨*proof*⟩

lemma *primerec_2*[*intro*]:
 primerec rec_quo (*Suc* (*Suc 0*)) *primerec rec_mod* (*Suc* (*Suc 0*))
 primerec rec_power (*Suc* (*Suc 0*))
 ⟨*proof*⟩

rec_lo is the recursive function used to implement *Lo*.

```
definition rec_lo :: recf
 where
  rec_lo = (let rR = Cn 3 rec_eq [Cn 3 rec_mod [id 3 0,
         Cn 3 rec_power [id 3 1, id 3 2]],
            Cn 3 (constn 0) [id 3 1]] in
        let rb =  Cn 2 (rec_maxr rR) [id 2 0, id 2 1, id 2 0] in
        let rcond = Cn 2 rec_conj [Cn 2 rec_less [Cn 2 (constn 1)
                                 [id 2 0], id 2 0],
                           Cn 2 rec_less [Cn 2 (constn 1)
                                 [id 2 0], id 2 1]] in
        let rcond2 = Cn 2 rec_minus
                    [Cn 2 (constn 1) [id 2 0], rcond]
        in Cn 2 rec_add [Cn 2 rec_mult [rb, rcond],
           Cn 2 rec_mult [Cn 2 (constn 0) [id 2 0], rcond2]])

lemma rec_lo_Maxr_lor:
 ⟦Suc 0 < x; Suc 0 < y⟧ ⟹
     rec_exec rec_lo [x, y] = Maxr loR [x, y] x
⟨proof⟩

lemma x_less_exp: ⟦y > Suc 0⟧ ⟹ x < y^x
⟨proof⟩


lemma uplimit_loR:
 assumes Suc 0 < x Suc 0 < y loR [x, y, xa]
 shows xa ≤ x
⟨proof⟩

lemma loR_set_strengthen[simp]: ⟦xa ≤ x; loR [x, y, xa]; Suc 0 < x; Suc 0 < y⟧ ⟹
 {u. loR [x, y, u]} = {ya. ya ≤ x ∧ loR [x, y, ya]}
 ⟨proof⟩

lemma Maxr_lo: ⟦Suc 0 < x; Suc 0 < y⟧ ⟹
 Maxr loR [x, y] x = lo x y
 ⟨proof⟩

lemma lo_lemma': ⟦Suc 0 < x; Suc 0 < y⟧ ⟹
 rec_exec rec_lo [x, y] = lo x y
 ⟨proof⟩

lemma lo_lemma'': ⟦¬ Suc 0 < x⟧ ⟹ rec_exec rec_lo [x, y] = lo x y
 ⟨proof⟩

lemma lo_lemma''': ⟦¬ Suc 0 < y⟧ ⟹ rec_exec rec_lo [x, y] = lo x y
 ⟨proof⟩
```

The correctness of *rec_lo*:

lemma *lo_lemma*: *rec_exec rec_lo* [*x*, *y*] = *lo x y*
⟨*proof*⟩

fun *lgR* :: *nat list* ⇒ *bool*
where
lgR [*x*, *y*, *u*] = (*y*^*u* ≤ *x*)

lg specifies the *lg* function given on page 79 of Boolos's book. It is one of the two notions of integeral logarithmetic operation on that page. The other is *lo*.

fun *lg* :: *nat* ⇒ *nat* ⇒ *nat*
where
lg x y = (*if x* > *1* ∧ *y* > *1* ∧ {*u*. *lgR* [*x*, *y*, *u*]} ≠ {} *then*
Max {*u*. *lgR* [*x*, *y*, *u*]}
else 0)

declare *lg.simps*[*simp del*] *lgR.simps*[*simp del*]

rec_lg is the recursive function used to implement *lg*.

definition *rec_lg* :: *recf*
where
rec_lg = (*let rec_lgR* = *Cn 3 rec_le*
[*Cn 3 rec_power* [*id 3 1*, *id 3 2*], *id 3 0*] *in*
let conR1 = *Cn 2 rec_conj* [*Cn 2 rec_less*
[*Cn 2* (*constn 1*) [*id 2 0*], *id 2 0*],
Cn 2 rec_less [*Cn 2* (*constn 1*)
[*id 2 0*], *id 2 1*]] *in*
let conR2 = *Cn 2 rec_not* [*conR1*] *in*
Cn 2 rec_add [*Cn 2 rec_mult*
[*conR1*, *Cn 2* (*rec_maxr rec_lgR*)
[*id 2 0*, *id 2 1*, *id 2 0*]],
Cn 2 rec_mult [*conR2*, *Cn 2* (*constn 0*)
[*id 2 0*]]])

lemma *lg_maxr*: ⟦*Suc 0* < *x*; *Suc 0* < *y*⟧ ⟹
rec_exec rec_lg [*x*, *y*] = *Maxr lgR* [*x*, *y*] *x*
⟨*proof*⟩

lemma *lgR_ok*: ⟦*Suc 0* < *y*; *lgR* [*x*, *y*, *xa*]⟧ ⟹ *xa* ≤ *x*
⟨*proof*⟩

lemma *lgR_set_strengthen*[*simp*]: ⟦*Suc 0* < *x*; *Suc 0* < *y*; *lgR* [*x*, *y*, *xa*]⟧ ⟹
{*u*. *lgR* [*x*, *y*, *u*]} = {*ya*. *ya* ≤ *x* ∧ *lgR* [*x*, *y*, *ya*]}
⟨*proof*⟩

lemma *maxr_lg*: ⟦*Suc 0* < *x*; *Suc 0* < *y*⟧ ⟹ *Maxr lgR* [*x*, *y*] *x* = *lg x y*
⟨*proof*⟩

lemma *lg_lemma'*: ⟦*Suc 0* < *x*; *Suc 0* < *y*⟧ ⟹ *rec_exec rec_lg* [*x*, *y*] = *lg x y*
⟨*proof*⟩

lemma *lg_lemma''*: $\neg$ *Suc 0* $<$ *x* $\Longrightarrow$ *rec_exec rec_lg* [*x*, *y*] = *lg x y*
⟨*proof*⟩

lemma *lg_lemma'''*: $\neg$ *Suc 0* $<$ *y* $\Longrightarrow$ *rec_exec rec_lg* [*x*, *y*] = *lg x y*
⟨*proof*⟩

The correctness of *rec_lg*.

lemma *lg_lemma*: *rec_exec rec_lg* [*x*, *y*] = *lg x y*
⟨*proof*⟩

Entry sr i returns the *i*-th entry of a list of natural numbers encoded by number *sr* using Godel's coding.

fun *Entry* :: *nat* ⇒ *nat* ⇒ *nat*
where
Entry sr i = *lo sr* (*Pi* (*Suc i*))

rec_entry is the recursive function used to implement *Entry*.

definition *rec_entry*:: *recf*
where
rec_entry = *Cn 2 rec_lo* [*id 2 0*, *Cn 2 rec_pi* [*Cn 2 s* [*id 2 1*]]]

declare *Pi.simps*[*simp del*]

The correctness of *rec_entry*.

lemma *entry_lemma*: *rec_exec rec_entry* [*str*, *i*] = *Entry str i*
⟨*proof*⟩

23.2 The construction of F

Using the auxilliary functions obtained in last section, we are going to contruct the function *F*, which is an interpreter of Turing Machines.

fun *listsum2* :: *nat list* ⇒ *nat* ⇒ *nat*
where
listsum2 xs 0 = *0*
| *listsum2 xs* (*Suc n*) = *listsum2 xs n* + *xs* ! *n*

fun *rec_listsum2* :: *nat* ⇒ *nat* ⇒ *recf*
where
rec_listsum2 vl 0 = *Cn vl z* [*id vl 0*]
| *rec_listsum2 vl* (*Suc n*) = *Cn vl rec_add* [*rec_listsum2 vl n*, *id vl n*]

declare *listsum2.simps*[*simp del*] *rec_listsum2.simps*[*simp del*]

lemma *listsum2_lemma*: ⟦*length xs* = *vl*; *n* ≤ *vl*⟧ $\Longrightarrow$
rec_exec (*rec_listsum2 vl n*) *xs* = *listsum2 xs n*
⟨*proof*⟩

fun *strt'* :: *nat list* ⇒ *nat* ⇒ *nat*
where

```
  strt' xs 0 = 0
| strt' xs (Suc n) = (let dbound = listsum2 xs n + n in
                      strt' xs n + (2^(xs ! n + dbound) - 2^dbound))

fun rec_strt' :: nat ⇒ nat ⇒ recf
 where
  rec_strt' vl 0 = Cn vl z [id vl 0]
| rec_strt' vl (Suc n) = (let rec_dbound =
Cn vl rec_add [rec_listsum2 vl n, Cn vl (constn n) [id vl 0]]
in Cn vl rec_add [rec_strt' vl n, Cn vl rec_minus
[Cn vl rec_power [Cn vl (constn 2) [id vl 0], Cn vl rec_add
[id vl (n), rec_dbound]],
Cn vl rec_power [Cn vl (constn 2) [id vl 0], rec_dbound]]])

declare strt'.simps[simp del] rec_strt'.simps[simp del]

lemma strt'_lemma: ⟦length xs = vl; n ≤ vl⟧ ⟹
 rec_exec (rec_strt' vl n) xs = strt' xs n
 ⟨proof⟩
```

strt corresponds to the *strt* function on page 90 of B book, but this definition generalises the original one to deal with multiple input arguments.

```
fun strt :: nat list ⇒ nat
 where
  strt xs = (let ys = map Suc xs in
        strt' ys (length ys))

fun rec_map :: recf ⇒ nat ⇒ recf list
 where
  rec_map rf vl = map (λ i. Cn vl rf [id vl i]) [0..<vl]
```

rec_strt is the recursive function used to implement *strt*.

```
fun rec_strt :: nat ⇒ recf
 where
  rec_strt vl = Cn vl (rec_strt' vl vl) (rec_map s vl)

lemma map_s_lemma: length xs = vl ⟹
 map ((λa. rec_exec a xs) ∘ (λi. Cn vl s [recf.id vl i]))
 [0..<vl]
    = map Suc xs
 ⟨proof⟩
```

The correctness of *rec_strt*.

```
lemma strt_lemma: length xs = vl ⟹
 rec_exec (rec_strt vl) xs = strt xs
 ⟨proof⟩
```

The *scan* function on page 90 of B book.

```
fun scan :: nat ⇒ nat
```

where
scan r = r mod 2

rec_scan is the implemention of *scan*.

definition *rec_scan* :: *recf*
where *rec_scan = Cn 1 rec_mod* [*id 1 0*, *constn 2*]

The correctness of *scan*.

lemma *scan_lemma*: *rec_exec rec_scan* [*r*] = *r mod 2*
⟨*proof*⟩

fun *newleft0* :: *nat list* ⇒ *nat*
where
newleft0 [*p*, *r*] = *p*

definition *rec_newleft0* :: *recf*
where
rec_newleft0 = id 2 0

fun *newrgt0* :: *nat list* ⇒ *nat*
where
newrgt0 [*p*, *r*] = *r* − *scan r*

definition *rec_newrgt0* :: *recf*
where
rec_newrgt0 = Cn 2 rec_minus [*id 2 1*, *Cn 2 rec_scan* [*id 2 1*]]

fun *newleft1* :: *nat list* ⇒ *nat*
where
newleft1 [*p*, *r*] = *p*

definition *rec_newleft1* :: *recf*
where
rec_newleft1 = id 2 0

fun *newrgt1* :: *nat list* ⇒ *nat*
where
newrgt1 [*p*, *r*] = *r* + 1 − *scan r*

definition *rec_newrgt1* :: *recf*
where
rec_newrgt1 =
Cn 2 rec_minus [*Cn 2 rec_add* [*id 2 1*, *Cn 2* (*constn 1*) [*id 2 0*]],
Cn 2 rec_scan [*id 2 1*]]

fun *newleft2* :: *nat list* ⇒ *nat*
where
newleft2 [*p*, *r*] = *p div 2*

```
definition rec_newleft2 :: recf
 where
  rec_newleft2 = Cn 2 rec_quo [id 2 0, Cn 2 (constn 2) [id 2 0]]

fun newrgt2 :: nat list ⇒ nat
 where
  newrgt2 [p, r] = 2 * r + p mod 2

definition rec_newrgt2 :: recf
 where
  rec_newrgt2 =
  Cn 2 rec_add [Cn 2 rec_mult [Cn 2 (constn 2) [id 2 0], id 2 1],
          Cn 2 rec_mod [id 2 0, Cn 2 (constn 2) [id 2 0]]]

fun newleft3 :: nat list ⇒ nat
 where
  newleft3 [p, r] = 2 * p + r mod 2

definition rec_newleft3 :: recf
 where
  rec_newleft3 =
 Cn 2 rec_add [Cn 2 rec_mult [Cn 2 (constn 2) [id 2 0], id 2 0],
          Cn 2 rec_mod [id 2 1, Cn 2 (constn 2) [id 2 0]]]

fun newrgt3 :: nat list ⇒ nat
 where
  newrgt3 [p, r] = r div 2

definition rec_newrgt3 :: recf
 where
  rec_newrgt3 = Cn 2 rec_quo [id 2 1, Cn 2 (constn 2) [id 2 0]]
```

The *new_left* function on page 91 of B book.

```
fun newleft :: nat ⇒ nat ⇒ nat ⇒ nat
 where
  newleft p r a = (if a = 0 ∨ a = 1 then newleft0 [p, r]
             else if a = 2 then newleft2 [p, r]
             else if a = 3 then newleft3 [p, r]
             else p)
```

rec_newleft is the recursive function used to implement *newleft*.

```
definition rec_newleft :: recf
 where
  rec_newleft =
 (let g0 =
    Cn 3 rec_newleft0 [id 3 0, id 3 1] in
 let g1 = Cn 3 rec_newleft2 [id 3 0, id 3 1] in
 let g2 = Cn 3 rec_newleft3 [id 3 0, id 3 1] in
 let g3 = id 3 0 in
 let r0 = Cn 3 rec_disj
```

```
      [Cn 3 rec_eq [id 3 2, Cn 3 (constn 0) [id 3 0]],
       Cn 3 rec_eq [id 3 2, Cn 3 (constn 1) [id 3 0]]] in
 let r1 = Cn 3 rec_eq [id 3 2, Cn 3 (constn 2) [id 3 0]] in
 let r2 = Cn 3 rec_eq [id 3 2, Cn 3 (constn 3) [id 3 0]] in
 let r3 = Cn 3 rec_less [Cn 3 (constn 3) [id 3 0], id 3 2] in
 let gs = [g0, g1, g2, g3] in
 let rs = [r0, r1, r2, r3] in
 rec_embranch (zip gs rs))
```

declare *newleft.simps*[*simp del*]

lemma *Suc_Suc_Suc_Suc_induct*:
$[\![i < Suc\ (Suc\ (Suc\ (Suc\ 0)))$; $i = 0 \Longrightarrow P\ i$;
$i = 1 \Longrightarrow P\ i$; $i = 2 \Longrightarrow P\ i$;
$i = 3 \Longrightarrow P\ i]\!] \Longrightarrow P\ i$
⟨*proof*⟩

declare *quo_lemma2*[*simp*] *mod_lemma*[*simp*]

The correctness of *rec_newleft*.

lemma *newleft_lemma*:
rec_exec rec_newleft [*p*, *r*, *a*] = *newleft p r a*
⟨*proof*⟩

The *newrght* function is one similar to *newleft*, but used to compute the right number.

```
fun newrght :: nat ⇒ nat ⇒ nat ⇒ nat
 where
  newrght p r a  = (if a = 0 then newrgt0 [p, r]
                    else if a = 1 then newrgt1 [p, r]
                    else if a = 2 then newrgt2 [p, r]
                    else if a = 3 then newrgt3 [p, r]
                    else r)
```

rec_newrght is the recursive function used to implement *newrgth*.

```
definition rec_newrght :: recf
 where
  rec_newrght =
 (let g0 = Cn 3 rec_newrgt0 [id 3 0, id 3 1] in
 let g1 = Cn 3 rec_newrgt1 [id 3 0, id 3 1] in
 let g2 = Cn 3 rec_newrgt2 [id 3 0, id 3 1] in
 let g3 = Cn 3 rec_newrgt3 [id 3 0, id 3 1] in
 let g4 = id 3 1 in
 let r0 = Cn 3 rec_eq [id 3 2, Cn 3 (constn 0) [id 3 0]] in
 let r1 = Cn 3 rec_eq [id 3 2, Cn 3 (constn 1) [id 3 0]] in
 let r2 = Cn 3 rec_eq [id 3 2, Cn 3 (constn 2) [id 3 0]] in
 let r3 = Cn 3 rec_eq [id 3 2, Cn 3 (constn 3) [id 3 0]] in
 let r4 = Cn 3 rec_less [Cn 3 (constn 3) [id 3 0], id 3 2] in
```

let gs = [g0, g1, g2, g3, g4] in
let rs = [r0, r1, r2, r3, r4] in
rec_embranch (zip gs rs))
declare *newrght.simps*[*simp del*]

lemma *numeral_4_eq_4*: *4 = Suc 3*
⟨*proof*⟩

lemma *Suc_5_induct*:
⟦*i < Suc (Suc (Suc (Suc (Suc 0))))*; *i = 0 ⟹ P 0*;
i = 1 ⟹ P 1; *i = 2 ⟹ P 2*; *i = 3 ⟹ P 3*; *i = 4 ⟹ P 4*⟧ *⟹ P i*
⟨*proof*⟩

lemma *primerec_rec_scan_1*[*intro*]: *primerec rec_scan (Suc 0)*
⟨*proof*⟩

The correctness of *rec_newrght*.

lemma *newrght_lemma*: *rec_exec rec_newrght [p, r, a] = newrght p r a*
⟨*proof*⟩

declare *Entry.simps*[*simp del*]

The *actn* function given on page 92 of B book, which is used to fetch Turing Machine intructions. In *actn m q r*, *m* is the Godel coding of a Turing Machine, *q* is the current state of Turing Machine, *r* is the right number of Turing Machine tape.

fun *actn* :: *nat ⇒ nat ⇒ nat ⇒ nat*
where
actn m q r = (if q ≠ 0 then Entry m (4∗(q − 1) + 2 ∗ scan r)
else 4)

rec_actn is the recursive function used to implement *actn*

definition *rec_actn* :: *recf*
where
rec_actn =
Cn 3 rec_add [Cn 3 rec_mult
[Cn 3 rec_entry [id 3 0, Cn 3 rec_add [Cn 3 rec_mult
[Cn 3 (constn 4) [id 3 0],
Cn 3 rec_minus [id 3 1, Cn 3 (constn 1) [id 3 0]]],
Cn 3 rec_mult [Cn 3 (constn 2) [id 3 0],
Cn 3 rec_scan [id 3 2]]]],
Cn 3 rec_noteq [id 3 1, Cn 3 (constn 0) [id 3 0]]],
Cn 3 rec_mult [Cn 3 (constn 4) [id 3 0],
Cn 3 rec_eq [id 3 1, Cn 3 (constn 0) [id 3 0]]]]

The correctness of *actn*.

lemma *actn_lemma*: *rec_exec rec_actn [m, q, r] = actn m q r*
⟨*proof*⟩

fun *newstat* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat*
 where
  newstat m q r = (*if q* ≠ *0 then Entry m* (*4*∗(*q* − *1*) + *2*∗*scan r* + *1*)
        else 0)

definition *rec_newstat* :: *recf*
 where
  rec_newstat = *Cn 3 rec_add*
  [*Cn 3 rec_mult* [*Cn 3 rec_entry* [*id 3 0*,
      Cn 3 rec_add [*Cn 3 rec_mult* [*Cn 3* (*constn 4*) [*id 3 0*],
      Cn 3 rec_minus [*id 3 1*, *Cn 3* (*constn 1*) [*id 3 0*]]],
      Cn 3 rec_add [*Cn 3 rec_mult* [*Cn 3* (*constn 2*) [*id 3 0*],
      Cn 3 rec_scan [*id 3 2*]], *Cn 3* (*constn 1*) [*id 3 0*]]]],
      Cn 3 rec_noteq [*id 3 1*, *Cn 3* (*constn 0*) [*id 3 0*]]],
      Cn 3 rec_mult [*Cn 3* (*constn 0*) [*id 3 0*],
      Cn 3 rec_eq [*id 3 1*, *Cn 3* (*constn 0*) [*id 3 0*]]]]

lemma *newstat_lemma*: *rec_exec rec_newstat* [*m*, *q*, *r*] = *newstat m q r*
 ⟨*proof*⟩

declare *newstat.simps*[*simp del*] *actn.simps*[*simp del*]

code the configuration

fun *trpl* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat*
 where
  trpl p q r = (*Pi 0*)^*p* ∗ (*Pi 1*)^*q* ∗ (*Pi 2*)^*r*

definition *rec_trpl* :: *recf*
 where
  rec_trpl = *Cn 3 rec_mult* [*Cn 3 rec_mult*
    [*Cn 3 rec_power* [*Cn 3* (*constn* (*Pi 0*)) [*id 3 0*], *id 3 0*],
     Cn 3 rec_power [*Cn 3* (*constn* (*Pi 1*)) [*id 3 0*], *id 3 1*]],
     Cn 3 rec_power [*Cn 3* (*constn* (*Pi 2*)) [*id 3 0*], *id 3 2*]]
declare *trpl.simps*[*simp del*]
lemma *trpl_lemma*: *rec_exec rec_trpl* [*p*, *q*, *r*] = *trpl p q r*
 ⟨*proof*⟩

left, stat, rght: decode func

fun *left* :: *nat* ⇒ *nat*
 where
  left c = *lo c* (*Pi 0*)

fun *stat* :: *nat* ⇒ *nat*
 where
  stat c = *lo c* (*Pi 1*)

fun *rght* :: *nat* ⇒ *nat*
 where
  rght c = *lo c* (*Pi 2*)

```
fun inpt :: nat ⇒ nat list ⇒ nat
 where
  inpt m xs = trpl 0 1 (strt xs)

fun newconf :: nat ⇒ nat ⇒ nat
 where
  newconf m c = trpl (newleft (left c) (rght c)
                      (actn m (stat c) (rght c)))
                     (newstat m (stat c) (rght c))
                     (newrght (left c) (rght c)
                         (actn m (stat c) (rght c)))

declare left.simps[simp del] stat.simps[simp del] rght.simps[simp del]
 inpt.simps[simp del] newconf.simps[simp del]

definition rec_left :: recf
 where
  rec_left = Cn 1 rec_lo [id 1 0, constn (Pi 0)]

definition rec_right :: recf
 where
  rec_right = Cn 1 rec_lo [id 1 0, constn (Pi 2)]

definition rec_stat :: recf
 where
  rec_stat = Cn 1 rec_lo [id 1 0, constn (Pi 1)]

definition rec_inpt :: nat ⇒ recf
 where
  rec_inpt vl = Cn vl rec_trpl
            [Cn vl (constn 0) [id vl 0],
             Cn vl (constn 1) [id vl 0],
             Cn vl (rec_strt (vl − 1))
                 (map (λ i. id vl (i)) [1..<vl])]

lemma left_lemma: rec_exec rec_left [c] = left c
 ⟨proof⟩

lemma right_lemma: rec_exec rec_right [c] = rght c
 ⟨proof⟩

lemma stat_lemma: rec_exec rec_stat [c] = stat c
 ⟨proof⟩

declare rec_strt.simps[simp del] strt.simps[simp del]

lemma map_cons_eq:
 (map ((λa. rec_exec a (m # xs)) ∘
  (λi. recf.id (Suc (length xs)) (i)))
      [Suc 0..<Suc (length xs)])
```

$= map\ (\lambda\ i.\ xs\ !\ (i - 1))\ [Suc\ 0..<Suc\ (length\ xs)]$
⟨*proof*⟩

lemma *list_map_eq*:
$vl = length\ (xs::nat\ list) \Longrightarrow map\ (\lambda\ i.\ xs\ !\ (i - 1))\ [Suc\ 0..<Suc\ vl] = xs$
⟨*proof*⟩

lemma *nonempty_listE*:
$Suc\ 0 \leq length\ xs \Longrightarrow$
$(map\ ((\lambda a.\ rec\_exec\ a\ (m\ \#\ xs)) \circ (\lambda i.\ recf.id\ (Suc\ (length\ xs))\ (i)))\ [Suc\ 0..<length\ xs]\ @\ [(m\ \#\ xs)\ !\ length\ xs]) = xs$
⟨*proof*⟩

lemma *inpt_lemma*:
$[\![Suc\ (length\ xs) = vl]\!] \Longrightarrow rec\_exec\ (rec\_inpt\ vl)\ (m\ \#\ xs) = inpt\ m\ xs$
⟨*proof*⟩

definition *rec_newconf*:: *recf*
where
rec_newconf =
Cn 2 rec_trpl
[*Cn 2 rec_newleft* [*Cn 2 rec_left* [*id 2 1*],
Cn 2 rec_right [*id 2 1*],
Cn 2 rec_actn [*id 2 0*,
Cn 2 rec_stat [*id 2 1*],
Cn 2 rec_right [*id 2 1*]]],
Cn 2 rec_newstat [*id 2 0*,
Cn 2 rec_stat [*id 2 1*],
Cn 2 rec_right [*id 2 1*]],
Cn 2 rec_newrght [*Cn 2 rec_left* [*id 2 1*],
Cn 2 rec_right [*id 2 1*],
Cn 2 rec_actn [*id 2 0*,
Cn 2 rec_stat [*id 2 1*],
Cn 2 rec_right [*id 2 1*]]]]

lemma *newconf_lemma*: *rec_exec rec_newconf* [*m* ,*c*] = *newconf m c*
⟨*proof*⟩

declare *newconf_lemma*[*simp*]

conf m r k computes the TM configuration after *k* steps of execution of TM coded as *m* starting from the initial configuration where the left number equals *0*, right number equals *r*.

fun *conf* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
conf m r 0 = *trpl 0* (*Suc 0*) *r*
| *conf m r* (*Suc t*) = *newconf m* (*conf m r t*)

declare *conf.simps*[*simp del*]

conf is implemented by the following recursive function *rec_conf*.

definition *rec_conf* :: *recf*
where
rec_conf = *Pr 2* (*Cn 2 rec_trpl* [*Cn 2* (*constn 0*) [*id 2 0*], *Cn 2* (*constn* (*Suc 0*)) [*id 2 0*], *id 2 1*])
(*Cn 4 rec_newconf* [*id 4 0*, *id 4 3*])

lemma *conf_step*:
rec_exec rec_conf [*m*, *r*, *Suc t*] =
rec_exec rec_newconf [*m*, *rec_exec rec_conf* [*m*, *r*, *t*]]
⟨*proof*⟩

The correctness of *rec_conf*.

lemma *conf_lemma*:
rec_exec rec_conf [*m*, *r*, *t*] = *conf m r t*
⟨*proof*⟩

NSTD c returns true if the configureation coded by *c* is no a stardard final configuration.

fun *NSTD* :: *nat* ⇒ *bool*
where
NSTD c = (*stat c* ≠ *0* ∨ *left c* ≠ *0* ∨
rght c ≠ *2*^(*lg* (*rght c* + *1*) *2*) − *1* ∨ *rght c* = *0*)

rec_NSTD is the recursive function implementing *NSTD*.

definition *rec_NSTD* :: *recf*
where
rec_NSTD =
Cn 1 rec_disj [
Cn 1 rec_disj [
Cn 1 rec_disj
[*Cn 1 rec_noteq* [*rec_stat*, *constn 0*],
Cn 1 rec_noteq [*rec_left*, *constn 0*]] ,
Cn 1 rec_noteq [*rec_right*,
Cn 1 rec_minus [*Cn 1 rec_power*
[*constn 2*, *Cn 1 rec_lg*
[*Cn 1 rec_add*
[*rec_right*, *constn 1*],
constn 2]], *constn 1*]]],
Cn 1 rec_eq [*rec_right*, *constn 0*]]

lemma *NSTD_lemma1*: *rec_exec rec_NSTD* [*c*] = *Suc 0* ∨
rec_exec rec_NSTD [*c*] = *0*
⟨*proof*⟩

declare *NSTD.simps*[*simp del*]

lemma *NSTD_lemma2'*: $(rec\_exec\ rec\_NSTD\ [c] = Suc\ 0) \Longrightarrow NSTD\ c$
⟨*proof*⟩

lemma *NSTD_lemma2''*:
$NSTD\ c \Longrightarrow (rec\_exec\ rec\_NSTD\ [c] = Suc\ 0)$
⟨*proof*⟩

The correctness of *NSTD*.

lemma *NSTD_lemma2*: $(rec\_exec\ rec\_NSTD\ [c] = Suc\ 0) = NSTD\ c$
⟨*proof*⟩

fun *nstd* :: $nat \Rightarrow nat$
where
$nstd\ c = (if\ NSTD\ c\ then\ 1\ else\ 0)$

lemma *nstd_lemma*: $rec\_exec\ rec\_NSTD\ [c] = nstd\ c$
⟨*proof*⟩

nonstep m r t means afer *t* steps of execution, the TM coded by *m* is not at a stardard final configuration.

fun *nonstop* :: $nat \Rightarrow nat \Rightarrow nat \Rightarrow nat$
where
$nonstop\ m\ r\ t = nstd\ (conf\ m\ r\ t)$

rec_nonstop is the recursive function implementing *nonstop*.

definition *rec_nonstop* :: *recf*
where
$rec\_nonstop = Cn\ 3\ rec\_NSTD\ [rec\_conf]$

The correctness of *rec_nonstop*.

lemma *nonstop_lemma*:
$rec\_exec\ rec\_nonstop\ [m, r, t] = nonstop\ m\ r\ t$
⟨*proof*⟩

rec_halt is the recursive function calculating the steps a TM needs to execute before to reach a stardard final configuration. This recursive function is the only one using *Mn* combinator. So it is the only non-primitive recursive function needs to be used in the construction of the universal function *F*.

definition *rec_halt* :: *recf*
where
$rec\_halt = Mn\ (Suc\ (Suc\ 0))\ (rec\_nonstop)$

declare *nonstop.simps*[*simp del*]

The lemma relates the interpreter of primitive functions with the calculation relation of general recursive functions.

declare *numeral_2_eq_2*[*simp*] *numeral_3_eq_3*[*simp*]

lemma *primerec_rec_right_1*[*intro*]: $primerec\ rec\_right\ (Suc\ 0)$

⟨*proof*⟩

lemma *primerec_rec_pi_helper*:
∀ *i*<*Suc* (*Suc 0*). *primerec* ([*recf*.*id* (*Suc 0*) *0*, *recf*.*id* (*Suc 0*) *0*] ! *i*) (*Suc 0*)
⟨*proof*⟩

lemmas *primerec_rec_pi_helpers* =
primerec_rec_pi_helper primerec_constn_1 primerec_rec_sg_1 primerec_rec_not_1 primerec_rec_conj_2

lemma *primrec_dummyfac*:
∀ *i*<*Suc* (*Suc 0*).
primerec
([*recf*.*id* (*Suc 0*) *0*,
Cn (*Suc 0*) *s*
[*Cn* (*Suc 0*) *rec_dummyfac*
[*recf*.*id* (*Suc 0*) *0*, *recf*.*id* (*Suc 0*) *0*]]] !
i)
(*Suc 0*)
⟨*proof*⟩

lemma *primerec_rec_pi_1*[*intro*]: *primerec rec_pi* (*Suc 0*)
⟨*proof*⟩

lemma *primerec_recs*[*intro*]:
primerec rec_trpl (*Suc* (*Suc* (*Suc 0*)))
primerec rec_newleft0 (*Suc* (*Suc 0*))
primerec rec_newleft1 (*Suc* (*Suc 0*))
primerec rec_newleft2 (*Suc* (*Suc 0*))
primerec rec_newleft3 (*Suc* (*Suc 0*))
primerec rec_newleft (*Suc* (*Suc* (*Suc 0*)))
primerec rec_left (*Suc 0*)
primerec rec_actn (*Suc* (*Suc* (*Suc 0*)))
primerec rec_stat (*Suc 0*)
primerec rec_newstat (*Suc* (*Suc* (*Suc 0*)))
⟨*proof*⟩

lemma *primerec_rec_newrght*[*intro*]: *primerec rec_newrght* (*Suc* (*Suc* (*Suc 0*)))
⟨*proof*⟩

lemma *primerec_rec_newconf*[*intro*]: *primerec rec_newconf* (*Suc* (*Suc 0*))
⟨*proof*⟩

lemma *primerec_rec_conf*[*intro*]: *primerec rec_conf* (*Suc* (*Suc* (*Suc 0*)))
⟨*proof*⟩

lemma *primerec_recs2*[*intro*]:
primerec rec_lg (*Suc* (*Suc 0*))
primerec rec_nonstop (*Suc* (*Suc* (*Suc 0*)))
⟨*proof*⟩

lemma *primerec_terminate*:
⟦*primerec f x*; *length xs* = *x*⟧ ⟹ *terminate f xs*
⟨*proof*⟩

The following lemma gives the correctness of *rec_halt*. It says: if *rec_halt* calculates that the TM coded by *m* will reach a standard final configuration after *t* steps of execution, then it is indeed so.

F: universal machine

valu r extracts computing result out of the right number *r*.

fun *valu* :: *nat* ⇒ *nat*
where
valu r = (*lg* (*r* + *1*) *2*) − *1*

rec_valu is the recursive function implementing *valu*.

definition *rec_valu* :: *recf*
where
rec_valu = *Cn 1 rec_minus* [*Cn 1 rec_lg* [*s*, *constn 2*], *constn 1*]

The correctness of *rec_valu*.

lemma *value_lemma*: *rec_exec rec_valu* [*r*] = *valu r*
⟨*proof*⟩

lemma *primerec_rec_valu_1*[*intro*]: *primerec rec_valu* (*Suc 0*)
⟨*proof*⟩

declare *valu.simps*[*simp del*]

The definition of the universal function *rec_F*.

definition *rec_F* :: *recf*
where
rec_F = *Cn* (*Suc* (*Suc 0*)) *rec_valu* [*Cn* (*Suc* (*Suc 0*)) *rec_right* [*Cn* (*Suc* (*Suc 0*))
rec_conf ([*id* (*Suc* (*Suc 0*)) *0*, *id* (*Suc* (*Suc 0*)) (*Suc 0*), *rec_halt*])]]

lemma *terminate_halt_lemma*:
⟦*rec_exec rec_nonstop* ([*m*, *r*] @ [*t*]) = *0*;
∀ *i*<*t*. *0* < *rec_exec rec_nonstop* ([*m*, *r*] @ [*i*])⟧ ⟹ *terminate rec_halt* [*m*, *r*]
⟨*proof*⟩

The correctness of *rec_F*, halt case.

lemma *F_lemma*: *rec_exec rec_halt* [*m*, *r*] = *t* ⟹ *rec_exec rec_F* [*m*, *r*] = (*valu* (*rght* (*conf m r t*)))
⟨*proof*⟩

lemma *terminate_F_lemma*: *terminate rec_halt* [*m*, *r*] ⟹ *terminate rec_F* [*m*, *r*]
⟨*proof*⟩

The correctness of *rec_F*, nonhalt case.

23.3 Coding function of TMs

The purpose of this section is to get the coding function of Turing Machine, which is going to be named *code*.

```
fun bl2nat :: cell list ⇒ nat ⇒ nat
 where
  bl2nat [] n = 0
| bl2nat (Bk#bl) n = bl2nat bl (Suc n)
| bl2nat (Oc#bl) n = 2^n + bl2nat bl (Suc n)

fun bl2wc :: cell list ⇒ nat
 where
  bl2wc xs = bl2nat xs 0

fun trpl_code :: config ⇒ nat
 where
  trpl_code (st, l, r) = trpl (bl2wc l) st (bl2wc r)

declare bl2nat.simps[simp del] bl2wc.simps[simp del]
 trpl_code.simps[simp del]

fun action_map :: action ⇒ nat
 where
  action_map W0 = 0
| action_map W1 = 1
| action_map L = 2
| action_map R = 3
| action_map Nop = 4

fun action_map_iff :: nat ⇒ action
 where
  action_map_iff (0::nat) = W0
| action_map_iff (Suc 0) = W1
| action_map_iff (Suc (Suc 0)) = L
| action_map_iff (Suc (Suc (Suc 0))) = R
| action_map_iff n = Nop

fun block_map :: cell ⇒ nat
 where
  block_map Bk = 0
| block_map Oc = 1

fun godel_code' :: nat list ⇒ nat ⇒ nat
 where
  godel_code' [] n = 1
| godel_code' (x#xs) n = (Pi n)^x * godel_code' xs (Suc n)

fun godel_code :: nat list ⇒ nat
 where
  godel_code xs = (let lh = length xs in
```

$2\hat{}lh * (godel\_code' xs\ (Suc\ 0)))$

fun *modify_tprog* :: *instr list* ⇒ *nat list*
where
modify_tprog [] = []
| *modify_tprog* ((*ac*, *ns*)#*nl*) = *action_map ac* # *ns* # *modify_tprog nl*

code tp gives the Godel coding of TM program *tp*.

fun *code* :: *instr list* ⇒ *nat*
where
code tp = (*let nl* = *modify_tprog tp in*
godel_code nl)

23.4 Relating interperter functions to the execution of TMs

lemma *bl2wc_0*[*simp*]: *bl2wc* [] = *0* ⟨*proof*⟩

lemma *fetch_action_map_4*[*simp*]: ⟦*fetch tp 0 b* = (*nact*, *ns*)⟧ ⟹ *action_map nact* = *4*
⟨*proof*⟩

lemma *Pi_gr_1*[*simp*]: *Pi n* > *Suc 0*
⟨*proof*⟩

lemma *Pi_not_0*[*simp*]: *Pi n* > *0*
⟨*proof*⟩

declare *godel_code.simps*[*simp del*]

lemma *godel_code'_nonzero*[*simp*]: *0* < *godel_code' nl n*
⟨*proof*⟩

lemma *godel_code_great*: *godel_code nl* > *0*
⟨*proof*⟩

lemma *godel_code_eq_1*: (*godel_code nl* = *1*) = (*nl* = [])
⟨*proof*⟩

lemma *godel_code_1_iff*[*elim*]:
⟦*i* < *length nl*; ¬ *Suc 0* < *godel_code nl*⟧ ⟹ *nl* ! *i* = *0*
⟨*proof*⟩

lemma *prime_coprime*: ⟦*Prime x*; *Prime y*; *x*≠*y*⟧ ⟹ *coprime x y*
⟨*proof*⟩

lemma *Pi_inc*: *Pi* (*Suc i*) > *Pi i*
⟨*proof*⟩

lemma *Pi_inc_gr*: *i* < *j* ⟹ *Pi i* < *Pi j*
⟨*proof*⟩

lemma *Pi_notEq*: $i \neq j \Longrightarrow Pi\ i \neq Pi\ j$
⟨*proof*⟩

lemma *prime_2*[*intro*]: *Prime* (*Suc* (*Suc 0*))
⟨*proof*⟩

lemma *Prime_Pi*[*intro*]: *Prime* (*Pi n*)
⟨*proof*⟩

lemma *Pi_coprime*: $i \neq j \Longrightarrow coprime\ (Pi\ i)\ (Pi\ j)$
⟨*proof*⟩

lemma *Pi_power_coprime*: $i \neq j \Longrightarrow coprime\ ((Pi\ i)\hat{}m)\ ((Pi\ j)\hat{}n)$
⟨*proof*⟩

lemma *coprime_dvd_mult_nat2*: $[\![coprime\ (k::nat)\ n;\ k\ dvd\ n * m]\!] \Longrightarrow k\ dvd\ m$
⟨*proof*⟩

declare *godel_code'.simps*[*simp del*]

lemma *godel_code'_butlast_last_id'*:
godel_code' (*ys* @ [*y*]) (*Suc j*) = *godel_code' ys* (*Suc j*) ∗
Pi (*Suc* (*length ys* + *j*)) ^ *y*
⟨*proof*⟩

lemma *godel_code'_butlast_last_id*:
$xs \neq [] \Longrightarrow godel\_code'\ xs\ (Suc\ j) =$
godel_code' (*butlast xs*) (*Suc j*) ∗ *Pi* (*length xs* + *j*)^(*last xs*)
⟨*proof*⟩

lemma *godel_code'_not0*: *godel_code' xs n* ≠ *0*
⟨*proof*⟩

lemma *godel_code_append_cons*:
length xs = *i* ⟹ *godel_code'* (*xs*@*y*#*ys*) (*Suc 0*)
= *godel_code' xs* (*Suc 0*) ∗ *Pi* (*Suc i*)^*y* ∗ *godel_code' ys* (*i* + *2*)
⟨*proof*⟩

lemma *Pi_coprime_pre*:
length ps ≤ *i* ⟹ *coprime* (*Pi* (*Suc i*)) (*godel_code' ps* (*Suc 0*))
⟨*proof*⟩

lemma *Pi_coprime_suf*: *i* < *j* ⟹ *coprime* (*Pi i*) (*godel_code' ps j*)
⟨*proof*⟩

lemma *godel_finite*:
finite {*u*. *Pi* (*Suc i*) ^ *u dvd godel_code' nl* (*Suc 0*)}
⟨*proof*⟩

lemma *godel_code_in*:

$i < \mathit{length}\ nl \Longrightarrow\ nl\ !\ i\ \in \{u.\ \mathit{Pi}\ (\mathit{Suc}\ i)\ \hat{}\ u\ \mathit{dvd}\ \mathit{godel\_code'}\ nl\ (\mathit{Suc}\ 0)\}$

⟨*proof*⟩

lemma *godel_code'_get_nth*:
$i < \mathit{length}\ nl \Longrightarrow \mathit{Max}\ \{u.\ \mathit{Pi}\ (\mathit{Suc}\ i)\ \hat{}\ u\ \mathit{dvd}\ \mathit{godel\_code'}\ nl\ (\mathit{Suc}\ 0)\} = nl\ !\ i$

⟨*proof*⟩

lemma *godel_code'_set*[*simp*]:
$\{u.\ \mathit{Pi}\ (\mathit{Suc}\ i)\ \hat{}\ u\ \mathit{dvd}\ (\mathit{Suc}\ (\mathit{Suc}\ 0))\ \hat{}\ \mathit{length}\ nl * \mathit{godel\_code'}\ nl\ (\mathit{Suc}\ 0)\} = \{u.\ \mathit{Pi}\ (\mathit{Suc}\ i)\ \hat{}\ u\ \mathit{dvd}\ \ \mathit{godel\_code'}\ nl\ (\mathit{Suc}\ 0)\}$

⟨*proof*⟩

lemma *godel_code_get_nth*:
$i < \mathit{length}\ nl \Longrightarrow \mathit{Max}\ \{u.\ \mathit{Pi}\ (\mathit{Suc}\ i)\ \hat{}\ u\ \mathit{dvd}\ \mathit{godel\_code}\ nl\} = nl\ !\ i$

⟨*proof*⟩

lemma *mod_dvd_simp*: $(x\ \mathit{mod}\ y = (0::\mathit{nat})) = (y\ \mathit{dvd}\ x)$

⟨*proof*⟩

lemma *dvd_power_le*: $[\![a > \mathit{Suc}\ 0;\ a\ \hat{}\ y\ \mathit{dvd}\ a\ \hat{}\ l]\!] \Longrightarrow y \leq l$

⟨*proof*⟩

lemma *Pi_nonzeroE*[*elim*]: $\mathit{Pi}\ n = 0 \Longrightarrow \mathit{RR}$

⟨*proof*⟩

lemma *Pi_not_oneE*[*elim*]: $\mathit{Pi}\ n = \mathit{Suc}\ 0 \Longrightarrow \mathit{RR}$

⟨*proof*⟩

lemma *finite_power_dvd*:
$[\![(a::\mathit{nat}) > \mathit{Suc}\ 0;\ y \neq 0]\!] \Longrightarrow \mathit{finite}\ \{u.\ a\hat{}u\ \mathit{dvd}\ y\}$

⟨*proof*⟩

lemma *conf_decode1*: $[\![m \neq n;\ m \neq k;\ k \neq n]\!] \Longrightarrow \mathit{Max}\ \{u.\ \mathit{Pi}\ m\ \hat{}\ u\ \mathit{dvd}\ \mathit{Pi}\ m\ \hat{}\ l * \mathit{Pi}\ n\ \hat{}\ st * \mathit{Pi}\ k\ \hat{}\ r\} = l$

⟨*proof*⟩

lemma *left_trpl_fst*[*simp*]: $\mathit{left}\ (\mathit{trpl}\ l\ st\ r) = l$

⟨*proof*⟩

lemma *stat_trpl_snd*[*simp*]: $\mathit{stat}\ (\mathit{trpl}\ l\ st\ r) = st$

⟨*proof*⟩

lemma *rght_trpl_trd*[*simp*]: $\mathit{rght}\ (\mathit{trpl}\ l\ st\ r) = r$

⟨*proof*⟩

lemma *max_lor*:
$i < length\ nl \Longrightarrow Max\ \{u.\ loR\ [godel\_code\ nl, Pi\ (Suc\ i), u]\}$
$= nl\ !\ i$
⟨*proof*⟩

lemma *godel_decode*:
$i < length\ nl \Longrightarrow Entry\ (godel\_code\ nl)\ i = nl\ !\ i$
⟨*proof*⟩

lemma *Four_Suc*: $4 = Suc\ (Suc\ (Suc\ (Suc\ 0)))$
⟨*proof*⟩

declare *numeral_2_eq_2*[*simp del*]

lemma *modify_tprog_fetch_even*:
$[\![st \leq length\ tp\ div\ 2; st > 0]\!] \Longrightarrow$
$modify\_tprog\ tp\ !\ (4 * (st - Suc\ 0)\ ) =$
$action\_map\ (fst\ (tp\ !\ (2 * (st - Suc\ 0))))$
⟨*proof*⟩

lemma *modify_tprog_fetch_odd*:
$[\![st \leq length\ tp\ div\ 2; st > 0]\!] \Longrightarrow$
$modify\_tprog\ tp\ !\ (Suc\ (Suc\ (4 * (st - Suc\ 0)))) =$
$action\_map\ (fst\ (tp\ !\ (Suc\ (2 * (st - Suc\ 0)))))$
⟨*proof*⟩

lemma *modify_tprog_fetch_action*:
$[\![st \leq length\ tp\ div\ 2; st > 0; b = 1 \vee b = 0]\!] \Longrightarrow$
$modify\_tprog\ tp\ !\ (4 * (st - Suc\ 0) + 2 * b) =$
$action\_map\ (fst\ (tp\ !\ ((2 * (st - Suc\ 0)) + b)))$
⟨*proof*⟩

lemma *length_modify*: $length\ (modify\_tprog\ tp) = 2 * length\ tp$
⟨*proof*⟩

declare *fetch.simps*[*simp del*]

lemma *fetch_action_eq*:
$[\![block\_map\ b = scan\ r; fetch\ tp\ st\ b = (nact, ns);$
$st \leq length\ tp\ div\ 2]\!] \Longrightarrow actn\ (code\ tp)\ st\ r = action\_map\ nact$
⟨*proof*⟩

lemma *fetch_zero_zero*[*simp*]: $fetch\ tp\ 0\ b = (nact, ns) \Longrightarrow ns = 0$
⟨*proof*⟩

lemma *modify_tprog_fetch_state*:
$[\![st \leq length\ tp\ div\ 2; st > 0; b = 1 \vee b = 0]\!] \Longrightarrow$
$modify\_tprog\ tp\ !\ Suc\ (4 * (st - Suc\ 0) + 2 * b) =$
$(snd\ (tp\ !\ (2 * (st - Suc\ 0) + b)))$
⟨*proof*⟩

lemma *fetch_state_eq*:
 ⟦*block_map b = scan r*;
 fetch tp st b = (nact, ns);
 st ≤ length tp div 2⟧ ⟹ *newstat (code tp) st r = ns*
⟨*proof*⟩

lemma *tpl_eqI*[*intro*!]:
 ⟦*a = a'*; *b = b'*; *c = c'*⟧ ⟹ *trpl a b c = trpl a' b' c'*
 ⟨*proof*⟩

lemma *bl2nat_double*: *bl2nat xs (Suc n) = 2 * bl2nat xs n*
⟨*proof*⟩

lemma *bl2wc_simps*[*simp*]:
 bl2wc (Oc # tl c) = Suc (bl2wc c) − bl2wc c mod 2
 *bl2wc (Bk # c) = 2*bl2wc (c)*
 *2 * bl2wc (tl c) = bl2wc c − bl2wc c mod 2*
 bl2wc [Oc] = Suc 0
 c ≠ [] ⟹ bl2wc (tl c) = bl2wc c div 2
 c ≠ [] ⟹ bl2wc [hd c] = bl2wc c mod 2
 *c ≠ [] ⟹ bl2wc (hd c # d) = 2 * bl2wc d + bl2wc c mod 2*
 *2 * (bl2wc c div 2) = bl2wc c − bl2wc c mod 2*
 bl2wc (Oc # list) mod 2 = Suc 0
 ⟨*proof*⟩

declare *code.simps*[*simp del*]
declare *nth_of.simps*[*simp del*]

The lemma relates the one step execution of TMs with the interpreter function *rec_newconf*.

lemma *rec_t_eq_step*:
 (λ (s, l, r). s ≤ length tp div 2) c ⟹
 trpl_code (step0 c tp) =
 rec_exec rec_newconf [code tp, trpl_code c]
⟨*proof*⟩

lemma *bl2nat_simps*[*simp*]: *bl2nat (Oc # Oc↑x) 0 = (2 * 2 ^ x − Suc 0)*
 bl2nat (Bk↑x) n = 0
 ⟨*proof*⟩

lemma *bl2nat_exp_zero*[*simp*]: *bl2nat (Oc↑y) 0 = 2^y − Suc 0*
⟨*proof*⟩

lemma *bl2nat_cons_bk*: *bl2nat (ks @ [Bk]) 0 = bl2nat ks 0*
⟨*proof*⟩

lemma *bl2nat_cons_oc*:

bl2nat (ks @ [Oc]) 0 = bl2nat ks 0 + 2 ^ length ks
⟨*proof*⟩

lemma *bl2nat_append*:
bl2nat (xs @ ys) 0 = bl2nat xs 0 + bl2nat ys (length xs)
⟨*proof*⟩

lemma *trpl_code_simp*[*simp*]:
trpl_code (steps0 (Suc 0, Bk↑l, <lm>) tp 0) =
rec_exec rec_conf [code tp, bl2wc (<lm>), 0]
⟨*proof*⟩

The following lemma relates the multi-step interpreter function *rec_conf* with the multi-step execution of TMs.

lemma *state_in_range_step*
: ⟦*a ≤ length A div 2*; *step0 (a, b, c) A = (st, l, r)*; *tm_wf (A,0)*⟧
⟹ *st ≤ length A div 2*
⟨*proof*⟩

lemma *state_in_range*: ⟦*steps0 (Suc 0, tp) A stp = (st, l, r)*; *tm_wf (A, 0)*⟧
⟹ *st ≤ length A div 2*
⟨*proof*⟩

lemma *rec_t_eq_steps*:
tm_wf (tp,0) ⟹
trpl_code (steps0 (Suc 0, Bk↑l, <lm>) tp stp) =
rec_exec rec_conf [code tp, bl2wc (<lm>), stp]
⟨*proof*⟩

lemma *bl2wc_Bk_0*[*simp*]: *bl2wc (Bk↑ m) = 0*
⟨*proof*⟩

lemma *bl2wc_Oc_then_Bk*[*simp*]: *bl2wc (Oc↑ rs@Bk↑ n) = bl2wc (Oc↑ rs)*
⟨*proof*⟩

lemma *lg_power*: *x > Suc 0* ⟹ *lg (x ^ rs) x = rs*
⟨*proof*⟩

The following lemma relates execution of TMs with the multi-step interpreter function *rec_nonstop*. Note, *rec_nonstop* is constructed using *rec_conf*.

declare *tm_wf.simps*[*simp del*]

lemma *nonstop_t_eq*:
⟦*steps0 (Suc 0, Bk↑l, <lm>) tp stp = (0, Bk↑ m, Oc↑ rs @ Bk↑ n)*;
tm_wf (tp, 0);
rs > 0⟧
⟹ *rec_exec rec_nonstop [code tp, bl2wc (<lm>), stp] = 0*
⟨*proof*⟩

lemma *actn_0_is_4*[*simp*]: *actn m 0 r = 4*

⟨*proof*⟩

lemma *newstat_0_0*[*simp*]: *newstat m 0 r* = *0*
⟨*proof*⟩

declare *step_red*[*simp del*]

lemma *halt_least_step*:
⟦*steps0* (*Suc 0*, *Bk*↑*l*, <*lm*>) *tp stp* =
(*0*, *Bk*↑ *m*, *Oc*↑*rs* @ *Bk*↑*n*);
tm_wf (*tp*, *0*);
0<*rs*⟧ ⟹
∃ *stp*. (*nonstop* (*code tp*) (*bl2wc* (<*lm*>)) *stp* = *0* ∧
(∀ *stp*′. *nonstop* (*code tp*) (*bl2wc* (<*lm*>)) *stp*′ = *0* ⟶ *stp* ≤ *stp*′))
⟨*proof*⟩

lemma *conf_trpl_ex*: ∃ *p q r*. *conf m* (*bl2wc* (<*lm*>)) *stp* = *trpl p q r*
⟨*proof*⟩

lemma *nonstop_rgt_ex*:
nonstop m (*bl2wc* (<*lm*>)) *stpa* = *0* ⟹ ∃ *r*. *conf m* (*bl2wc* (<*lm*>)) *stpa* = *trpl 0 0 r*
⟨*proof*⟩

lemma *max_divisors*: *x* > *Suc 0* ⟹ *Max* {*u*. *x* ^ *u dvd x* ^ *r*} = *r*
⟨*proof*⟩

lemma *lo_power*:
assumes *x* > *Suc 0* **shows** *lo* (*x* ^ *r*) *x* = *r*
⟨*proof*⟩

lemma *lo_rgt*: *lo* (*trpl 0 0 r*) (*Pi 2*) = *r*
⟨*proof*⟩

lemma *conf_keep*:
conf m lm stp = *trpl 0 0 r* ⟹
conf m lm (*stp* + *n*) = *trpl 0 0 r*
⟨*proof*⟩

lemma *halt_state_keep_steps_add*:
⟦*nonstop m* (*bl2wc* (<*lm*>)) *stpa* = *0*⟧ ⟹
conf m (*bl2wc* (<*lm*>)) *stpa* = *conf m* (*bl2wc* (<*lm*>)) (*stpa* + *n*)
⟨*proof*⟩

lemma *halt_state_keep*:
⟦*nonstop m* (*bl2wc* (<*lm*>)) *stpa* = *0*; *nonstop m* (*bl2wc* (<*lm*>)) *stpb* = *0*⟧ ⟹
conf m (*bl2wc* (<*lm*>)) *stpa* = *conf m* (*bl2wc* (<*lm*>)) *stpb*
⟨*proof*⟩

The correntess of *rec_F* which relates the interpreter function *rec_F* with the execution of of TMs.

lemma *terminate_halt*:
⟦*steps0* (*Suc 0*, *Bk*↑*l*, <*lm*>) *tp stp* = (*0*, *Bk*↑*m*, *Oc*↑*rs*@*Bk*↑*n*);
tm_wf (*tp*,*0*); *0*<*rs*⟧ ⟹ *terminate rec_halt* [*code tp*, (*bl2wc* (<*lm*>))]
⟨*proof*⟩

lemma *terminate_F*:
⟦*steps0* (*Suc 0*, *Bk*↑*l*, <*lm*>) *tp stp* = (*0*, *Bk*↑*m*, *Oc*↑*rs*@*Bk*↑*n*);
tm_wf (*tp*,*0*); *0*<*rs*⟧ ⟹ *terminate rec_F* [*code tp*, (*bl2wc* (<*lm*>))]
⟨*proof*⟩

lemma *F_correct*:
⟦*steps0* (*Suc 0*, *Bk*↑*l*, <*lm*>) *tp stp* = (*0*, *Bk*↑*m*, *Oc*↑*rs*@*Bk*↑*n*);
tm_wf (*tp*,*0*); *0*<*rs*⟧
⟹ *rec_exec rec_F* [*code tp*, (*bl2wc* (<*lm*>))] = (*rs* − *Suc 0*)
⟨*proof*⟩

end

24 Construction of a Universal Turing Machine

theory *UTM*
imports *Recursive Abacus UF HOL.GCD Turing_Hoare*
begin

25 Wang coding of input arguments

The direct compilation of the universal function *rec_F* can not give us UTM, because *rec_F* is of arity 2, where the first argument represents the Godel coding of the TM being simulated and the second argument represents the right number (in Wang's coding) of the TM tape. (Notice, left number is always *0* at the very beginning). However, UTM needs to simulate the execution of any TM which may very well take many input arguments. Therefore, a initialization TM needs to run before the TM compiled from *rec_F*, and the sequential composition of these two TMs will give rise to the UTM we are seeking. The purpose of this initialization TM is to transform the multiple input arguments of the TM being simulated into Wang's coding, so that it can be consumed by the TM compiled from *rec_F* as the second argument.

However, this initialization TM (named *t_wcode*) can not be constructed by compiling from any recursive function, because every recursive function takes a fixed number of input arguments, while *t_wcode* needs to take varying number of arguments and tranform them into Wang's coding. Therefore, this section give a direct construction of *t_wcode* with just some parts being obtained from recursive functions.

The TM used to generate the Wang's code of input arguments is divided into three TMs executed sequentially, namely $prepare$, $mainwork$ and $adjust$. According to the convention, the start state of ever TM is fixed to state 1 while the final state is fixed to 0.

The input and output of $prepare$ are illustrated respectively by Figure 1 and 2.

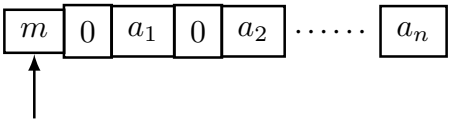

Figure 1: The input of TM $prepare$

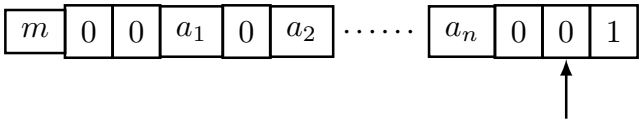

Figure 2: The output of TM $prepare$

As shown in Figure 1, the input of $prepare$ is the same as the the input of UTM, where m is the Godel coding of the TM being interpreted and a_1 through a_n are the n input arguments of the TM under interpretation. The purpose of $purpose$ is to transform this initial tape layout to the one shown in Figure 2, which is convenient for the generation of Wang's codding of $a_1, \ldots, a_n$. The coding procedure starts from a_n and ends after a_1 is encoded. The coding result is stored in an accumulator at the end of the tape (initially represented by the 1 two blanks right to a_n in Figure 2). In Figure 2, arguments $a_1, \ldots, a_n$ are separated by two blanks on both ends with the rest so that movement conditions can be implemented conveniently in subsequent TMs, because, by convention, two consecutive blanks are usually used to signal the end or start of a large chunk of data. The diagram of $prepare$ is given in Figure 3.

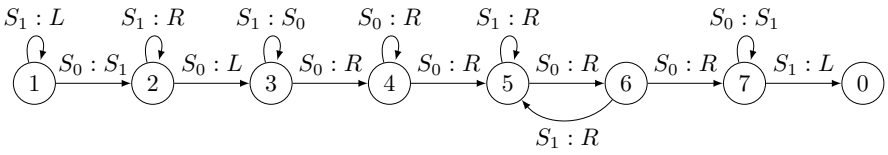

Figure 3: The diagram of TM $prepare$

The purpose of TM $mainwork$ is to compute the Wang's encoding of $a_1, \ldots, a_n$. Every bit of $a_1, \ldots, a_n$, including the separating bits, is processed from left to right. In order to detect the termination condition when the left most bit of a_1 is reached, TM $mainwork$ needs to look ahead and consider three different situations at the start of every iteration:

1. The TM configuration for the first situation is shown in Figure 4, where the accumulator is stored in r, both of the next two bits to be encoded are 1. The configuration at the end of the iteration is shown in Figure 5, where the first 1-bit has been encoded and cleared. Notice that the accumulator has been changed to $(r + 1) \times 2$ to reflect the encoded bit.

2. The TM configuration for the second situation is shown in Figure 6, where the

accumulator is stored in r, the next two bits to be encoded are 1 and 0. After the first 1-bit was encoded and cleared, the second 0-bit is difficult to detect and process. To solve this problem, these two consecutive bits are encoded in one iteration. In this situation, only the first 1-bit needs to be cleared since the second one is cleared by definition. The configuration at the end of the iteration is shown in Figure 7. Notice that the accumulator has been changed to $(r+1)\times 4$ to reflect the two encoded bits.

3. The third situation corresponds to the case when the last bit of a_1 is reached. The TM configurations at the start and end of the iteration are shown in Figure 8 and 9 respectively. For this situation, only the read write head needs to be moved to the left to prepare a initial configuration for TM $adjust$ to start with.

The diagram of $mainwork$ is given in Figure 10. The two rectangular nodes labeled with $2 \times x$ and $4 \times x$ are two TMs compiling from recursive functions so that we do not have to design and verify two quite complicated TMs.

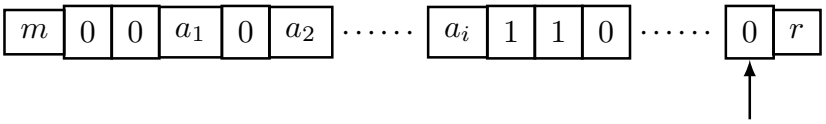

Figure 4: The first situation for TM $mainwork$ to consider

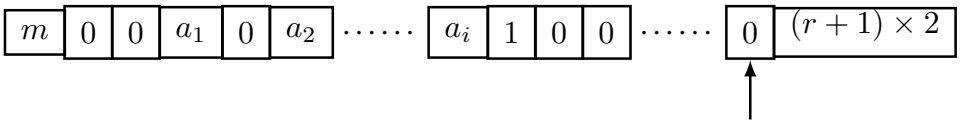

Figure 5: The output for the first case of TM $mainwork$'s processing

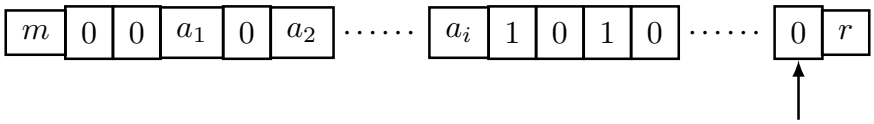

Figure 6: The second situation for TM $mainwork$ to consider

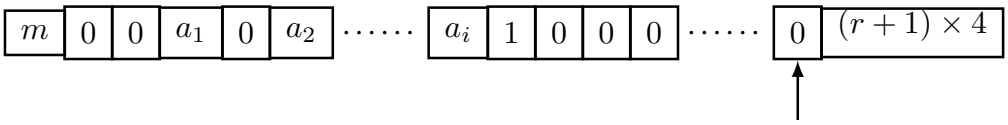

Figure 7: The output for the second case of TM $mainwork$'s processing

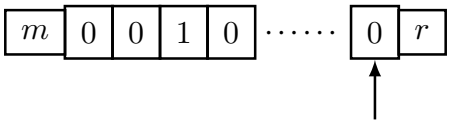

Figure 8: The third situation for TM $mainwork$ to consider

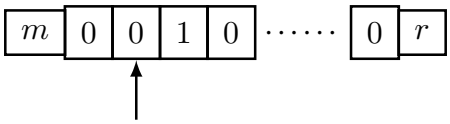

Figure 9: The output for the third case of TM $mainwork$'s processing

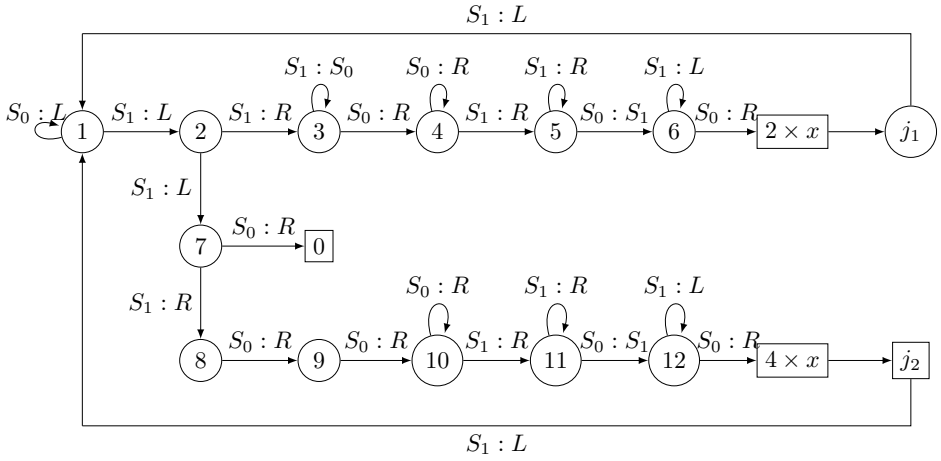

Figure 10: The diagram of TM $mainwork$

The purpose of TM $adjust$ is to encode the last bit of a_1. The initial and final configuration of this TM are shown in Figure 11 and 12 respectively. The diagram of TM $adjust$ is shown in Figure 13.

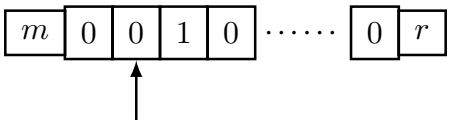

Figure 11: Initial configuration of TM $adjust$

definition *rec_twice* :: *recf*
where
rec_twice = *Cn 1 rec_mult* [*id 1 0*, *constn 2*]

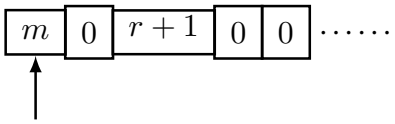

Figure 12: Final configuration of TM $adjust$

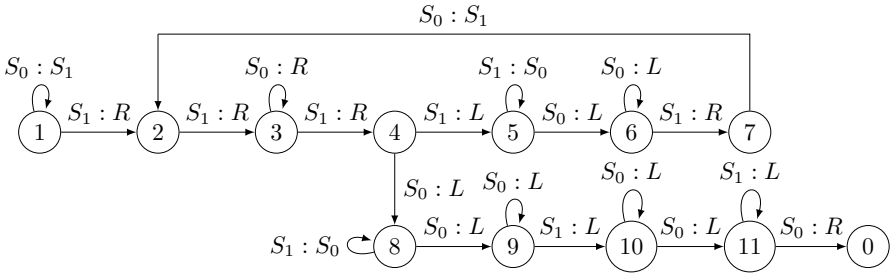

Figure 13: Diagram of TM $adjust$

definition *rec_fourtimes* :: *recf*
where
rec_fourtimes = *Cn 1 rec_mult* [*id 1 0*, *constn 4*]

definition *abc_twice* :: *abc_prog*
where
abc_twice = (*let* (*aprog*, *ary*, *fp*) = *rec_ci rec_twice in*
aprog [+] *dummy_abc* ((*Suc 0*)))

definition *abc_fourtimes* :: *abc_prog*
where
abc_fourtimes = (*let* (*aprog*, *ary*, *fp*) = *rec_ci rec_fourtimes in*
aprog [+] *dummy_abc* ((*Suc 0*)))

definition *twice_ly* :: *nat list*
where
twice_ly = *layout_of abc_twice*

definition *fourtimes_ly* :: *nat list*
where
fourtimes_ly = *layout_of abc_fourtimes*

definition *t_twice_compile* :: *instr list*
where
t_twice_compile= (*tm_of abc_twice* @ (*shift* (*mopup 1*) (*length* (*tm_of abc_twice*) *div 2*)))

definition *t_twice* :: *instr list*
where
t_twice = *adjust0 t_twice_compile*

definition *t_fourtimes_compile* :: *instr list*
 where
 t_fourtimes_compile= (*tm_of abc_fourtimes* @ (*shift* (*mopup 1*) (*length* (*tm_of abc_fourtimes*) *div 2*)))

definition *t_fourtimes* :: *instr list*
 where
 t_fourtimes = *adjust0 t_fourtimes_compile*

definition *t_twice_len* :: *nat*
 where
 t_twice_len = *length t_twice div 2*

definition *t_wcode_main_first_part*:: *instr list*
 where
 t_wcode_main_first_part $\stackrel{def}{=}$
 [(*L*, *1*), (*L*, *2*), (*L*, *7*), (*R*, *3*),
 (*R*, *4*), (*W0*, *3*), (*R*, *4*), (*R*, *5*),
 (*W1*, *6*), (*R*, *5*), (*R*, *13*), (*L*, *6*),
 (*R*, *0*), (*R*, *8*), (*R*, *9*), (*Nop*, *8*),
 (*R*, *10*), (*W0*, *9*), (*R*, *10*), (*R*, *11*),
 (*W1*, *12*), (*R*, *11*), (*R*, *t_twice_len* + *14*), (*L*, *12*)]

definition *t_wcode_main* :: *instr list*
 where
 t_wcode_main = (*t_wcode_main_first_part* @ *shift t_twice 12* @ [(*L*, *1*), (*L*, *1*)]
 @ *shift t_fourtimes* (*t_twice_len* + *13*) @ [(*L*, *1*), (*L*, *1*)])

fun *bl_bin* :: *cell list* ⇒ *nat*
 where
 bl_bin [] = *0*
 | *bl_bin* (*Bk* # *xs*) = *2* ∗ *bl_bin xs*
 | *bl_bin* (*Oc* # *xs*) = *Suc* (*2* ∗ *bl_bin xs*)

declare *bl_bin.simps*[*simp del*]

type-synonym *bin_inv_t* = *cell list* ⇒ *nat* ⇒ *tape* ⇒ *bool*

fun *wcode_before_double* :: *bin_inv_t*
 where
 wcode_before_double ires rs (*l*, *r*) =
 (∃ *ln rn*. *l* = *Bk* # *Bk* # *Bk*↑(*ln*) @ *Oc* # *ires* ∧
 r = *Oc*↑((*Suc* (*Suc rs*))) @ *Bk*↑(*rn*))

declare *wcode_before_double.simps*[*simp del*]

fun *wcode_after_double* :: *bin_inv_t*
 where

```
  wcode_after_double ires rs (l, r) =
   (∃ ln rn. l = Bk # Bk # Bk↑(ln) @ Oc # ires ∧
     r = Oc↑(Suc (Suc (Suc 2∗rs))) @ Bk↑(rn))

declare wcode_after_double.simps[simp del]

fun wcode_on_left_moving_1_B :: bin_inv_t
 where
  wcode_on_left_moving_1_B ires rs (l, r) =
   (∃ ml mr rn. l = Bk↑(ml) @ Oc # Oc # ires ∧
         r = Bk↑(mr) @ Oc↑(Suc rs) @ Bk↑(rn) ∧
         ml + mr > Suc 0 ∧ mr > 0)

declare wcode_on_left_moving_1_B.simps[simp del]

fun wcode_on_left_moving_1_O :: bin_inv_t
 where
  wcode_on_left_moving_1_O ires rs (l, r) =
   (∃ ln rn.
         l = Oc # ires ∧
         r = Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

declare wcode_on_left_moving_1_O.simps[simp del]

fun wcode_on_left_moving_1 :: bin_inv_t
 where
  wcode_on_left_moving_1 ires rs (l, r) =
      (wcode_on_left_moving_1_B ires rs (l, r) ∨ wcode_on_left_moving_1_O ires rs (l, r))

declare wcode_on_left_moving_1.simps[simp del]

fun wcode_on_checking_1 :: bin_inv_t
 where
  wcode_on_checking_1 ires rs (l, r) =
  (∃ ln rn. l = ires ∧
         r = Oc # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun wcode_erase1 :: bin_inv_t
 where
  wcode_erase1 ires rs (l, r) =
    (∃ ln rn. l = Oc # ires ∧
          tl r = Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

declare wcode_erase1.simps [simp del]

fun wcode_on_right_moving_1 :: bin_inv_t
 where
  wcode_on_right_moving_1 ires rs (l, r) =
    (∃ ml mr rn.
       l = Bk↑(ml) @ Oc # ires ∧
```

$r = Bk\uparrow(mr)$ @ $Oc\uparrow(Suc\ rs)$ @ $Bk\uparrow(rn) \wedge$
$ml + mr > Suc\ 0)$

declare *wcode_on_right_moving_1.simps* [*simp del*]

declare *wcode_on_right_moving_1.simps*[*simp del*]

fun *wcode_goon_right_moving_1* :: *bin_inv_t*
where
wcode_goon_right_moving_1 ires rs $(l, r) =$
$(\exists\ ml\ mr\ ln\ rn.$
$l = Oc\uparrow(ml)$ @ $Bk \# Bk \# Bk\uparrow(ln)$ @ $Oc \# ires \wedge$
$r = Oc\uparrow(mr)$ @ $Bk\uparrow(rn) \wedge$
$ml + mr = Suc\ rs)$

declare *wcode_goon_right_moving_1.simps*[*simp del*]

fun *wcode_backto_standard_pos_B* :: *bin_inv_t*
where
wcode_backto_standard_pos_B ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l =\ Bk \# Bk\uparrow(ln)$ @ $Oc \# ires \wedge$
$r =\ Bk \# Oc\uparrow((Suc\ (Suc\ rs)))$ @ $Bk\uparrow(rn))$

declare *wcode_backto_standard_pos_B.simps*[*simp del*]

fun *wcode_backto_standard_pos_O* :: *bin_inv_t*
where
wcode_backto_standard_pos_O ires rs $(l, r) =$
$(\exists\ ml\ mr\ ln\ rn.$
$l = Oc\uparrow(ml)$ @ $Bk \# Bk \# Bk\uparrow(ln)$ @ $Oc \# ires \wedge$
$r = Oc\uparrow(mr)$ @ $Bk\uparrow(rn) \wedge$
$ml + mr = Suc\ (Suc\ rs) \wedge mr > 0)$

declare *wcode_backto_standard_pos_O.simps*[*simp del*]

fun *wcode_backto_standard_pos* :: *bin_inv_t*
where
wcode_backto_standard_pos ires rs $(l, r) =$ (*wcode_backto_standard_pos_B ires rs* $(l, r) \vee$
wcode_backto_standard_pos_O ires rs (l, r))

declare *wcode_backto_standard_pos.simps*[*simp del*]

lemma *bin_wc_eq*: *bl_bin xs* = *bl2wc xs*
⟨*proof*⟩

lemma *tape_of_nl_append_one*: $lm \neq [] \Longrightarrow$ <*lm* @ [*a*]> = <*lm*> @ *Bk* # *Oc*↑*Suc a*
⟨*proof*⟩

lemma *tape_of_nl_rev*: *rev* (<*lm*::*nat list*>) = (<*rev lm*>)
⟨*proof*⟩

lemma *exp_1*[*simp*]: *a*↑(*Suc 0*) = [*a*]
⟨*proof*⟩

lemma *tape_of_nl_cons_app1*: (<*a* # *xs* @ [*b*]>) = (*Oc*↑(*Suc a*) @ *Bk* # (<*xs*@ [*b*]>))
⟨*proof*⟩

lemma *bl_bin_bk_oc*[*simp*]:
bl_bin (*xs* @ [*Bk*, *Oc*]) =
bl_bin xs + 2∗2^(*length xs*)
⟨*proof*⟩

lemma *tape_of_nat*[*simp*]: (<*a*::*nat*>) = *Oc*↑(*Suc a*)
⟨*proof*⟩

lemma *tape_of_nl_cons_app2*: (<*c* # *xs* @ [*b*]>) = (<*c* # *xs*> @ *Bk* # *Oc*↑(*Suc b*))
⟨*proof*⟩

lemma *length_2_elems*[*simp*]: *length* (<*aa* # *a* # *list*>) = *Suc* (*Suc aa*) + *length* (<*a* # *list*>)
⟨*proof*⟩

lemma *bl_bin_addition*[*simp*]: *bl_bin* (*Oc*↑(*Suc aa*) @ *Bk* # *tape_of_nat_list* (*a* # *lista*) @ [*Bk*, *Oc*]) =
bl_bin (*Oc*↑(*Suc aa*) @ *Bk* # *tape_of_nat_list* (*a* # *lista*)) +
2∗ 2^(*length* (*Oc*↑(*Suc aa*) @ *Bk* # *tape_of_nat_list* (*a* # *lista*)))
⟨*proof*⟩

declare *replicate_Suc*[*simp del*]

lemma *bl_bin_2*[*simp*]:
bl_bin (<*aa* # *list*>) + (*4* ∗ *rs* + *4*) ∗ *2* ^ (*length* (<*aa* # *list*>) − *Suc 0*)
= *bl_bin* (*Oc*↑(*Suc aa*) @ *Bk* # <*list* @ [*0*]>) + *rs* ∗ (*2* ∗ *2* ^ (*aa* + *length* (<*list* @ [*0*]>)))
⟨*proof*⟩

lemma *tape_of_nl_app_Suc*: ((<*list* @ [*Suc ab*]>)) = (<*list* @ [*ab*]>) @ [*Oc*]
⟨*proof*⟩

lemma *bl_bin_3*[*simp*]: *bl_bin* (*Oc* # *Oc*↑(*aa*) @ *Bk* # <*list* @ [*ab*]> @ [*Oc*])
= *bl_bin* (*Oc* # *Oc*↑(*aa*) @ *Bk* # <*list* @ [*ab*]>) +
2^(*length* (*Oc* # *Oc*↑(*aa*) @ *Bk* # <*list* @ [*ab*]>))
⟨*proof*⟩
lemma *bl_bin_4*[*simp*]: *bl_bin* (*Oc* # *Oc*↑(*aa*) @ *Bk* # <*list* @ [*ab*]>) + (*4* ∗ *2* ^ (*aa* + *length* (<*list* @ [*ab*]>)) +
4 ∗ (*rs* ∗ *2* ^ (*aa* + *length* (<*list* @ [*ab*]>)))) =
bl_bin (*Oc* # *Oc*↑(*aa*) @ *Bk* # <*list* @ [*Suc ab*]>) +
rs ∗ (*2* ∗ *2* ^ (*aa* + *length* (<*list* @ [*Suc ab*]>)))
⟨*proof*⟩

declare *tape_of_nat*[*simp del*]

fun *wcode_double_case_inv* :: *nat* ⇒ *bin_inv_t*
 where
  wcode_double_case_inv st ires rs (*l*, *r*) =
      (*if st* = *Suc 0 then wcode_on_left_moving_1 ires rs* (*l*, *r*)
      else if st = *Suc* (*Suc 0*) *then wcode_on_checking_1 ires rs* (*l*, *r*)
      else if st = *3 then wcode_erase1 ires rs* (*l*, *r*)
      else if st = *4 then wcode_on_right_moving_1 ires rs* (*l*, *r*)
      else if st = *5 then wcode_goon_right_moving_1 ires rs* (*l*, *r*)
      else if st = *6 then wcode_backto_standard_pos ires rs* (*l*, *r*)
      else if st = *13 then wcode_before_double ires rs* (*l*, *r*)
      else False)

declare *wcode_double_case_inv.simps*[*simp del*]

fun *wcode_double_case_state* :: *config* ⇒ *nat*
 where
  wcode_double_case_state (*st*, *l*, *r*) =
  13 − *st*

fun *wcode_double_case_step* :: *config* ⇒ *nat*
 where
  wcode_double_case_step (*st*, *l*, *r*) =
   (*if st* = *Suc 0 then* (*length l*)
   else if st = *Suc* (*Suc 0*) *then* (*length r*)
   else if st = *3 then*
         if hd r = *Oc then 1 else 0*
   else if st = *4 then* (*length r*)
   else if st = *5 then* (*length r*)
   else if st = *6 then* (*length l*)
   else 0)

fun *wcode_double_case_measure* :: *config* ⇒ *nat* × *nat*
 where
  wcode_double_case_measure (*st*, *l*, *r*) =
   (*wcode_double_case_state* (*st*, *l*, *r*),
    wcode_double_case_step (*st*, *l*, *r*))

definition *wcode_double_case_le* :: (*config* × *config*) *set*
 where *wcode_double_case_le* $\stackrel{def}{=}$ (*inv_image lex_pair wcode_double_case_measure*)

lemma *wf_lex_pair*[*intro*]: *wf lex_pair*
 ⟨*proof*⟩

lemma *wf_wcode_double_case_le*[*intro*]: *wf wcode_double_case_le*
 ⟨*proof*⟩

lemma *fetch_t_wcode_main*[*simp*]:
 fetch t_wcode_main (*Suc 0*) *Bk* = (*L*, *Suc 0*)
 fetch t_wcode_main (*Suc 0*) *Oc* = (*L*, *Suc* (*Suc 0*))

fetch t_wcode_main (Suc (Suc 0)) Oc = (R, 3)
fetch t_wcode_main (Suc (Suc 0)) Bk = (L, 7)
fetch t_wcode_main (Suc (Suc (Suc 0))) Bk = (R, 4)
fetch t_wcode_main (Suc (Suc (Suc 0))) Oc = (W0, 3)
fetch t_wcode_main 4 Bk = (R, 4)
fetch t_wcode_main 4 Oc = (R, 5)
fetch t_wcode_main 5 Oc = (R, 5)
fetch t_wcode_main 5 Bk = (W1, 6)
fetch t_wcode_main 6 Bk = (R, 13)
fetch t_wcode_main 6 Oc = (L, 6)
fetch t_wcode_main 7 Oc = (R, 8)
fetch t_wcode_main 7 Bk = (R, 0)
fetch t_wcode_main 8 Bk = (R, 9)
fetch t_wcode_main 9 Bk = (R, 10)
fetch t_wcode_main 9 Oc = (W0, 9)
fetch t_wcode_main 10 Bk = (R, 10)
fetch t_wcode_main 10 Oc = (R, 11)
fetch t_wcode_main 11 Bk = (W1, 12)
fetch t_wcode_main 11 Oc = (R, 11)
fetch t_wcode_main 12 Oc = (L, 12)
fetch t_wcode_main 12 Bk = (R, t_twice_len + 14)
⟨*proof*⟩

declare *wcode_on_checking_1.simps*[*simp del*]

lemmas *wcode_double_case_inv_simps* =
wcode_on_left_moving_1.simps wcode_on_left_moving_1_O.simps
wcode_on_left_moving_1_B.simps wcode_on_checking_1.simps
wcode_erase1.simps wcode_on_right_moving_1.simps
wcode_goon_right_moving_1.simps wcode_backto_standard_pos.simps

lemma *wcode_on_left_moving_1*[*simp*]:
wcode_on_left_moving_1 ires rs (*b*, []) = *False*
wcode_on_left_moving_1 ires rs (*b*, *r*) ⟹ *b* ≠ []
⟨*proof*⟩

lemma *wcode_on_left_moving_1E*[*elim*]: ⟦*wcode_on_left_moving_1 ires rs* (*b*, *Bk* # *list*);
tl b = *aa* ∧ *hd b* # *Bk* # *list* = *ba*⟧ ⟹
wcode_on_left_moving_1 ires rs (*aa*, *ba*)
⟨*proof*⟩

declare *replicate_Suc*[*simp*]

lemma *wcode_on_moving_1_Elim*[*elim*]:
⟦*wcode_on_left_moving_1 ires rs* (*b*, *Oc* # *list*); *tl b* = *aa* ∧ *hd b* # *Oc* # *list* = *ba*⟧
⟹ *wcode_on_checking_1 ires rs* (*aa*, *ba*)
⟨*proof*⟩

lemma *wcode_on_checking_1_Elim*[*elim*]: ⟦*wcode_on_checking_1 ires rs* (*b*, *Oc* # *ba*);*Oc* # *b*

$= aa \wedge list = ba\rrbracket$
$\Longrightarrow$ *wcode_erase1 ires rs* (aa, ba)
$\langle proof \rangle$

lemma *wcode_on_checking_1_simp*[*simp*]:
wcode_on_checking_1 ires rs $(b, [])$ = *False*
wcode_on_checking_1 ires rs $(b, Bk \# list)$ = *False*
$\langle proof \rangle$

lemma *wcode_erase1_nonempty_snd*[*simp*]: *wcode_erase1 ires rs* $(b, [])$ = *False*
$\langle proof \rangle$

lemma *wcode_on_right_moving_1_nonempty_snd*[*simp*]: *wcode_on_right_moving_1 ires rs* $(b,$ $[])$ = *False*
$\langle proof \rangle$

lemma *wcode_on_right_moving_1_BkE*[*elim*]:
$\llbracket$*wcode_on_right_moving_1 ires rs* $(b, Bk \# ba)$; $Bk \# b = aa \wedge list = b\rrbracket \Longrightarrow$
wcode_on_right_moving_1 ires rs (aa, ba)
$\langle proof \rangle$

lemma *wcode_on_right_moving_1_OcE*[*elim*]:
$\llbracket$*wcode_on_right_moving_1 ires rs* $(b, Oc \# ba)$; $Oc \# b = aa \wedge list = ba\rrbracket$
$\Longrightarrow$ *wcode_goon_right_moving_1 ires rs* (aa, ba)
$\langle proof \rangle$

lemma *wcode_erase1_BkE*[*elim*]:
assumes *wcode_erase1 ires rs* $(b, Bk \# ba)$ $Bk \# b = aa \wedge list = ba$ $c = Bk \# ba$
shows *wcode_on_right_moving_1 ires rs* (aa, ba)
$\langle proof \rangle$

lemma *wcode_erase1_OcE*[*elim*]: $\llbracket$*wcode_erase1 ires rs* $(aa, Oc \# list)$; $b = aa \wedge Bk \# list =$ $ba\rrbracket \Longrightarrow$
wcode_erase1 ires rs (aa, ba)
$\langle proof \rangle$

lemma *wcode_goon_right_moving_1_emptyE*[*elim*]:
assumes *wcode_goon_right_moving_1 ires rs* $(aa, [])$ $b = aa \wedge [Oc] = ba$
shows *wcode_backto_standard_pos ires rs* (aa, ba)
$\langle proof \rangle$

lemma *wcode_goon_right_moving_1_BkE*[*elim*]:
assumes *wcode_goon_right_moving_1 ires rs* $(aa, Bk \# list)$ $b = aa \wedge Oc \# list = ba$
shows *wcode_backto_standard_pos ires rs* (aa, ba)
$\langle proof \rangle$

lemma *wcode_goon_right_moving_1_OcE*[*elim*]:
assumes *wcode_goon_right_moving_1 ires rs* $(b, Oc \# ba)$ $Oc \# b = aa \wedge list = ba$
shows *wcode_goon_right_moving_1 ires rs* (aa, ba)
$\langle proof \rangle$

lemma *wcode_backto_standard_pos_BkE*[*elim*]: ⟦*wcode_backto_standard_pos ires rs* (*b*, *Bk* # *ba*); *Bk* # *b* = *aa* ∧ *list* = *ba*⟧
⟹ *wcode_before_double ires rs* (*aa*, *ba*)
⟨*proof*⟩

lemma *wcode_backto_standard_pos_no_Oc*[*simp*]: *wcode_backto_standard_pos ires rs* ([], *Oc* # *list*) = *False*
⟨*proof*⟩

lemma *wcode_backto_standard_pos_nonempty_snd*[*simp*]: *wcode_backto_standard_pos ires rs* (*b*, []) = *False*
⟨*proof*⟩

lemma *wcode_backto_standard_pos_OcE*[*elim*]: ⟦*wcode_backto_standard_pos ires rs* (*b*, *Oc* # *list*); *tl b* = *aa*; *hd b* # *Oc* # *list* = *ba*⟧
⟹ *wcode_backto_standard_pos ires rs* (*aa*, *ba*)
⟨*proof*⟩

declare *nth_of*.*simps*[*simp del*] *fetch*.*simps*[*simp del*]
lemma *wcode_double_case_first_correctness*:
let P = (λ (*st*, *l*, *r*). *st* = *13*) *in*
let Q = (λ (*st*, *l*, *r*). *wcode_double_case_inv st ires rs* (*l*, *r*)) *in*
let f = (λ *stp*. *steps0* (*Suc 0*, *Bk* # *Bk*↑(*m*) @ *Oc* # *Oc* # *ires*, *Bk* # *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *t_wcode_main stp*) *in*
∃ *n* .*P* (*f n*) ∧ *Q* (*f* (*n*::*nat*))
⟨*proof*⟩

lemma *tm_append_shift_append_steps*:
⟦*steps0* (*st*, *l*, *r*) *tp stp* = (*st*′, *l*′, *r*′);
0 < *st*′;
length tp1 mod 2 = *0*
⟧
⟹ *steps0* (*st* + *length tp1 div 2*, *l*, *r*) (*tp1* @ *shift tp* (*length tp1 div 2*) @ *tp2*) *stp*
= (*st*′ + *length tp1 div 2*, *l*′, *r*′)
⟨*proof*⟩

declare *start_of*.*simps*[*simp del*]

lemma *twice_lemma*: *rec_exec rec_twice* [*rs*] = *2*∗*rs*
⟨*proof*⟩

lemma *t_twice_correct*:
∃ *stp ln rn*. *steps0* (*Suc 0*, *Bk* # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))
(*tm_of abc_twice* @ *shift* (*mopup* (*Suc 0*)) ((*length* (*tm_of abc_twice*) *div 2*))) *stp* =
(*0*, *Bk*↑(*ln*) @ *Bk* # *Bk* # *ires*, *Oc*↑(*Suc* (*2* ∗ *rs*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

declare *adjust*.*simps*[*simp*]

lemma *adjust_fetch0*:
⟦*0 < a*; *a ≤ length ap div 2*; *fetch ap a b = (aa, 0)*⟧
⟹ *fetch (adjust0 ap) a b = (aa, Suc (length ap div 2))*
⟨*proof*⟩

lemma *adjust_fetch_norm*:
⟦*st > 0*; *st ≤ length tp div 2*; *fetch ap st b = (aa, ns)*; *ns ≠ 0*⟧
⟹ *fetch (adjust0 ap) st b = (aa, ns)*
⟨*proof*⟩

declare *adjust.simps*[*simp del*]

lemma *adjust_step_eq*:
assumes *exec*: *step0 (st,l,r) ap = (st′, l′, r′)*
and *wf_tm*: *tm_wf (ap, 0)*
and *notfinal*: *st′ > 0*
shows *step0 (st, l, r) (adjust0 ap) = (st′, l′, r′)*
⟨*proof*⟩

declare *adjust.simps*[*simp del*]

lemma *adjust_steps_eq*:
assumes *exec*: *steps0 (st,l,r) ap stp = (st′, l′, r′)*
and *wf_tm*: *tm_wf (ap, 0)*
and *notfinal*: *st′ > 0*
shows *steps0 (st, l, r) (adjust0 ap) stp = (st′, l′, r′)*
⟨*proof*⟩

lemma *adjust_halt_eq*:
assumes *exec*: *steps0 (1, l, r) ap stp = (0, l′, r′)*
and *tm_wf*: *tm_wf (ap, 0)*
shows ∃ *stp*. *steps0 (Suc 0, l, r) (adjust0 ap) stp =*
(Suc (length ap div 2), l′, r′)
⟨*proof*⟩

declare *tm_wf.simps*[*simp del*]

lemma *tm_wf_t_twice_compile* [*simp*]: *tm_wf (t_twice_compile, 0)*
⟨*proof*⟩

lemma *t_twice_change_term_state*:
∃ *stp ln rn*. *steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n)) t_twice stp*
= (Suc t_twice_len, Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (2 ∗ rs)) @ Bk↑(rn))
⟨*proof*⟩

lemma *length_t_wcode_main_first_part_even*[*intro*]: *length t_wcode_main_first_part mod 2 = 0*
⟨*proof*⟩

lemma *t_twice_append_pre*:
steps0 (*Suc 0*, *Bk # Bk # ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *t_twice stp*
= (*Suc t_twice_len*, *Bk*↑(*ln*) @ *Bk # Bk # ires*, *Oc*↑(*Suc* (*2 * rs*)) @ *Bk*↑(*rn*))
⟹ *steps0* (*Suc 0 + length t_wcode_main_first_part div 2*, *Bk # Bk # ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))
(*t_wcode_main_first_part* @ *shift t_twice* (*length t_wcode_main_first_part div 2*) @
([(*L*, *1*), (*L*, *1*)] @ *shift t_fourtimes* (*t_twice_len + 13*) @ [(*L*, *1*), (*L*, *1*)])) *stp*
= (*Suc* (*t_twice_len*) + *length t_wcode_main_first_part div 2*,
Bk↑(*ln*) @ *Bk # Bk # ires*, *Oc*↑(*Suc* (*2 * rs*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

lemma *t_twice_append*:
∃ *stp ln rn*. *steps0* (*Suc 0 + length t_wcode_main_first_part div 2*, *Bk # Bk # ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))
(*t_wcode_main_first_part* @ *shift t_twice* (*length t_wcode_main_first_part div 2*) @
([(*L*, *1*), (*L*, *1*)] @ *shift t_fourtimes* (*t_twice_len + 13*) @ [(*L*, *1*), (*L*, *1*)])) *stp*
= (*Suc* (*t_twice_len*) + *length t_wcode_main_first_part div 2*, *Bk*↑(*ln*) @ *Bk # Bk # ires*, *Oc*↑(*Suc* (*2 * rs*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

lemma *mopup_mod2*: *length* (*mopup k*) *mod 2* = *0*
⟨*proof*⟩

lemma *fetch_t_wcode_main_Oc*[*simp*]: *fetch t_wcode_main* (*Suc* (*t_twice_len* + *length t_wcode_main_first_part div 2*)) *Oc*
= (*L*, *Suc 0*)
⟨*proof*⟩

lemma *wcode_jump1*:
∃ *stp ln rn*. *steps0* (*Suc* (*t_twice_len*) + *length t_wcode_main_first_part div 2*,
Bk↑(*m*) @ *Bk # Bk # ires*, *Oc*↑(*Suc* (*2 * rs*)) @ *Bk*↑(*n*))
t_wcode_main stp
= (*Suc 0*, *Bk*↑(*ln*) @ *Bk # ires*, *Bk # Oc*↑(*Suc* (*2 * rs*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

lemma *wcode_main_first_part_len*[*simp*]:
length t_wcode_main_first_part = *24*
⟨*proof*⟩

lemma *wcode_double_case*:
shows ∃ *stp ln rn*. *steps0* (*Suc 0*, *Bk # Bk*↑(*m*) @ *Oc # Oc # ires*, *Bk # Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *t_wcode_main stp* =
(*Suc 0*, *Bk # Bk*↑(*ln*) @ *Oc # ires*, *Bk # Oc*↑(*Suc* (*2 * rs + 2*)) @ *Bk*↑(*rn*))
(**is** ∃ *stp ln rn*. *?tm stp ln rn*)
⟨*proof*⟩

fun *wcode_on_left_moving_2_B* :: *bin_inv_t*
where

```
  wcode_on_left_moving_2_B ires rs (l, r) =
   (∃ ml mr rn. l = Bk↑(ml) @ Oc # Bk # Oc # ires ∧
          r = Bk↑(mr) @ Oc↑(Suc rs) @ Bk↑(rn) ∧
          ml + mr > Suc 0 ∧ mr > 0)

fun wcode_on_left_moving_2_O :: bin_inv_t
 where
  wcode_on_left_moving_2_O ires rs (l, r) =
   (∃ ln rn. l = Bk # Oc # ires ∧
         r = Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun wcode_on_left_moving_2 :: bin_inv_t
 where
  wcode_on_left_moving_2 ires rs (l, r) =
   (wcode_on_left_moving_2_B ires rs (l, r) ∨
    wcode_on_left_moving_2_O ires rs (l, r))

fun wcode_on_checking_2 :: bin_inv_t
 where
  wcode_on_checking_2 ires rs (l, r) =
    (∃ ln rn. l = Oc#ires ∧
          r = Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun wcode_goon_checking :: bin_inv_t
 where
  wcode_goon_checking ires rs (l, r) =
    (∃ ln rn. l = ires ∧
          r = Oc # Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun wcode_right_move :: bin_inv_t
 where
  wcode_right_move ires rs (l, r) =
   (∃ ln rn. l = Oc # ires ∧
          r = Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun wcode_erase2 :: bin_inv_t
 where
  wcode_erase2 ires rs (l, r) =
     (∃ ln rn. l = Bk # Oc # ires ∧
          tl r = Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun wcode_on_right_moving_2 :: bin_inv_t
 where
  wcode_on_right_moving_2 ires rs (l, r) =
     (∃ ml mr rn. l = Bk↑(ml) @ Oc # ires ∧
            r = Bk↑(mr) @ Oc↑(Suc rs) @ Bk↑(rn) ∧ ml + mr > Suc 0)

fun wcode_goon_right_moving_2 :: bin_inv_t
 where
  wcode_goon_right_moving_2 ires rs (l, r) =
```

$(\exists\ ml\ mr\ ln\ rn.\ l = Oc\uparrow(ml)\ @\ Bk\ \#\ Bk\ \#\ Bk\uparrow(ln)\ @\ Oc\ \#\ ires \wedge$
$r = Oc\uparrow(mr)\ @\ Bk\uparrow(rn) \wedge ml + mr = Suc\ rs)$

fun *wcode_backto_standard_pos_2_B* :: *bin_inv_t*
where
wcode_backto_standard_pos_2_B ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l = Bk\ \#\ Bk\uparrow(ln)\ @\ Oc\ \#\ ires \wedge$
$r = Bk\ \#\ Oc\uparrow(Suc\ (Suc\ rs))\ @\ Bk\uparrow(rn))$

fun *wcode_backto_standard_pos_2_O* :: *bin_inv_t*
where
wcode_backto_standard_pos_2_O ires rs $(l, r) =$
$(\exists\ ml\ mr\ ln\ rn.\ l = Oc\uparrow(ml)@\ Bk\ \#\ Bk\ \#\ Bk\uparrow(ln)\ @\ Oc\ \#\ ires \wedge$
$r = Oc\uparrow(mr)\ @\ Bk\uparrow(rn) \wedge$
$ml + mr = (Suc\ (Suc\ rs)) \wedge mr > 0)$

fun *wcode_backto_standard_pos_2* :: *bin_inv_t*
where
wcode_backto_standard_pos_2 ires rs $(l, r) =$
$(wcode\_backto\_standard\_pos\_2\_O\ ires\ rs\ (l, r) \vee$
wcode_backto_standard_pos_2_B ires rs $(l, r))$

fun *wcode_before_fourtimes* :: *bin_inv_t*
where
wcode_before_fourtimes ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l = Bk\ \#\ Bk\ \#\ Bk\uparrow(ln)\ @\ Oc\ \#\ ires \wedge$
$r = Oc\uparrow(Suc\ (Suc\ rs))\ @\ Bk\uparrow(rn))$

declare *wcode_on_left_moving_2_B.simps*[*simp del*] *wcode_on_left_moving_2.simps*[*simp del*]
wcode_on_left_moving_2_O.simps[*simp del*] *wcode_on_checking_2.simps*[*simp del*]
wcode_goon_checking.simps[*simp del*] *wcode_right_move.simps*[*simp del*]
wcode_erase2.simps[*simp del*]
wcode_on_right_moving_2.simps[*simp del*] *wcode_goon_right_moving_2.simps*[*simp del*]
wcode_backto_standard_pos_2_B.simps[*simp del*] *wcode_backto_standard_pos_2_O.simps*[*simp del*]
wcode_backto_standard_pos_2.simps[*simp del*]

lemmas *wcode_fourtimes_invs* =
wcode_on_left_moving_2_B.simps wcode_on_left_moving_2.simps
wcode_on_left_moving_2_O.simps wcode_on_checking_2.simps
wcode_goon_checking.simps wcode_right_move.simps
wcode_erase2.simps
wcode_on_right_moving_2.simps wcode_goon_right_moving_2.simps
wcode_backto_standard_pos_2_B.simps wcode_backto_standard_pos_2_O.simps
wcode_backto_standard_pos_2.simps

fun *wcode_fourtimes_case_inv* :: *nat* $\Rightarrow$ *bin_inv_t*
where
wcode_fourtimes_case_inv st ires rs $(l, r) =$
$(if\ st = Suc\ 0\ then\ wcode\_on\_left\_moving\_2\ ires\ rs\ (l, r)$

else if st = Suc (Suc 0) then wcode_on_checking_2 ires rs (l, r)
else if st = 7 then wcode_goon_checking ires rs (l, r)
else if st = 8 then wcode_right_move ires rs (l, r)
else if st = 9 then wcode_erase2 ires rs (l, r)
else if st = 10 then wcode_on_right_moving_2 ires rs (l, r)
else if st = 11 then wcode_goon_right_moving_2 ires rs (l, r)
else if st = 12 then wcode_backto_standard_pos_2 ires rs (l, r)
else if st = t_twice_len + 14 then wcode_before_fourtimes ires rs (l, r)
else False)

declare *wcode_fourtimes_case_inv.simps*[*simp del*]

fun *wcode_fourtimes_case_state* :: *config* ⇒ *nat*
where
wcode_fourtimes_case_state (*st*, *l*, *r*) = *13* − *st*

fun *wcode_fourtimes_case_step* :: *config* ⇒ *nat*
where
wcode_fourtimes_case_step (*st*, *l*, *r*) =
(*if st = Suc 0 then length l*
else if st = 9 then
(*if hd r = Oc then 1*
else 0)
else if st = 10 then length r
else if st = 11 then length r
else if st = 12 then length l
else 0)

fun *wcode_fourtimes_case_measure* :: *config* ⇒ *nat* × *nat*
where
wcode_fourtimes_case_measure (*st*, *l*, *r*) =
(*wcode_fourtimes_case_state* (*st*, *l*, *r*),
wcode_fourtimes_case_step (*st*, *l*, *r*))

definition *wcode_fourtimes_case_le* :: (*config* × *config*) *set*
where *wcode_fourtimes_case_le* $\stackrel{def}{=}$ (*inv_image lex_pair wcode_fourtimes_case_measure*)

lemma *wf_wcode_fourtimes_case_le*[*intro*]: *wf wcode_fourtimes_case_le*
⟨*proof*⟩

lemma *nonempty_snd* [*simp*]:
wcode_on_left_moving_2 ires rs (*b*, []) = *False*
wcode_on_checking_2 ires rs (*b*, []) = *False*
wcode_goon_checking ires rs (*b*, []) = *False*
wcode_right_move ires rs (*b*, []) = *False*
wcode_erase2 ires rs (*b*, []) = *False*
wcode_on_right_moving_2 ires rs (*b*, []) = *False*
wcode_backto_standard_pos_2 ires rs (*b*, []) = *False*
wcode_on_checking_2 ires rs (*b*, *Oc* # *list*) = *False*

⟨*proof*⟩

lemma *wcode_on_left_moving_2*[*simp*]:
wcode_on_left_moving_2 ires rs (*b*, *Bk* # *list*) ⟹ *wcode_on_left_moving_2 ires rs* (*tl b*, *hd b* # *Bk* # *list*)
⟨*proof*⟩

lemma *wcode_goon_checking_via_2* [*simp*]: *wcode_on_checking_2 ires rs* (*b*, *Bk* # *list*)
⟹ *wcode_goon_checking ires rs* (*tl b*, *hd b* # *Bk* # *list*)
⟨*proof*⟩

lemma *wcode_erase2_via_move* [*simp*]: *wcode_right_move ires rs* (*b*, *Bk* # *list*) ⟹ *wcode_erase2 ires rs* (*Bk* # *b*, *list*)
⟨*proof*⟩

lemma *wcode_on_right_moving_2_via_erase2*[*simp*]:
wcode_erase2 ires rs (*b*, *Bk* # *list*) ⟹ *wcode_on_right_moving_2 ires rs* (*Bk* # *b*, *list*)
⟨*proof*⟩

lemma *wcode_on_right_moving_2_move_Bk*[*simp*]: *wcode_on_right_moving_2 ires rs* (*b*, *Bk* # *list*)
⟹ *wcode_on_right_moving_2 ires rs* (*Bk* # *b*, *list*)
⟨*proof*⟩

lemma *wcode_backto_standard_pos_2_via_right*[*simp*]:
wcode_goon_right_moving_2 ires rs (*b*, *Bk* # *list*) ⟹
wcode_backto_standard_pos_2 ires rs (*b*, *Oc* # *list*)
⟨*proof*⟩

lemma *wcode_on_checking_2_via_left*[*simp*]: *wcode_on_left_moving_2 ires rs* (*b*, *Oc* # *list*)
⟹
wcode_on_checking_2 ires rs (*tl b*, *hd b* # *Oc* # *list*)
⟨*proof*⟩

lemma *wcode_backto_standard_pos_2_empty_via_right*[*simp*]:
wcode_goon_right_moving_2 ires rs (*b*, []) ⟹
wcode_backto_standard_pos_2 ires rs (*b*, [*Oc*])
⟨*proof*⟩

lemma *wcode_goon_checking_cases*[*simp*]: *wcode_goon_checking ires rs* (*b*, *Oc* # *list*) ⟹
(*b* = [] ⟶ *wcode_right_move ires rs* ([*Oc*], *list*)) ∧
(*b* ≠ [] ⟶ *wcode_right_move ires rs* (*Oc* # *b*, *list*))
⟨*proof*⟩

lemma *wcode_right_move_no_Oc*[*simp*]: *wcode_right_move ires rs* (*b*, *Oc* # *list*) = *False*
⟨*proof*⟩

lemma *wcode_erase2_Bk_via_Oc*[*simp*]: *wcode_erase2 ires rs* (*b*, *Oc* # *list*)
⟹ *wcode_erase2 ires rs* (*b*, *Bk* # *list*)

⟨*proof*⟩

lemma *wcode_goon_right_moving_2_Oc_move*[*simp*]:
wcode_on_right_moving_2 ires rs (*b*, *Oc* # *list*)
⟹ *wcode_goon_right_moving_2 ires rs* (*Oc* # *b*, *list*)
⟨*proof*⟩

lemma *wcode_backto_standard_pos_2_exists*[*simp*]: *wcode_backto_standard_pos_2 ires rs* (*b*, *Bk* # *list*)
⟹ (∃ *ln*. *b* = *Bk* # *Bk*↑(*ln*) @ *Oc* # *ires*) ∧ (∃ *rn*. *list* = *Oc*↑(*Suc* (*Suc rs*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

lemma *wcode_goon_right_moving_2_move_Oc*[*simp*]: *wcode_goon_right_moving_2 ires rs* (*b*, *Oc* # *list*) ⟹
wcode_goon_right_moving_2 ires rs (*Oc* # *b*, *list*)
⟨*proof*⟩

lemma *wcode_backto_standard_pos_2_Oc_mv_hd*[*simp*]:
wcode_backto_standard_pos_2 ires rs (*b*, *Oc* # *list*)
⟹ *wcode_backto_standard_pos_2 ires rs* (*tl b*, *hd b* # *Oc* # *list*)
⟨*proof*⟩

lemma *nonempty_fst*[*simp*]:
wcode_on_left_moving_2 ires rs (*b*, *Bk* # *list*) ⟹ *b* ≠ []
wcode_on_checking_2 ires rs (*b*, *Bk* # *list*) ⟹ *b* ≠ []
wcode_goon_checking ires rs (*b*, *Bk* # *list*) = *False*
wcode_right_move ires rs (*b*, *Bk* # *list*) ⟹ *b*≠ []
wcode_erase2 ires rs (*b*, *Bk* # *list*) ⟹ *b* ≠ []
wcode_on_right_moving_2 ires rs (*b*, *Bk* # *list*) ⟹ *b* ≠ []
wcode_goon_right_moving_2 ires rs (*b*, *Bk* # *list*) ⟹ *b* ≠ []
wcode_backto_standard_pos_2 ires rs (*b*, *Bk* # *list*) ⟹ *b* ≠ []
wcode_on_left_moving_2 ires rs (*b*, *Oc* # *list*) ⟹ *b* ≠ []
wcode_goon_right_moving_2 ires rs (*b*, []) ⟹ *b* ≠ []
wcode_erase2 ires rs (*b*, *Oc* # *list*) ⟹ *b* ≠ []
wcode_on_right_moving_2 ires rs (*b*, *Oc* # *list*) ⟹ *b* ≠ []
wcode_goon_right_moving_2 ires rs (*b*, *Oc* # *list*) ⟹ *b* ≠ []
wcode_backto_standard_pos_2 ires rs (*b*, *Oc* # *list*) ⟹ *b* ≠ []
⟨*proof*⟩

lemma *wcode_fourtimes_case_first_correctness*:
shows *let P* = (λ (*st*, *l*, *r*). *st* = *t_twice_len* + *14*) *in*
let Q = (λ (*st*, *l*, *r*). *wcode_fourtimes_case_inv st ires rs* (*l*, *r*)) *in*
let f = (λ *stp*. *steps0* (*Suc 0*, *Bk* # *Bk*↑(*m*) @ *Oc* # *Bk* # *Oc* # *ires*, *Bk* # *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *t_wcode_main stp*) *in*
∃ *n* .*P* (*f n*) ∧ *Q* (*f* (*n*::*nat*))
⟨*proof*⟩

definition *t_fourtimes_len* :: *nat*

where
t_fourtimes_len = (*length t_fourtimes div 2*)

lemma *primerec_rec_fourtimes_1*[*intro*]: *primerec rec_fourtimes* (*Suc 0*)
⟨*proof*⟩

lemma *fourtimes_lemma*: *rec_exec rec_fourtimes* [*rs*] = *4* ∗ *rs*
⟨*proof*⟩

lemma *t_fourtimes_correct*:
∃ *stp ln rn*. *steps0* (*Suc 0*, *Bk* # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))
(*tm_of abc_fourtimes* @ *shift* (*mopup 1*) (*length* (*tm_of abc_fourtimes*) *div 2*)) *stp* =
(*0*, *Bk*↑(*ln*) @ *Bk* # *Bk* # *ires*, *Oc*↑(*Suc* (*4* ∗ *rs*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

lemma *wf_fourtimes*[*intro*]: *tm_wf* (*t_fourtimes_compile*, *0*)
⟨*proof*⟩

lemma *t_fourtimes_change_term_state*:
∃ *stp ln rn*. *steps0* (*Suc 0*, *Bk* # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *t_fourtimes stp*
= (*Suc t_fourtimes_len*, *Bk*↑(*ln*) @ *Bk* # *Bk* # *ires*, *Oc*↑(*Suc* (*4* ∗ *rs*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

lemma *length_t_twice_even*[*intro*]: *is_even* (*length t_twice*)
⟨*proof*⟩

lemma *t_fourtimes_append_pre*:
steps0 (*Suc 0*, *Bk* # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *t_fourtimes stp*
= (*Suc t_fourtimes_len*, *Bk*↑(*ln*) @ *Bk* # *Bk* # *ires*, *Oc*↑(*Suc* (*4* ∗ *rs*)) @ *Bk*↑(*rn*))
⟹ *steps0* (*Suc 0* + *length* (*t_wcode_main_first_part* @
shift t_twice (*length t_wcode_main_first_part div 2*) @ [(*L*, *1*), (*L*, *1*)]) *div 2*,
Bk # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))
((*t_wcode_main_first_part* @
shift t_twice (*length t_wcode_main_first_part div 2*) @ [(*L*, *1*), (*L*, *1*)]) @
shift t_fourtimes (*length* (*t_wcode_main_first_part* @
shift t_twice (*length t_wcode_main_first_part div 2*) @ [(*L*, *1*), (*L*, *1*)]) *div 2*) @ ([(*L*, *1*), (*L*,
1)])) *stp*
= ((*Suc t_fourtimes_len*) + *length* (*t_wcode_main_first_part* @
shift t_twice (*length t_wcode_main_first_part div 2*) @ [(*L*, *1*), (*L*, *1*)]) *div 2*,
Bk↑(*ln*) @ *Bk* # *Bk* # *ires*, *Oc*↑(*Suc* (*4* ∗ *rs*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

lemma *split_26_even*[*simp*]: (*26* + *l*::*nat*) *div 2* = *l div 2* + *13* ⟨*proof*⟩

lemma *t_twice_len_plust_14*[*simp*]: *t_twice_len* + *14* = *14* + *length* (*shift t_twice 12*) *div 2*
⟨*proof*⟩

lemma *t_fourtimes_append*:
∃ *stp ln rn*.
steps0 (*Suc 0* + *length* (*t_wcode_main_first_part* @ *shift t_twice*

(*length t_wcode_main_first_part div 2*) @ [(*L*, *1*), (*L*, *1*)]) *div 2*,
Bk # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))
((*t_wcode_main_first_part* @ *shift t_twice* (*length t_wcode_main_first_part div 2*) @
[(*L*, *1*), (*L*, *1*)]) @ *shift t_fourtimes* (*t_twice_len* + *13*) @ [(*L*, *1*), (*L*, *1*)]) *stp*
= (*Suc t_fourtimes_len* + *length* (*t_wcode_main_first_part* @ *shift t_twice*
(*length t_wcode_main_first_part div 2*) @ [(*L*, *1*), (*L*, *1*)]) *div 2*, *Bk*↑(*ln*) @ *Bk* # *Bk* # *ires*,
Oc↑(*Suc* (*4* ∗ *rs*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

lemma *even_fourtimes_len*: *length t_fourtimes mod 2* = *0*
⟨*proof*⟩

lemma *t_twice_even*[*simp*]: *2* ∗ (*length t_twice div 2*) = *length t_twice*
⟨*proof*⟩

lemma *t_fourtimes_even*[*simp*]: *2* ∗ (*length t_fourtimes div 2*) = *length t_fourtimes*
⟨*proof*⟩

lemma *fetch_t_wcode_14_Oc*: *fetch t_wcode_main* (*14* + *length t_twice div 2* + *t_fourtimes_len*)
Oc
= (*L*, *Suc 0*)
⟨*proof*⟩

lemma *fetch_t_wcode_14_Bk*: *fetch t_wcode_main* (*14* + *length t_twice div 2* + *t_fourtimes_len*)
Bk
= (*L*, *Suc 0*)
⟨*proof*⟩

lemma *fetch_t_wcode_14* [*simp*]: *fetch t_wcode_main* (*14* + *length t_twice div 2* + *t_fourtimes_len*)
b
= (*L*, *Suc 0*)
⟨*proof*⟩

lemma *wcode_jump2*:
∃ *stp ln rn*. *steps0* (*t_twice_len* + *14* + *t_fourtimes_len*
, *Bk* # *Bk* # *Bk*↑(*lnb*) @ *Oc* # *ires*, *Oc*↑(*Suc* (*4* ∗ *rs* + *4*)) @ *Bk*↑(*rnb*)) *t_wcode_main stp* =
(*Suc 0*, *Bk* # *Bk*↑(*ln*) @ *Oc* # *ires*, *Bk* # *Oc*↑(*Suc* (*4* ∗ *rs* + *4*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

lemma *wcode_fourtimes_case*:
shows ∃ *stp ln rn*.
steps0 (*Suc 0*, *Bk* # *Bk*↑(*m*) @ *Oc* # *Bk* # *Oc* # *ires*, *Bk* # *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))
t_wcode_main stp =
(*Suc 0*, *Bk* # *Bk*↑(*ln*) @ *Oc* # *ires*, *Bk* # *Oc*↑(*Suc* (*4*∗*rs* + *4*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

fun *wcode_on_left_moving_3_B* :: *bin_inv_t*
where
wcode_on_left_moving_3_B ires rs (*l*, *r*) =
(∃ *ml mr rn*. *l* = *Bk*↑(*ml*) @ *Oc* # *Bk* # *Bk* # *ires* ∧

```
                r = Bk↑(mr) @ Oc↑(Suc rs) @ Bk↑(rn) ∧
                ml + mr > Suc 0 ∧ mr > 0 )

fun wcode_on_left_moving_3_O :: bin_inv_t
 where
  wcode_on_left_moving_3_O ires rs (l, r) =
      (∃ ln rn. l = Bk # Bk # ires ∧
            r = Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun wcode_on_left_moving_3 :: bin_inv_t
 where
  wcode_on_left_moving_3 ires rs (l, r) =
    (wcode_on_left_moving_3_B ires rs (l, r) ∨
     wcode_on_left_moving_3_O ires rs (l, r))

fun wcode_on_checking_3 :: bin_inv_t
 where
  wcode_on_checking_3 ires rs (l, r) =
      (∃ ln rn. l = Bk # ires ∧
        r = Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun wcode_goon_checking_3 :: bin_inv_t
 where
  wcode_goon_checking_3 ires rs (l, r) =
      (∃ ln rn. l = ires ∧
        r = Bk # Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun wcode_stop :: bin_inv_t
 where
  wcode_stop ires rs (l, r) =
      (∃ ln rn. l = Bk # ires ∧
        r = Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun wcode_halt_case_inv :: nat ⇒ bin_inv_t
 where
  wcode_halt_case_inv st ires rs (l, r) =
      (if st = 0 then wcode_stop ires rs (l, r)
       else if st = Suc 0 then wcode_on_left_moving_3 ires rs (l, r)
       else if st = Suc (Suc 0) then wcode_on_checking_3 ires rs (l, r)
       else if st = 7 then wcode_goon_checking_3 ires rs (l, r)
       else False)

fun wcode_halt_case_state :: config ⇒ nat
 where
  wcode_halt_case_state (st, l, r) =
      (if st = 1 then 5
       else if st = Suc (Suc 0) then 4
       else if st = 7 then 3
       else 0)
```

fun *wcode_halt_case_step* :: *config* ⇒ *nat*
where
wcode_halt_case_step (*st*, *l*, *r*) =
(*if st* = *1 then length l*
else 0)

fun *wcode_halt_case_measure* :: *config* ⇒ *nat* × *nat*
where
wcode_halt_case_measure (*st*, *l*, *r*) =
(*wcode_halt_case_state* (*st*, *l*, *r*),
wcode_halt_case_step (*st*, *l*, *r*))

definition *wcode_halt_case_le* :: (*config* × *config*) *set*
where *wcode_halt_case_le* $\stackrel{def}{=}$ (*inv_image lex_pair wcode_halt_case_measure*)

lemma *wf_wcode_halt_case_le*[*intro*]: *wf wcode_halt_case_le*
⟨*proof*⟩

declare *wcode_on_left_moving_3_B.simps*[*simp del*] *wcode_on_left_moving_3_O.simps*[*simp del*]

wcode_on_checking_3.simps[*simp del*] *wcode_goon_checking_3.simps*[*simp del*]
wcode_on_left_moving_3.simps[*simp del*] *wcode_stop.simps*[*simp del*]

lemmas *wcode_halt_invs* =
wcode_on_left_moving_3_B.simps wcode_on_left_moving_3_O.simps
wcode_on_checking_3.simps wcode_goon_checking_3.simps
wcode_on_left_moving_3.simps wcode_stop.simps

lemma *wcode_on_left_moving_3_mv_Bk*[*simp*]: *wcode_on_left_moving_3 ires rs* (*b*, *Bk* # *list*)
⟹ *wcode_on_left_moving_3 ires rs* (*tl b*, *hd b* # *Bk* # *list*)
⟨*proof*⟩

lemma *wcode_goon_checking_3_cases*[*simp*]: *wcode_goon_checking_3 ires rs* (*b*, *Bk* # *list*)
⟹
(*b* = [] ⟶ *wcode_stop ires rs* ([*Bk*], *list*)) ∧
(*b* ≠ [] ⟶ *wcode_stop ires rs* (*Bk* # *b*, *list*))
⟨*proof*⟩

lemma *wcode_on_checking_3_mv_Oc*[*simp*]: *wcode_on_left_moving_3 ires rs* (*b*, *Oc* # *list*)
⟹
wcode_on_checking_3 ires rs (*tl b*, *hd b* # *Oc* # *list*)
⟨*proof*⟩

lemma *wcode_3_nonempty*[*simp*]:
wcode_on_left_moving_3 ires rs (*b*, []) = *False*
wcode_on_checking_3 ires rs (*b*, []) = *False*
wcode_goon_checking_3 ires rs (*b*, []) = *False*
wcode_on_left_moving_3 ires rs (*b*, *Oc* # *list*) ⟹ *b* ≠ []

wcode_on_checking_3 ires rs (b, Oc # list) = False
wcode_on_left_moving_3 ires rs (b, Bk # list) ⟹ b ≠ []
wcode_on_checking_3 ires rs (b, Bk # list) ⟹ b ≠ []
wcode_goon_checking_3 ires rs (b, Oc # list) = False
⟨*proof*⟩

lemma *wcode_goon_checking_3_mv_Bk*[*simp*]: *wcode_on_checking_3 ires rs (b, Bk # list)* ⟹
wcode_goon_checking_3 ires rs (tl b, hd b # Bk # list)
⟨*proof*⟩

lemma *t_halt_case_correctness*:
shows *let P = (λ (st, l, r). st = 0) in*
let Q = (λ (st, l, r). wcode_halt_case_inv st ires rs (l, r)) in
let f = (λ stp. steps0 (Suc 0, Bk # Bk↑(m) @ Oc # Bk # Bk # ires, Bk # Oc↑(Suc rs) @ Bk↑(n)) t_wcode_main stp) in
∃ *n .P (f n) ∧ Q (f (n::nat))*
⟨*proof*⟩

declare *wcode_halt_case_inv.simps*[*simp del*]
lemma *leading_Oc*[*intro*]: ∃ *xs*. (<*rev list* @ [*aa*::*nat*]> :: *cell list*) = *Oc* # *xs*
⟨*proof*⟩

lemma *wcode_halt_case*:
∃ *stp ln rn. steps0 (Suc 0, Bk # Bk↑(m) @ Oc # Bk # Bk # ires, Bk # Oc↑(Suc rs) @ Bk↑(n))*
t_wcode_main stp = *(0, Bk # ires, Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))*
⟨*proof*⟩

lemma *bl_bin_one*[*simp*]: *bl_bin [Oc] = 1*
⟨*proof*⟩

lemma *twice_power*[*intro*]: *2 * 2 ^ a = Suc (Suc (2 * bl_bin (Oc ↑ a)))*
⟨*proof*⟩
declare *replicate_Suc*[*simp del*]

lemma *t_wcode_main_lemma_pre*:
⟦*args ≠ []*; *lm* = <*args*::*nat list*>⟧ ⟹
∃ *stp ln rn. steps0 (Suc 0, Bk # Bk↑(m) @ rev lm @ Bk # Bk # ires, Bk # Oc↑(Suc rs) @ Bk↑(n)) t_wcode_main*
stp
= *(0, Bk # ires, Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(bl_bin lm + rs * 2^(length lm − 1))) @ Bk↑(rn))*
⟨*proof*⟩

definition *t_wcode_prepare* :: *instr list*
where
t_wcode_prepare $\stackrel{def}{=}$
[(W1, 2), (L, 1), (L, 3), (R, 2), (R, 4), (W0, 3),

```
    (R, 4), (R, 5), (R, 6), (R, 5), (R, 7), (R, 5),
    (W1, 7), (L, 0)]

fun wprepare_add_one :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_add_one m lm (l, r) =
   (∃ rn. l = [] ∧
        (r = <m # lm> @ Bk↑(rn) ∨
         r = Bk # <m # lm> @ Bk↑(rn)))

fun wprepare_goto_first_end :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_goto_first_end m lm (l, r) =
   (∃ ml mr rn. l = Oc↑(ml) ∧
              r = Oc↑(mr) @ Bk # <lm> @ Bk↑(rn) ∧
              ml + mr = Suc (Suc m))

fun wprepare_erase :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_erase m lm (l, r) =
   (∃ rn. l = Oc↑(Suc m) ∧
        tl r = Bk # <lm> @ Bk↑(rn))

fun wprepare_goto_start_pos_B :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_goto_start_pos_B m lm (l, r) =
   (∃ rn. l = Bk # Oc↑(Suc m) ∧
        r = Bk # <lm> @ Bk↑(rn))

fun wprepare_goto_start_pos_O :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_goto_start_pos_O m lm (l, r) =
   (∃ rn. l = Bk # Bk # Oc↑(Suc m) ∧
        r = <lm> @ Bk↑(rn))

fun wprepare_goto_start_pos :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_goto_start_pos m lm (l, r) =
    (wprepare_goto_start_pos_B m lm (l, r) ∨
     wprepare_goto_start_pos_O m lm (l, r))

fun wprepare_loop_start_on_rightmost :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_loop_start_on_rightmost m lm (l, r) =
   (∃ rn mr. rev l @ r = Oc↑(Suc m) @ Bk # Bk # <lm> @ Bk↑(rn) ∧ l ≠ [] ∧
             r = Oc↑(mr) @ Bk↑(rn))

fun wprepare_loop_start_in_middle :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_loop_start_in_middle m lm (l, r) =
```

```
   (∃ rn (mr:: nat) (lm1::nat list).
 rev l @ r = Oc↑(Suc m) @ Bk # Bk # <lm> @ Bk↑(rn) ∧ l ≠ [] ∧
 r = Oc↑(mr) @ Bk # <lm1> @ Bk↑(rn) ∧ lm1 ≠ [])

fun wprepare_loop_start :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_loop_start m lm (l, r) = (wprepare_loop_start_on_rightmost m lm (l, r) ∨
                                     wprepare_loop_start_in_middle m lm (l, r))

fun wprepare_loop_goon_on_rightmost :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_loop_goon_on_rightmost m lm (l, r) =
   (∃ rn. l = Bk # <rev lm> @ Bk # Bk # Oc↑(Suc m) ∧
         r = Bk↑(rn))

fun wprepare_loop_goon_in_middle :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_loop_goon_in_middle m lm (l, r) =
   (∃ rn (mr:: nat) (lm1::nat list).
 rev l @ r = Oc↑(Suc m) @ Bk # Bk # <lm> @ Bk↑(rn) ∧ l ≠ [] ∧
             (if lm1 = [] then r = Oc↑(mr) @ Bk↑(rn)
             else r = Oc↑(mr) @ Bk # <lm1> @ Bk↑(rn)) ∧ mr > 0)

fun wprepare_loop_goon :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_loop_goon m lm (l, r) =
        (wprepare_loop_goon_in_middle m lm (l, r) ∨
         wprepare_loop_goon_on_rightmost m lm (l, r))

fun wprepare_add_one2 :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_add_one2 m lm (l, r) =
      (∃ rn. l = Bk # Bk # <rev lm> @ Bk # Bk # Oc↑(Suc m) ∧
          (r = [] ∨ tl r = Bk↑(rn)))

fun wprepare_stop :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_stop m lm (l, r) =
      (∃ rn. l = Bk # <rev lm> @ Bk # Bk # Oc↑(Suc m) ∧
          r = Bk # Oc # Bk↑(rn))

fun wprepare_inv :: nat ⇒ nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_inv st m lm (l, r) =
     (if st = 0 then wprepare_stop m lm (l, r)
      else if st = Suc 0 then wprepare_add_one m lm (l, r)
      else if st = Suc (Suc 0) then wprepare_goto_first_end m lm (l, r)
      else if st = Suc (Suc (Suc 0)) then wprepare_erase m lm (l, r)
      else if st = 4 then wprepare_goto_start_pos m lm (l, r)
      else if st = 5 then wprepare_loop_start m lm (l, r)
```

```
    else if st = 6 then wprepare_loop_goon m lm (l, r)
    else if st = 7 then wprepare_add_one2 m lm (l, r)
    else False)

fun wprepare_stage :: config ⇒ nat
 where
  wprepare_stage (st, l, r) =
   (if st ≥ 1 ∧ st ≤ 4 then 3
    else if st = 5 ∨ st = 6 then 2
    else 1)

fun wprepare_state :: config ⇒ nat
 where
  wprepare_state (st, l, r) =
    (if st = 1 then 4
     else if st = Suc (Suc 0) then 3
     else if st = Suc (Suc (Suc 0)) then 2
     else if st = 4 then 1
     else if st = 7 then 2
     else 0)

fun wprepare_step :: config ⇒ nat
 where
  wprepare_step (st, l, r) =
   (if st = 1 then (if hd r = Oc then Suc (length l)
                    else 0)
    else if st = Suc (Suc 0) then length r
    else if st = Suc (Suc (Suc 0)) then (if hd r = Oc then 1
                                         else 0)
    else if st = 4 then length r
    else if st = 5 then Suc (length r)
    else if st = 6 then (if r = [] then 0 else Suc (length r))
    else if st = 7 then (if (r ≠ [] ∧ hd r = Oc) then 0
                         else 1)
    else 0)

fun wcode_prepare_measure :: config ⇒ nat × nat × nat
 where
  wcode_prepare_measure (st, l, r) =
   (wprepare_stage (st, l, r),
    wprepare_state (st, l, r),
    wprepare_step (st, l, r))

definition wcode_prepare_le :: (config × config) set
 where wcode_prepare_le ≡ (inv_image lex_triple wcode_prepare_measure)

lemma wf_wcode_prepare_le[intro]: wf wcode_prepare_le
 ⟨proof⟩
```

declare *wprepare_add_one.simps*[*simp del*] *wprepare_goto_first_end.simps*[*simp del*]
wprepare_erase.simps[*simp del*] *wprepare_goto_start_pos.simps*[*simp del*]
wprepare_loop_start.simps[*simp del*] *wprepare_loop_goon.simps*[*simp del*]
wprepare_add_one2.simps[*simp del*]

lemmas *wprepare_invs* = *wprepare_add_one.simps wprepare_goto_first_end.simps*
wprepare_erase.simps wprepare_goto_start_pos.simps
wprepare_loop_start.simps wprepare_loop_goon.simps
wprepare_add_one2.simps

declare *wprepare_inv.simps*[*simp del*]

lemma *fetch_t_wcode_prepare*[*simp*]:
fetch t_wcode_prepare (*Suc 0*) *Bk* = (*W1*, *2*)
fetch t_wcode_prepare (*Suc 0*) *Oc* = (*L*, *1*)
fetch t_wcode_prepare (*Suc* (*Suc 0*)) *Bk* = (*L*, *3*)
fetch t_wcode_prepare (*Suc* (*Suc 0*)) *Oc* = (*R*, *2*)
fetch t_wcode_prepare (*Suc* (*Suc* (*Suc 0*))) *Bk* = (*R*, *4*)
fetch t_wcode_prepare (*Suc* (*Suc* (*Suc 0*))) *Oc* = (*W0*, *3*)
fetch t_wcode_prepare 4 Bk = (*R*, *4*)
fetch t_wcode_prepare 4 Oc = (*R*, *5*)
fetch t_wcode_prepare 5 Oc = (*R*, *5*)
fetch t_wcode_prepare 5 Bk = (*R*, *6*)
fetch t_wcode_prepare 6 Oc = (*R*, *5*)
fetch t_wcode_prepare 6 Bk = (*R*, *7*)
fetch t_wcode_prepare 7 Oc = (*L*, *0*)
fetch t_wcode_prepare 7 Bk = (*W1*, *7*)
⟨*proof*⟩

lemma *wprepare_add_one_nonempty_snd*[*simp*]: *lm* ≠ [] ⟹ *wprepare_add_one m lm* (*b*, []) = *False*
⟨*proof*⟩

lemma *wprepare_goto_first_end_nonempty_snd*[*simp*]: *lm* ≠ [] ⟹ *wprepare_goto_first_end m lm* (*b*, []) = *False*
⟨*proof*⟩

lemma *wprepare_erase_nonempty_snd*[*simp*]: *lm* ≠ [] ⟹ *wprepare_erase m lm* (*b*, []) = *False*
⟨*proof*⟩

lemma *wprepare_goto_start_pos_nonempty_snd*[*simp*]: *lm* ≠ [] ⟹ *wprepare_goto_start_pos m lm* (*b*, []) = *False*
⟨*proof*⟩

lemma *wprepare_loop_start_empty_nonempty_fst*[*simp*]: ⟦*lm* ≠ []; *wprepare_loop_start m lm* (*b*, [])⟧ ⟹ *b* ≠ []
⟨*proof*⟩

lemma *rev_eq*: *rev xs* = *rev ys* ⟹ *xs* = *ys* ⟨*proof*⟩

lemma *wprepare_loop_goon_Bk_empty_snd*[*simp*]: ⟦*lm* ≠ []; *wprepare_loop_start m lm* (*b*, [])⟧
⟹
wprepare_loop_goon m lm (*Bk* # *b*, [])
⟨*proof*⟩

lemma *wprepare_loop_goon_nonempty_fst*[*simp*]: ⟦*lm* ≠ []; *wprepare_loop_goon m lm* (*b*, [])⟧
⟹ *b* ≠ []
⟨*proof*⟩

lemma *wprepare_add_one2_Bk_empty*[*simp*]:⟦*lm* ≠ []; *wprepare_loop_goon m lm* (*b*, [])⟧ ⟹
wprepare_add_one2 m lm (*Bk* # *b*, [])
⟨*proof*⟩

lemma *wprepare_add_one2_nonempty_fst*[*simp*]: *wprepare_add_one2 m lm* (*b*, []) ⟹ *b* ≠ []
⟨*proof*⟩

lemma *wprepare_add_one2_Oc*[*simp*]: *wprepare_add_one2 m lm* (*b*, []) ⟹ *wprepare_add_one2*
m lm (*b*, [*Oc*])
⟨*proof*⟩

lemma *Bk_not_tape_start*[*simp*]: (*Bk* # *list* = <(*m*::*nat*) # *lm*> @ *ys*) = *False*
⟨*proof*⟩

lemma *wprepare_goto_first_end_cases*[*simp*]:
⟦*lm* ≠ []; *wprepare_add_one m lm* (*b*, *Bk* # *list*)⟧
⟹ (*b* = [] ⟶ *wprepare_goto_first_end m lm* ([], *Oc* # *list*)) ∧
(*b* ≠ [] ⟶ *wprepare_goto_first_end m lm* (*b*, *Oc* # *list*))
⟨*proof*⟩

lemma *wprepare_goto_first_end_Bk_nonempty_fst*[*simp*]:
wprepare_goto_first_end m lm (*b*, *Bk* # *list*) ⟹ *b* ≠ []
⟨*proof*⟩

declare *replicate_Suc*[*simp*]

lemma *wprepare_erase_elem_Bk_rest*[*simp*]: *wprepare_goto_first_end m lm* (*b*, *Bk* # *list*) ⟹
wprepare_erase m lm (*tl b*, *hd b* # *Bk* # *list*)
⟨*proof*⟩

lemma *wprepare_erase_Bk_nonempty_fst*[*simp*]: *wprepare_erase m lm* (*b*, *Bk* # *list*) ⟹ *b* ≠
[]
⟨*proof*⟩

lemma *wprepare_goto_start_pos_Bk*[*simp*]: *wprepare_erase m lm* (*b*, *Bk* # *list*) ⟹
wprepare_goto_start_pos m lm (*Bk* # *b*, *list*)
⟨*proof*⟩

lemma *wprepare_add_one_Bk_nonempty_snd*[*simp*]: ⟦*wprepare_add_one m lm* (*b*, *Bk* # *list*)⟧
⟹ *list* ≠ []
⟨*proof*⟩

lemma *wprepare_goto_first_end_nonempty_snd_tl*[*simp*]:
⟦*lm* ≠ []; *wprepare_goto_first_end m lm* (*b*, *Bk* # *list*)⟧ ⟹ *list* ≠ []
⟨*proof*⟩

lemma *wprepare_erase_Bk_nonempty_list*[*simp*]: ⟦*lm* ≠ []; *wprepare_erase m lm* (*b*, *Bk* # *list*)⟧
⟹ *list* ≠ []
⟨*proof*⟩

lemma *wprepare_goto_start_pos_Bk_nonempty*[*simp*]: ⟦*lm* ≠ []; *wprepare_goto_start_pos m lm* (*b*, *Bk* # *list*)⟧ ⟹ *list* ≠ []
⟨*proof*⟩

lemma *wprepare_goto_start_pos_Bk_nonempty_fst*[*simp*]: ⟦*lm* ≠ []; *wprepare_goto_start_pos m lm* (*b*, *Bk* # *list*)⟧ ⟹ *b* ≠ []
⟨*proof*⟩

lemma *wprepare_loop_goon_Bk_nonempty*[*simp*]: ⟦*lm* ≠ []; *wprepare_loop_goon m lm* (*b*, *Bk* # *list*)⟧ ⟹ *b* ≠ []
⟨*proof*⟩

lemma *wprepare_loop_goon_wprepare_add_one2_cases*[*simp*]: ⟦*lm* ≠ []; *wprepare_loop_goon m lm* (*b*, *Bk* # *list*)⟧ ⟹
(*list* = [] ⟶ *wprepare_add_one2 m lm* (*Bk* # *b*, [])) ∧
(*list* ≠ [] ⟶ *wprepare_add_one2 m lm* (*Bk* # *b*, *list*))
⟨*proof*⟩

lemma *wprepare_add_one2_nonempty*[*simp*]: *wprepare_add_one2 m lm* (*b*, *Bk* # *list*) ⟹ *b* ≠ []
⟨*proof*⟩

lemma *wprepare_add_one2_cases*[*simp*]: *wprepare_add_one2 m lm* (*b*, *Bk* # *list*) ⟹
(*list* = [] ⟶ *wprepare_add_one2 m lm* (*b*, [*Oc*])) ∧
(*list* ≠ [] ⟶ *wprepare_add_one2 m lm* (*b*, *Oc* # *list*))
⟨*proof*⟩

lemma *wprepare_goto_first_end_cases_Oc*[*simp*]: *wprepare_goto_first_end m lm* (*b*, *Oc* # *list*)
⟹ (*b* = [] ⟶ *wprepare_goto_first_end m lm* ([*Oc*], *list*)) ∧
(*b* ≠ [] ⟶ *wprepare_goto_first_end m lm* (*Oc* # *b*, *list*))
⟨*proof*⟩

lemma *wprepare_erase_nonempty*[*simp*]: *wprepare_erase m lm* (*b*, *Oc* # *list*) ⟹ *b* ≠ []
⟨*proof*⟩

lemma *wprepare_erase_Bk*[*simp*]: *wprepare_erase m lm* (*b*, *Oc* # *list*)
⟹ *wprepare_erase m lm* (*b*, *Bk* # *list*)
⟨*proof*⟩

lemma *wprepare_goto_start_pos_Bk_move*[*simp*]: ⟦*lm* ≠ []; *wprepare_goto_start_pos m lm* (*b*,

Bk # list)⟧
⟹ *wprepare_goto_start_pos m lm* (*Bk # b*, *list*)
⟨*proof*⟩

lemma *wprepare_loop_start_b_nonempty*[*simp*]: *wprepare_loop_start m lm* (*b*, *aa*) ⟹ *b* ≠ []
⟨*proof*⟩
lemma *exists_exp_of_Bk*[*elim*]: *Bk # list* = *Oc*↑(*mr*) @ *Bk*↑(*rn*) ⟹ ∃ *rn*. *list* = *Bk*↑(*rn*)
⟨*proof*⟩

lemma *wprepare_loop_start_in_middle_Bk_False*[*simp*]: *wprepare_loop_start_in_middle m lm*
(*b*, [*Bk*]) = *False*
⟨*proof*⟩

declare *wprepare_loop_start_in_middle.simps*[*simp del*]

declare *wprepare_loop_start_on_rightmost.simps*[*simp del*]
wprepare_loop_goon_in_middle.simps[*simp del*]
wprepare_loop_goon_on_rightmost.simps[*simp del*]

lemma *wprepare_loop_goon_in_middle_Bk_False*[*simp*]: *wprepare_loop_goon_in_middle m lm*
(*Bk # b*, []) = *False*
⟨*proof*⟩

lemma *wprepare_loop_goon_Bk*[*simp*]: ⟦*lm* ≠ []; *wprepare_loop_start m lm* (*b*, [*Bk*])⟧ ⟹
wprepare_loop_goon m lm (*Bk # b*, [])
⟨*proof*⟩

lemma *wprepare_loop_goon_in_middle_Bk_False2*[*simp*]: *wprepare_loop_start_on_rightmost*
m lm (*b*, *Bk # a # lista*)
⟹ *wprepare_loop_goon_in_middle m lm* (*Bk # b*, *a # lista*) = *False*
⟨*proof*⟩

lemma *wprepare_loop_goon_on_rightbmost_Bk_False*[*simp*]: ⟦*lm* ≠ []; *wprepare_loop_start_on_rightmost*
m lm (*b*, *Bk # a # lista*)⟧
⟹ *wprepare_loop_goon_on_rightmost m lm* (*Bk # b*, *a # lista*)
⟨*proof*⟩

lemma *wprepare_loop_goon_in_middle_Bk_False3*[*simp*]:
assumes *lm* ≠ [] *wprepare_loop_start_in_middle m lm* (*b*, *Bk # a # lista*)
shows *wprepare_loop_goon_in_middle m lm* (*Bk # b*, *a # lista*) (**is** *?t1*)
and *wprepare_loop_goon_on_rightmost m lm* (*Bk # b*, *a # lista*) = *False* (**is** *?t2*)
⟨*proof*⟩

lemma *wprepare_loop_goon_Bk2*[*simp*]: ⟦*lm* ≠ []; *wprepare_loop_start m lm* (*b*, *Bk # a #*
lista)⟧ ⟹
wprepare_loop_goon m lm (*Bk # b*, *a # lista*)
⟨*proof*⟩

lemma *start_2_goon*:

⟦*lm* ≠ []; *wprepare_loop_start m lm* (*b*, *Bk* # *list*)⟧ ⟹
(*list* = [] ⟶ *wprepare_loop_goon m lm* (*Bk* # *b*, [])) ∧
(*list* ≠ [] ⟶ *wprepare_loop_goon m lm* (*Bk* # *b*, *list*))
⟨*proof*⟩

lemma *add_one_2_add_one*: *wprepare_add_one m lm* (*b*, *Oc* # *list*)
⟹ (*hd b* = *Oc* ⟶ (*b* = [] ⟶ *wprepare_add_one m lm* ([], *Bk* # *Oc* # *list*)) ∧
(*b* ≠ [] ⟶ *wprepare_add_one m lm* (*tl b*, *Oc* # *Oc* # *list*))) ∧
(*hd b* ≠ *Oc* ⟶ (*b* = [] ⟶ *wprepare_add_one m lm* ([], *Bk* # *Oc* # *list*)) ∧
(*b* ≠ [] ⟶ *wprepare_add_one m lm* (*tl b*, *hd b* # *Oc* # *list*)))
⟨*proof*⟩

lemma *wprepare_loop_start_on_rightmost_Oc*[*simp*]: *wprepare_loop_start_on_rightmost m lm*
(*b*, *Oc* # *list*) ⟹
wprepare_loop_start_on_rightmost m lm (*Oc* # *b*, *list*)
⟨*proof*⟩

lemma *wprepare_loop_start_in_middle_Oc*[*simp*]:
assumes *wprepare_loop_start_in_middle m lm* (*b*, *Oc* # *list*)
shows *wprepare_loop_start_in_middle m lm* (*Oc* # *b*, *list*)
⟨*proof*⟩

lemma *start_2_start*: *wprepare_loop_start m lm* (*b*, *Oc* # *list*) ⟹
wprepare_loop_start m lm (*Oc* # *b*, *list*)
⟨*proof*⟩

lemma *wprepare_loop_goon_Oc_nonempty*[*simp*]: *wprepare_loop_goon m lm* (*b*, *Oc* # *list*)
⟹ *b* ≠ []
⟨*proof*⟩

lemma *wprepare_goto_start_pos_Oc_nonempty*[*simp*]: *wprepare_goto_start_pos m lm* (*b*, *Oc*
list) ⟹ *b* ≠ []
⟨*proof*⟩

lemma *wprepare_loop_goon_on_rightmost_Oc_False*[*simp*]: *wprepare_loop_goon_on_rightmost*
m lm (*b*, *Oc* # *list*) = *False*
⟨*proof*⟩

lemma *wprepare_loop1*: ⟦*rev b* @ *Oc*↑(*mr*) = *Oc*↑(*Suc m*) @ *Bk* # *Bk* # <*lm*>;
b ≠ []; 0 < *mr*; *Oc* # *list* = *Oc*↑(*mr*) @ *Bk*↑(*rn*)⟧
⟹ *wprepare_loop_start_on_rightmost m lm* (*Oc* # *b*, *list*)
⟨*proof*⟩

lemma *wprepare_loop2*: ⟦*rev b* @ *Oc*↑(*mr*) @ *Bk* # <*a* # *lista*> = *Oc*↑(*Suc m*) @ *Bk* # *Bk* #
<*lm*>;
b ≠ []; *Oc* # *list* = *Oc*↑(*mr*) @ *Bk* # <(*a*::*nat*) # *lista*> @ *Bk*↑(*rn*)⟧
⟹ *wprepare_loop_start_in_middle m lm* (*Oc* # *b*, *list*)
⟨*proof*⟩

lemma *wprepare_loop_goon_in_middle_cases*[*simp*]: *wprepare_loop_goon_in_middle m lm* (*b*,

Oc # list) ⟹
wprepare_loop_start_on_rightmost m lm (*Oc # b*, *list*) ∨
wprepare_loop_start_in_middle m lm (*Oc # b*, *list*)
⟨*proof*⟩

lemma *wprepare_add_one_b*[*simp*]: *wprepare_add_one m lm* (*b*, *Oc # list*)
⟹ *b* = [] ⟶ *wprepare_add_one m lm* ([], *Bk # Oc # list*)
wprepare_loop_goon m lm (*b*, *Oc # list*)
⟹ *wprepare_loop_start m lm* (*Oc # b*, *list*)
⟨*proof*⟩

lemma *wprepare_loop_start_on_rightmost_Oc2*[*simp*]: *wprepare_goto_start_pos m* [*a*] (*b*, *Oc # list*)
⟹ *wprepare_loop_start_on_rightmost m* [*a*] (*Oc # b*, *list*)
⟨*proof*⟩

lemma *wprepare_loop_start_in_middle_2_Oc*[*simp*]: *wprepare_goto_start_pos m* (*a # aa # listaa*) (*b*, *Oc # list*)
⟹*wprepare_loop_start_in_middle m* (*a # aa # listaa*) (*Oc # b*, *list*)
⟨*proof*⟩

lemma *wprepare_loop_start_Oc2*[*simp*]: ⟦*lm* ≠ []; *wprepare_goto_start_pos m lm* (*b*, *Oc # list*)⟧
⟹ *wprepare_loop_start m lm* (*Oc # b*, *list*)
⟨*proof*⟩

lemma *wprepare_add_one2_Oc_nonempty*[*simp*]: *wprepare_add_one2 m lm* (*b*, *Oc # list*) ⟹ *b* ≠ []
⟨*proof*⟩

lemma *add_one_2_stop*:
wprepare_add_one2 m lm (*b*, *Oc # list*)
⟹ *wprepare_stop m lm* (*tl b*, *hd b # Oc # list*)
⟨*proof*⟩

declare *wprepare_stop.simps*[*simp del*]

lemma *wprepare_correctness*:
assumes *h*: *lm* ≠ []
shows *let P* = (λ (*st*, *l*, *r*). *st* = *0*) *in*
let Q = (λ (*st*, *l*, *r*). *wprepare_inv st m lm* (*l*, *r*)) *in*
let f = (λ *stp*. *steps0* (*Suc 0*, [], (<*m # lm*>)) *t_wcode_prepare stp*) *in*
∃ *n* .*P* (*f n*) ∧ *Q* (*f n*)
⟨*proof*⟩

lemma *tm_wf_t_wcode_prepare*[*intro*]: *tm_wf* (*t_wcode_prepare*, *0*)
⟨*proof*⟩

lemma *is_28_even*[*intro*]: (*28* + (*length t_twice_compile* + *length t_fourtimes_compile*)) *mod 2* = *0*

⟨*proof*⟩

lemma *b_le_28*[*elim*]: (*a*, *b*) ∈ *set t_wcode_main_first_part* ⟹
b ≤ (*28* + (*length t_twice_compile* + *length t_fourtimes_compile*)) *div 2*
⟨*proof*⟩

lemma *tm_wf_change_termi*:
assumes *tm_wf* (*tp*, *0*)
shows *list_all* (λ(*acn*, *st*). (*st* ≤ *Suc* (*length tp div 2*))) (*adjust0 tp*)
⟨*proof*⟩

lemma *tm_wf_shift*:
assumes *list_all* (λ(*acn*, *st*). (*st* ≤ *y*)) *tp*
shows *list_all* (λ(*acn*, *st*). (*st* ≤ *y* + *off*)) (*shift tp off*)
⟨*proof*⟩

declare *length_tp'*[*simp del*]

lemma *length_mopup_1*[*simp*]: *length* (*mopup* (*Suc 0*)) = *16*
⟨*proof*⟩

lemma *twice_plus_28_elim*[*elim*]: (*a*, *b*) ∈ *set* (*shift* (*adjust0 t_twice_compile*) *12*) ⟹
b ≤ (*28* + (*length t_twice_compile* + *length t_fourtimes_compile*)) *div 2*
⟨*proof*⟩

lemma *length_plus_28_elim2*[*elim*]: (*a*, *b*) ∈ *set* (*shift* (*adjust0 t_fourtimes_compile*) (*t_twice_len* + *13*))
⟹ *b* ≤ (*28* + (*length t_twice_compile* + *length t_fourtimes_compile*)) *div 2*
⟨*proof*⟩

lemma *tm_wf_t_wcode_main*[*intro*]: *tm_wf* (*t_wcode_main*, *0*)
⟨*proof*⟩

declare *tm_comp.simps*[*simp del*]

lemma *prepare_mainpart_lemma*:
args ≠ [] ⟹
∃ *stp ln rn*. *steps0* (*Suc 0*, [], <*m* # *args*>) (*t_wcode_prepare* |+| *t_wcode_main*) *stp*
= (*0*, *Bk* # *Oc*↑(*Suc m*), *Bk* # *Oc* # *Bk*↑(*ln*) @ *Bk* # *Bk* # *Oc*↑(*bl_bin* (<*args*>))
@ *Bk*↑(*rn*))
⟨*proof*⟩

definition *tinres* :: *cell list* ⇒ *cell list* ⇒ *bool*
where
tinres xs ys = (∃ *n*. *xs* = *ys* @ *Bk* ↑ *n* ∨ *ys* = *xs* @ *Bk* ↑ *n*)

lemma *tinres_fetch_congr*[*simp*]: *tinres r r'* ⟹
fetch t ss (*read r*) =

fetch t ss (read r′)
⟨*proof*⟩

lemma *nonempty_hd_tinres*[*simp*]: ⟦*tinres r r′*; *r* ≠ []; *r′* ≠ []⟧ ⟹ *hd r* = *hd r′*
⟨*proof*⟩

lemma *tinres_nonempty*[*simp*]:
⟦*tinres r* []; *r* ≠ []⟧ ⟹ *hd r* = *Bk*
⟦*tinres* [] *r′*; *r′* ≠ []⟧ ⟹ *hd r′* = *Bk*
⟦*tinres r* []; *r* ≠ []⟧ ⟹ *tinres* (*tl r*) []
tinres r r′ ⟹ *tinres* (*b* # *r*) (*b* # *r′*)
⟨*proof*⟩

lemma *ex_move_tl*[*intro*]: ∃ *na*. *tl r* = *tl* (*r* @ *Bk*↑(*n*)) @ *Bk*↑(*na*) ∨ *tl* (*r* @ *Bk*↑(*n*)) = *tl r* @ *Bk*↑(*na*)
⟨*proof*⟩

lemma *tinres_tails*[*simp*]: *tinres r r′* ⟹ *tinres* (*tl r*) (*tl r′*)
⟨*proof*⟩

lemma *tinres_empty*[*simp*]:
⟦*tinres* [] *r′*⟧ ⟹ *tinres* [] (*tl r′*)
tinres r [] ⟹ *tinres* (*Bk* # *tl r*) [*Bk*]
tinres r [] ⟹ *tinres* (*Oc* # *tl r*) [*Oc*]
⟨*proof*⟩

lemma *tinres_step2*:
assumes *tinres r r′ step0* (*ss*, *l*, *r*) *t* = (*sa*, *la*, *ra*) *step0* (*ss*, *l*, *r′*) *t* = (*sb*, *lb*, *rb*)
shows *la* = *lb* ∧ *tinres ra rb* ∧ *sa* = *sb*
⟨*proof*⟩

lemma *tinres_steps2*:
⟦*tinres r r′*; *steps0* (*ss*, *l*, *r*) *t stp* = (*sa*, *la*, *ra*); *steps0* (*ss*, *l*, *r′*) *t stp* = (*sb*, *lb*, *rb*)⟧
⟹ *la* = *lb* ∧ *tinres ra rb* ∧ *sa* = *sb*
⟨*proof*⟩

definition *t_wcode_adjust* :: *instr list*
where
t_wcode_adjust = [(*W1*, *1*), (*R*, *2*), (*Nop*, *2*), (*R*, *3*), (*R*, *3*), (*R*, *4*),
(*L*, *8*), (*L*, *5*), (*L*, *6*), (*W0*, *5*), (*L*, *6*), (*R*, *7*),
(*W1*, *2*), (*Nop*, *7*), (*L*, *9*), (*W0*, *8*), (*L*, *9*), (*L*, *10*),
(*L*, *11*), (*L*, *10*), (*R*, *0*), (*L*, *11*)]

lemma *fetch_t_wcode_adjust*[*simp*]:
fetch t_wcode_adjust (*Suc 0*) *Bk* = (*W1*, *1*)
fetch t_wcode_adjust (*Suc 0*) *Oc* = (*R*, *2*)
fetch t_wcode_adjust (*Suc* (*Suc 0*)) *Oc* = (*R*, *3*)
fetch t_wcode_adjust (*Suc* (*Suc* (*Suc 0*))) *Oc* = (*R*, *4*)
fetch t_wcode_adjust (*Suc* (*Suc* (*Suc 0*))) *Bk* = (*R*, *3*)

fetch t_wcode_adjust 4 Bk = (*L*, *8*)
fetch t_wcode_adjust 4 Oc = (*L*, *5*)
fetch t_wcode_adjust 5 Oc = (*W0*, *5*)
fetch t_wcode_adjust 5 Bk = (*L*, *6*)
fetch t_wcode_adjust 6 Oc = (*R*, *7*)
fetch t_wcode_adjust 6 Bk = (*L*, *6*)
fetch t_wcode_adjust 7 Bk = (*W1*, *2*)
fetch t_wcode_adjust 8 Bk = (*L*, *9*)
fetch t_wcode_adjust 8 Oc = (*W0*, *8*)
fetch t_wcode_adjust 9 Oc = (*L*, *10*)
fetch t_wcode_adjust 9 Bk = (*L*, *9*)
fetch t_wcode_adjust 10 Bk = (*L*, *11*)
fetch t_wcode_adjust 10 Oc = (*L*, *10*)
fetch t_wcode_adjust 11 Oc = (*L*, *11*)
fetch t_wcode_adjust 11 Bk = (*R*, *0*)
⟨*proof*⟩

fun *wadjust_start* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust_start m rs (*l*, *r*) =
(∃ *ln rn*. *l* = *Bk* # *Oc*↑(*Suc m*) ∧
tl r = *Oc* # *Bk*↑(*ln*) @ *Bk* # *Oc*↑(*Suc rs*) @ *Bk*↑(*rn*))

fun *wadjust_loop_start* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust_loop_start m rs (*l*, *r*) =
(∃ *ln rn ml mr*. *l* = *Oc*↑(*ml*) @ *Bk* # *Oc*↑(*Suc m*) ∧
r = *Oc* # *Bk*↑(*ln*) @ *Bk* # *Oc*↑(*mr*) @ *Bk*↑(*rn*) ∧
ml + *mr* = *Suc* (*Suc rs*) ∧ *mr* > *0*)

fun *wadjust_loop_right_move* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust_loop_right_move m rs (*l*, *r*) =
(∃ *ml mr nl nr rn*. *l* = *Bk*↑(*nl*) @ *Oc* # *Oc*↑(*ml*) @ *Bk* # *Oc*↑(*Suc m*) ∧
r = *Bk*↑(*nr*) @ *Oc*↑(*mr*) @ *Bk*↑(*rn*) ∧
ml + *mr* = *Suc* (*Suc rs*) ∧ *mr* > *0* ∧
nl + *nr* > *0*)

fun *wadjust_loop_check* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust_loop_check m rs (*l*, *r*) =
(∃ *ml mr ln rn*. *l* = *Oc* # *Bk*↑(*ln*) @ *Bk* # *Oc* # *Oc*↑(*ml*) @ *Bk* # *Oc*↑(*Suc m*) ∧
r = *Oc*↑(*mr*) @ *Bk*↑(*rn*) ∧ *ml* + *mr* = (*Suc rs*))

fun *wadjust_loop_erase* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust_loop_erase m rs (*l*, *r*) =
(∃ *ml mr ln rn*. *l* = *Bk*↑(*ln*) @ *Bk* # *Oc* # *Oc*↑(*ml*) @ *Bk* # *Oc*↑(*Suc m*) ∧
tl r = *Oc*↑(*mr*) @ *Bk*↑(*rn*) ∧ *ml* + *mr* = (*Suc rs*) ∧ *mr* > *0*)

```
fun wadjust_loop_on_left_moving_O :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_loop_on_left_moving_O m rs (l, r) =
   (∃ ml mr ln rn. l = Oc↑(ml) @ Bk # Oc↑(Suc m )∧
              r = Oc # Bk↑(ln) @ Bk # Bk # Oc↑(mr) @ Bk↑(rn) ∧
              ml + mr = Suc rs ∧ mr > 0)

fun wadjust_loop_on_left_moving_B :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_loop_on_left_moving_B m rs (l, r) =
   (∃ ml mr nl nr rn. l = Bk↑(nl) @ Oc # Oc↑(ml) @ Bk # Oc↑(Suc m) ∧
               r = Bk↑(nr) @ Bk # Bk # Oc↑(mr) @ Bk↑(rn) ∧
               ml + mr = Suc rs ∧ mr > 0)

fun wadjust_loop_on_left_moving :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_loop_on_left_moving m rs (l, r) =
    (wadjust_loop_on_left_moving_O m rs (l, r) ∨
    wadjust_loop_on_left_moving_B m rs (l, r))

fun wadjust_loop_right_move2 :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_loop_right_move2 m rs (l, r) =
     (∃ ml mr ln rn. l = Oc # Oc↑(ml) @ Bk # Oc↑(Suc m) ∧
               r = Bk↑(ln) @ Bk # Bk # Oc↑(mr) @ Bk↑(rn) ∧
               ml + mr = Suc rs ∧ mr > 0)

fun wadjust_erase2 :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_erase2 m rs (l, r) =
   (∃ ln rn. l = Bk↑(ln) @ Bk # Oc # Oc↑(Suc rs) @ Bk # Oc↑(Suc m) ∧
            tl r = Bk↑(rn))

fun wadjust_on_left_moving_O :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_on_left_moving_O m rs (l, r) =
     (∃ rn. l = Oc↑(Suc rs) @ Bk # Oc↑(Suc m) ∧
          r = Oc # Bk↑(rn))

fun wadjust_on_left_moving_B :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_on_left_moving_B m rs (l, r) =
     (∃ ln rn. l = Bk↑(ln) @ Oc # Oc↑(Suc rs) @ Bk # Oc↑(Suc m) ∧
          r = Bk↑(rn))

fun wadjust_on_left_moving :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_on_left_moving m rs (l, r) =
   (wadjust_on_left_moving_O m rs (l, r) ∨
```

```
    wadjust_on_left_moving_B m rs (l, r))

fun wadjust_goon_left_moving_B :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_goon_left_moving_B m rs (l, r) =
    (∃ rn. l = Oc↑(Suc m) ∧
        r = Bk # Oc↑(Suc (Suc rs)) @ Bk↑(rn))

fun wadjust_goon_left_moving_O :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_goon_left_moving_O m rs (l, r) =
   (∃ ml mr rn. l = Oc↑(ml) @ Bk # Oc↑(Suc m) ∧
            r = Oc↑(mr) @ Bk↑(rn) ∧
            ml + mr = Suc (Suc rs) ∧ mr > 0)

fun wadjust_goon_left_moving :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_goon_left_moving m rs (l, r) =
       (wadjust_goon_left_moving_B m rs (l, r) ∨
        wadjust_goon_left_moving_O m rs (l, r))

fun wadjust_backto_standard_pos_B :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_backto_standard_pos_B m rs (l, r) =
    (∃ rn. l = [] ∧
        r = Bk # Oc↑(Suc m )@ Bk # Oc↑(Suc (Suc rs)) @ Bk↑(rn))

fun wadjust_backto_standard_pos_O :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_backto_standard_pos_O m rs (l, r) =
   (∃ ml mr rn. l = Oc↑(ml) ∧
            r = Oc↑(mr) @ Bk # Oc↑(Suc (Suc rs)) @ Bk↑(rn) ∧
            ml + mr = Suc m ∧ mr > 0)

fun wadjust_backto_standard_pos :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_backto_standard_pos m rs (l, r) =
    (wadjust_backto_standard_pos_B m rs (l, r) ∨
    wadjust_backto_standard_pos_O m rs (l, r))

fun wadjust_stop :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_stop m rs (l, r) =
    (∃ rn. l = [Bk] ∧
        r = Oc↑(Suc m )@ Bk # Oc↑(Suc (Suc rs)) @ Bk↑(rn))

declare wadjust_start.simps[simp del]  wadjust_loop_start.simps[simp del]
 wadjust_loop_right_move.simps[simp del]  wadjust_loop_check.simps[simp del]
 wadjust_loop_erase.simps[simp del] wadjust_loop_on_left_moving.simps[simp del]
 wadjust_loop_right_move2.simps[simp del] wadjust_erase2.simps[simp del]
```

```
 wadjust_on_left_moving_O.simps[simp del] wadjust_on_left_moving_B.simps[simp del]
 wadjust_on_left_moving.simps[simp del] wadjust_goon_left_moving_B.simps[simp del]
 wadjust_goon_left_moving_O.simps[simp del] wadjust_goon_left_moving.simps[simp del]
 wadjust_backto_standard_pos.simps[simp del] wadjust_backto_standard_pos_B.simps[simp del]
 wadjust_backto_standard_pos_O.simps[simp del] wadjust_stop.simps[simp del]

fun wadjust_inv :: nat ⇒ nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_inv st m rs (l, r) =
    (if st = Suc 0 then wadjust_start m rs (l, r)
     else if st = Suc (Suc 0) then wadjust_loop_start m rs (l, r)
     else if st = Suc (Suc (Suc 0)) then wadjust_loop_right_move m rs (l, r)
     else if st = 4 then wadjust_loop_check m rs (l, r)
     else if st = 5 then wadjust_loop_erase m rs (l, r)
     else if st = 6 then wadjust_loop_on_left_moving m rs (l, r)
     else if st = 7 then wadjust_loop_right_move2 m rs (l, r)
     else if st = 8 then wadjust_erase2 m rs (l, r)
     else if st = 9 then wadjust_on_left_moving m rs (l, r)
     else if st = 10 then wadjust_goon_left_moving m rs (l, r)
     else if st = 11 then wadjust_backto_standard_pos m rs (l, r)
     else if st = 0 then wadjust_stop m rs (l, r)
     else False
)

declare wadjust_inv.simps[simp del]

fun wadjust_phase :: nat ⇒ config ⇒ nat
 where
  wadjust_phase rs (st, l, r) =
      (if st = 1 then 3
       else if st ≥ 2 ∧ st ≤ 7 then 2
       else if st ≥ 8 ∧ st ≤ 11 then 1
       else 0)

fun wadjust_stage :: nat ⇒ config ⇒ nat
 where
  wadjust_stage rs (st, l, r) =
       (if st ≥ 2 ∧ st ≤ 7 then
            rs − length (takeWhile (λ a. a = Oc)
                 (tl (dropWhile (λ a. a = Oc) (rev l @ r))))
        else 0)

fun wadjust_state :: nat ⇒ config ⇒ nat
 where
  wadjust_state rs (st, l, r) =
    (if st ≥ 2 ∧ st ≤ 7 then 8 − st
     else if st ≥ 8 ∧ st ≤ 11 then 12 − st
     else 0)

fun wadjust_step :: nat ⇒ config ⇒ nat
```

where
wadjust_step rs $(st, l, r) =$
$($*if st* $= 1$ *then* $($*if hd r* $=$ *Bk then 1*
else 0$)$
else if st $= 3$ *then length r*
else if st $= 5$ *then* $($*if hd r* $=$ *Oc then 1*
else 0$)$
else if st $= 6$ *then length l*
else if st $= 8$ *then* $($*if hd r* $=$ *Oc then 1*
else 0$)$
else if st $= 9$ *then length l*
else if st $= 10$ *then length l*
else if st $= 11$ *then* $($*if hd r* $=$ *Bk then 0*
else Suc $($*length l*$))$
else 0$)$

fun *wadjust_measure* :: $(nat \times config) \Rightarrow nat \times nat \times nat \times nat$
where
wadjust_measure $(rs, (st, l, r)) =$
$($*wadjust_phase rs* (st, l, r),
wadjust_stage rs (st, l, r),
wadjust_state rs (st, l, r),
wadjust_step rs $(st, l, r))$

definition *wadjust_le* :: $((nat \times config) \times nat \times config)$ *set*
where *wadjust_le* $\stackrel{def}{=}$ $($*inv_image lex_square wadjust_measure*$)$

lemma *wf_lex_square*[*intro*]: *wf lex_square*
⟨*proof*⟩

lemma *wf_wadjust_le*[*intro*]: *wf wadjust_le*
⟨*proof*⟩

lemma *wadjust_start_snd_nonempty*[*simp*]: *wadjust_start m rs* $(c, [])$ = *False*
⟨*proof*⟩

lemma *wadjust_loop_right_move_fst_nonempty*[*simp*]: *wadjust_loop_right_move m rs* $(c, [])$ $\Longrightarrow c \neq []$
⟨*proof*⟩

lemma *wadjust_loop_check_fst_nonempty*[*simp*]: *wadjust_loop_check m rs* $(c, []) \Longrightarrow c \neq []$
⟨*proof*⟩

lemma *wadjust_loop_start_snd_nonempty*[*simp*]: *wadjust_loop_start m rs* $(c, [])$ = *False*
⟨*proof*⟩

lemma *wadjust_erase2_singleton*[*simp*]: *wadjust_loop_check m rs* $(c, []) \Longrightarrow$ *wadjust_erase2 m rs* $(tl\ c, [hd\ c])$
⟨*proof*⟩

lemma *wadjust_loop_on_left_moving_snd_nonempty*[*simp*]:
wadjust_loop_on_left_moving m rs (*c*, []) = *False*
wadjust_loop_right_move2 m rs (*c*, []) = *False*
wadjust_erase2 m rs ([], []) = *False*
⟨*proof*⟩

lemma *wadjust_on_left_moving_B_Bk1*[*simp*]: *wadjust_on_left_moving_B m rs*
(*Oc* # *Oc* # *Oc*↑(*rs*) @ *Bk* # *Oc* # *Oc*↑(*m*), [*Bk*])
⟨*proof*⟩

lemma *wadjust_on_left_moving_B_Bk2*[*simp*]: *wadjust_on_left_moving_B m rs*
(*Bk*↑(*n*) @ *Bk* # *Oc* # *Oc* # *Oc*↑(*rs*) @ *Bk* # *Oc* # *Oc*↑(*m*), [*Bk*])
⟨*proof*⟩

lemma *wadjust_on_left_moving_singleton*[*simp*]: ⟦*wadjust_erase2 m rs* (*c*, []); *c* ≠ []⟧ ⟹
wadjust_on_left_moving m rs (*tl c*, [*hd c*]) ⟨*proof*⟩

lemma *wadjust_erase2_cases*[*simp*]: *wadjust_erase2 m rs* (*c*, [])
⟹ (*c* = [] ⟶ *wadjust_on_left_moving m rs* ([], [*Bk*])) ∧
(*c* ≠ [] ⟶ *wadjust_on_left_moving m rs* (*tl c*, [*hd c*]))
⟨*proof*⟩

lemma *wadjust_on_left_moving_nonempty*[*simp*]:
wadjust_on_left_moving m rs ([], []) = *False*
wadjust_on_left_moving_O m rs (*c*, []) = *False*
⟨*proof*⟩

lemma *wadjust_on_left_moving_B_singleton_Bk*[*simp*]:
⟦*wadjust_on_left_moving_B m rs* (*c*, []); *c* ≠ []; *hd c* = *Bk*⟧ ⟹
wadjust_on_left_moving_B m rs (*tl c*, [*Bk*])
⟨*proof*⟩

lemma *wadjust_on_left_moving_B_singleton_Oc*[*simp*]:
⟦*wadjust_on_left_moving_B m rs* (*c*, []); *c* ≠ []; *hd c* = *Oc*⟧ ⟹
wadjust_on_left_moving_O m rs (*tl c*, [*Oc*])
⟨*proof*⟩

lemma *wadjust_on_left_moving_singleton2*[*simp*]:
⟦*wadjust_on_left_moving m rs* (*c*, []); *c* ≠ []⟧ ⟹
wadjust_on_left_moving m rs (*tl c*, [*hd c*])
⟨*proof*⟩

lemma *wadjust_nonempty*[*simp*]: *wadjust_goon_left_moving m rs* (*c*, []) = *False*
wadjust_backto_standard_pos m rs (*c*, []) = *False*
⟨*proof*⟩

lemma *wadjust_loop_start_no_Bk*[*simp*]: *wadjust_loop_start m rs* (*c*, *Bk* # *list*) = *False*
⟨*proof*⟩

lemma *wadjust_loop_check_nonempty*[*simp*]: *wadjust_loop_check m rs* (*c*, *b*) ⟹ *c* ≠ []
⟨*proof*⟩

lemma *wadjust_erase2_via_loop_check_Bk*[*simp*]: *wadjust_loop_check m rs* (*c*, *Bk* # *list*)
⟹ *wadjust_erase2 m rs* (*tl c*, *hd c* # *Bk* # *list*)
⟨*proof*⟩

declare *wadjust_loop_on_left_moving_O.simps*[*simp del*]
wadjust_loop_on_left_moving_B.simps[*simp del*]

lemma *wadjust_loop_on_left_moving_B_via_erase*[*simp*]: ⟦*wadjust_loop_erase m rs* (*c*, *Bk* # *list*); *hd c* = *Bk*⟧
⟹ *wadjust_loop_on_left_moving_B m rs* (*tl c*, *Bk* # *Bk* # *list*)
⟨*proof*⟩

lemma *wadjust_loop_on_left_moving_O_Bk_via_erase*[*simp*]:
⟦*wadjust_loop_erase m rs* (*c*, *Bk* # *list*); *c* ≠ []; *hd c* = *Oc*⟧ ⟹
wadjust_loop_on_left_moving_O m rs (*tl c*, *Oc* # *Bk* # *list*)
⟨*proof*⟩

lemma *wadjust_loop_on_left_moving_Bk_via_erase*[*simp*]: ⟦*wadjust_loop_erase m rs* (*c*, *Bk* # *list*); *c* ≠ []⟧ ⟹
wadjust_loop_on_left_moving m rs (*tl c*, *hd c* # *Bk* # *list*)
⟨*proof*⟩

lemma *wadjust_loop_on_left_moving_B_Bk_move*[*simp*]:
⟦*wadjust_loop_on_left_moving_B m rs* (*c*, *Bk* # *list*); *hd c* = *Bk*⟧
⟹ *wadjust_loop_on_left_moving_B m rs* (*tl c*, *Bk* # *Bk* # *list*)
⟨*proof*⟩

lemma *wadjust_loop_on_left_moving_O_Oc_move*[*simp*]:
⟦*wadjust_loop_on_left_moving_B m rs* (*c*, *Bk* # *list*); *hd c* = *Oc*⟧
⟹ *wadjust_loop_on_left_moving_O m rs* (*tl c*, *Oc* # *Bk* # *list*)
⟨*proof*⟩

lemma *wadjust_loop_erase_nonempty*[*simp*]: *wadjust_loop_erase m rs* (*c*, *b*) ⟹ *c* ≠ []
wadjust_loop_on_left_moving m rs (*c*, *b*) ⟹ *c* ≠ []
wadjust_loop_right_move2 m rs (*c*, *b*) ⟹ *c* ≠ []
wadjust_erase2 m rs (*c*, *Bk* # *list*) ⟹ *c* ≠ []
wadjust_on_left_moving m rs (*c*,*b*) ⟹ *c* ≠ []
wadjust_on_left_moving_O m rs (*c*, *Bk* # *list*) = *False*
wadjust_goon_left_moving m rs (*c*, *b*) ⟹ *c* ≠ []
wadjust_loop_on_left_moving_O m rs (*c*, *Bk* # *list*) = *False*
⟨*proof*⟩

lemma *wadjust_loop_on_left_moving_Bk_move*[*simp*]:
wadjust_loop_on_left_moving m rs (*c*, *Bk* # *list*)
⟹ *wadjust_loop_on_left_moving m rs* (*tl c*, *hd c* # *Bk* # *list*)

⟨*proof*⟩

lemma *wadjust_loop_start_Oc_via_Bk_move*[*simp*]:
wadjust_loop_right_move2 m rs (*c*, *Bk* # *list*) ⟹ *wadjust_loop_start m rs* (*c*, *Oc* # *list*)
⟨*proof*⟩

lemma *wadjust_on_left_moving_Bk_via_erase*[*simp*]: *wadjust_erase2 m rs* (*c*, *Bk* # *list*) ⟹
wadjust_on_left_moving m rs (*tl c*, *hd c* # *Bk* # *list*)
⟨*proof*⟩

lemma *wadjust_on_left_moving_B_Bk_drop_one*: ⟦*wadjust_on_left_moving_B m rs* (*c*, *Bk* # *list*); *hd c* = *Bk*⟧
⟹ *wadjust_on_left_moving_B m rs* (*tl c*, *Bk* # *Bk* # *list*)
⟨*proof*⟩

lemma *wadjust_on_left_moving_B_Bk_drop_Oc*: ⟦*wadjust_on_left_moving_B m rs* (*c*, *Bk* # *list*); *hd c* = *Oc*⟧
⟹ *wadjust_on_left_moving_O m rs* (*tl c*, *Oc* # *Bk* # *list*)
⟨*proof*⟩

lemma *wadjust_on_left_moving_B_drop*[*simp*]: *wadjust_on_left_moving m rs* (*c*, *Bk* # *list*)
⟹
wadjust_on_left_moving m rs (*tl c*, *hd c* # *Bk* # *list*)
⟨*proof*⟩

lemma *wadjust_goon_left_moving_O_no_Bk*[*simp*]: *wadjust_goon_left_moving_O m rs* (*c*, *Bk* # *list*) = *False*
⟨*proof*⟩

lemma *wadjust_backto_standard_pos_via_left_Bk*[*simp*]:
wadjust_goon_left_moving m rs (*c*, *Bk* # *list*) ⟹
wadjust_backto_standard_pos m rs (*tl c*, *hd c* # *Bk* # *list*)
⟨*proof*⟩

lemma *wadjust_loop_right_move_Oc*[*simp*]:
wadjust_loop_start m rs (*c*, *Oc* # *list*) ⟹ *wadjust_loop_right_move m rs* (*Oc* # *c*, *list*)
⟨*proof*⟩

lemma *wadjust_loop_check_Oc*[*simp*]:
assumes *wadjust_loop_right_move m rs* (*c*, *Oc* # *list*)
shows *wadjust_loop_check m rs* (*Oc* # *c*, *list*)
⟨*proof*⟩

lemma *wadjust_loop_erase_move_Oc*[*simp*]: *wadjust_loop_check m rs* (*c*, *Oc* # *list*) ⟹
wadjust_loop_erase m rs (*tl c*, *hd c* # *Oc* # *list*)
⟨*proof*⟩

lemma *wadjust_loop_on_move_no_Oc*[*simp*]:
wadjust_loop_on_left_moving_B m rs (*c*, *Oc* # *list*) = *False*

wadjust_loop_right_move2 m rs (*c*, *Oc* # *list*) = *False*
wadjust_loop_on_left_moving m rs (*c*, *Oc* # *list*)
⟹ *wadjust_loop_right_move2 m rs* (*Oc* # *c*, *list*)
wadjust_on_left_moving_B m rs (*c*, *Oc* # *list*) = *False*
wadjust_loop_erase m rs (*c*, *Oc* # *list*) ⟹
wadjust_loop_erase m rs (*c*, *Bk* # *list*)
⟨*proof*⟩

lemma *wadjust_goon_left_moving_B_Bk_Oc*: ⟦*wadjust_on_left_moving_O m rs* (*c*, *Oc* # *list*); *hd c* = *Bk*⟧ ⟹
wadjust_goon_left_moving_B m rs (*tl c*, *Bk* # *Oc* # *list*)
⟨*proof*⟩

lemma *wadjust_goon_left_moving_O_Oc_Oc*: ⟦*wadjust_on_left_moving_O m rs* (*c*, *Oc* # *list*); *hd c* = *Oc*⟧
⟹ *wadjust_goon_left_moving_O m rs* (*tl c*, *Oc* # *Oc* # *list*)
⟨*proof*⟩

lemma *wadjust_goon_left_moving_Oc*[*simp*]: *wadjust_on_left_moving m rs* (*c*, *Oc* # *list*) ⟹
wadjust_goon_left_moving m rs (*tl c*, *hd c* # *Oc* # *list*)
⟨*proof*⟩

lemma *left_moving_Bk_Oc*[*simp*]: ⟦*wadjust_goon_left_moving_O m rs* (*c*, *Oc* # *list*); *hd c* = *Bk*⟧
⟹ *wadjust_goon_left_moving_B m rs* (*tl c*, *Bk* # *Oc* # *list*)
⟨*proof*⟩

lemma *left_moving_Oc_Oc*[*simp*]: ⟦*wadjust_goon_left_moving_O m rs* (*c*, *Oc* # *list*); *hd c* = *Oc*⟧ ⟹
wadjust_goon_left_moving_O m rs (*tl c*, *Oc* # *Oc* # *list*)
⟨*proof*⟩

lemma *wadjust_goon_left_moving_B_no_Oc*[*simp*]:
wadjust_goon_left_moving_B m rs (*c*, *Oc* # *list*) = *False*
⟨*proof*⟩

lemma *wadjust_goon_left_moving_Oc_move*[*simp*]: *wadjust_goon_left_moving m rs* (*c*, *Oc* # *list*) ⟹
wadjust_goon_left_moving m rs (*tl c*, *hd c* # *Oc* # *list*)
⟨*proof*⟩

lemma *wadjust_backto_standard_pos_B_no_Oc*[*simp*]:
wadjust_backto_standard_pos_B m rs (*c*, *Oc* # *list*) = *False*
⟨*proof*⟩

lemma *wadjust_backto_standard_pos_O_no_Bk*[*simp*]:
wadjust_backto_standard_pos_O m rs (*c*, *Bk* # *xs*) = *False*
⟨*proof*⟩

lemma *wadjust_backto_standard_pos_B_Bk_Oc*[*simp*]:
wadjust_backto_standard_pos_O m rs ([], *Oc* # *list*) ⟹
wadjust_backto_standard_pos_B m rs ([], *Bk* # *Oc* # *list*)
⟨*proof*⟩

lemma *wadjust_backto_standard_pos_B_Bk_Oc_via_O*[*simp*]:
⟦*wadjust_backto_standard_pos_O m rs* (*c*, *Oc* # *list*); *c* ≠ []; *hd c* = *Bk*⟧
⟹ *wadjust_backto_standard_pos_B m rs* (*tl c*, *Bk* # *Oc* # *list*)
⟨*proof*⟩

lemma *wadjust_backto_standard_pos_B_Oc_Oc_via_O*[*simp*]: ⟦*wadjust_backto_standard_pos_O m rs* (*c*, *Oc* # *list*); *c* ≠ []; *hd c* = *Oc*⟧
⟹ *wadjust_backto_standard_pos_O m rs* (*tl c*, *Oc* # *Oc* # *list*)
⟨*proof*⟩

lemma *wadjust_backto_standard_pos_cases*[*simp*]: *wadjust_backto_standard_pos m rs* (*c*, *Oc* # *list*)
⟹ (*c* = [] ⟶ *wadjust_backto_standard_pos m rs* ([], *Bk* # *Oc* # *list*)) ∧
(*c* ≠ [] ⟶ *wadjust_backto_standard_pos m rs* (*tl c*, *hd c* # *Oc* # *list*))
⟨*proof*⟩

lemma *wadjust_loop_right_move_nonempty_snd*[*simp*]: *wadjust_loop_right_move m rs* (*c*, []) = *False*
⟨*proof*⟩

lemma *wadjust_loop_erase_nonempty_snd*[*simp*]: *wadjust_loop_erase m rs* (*c*, []) = *False*
⟨*proof*⟩

lemma *wadjust_loop_erase_cases2*[*simp*]: ⟦*Suc* (*Suc rs*) = *a*; *wadjust_loop_erase m rs* (*c*, *Bk* # *list*)⟧
⟹ *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev* (*tl c*) @ *hd c* # *Bk* # *list*))))
< *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Bk* # *list*)))) ∨
a − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev* (*tl c*) @ *hd c* # *Bk* # *list*)))) =
a − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Bk* # *list*))))
⟨*proof*⟩

lemma *dropWhile_exp1*: *dropWhile* (λ*a*. *a* = *Oc*) (*Oc*↑(*n*) @ *xs*) = *dropWhile* (λ*a*. *a* = *Oc*) *xs*
⟨*proof*⟩
lemma *takeWhile_exp1*: *takeWhile* (λ*a*. *a* = *Oc*) (*Oc*↑(*n*) @ *xs*) = *Oc*↑(*n*) @ *takeWhile* (λ*a*. *a* = *Oc*) *xs*
⟨*proof*⟩

lemma *wadjust_correctness_helper_1*:
assumes *Suc* (*Suc rs*) = *a* *wadjust_loop_right_move2 m rs* (*c*, *Bk* # *list*)
shows *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Oc* # *list*))))
< *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Bk* # *list*))))
⟨*proof*⟩

lemma *wadjust_correctness_helper_2*:
⟦*Suc* (*Suc rs*) = *a*; *wadjust_loop_on_left_moving m rs* (*c*, *Bk* # *list*)⟧
⟹ *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev* (*tl c*) @ *hd c* # *Bk* # *list*))))
< *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Bk* # *list*)))) ∨
a − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev* (*tl c*) @ *hd c* # *Bk* # *list*)))) =
a − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Bk* # *list*))))
⟨*proof*⟩

lemma *wadjust_loop_check_empty_false*[*simp*]: *wadjust_loop_check m rs* ([], *b*) = *False*
⟨*proof*⟩

lemma *wadjust_loop_check_cases*: ⟦*Suc* (*Suc rs*) = *a*; *wadjust_loop_check m rs* (*c*, *Oc* # *list*)⟧
⟹ *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev* (*tl c*) @ *hd c* # *Oc* # *list*))))
< *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Oc* # *list*)))) ∨
a − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev* (*tl c*) @ *hd c* # *Oc* # *list*)))) =
a − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Oc* # *list*))))
⟨*proof*⟩

lemma *wadjust_loop_erase_cases_or*:
⟦*Suc* (*Suc rs*) = *a*; *wadjust_loop_erase m rs* (*c*, *Oc* # *list*)⟧
⟹ *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Bk* # *list*))))
< *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Oc* # *list*)))) ∨
a − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Bk* # *list*)))) =
a − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Oc* # *list*))))
⟨*proof*⟩

lemmas *wadjust_correctness_helpers* = *wadjust_correctness_helper_2 wadjust_correctness_helper_1 wadjust_loop_erase_cases_or wadjust_loop_check_cases*

declare *numeral_2_eq_2*[*simp del*]

lemma *wadjust_start_Oc*[*simp*]: *wadjust_start m rs* (*c*, *Bk* # *list*)
⟹ *wadjust_start m rs* (*c*, *Oc* # *list*)
⟨*proof*⟩

lemma *wadjust_stop_Bk*[*simp*]: *wadjust_backto_standard_pos m rs* (*c*, *Bk* # *list*)
⟹ *wadjust_stop m rs* (*Bk* # *c*, *list*)
⟨*proof*⟩

lemma *wadjust_loop_start_Oc*[*simp*]:
assumes *wadjust_start m rs* (*c*, *Oc* # *list*)
shows *wadjust_loop_start m rs* (*Oc* # *c*, *list*)
⟨*proof*⟩

lemma *erase2_Bk_if_Oc*[*simp*]: *wadjust_erase2 m rs* (*c*, *Oc* # *list*)

$\Longrightarrow$ *wadjust_erase2 m rs* $(c, Bk \# list)$
⟨*proof*⟩

lemma *wadjust_loop_right_move_Bk*[*simp*]: *wadjust_loop_right_move m rs* $(c, Bk \# list)$
$\Longrightarrow$ *wadjust_loop_right_move m rs* $(Bk \# c, list)$
⟨*proof*⟩

lemma *wadjust_correctness*:
shows *let P* $= (\lambda\ (len, st, l, r).\ st = 0)$ *in*
let Q $= (\lambda\ (len, st, l, r).$ *wadjust_inv st m rs* $(l, r))$ *in*
let f $= (\lambda$ *stp*. $(Suc\ (Suc\ rs),$ *steps0* $(Suc\ 0, Bk \# Oc\uparrow(Suc\ m),$
$Bk \# Oc \# Bk\uparrow(ln) \mathbin{@} Bk \# \ Oc\uparrow(Suc\ rs) \mathbin{@} Bk\uparrow(rn))$ *t_wcode_adjust stp*)) *in*
$\exists\ n\ .P\ (f\ n) \wedge Q\ (f\ n)$
⟨*proof*⟩

lemma *tm_wf_t_wcode_adjust*[*intro*]: *tm_wf* $(t\_wcode\_adjust, 0)$
⟨*proof*⟩

lemma *bl_bin_nonzero*[*simp*]: $args \neq [] \Longrightarrow bl\_bin\ (<args::nat\ list>) > 0$
⟨*proof*⟩

lemma *wcode_lemma_pre'*:
$args \neq [] \Longrightarrow$
$\exists$ *stp rn*. *steps0* $(Suc\ 0, [], <m \# args>)$
$((t\_wcode\_prepare\ |+|\ t\_wcode\_main)\ |+|\ t\_wcode\_adjust)$ *stp*
$= (0,\ [Bk],\ Oc\uparrow(Suc\ m) \mathbin{@} Bk \# Oc\uparrow(Suc\ (bl\_bin\ (<args>))) \mathbin{@} Bk\uparrow(rn))$
⟨*proof*⟩

The initialization TM *t_wcode*.

definition *t_wcode* :: *instr list*
where
t_wcode $= (t\_wcode\_prepare\ |+|\ t\_wcode\_main)\ |+|\ t\_wcode\_adjust$

The correctness of *t_wcode*.

lemma *wcode_lemma_1*:
$args \neq [] \Longrightarrow$
$\exists$ *stp ln rn*. *steps0* $(Suc\ 0, [], <m \# args>)$ (t_wcode) *stp* $=$
$(0,\ [Bk],\ Oc\uparrow(Suc\ m) \mathbin{@} Bk \# Oc\uparrow(Suc\ (bl\_bin\ (<args>))) \mathbin{@} Bk\uparrow(rn))$
⟨*proof*⟩

lemma *wcode_lemma*:
$args \neq [] \Longrightarrow$
$\exists$ *stp ln rn*. *steps0* $(Suc\ 0, [], <m \# args>)$ (t_wcode) *stp* $=$
$(0,\ [Bk],\ <[m, bl\_bin\ (<args>)]> \mathbin{@} Bk\uparrow(rn))$
⟨*proof*⟩

26 The universal TM

This section gives the explicit construction of *Universal Turing Machine*, defined as *UTM* and proves its correctness. It is pretty easy by composing the partial results we have got so far.

definition *UTM* :: *instr list*
 where
  UTM = (*let* (*aprog*, *rs_pos*, *a_md*) = *rec_ci rec_F in*
      let abc_F = *aprog* [+] *dummy_abc* (*Suc* (*Suc 0*)) *in*
       (*t_wcode* |+| (*tm_of abc_F* @ *shift* (*mopup* (*Suc* (*Suc 0*))) (*length* (*tm_of abc_F*) *div* *2*))))

definition *F_aprog* :: *abc_prog*
 where
  F_aprog $\stackrel{def}{=}$ (*let* (*aprog*, *rs_pos*, *a_md*) = *rec_ci rec_F in*
          aprog [+] *dummy_abc* (*Suc* (*Suc 0*)))

definition *F_tprog* :: *instr list*
 where
  F_tprog = *tm_of* (*F_aprog*)

definition *t_utm* :: *instr list*
 where
  t_utm $\stackrel{def}{=}$
   F_tprog @ *shift* (*mopup* (*Suc* (*Suc 0*))) (*length F_tprog div 2*)

definition *UTM_pre* :: *instr list*
 where
  UTM_pre = *t_wcode* |+| *t_utm*

lemma *tinres_step1*:
 assumes *tinres l l' step* (*ss*, *l*, *r*) (*t*, *0*) = (*sa*, *la*, *ra*)
  step (*ss*, *l'*, *r*) (*t*, *0*) = (*sb*, *lb*, *rb*)
 shows *tinres la lb* ∧ *ra* = *rb* ∧ *sa* = *sb*
⟨*proof*⟩

lemma *tinres_steps1*:
 ⟦*tinres l l'*; *steps* (*ss*, *l*, *r*) (*t*, *0*) *stp* = (*sa*, *la*, *ra*);
        steps (*ss*, *l'*, *r*) (*t*, *0*) *stp* = (*sb*, *lb*, *rb*)⟧
  ⟹ *tinres la lb* ∧ *ra* = *rb* ∧ *sa* = *sb*
⟨*proof*⟩

lemma *tinres_some_exp*[*simp*]:
 tinres (*Bk* ↑ *m* @ [*Bk*, *Bk*]) *la* ⟹ ∃ *m*. *la* = *Bk* ↑ *m* ⟨*proof*⟩

lemma *t_utm_halt_eq*:
 assumes *tm_wf*: *tm_wf* (*tp*, *0*)
  and *exec*: *steps0* (*Suc 0*, *Bk*↑(*l*), <*lm*::*nat list*>) *tp stp* = (*0*, *Bk*↑(*m*), *Oc*↑(*rs*)@*Bk*↑(*n*))
  and *resutl*: *0* < *rs*

shows $\exists$ *stp m n. steps0* (*Suc 0*, [*Bk*], <[*code tp*, *bl2wc* (<*lm*>)]> @ *Bk*↑(*i*)) *t_utm stp* =
(*0*, *Bk*↑(*m*), *Oc*↑(*rs*) @ *Bk*↑(*n*))
⟨*proof*⟩

lemma *tm_wf_t_wcode*[*intro*]: *tm_wf* (*t_wcode*, *0*)
⟨*proof*⟩

lemma *UTM_halt_lemma_pre*:
assumes *wf_tm*: *tm_wf* (*tp*, *0*)
and *result*: *0* < *rs*
and *args*: *args* ≠ []
and *exec*: *steps0* (*Suc 0*, *Bk*↑(*i*), <*args*::*nat list*>) *tp stp* = (*0*, *Bk*↑(*m*), *Oc*↑(*rs*)@*Bk*↑(*k*))
shows $\exists$ *stp m n. steps0* (*Suc 0*, [], <*code tp* # *args*>) *UTM_pre stp* =
(*0*, *Bk*↑(*m*), *Oc*↑(*rs*) @ *Bk*↑(*n*))
⟨*proof*⟩

The correctness of *UTM*, the halt case.

lemma *UTM_halt_lemma′*:
assumes *tm_wf*: *tm_wf* (*tp*, *0*)
and *result*: *0* < *rs*
and *args*: *args* ≠ []
and *exec*: *steps0* (*Suc 0*, *Bk*↑(*i*), <*args*::*nat list*>) *tp stp* = (*0*, *Bk*↑(*m*), *Oc*↑(*rs*)@*Bk*↑(*k*))
shows $\exists$ *stp m n. steps0* (*Suc 0*, [], <*code tp* # *args*>) *UTM stp* =
(*0*, *Bk*↑(*m*), *Oc*↑(*rs*) @ *Bk*↑(*n*))
⟨*proof*⟩

definition *TSTD*:: *config* ⇒ *bool*
where
TSTD c = (*let* (*st*, *l*, *r*) = *c in*
st = *0* ∧ ($\exists$ *m*. *l* = *Bk*↑(*m*)) ∧ ($\exists$ *rs n*. *r* = *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)))

lemma *nstd_case1*: *0* < *a* ⟹ *NSTD* (*trpl_code* (*a*, *b*, *c*))
⟨*proof*⟩

lemma *nonzero_bl2wc*[*simp*]: ∀ *m*. *b* ≠ *Bk*↑(*m*) ⟹ *0* < *bl2wc b*
⟨*proof*⟩

lemma *nstd_case2*: ∀ *m*. *b* ≠ *Bk*↑(*m*) ⟹ *NSTD* (*trpl_code* (*a*, *b*, *c*))
⟨*proof*⟩

lemma *even_not_odd*[*elim*]: *Suc* (*2* ∗ *x*) = *2* ∗ *y* ⟹ *RR*
⟨*proof*⟩

declare *replicate_Suc*[*simp del*]

lemma *bl2nat_zero_eq*[*simp*]: (*bl2nat c 0* = *0*) = ($\exists$ *n*. *c* = *Bk*↑(*n*))
⟨*proof*⟩

lemma *bl2wc_exp_ex*:
⟦*Suc* (*bl2wc c*) = *2* ^ *m*⟧ ⟹ $\exists$ *rs n*. *c* = *Oc*↑(*rs*) @ *Bk*↑(*n*)

⟨*proof*⟩

lemma *lg_bin*:
assumes ∀ *rs n*. *c* ≠ *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)
bl2wc c = 2 ^ *lg* (*Suc* (*bl2wc c*)) 2 − *Suc 0*
shows *bl2wc c* = *0*
⟨*proof*⟩

lemma *nstd_case3*:
∀ *rs n*. *c* ≠ *Oc*↑(*Suc rs*) @ *Bk*↑(*n*) ⟹ *NSTD* (*trpl_code* (*a*, *b*, *c*))
⟨*proof*⟩

lemma *NSTD_1*: ¬ *TSTD* (*a*, *b*, *c*)
⟹ *rec_exec rec_NSTD* [*trpl_code* (*a*, *b*, *c*)] = *Suc 0*
⟨*proof*⟩

lemma *nonstop_t_uhalt_eq*:
⟦*tm_wf* (*tp*, *0*);
steps0 (*Suc 0*, *Bk*↑(*l*), <*lm*>) *tp stp* = (*a*, *b*, *c*);
¬ *TSTD* (*a*, *b*, *c*)⟧
⟹ *rec_exec rec_nonstop* [*code tp*, *bl2wc* (<*lm*>), *stp*] = *Suc 0*
⟨*proof*⟩

lemma *nonstop_true*:
⟦*tm_wf* (*tp*, *0*);
∀ *stp*. (¬ *TSTD* (*steps0* (*Suc 0*, *Bk*↑(*l*), <*lm*>) *tp stp*))⟧
⟹ ∀ *y*. *rec_exec rec_nonstop* ([*code tp*, *bl2wc* (<*lm*>), *y*]) = (*Suc 0*)
⟨*proof*⟩

lemma *cn_arity*: *rec_ci* (*Cn n f gs*) = (*a*, *b*, *c*) ⟹ *b* = *n*
⟨*proof*⟩

lemma *mn_arity*: *rec_ci* (*Mn n f*) = (*a*, *b*, *c*) ⟹ *b* = *n*
⟨*proof*⟩

lemma *F_aprog_uhalt*:
assumes *wf_tm*: *tm_wf* (*tp*,*0*)
and *unhalt*: ∀ *stp*. (¬ *TSTD* (*steps0* (*Suc 0*, *Bk*↑(*l*), <*lm*>) *tp stp*))
and *compile*: *rec_ci rec_F* = (*F_ap*, *rs_pos*, *a_md*)
shows {λ *nl*. *nl* = [*code tp*, *bl2wc* (<*lm*>)] @ *0*↑(*a_md* − *rs_pos*) @ *suflm*} (*F_ap*) ↑
⟨*proof*⟩

lemma *uabc_uhalt′*:
⟦*tm_wf* (*tp*, *0*);
∀ *stp*. (¬ *TSTD* (*steps0* (*Suc 0*, *Bk*↑(*l*), <*lm*>) *tp stp*));
rec_ci rec_F = (*ap*, *pos*, *md*)⟧
⟹ {λ *nl*. *nl* = [*code tp*, *bl2wc* (<*lm*>)]} *ap* ↑
⟨*proof*⟩

lemma *uabc_uhalt*:

⟦*tm_wf* (*tp*, *0*);
∀ *stp*. (¬ *TSTD* (*steps0* (*Suc 0*, *Bk*↑(*l*), <*lm*>) *tp stp*))⟧
⟹ {λ *nl*. *nl* = [*code tp*, *bl2wc* (<*lm*>)]} *F_aprog* ↑
⟨*proof*⟩

lemma *tutm_uhalt'*:
assumes *tm_wf*: *tm_wf* (*tp*,*0*)
and *unhalt*: ∀ *stp*. (¬ *TSTD* (*steps0* (*1*, *Bk*↑(*l*), <*lm*>) *tp stp*))
shows ∀ *stp*. ¬ *is_final* (*steps0* (*1*, [*Bk*, *Bk*], <[*code tp*, *bl2wc* (<*lm*>)]>) *t_utm stp*)
⟨*proof*⟩

lemma *tinres_commute*: *tinres r r'* ⟹ *tinres r' r*
⟨*proof*⟩

lemma *inres_tape*:
⟦*steps0* (*st*, *l*, *r*) *tp stp* = (*a*, *b*, *c*); *steps0* (*st*, *l'*, *r'*) *tp stp* = (*a'*, *b'*, *c'*);
tinres l l'; *tinres r r'*⟧
⟹ *a* = *a'* ∧ *tinres b b'* ∧ *tinres c c'*
⟨*proof*⟩

lemma *tape_normalize*:
assumes ∀ *stp*. ¬ *is_final*(*steps0* (*Suc 0*, [*Bk*,*Bk*], <[*code tp*, *bl2wc* (<*lm*>)]>) *t_utm stp*)
shows ∀ *stp*. ¬ *is_final* (*steps0* (*Suc 0*, *Bk*↑(*m*), <[*code tp*, *bl2wc* (<*lm*>)]> @ *Bk*↑(*n*)) *t_utm stp*)
(**is** ∀ *stp*. *?P stp*)
⟨*proof*⟩

lemma *tutm_uhalt*:
⟦*tm_wf* (*tp*,*0*);
∀ *stp*. (¬ *TSTD* (*steps0* (*Suc 0*, *Bk*↑(*l*), <*args*>) *tp stp*))⟧
⟹ ∀ *stp*. ¬ *is_final* (*steps0* (*Suc 0*, *Bk*↑(*m*), <[*code tp*, *bl2wc* (<*args*>)]> @ *Bk*↑(*n*)) *t_utm stp*)
⟨*proof*⟩

lemma *UTM_uhalt_lemma_pre*:
assumes *tm_wf*: *tm_wf* (*tp*, *0*)
and *exec*: ∀ *stp*. (¬ *TSTD* (*steps0* (*Suc 0*, *Bk*↑(*l*), <*args*>) *tp stp*))
and *args*: *args* ≠ []
shows ∀ *stp*. ¬ *is_final* (*steps0* (*Suc 0*, [], <*code tp* # *args*>) *UTM_pre stp*)
⟨*proof*⟩

The correctness of *UTM*, the unhalt case.

lemma *UTM_uhalt_lemma'*:
assumes *tm_wf*: *tm_wf* (*tp*, *0*)
and *unhalt*: ∀ *stp*. (¬ *TSTD* (*steps0* (*Suc 0*, *Bk*↑(*l*), <*args*>) *tp stp*))
and *args*: *args* ≠ []
shows ∀ *stp*. ¬ *is_final* (*steps0* (*Suc 0*, [], <*code tp* # *args*>) *UTM stp*)
⟨*proof*⟩

lemma *UTM_halt_lemma*:

assumes *tm_wf*: *tm_wf* (*p*, *0*)
and *resut*: *rs* > *0*
and *args*: (*args*::*nat list*) ≠ []
and *exec*: {(λ*tp*. *tp* = (*Bk*↑*i*, <*args*>))} *p* {(λ*tp*. *tp* = (*Bk*↑*m*, *Oc*↑*rs* @ *Bk*↑*k*))}
shows {(λ*tp*. *tp* = ([], <*code p* # *args*>))} *UTM* {(λ*tp*. (∃ *m n*. *tp* = (*Bk*↑*m*, *Oc*↑*rs* @ *Bk*↑*n*)))}
⟨*proof*⟩

lemma *UTM_halt_lemma2*:
assumes *tm_wf*: *tm_wf* (*p*, *0*)
and *args*: (*args*::*nat list*) ≠ []
and *exec*: {(λ*tp*. *tp* = ([], <*args*>))} *p* {(λ*tp*. *tp* = (*Bk*↑*m*, <(*n*::*nat*)> @ *Bk*↑*k*))}
shows {(λ*tp*. *tp* = ([], <*code p* # *args*>))} *UTM* {(λ*tp*. (∃ *m k*. *tp* = (*Bk*↑*m*, <*n*> @ *Bk*↑*k*)))}
⟨*proof*⟩

lemma *UTM_unhalt_lemma*:
assumes *tm_wf*: *tm_wf* (*p*, *0*)
and *unhalt*: {(λ*tp*. *tp* = (*Bk*↑*i*, <*args*>))} *p* ↑
and *args*: *args* ≠ []
shows {(λ*tp*. *tp* = ([], <*code p* # *args*>))} *UTM* ↑
⟨*proof*⟩

lemma *UTM_unhalt_lemma2*:
assumes *tm_wf*: *tm_wf* (*p*, *0*)
and *unhalt*: {(λ*tp*. *tp* = ([], <*args*>))} *p* ↑
and *args*: *args* ≠ []
shows {(λ*tp*. *tp* = ([], <*code p* # *args*>))} *UTM* ↑
⟨*proof*⟩

end

References

[1] G. Boolos, J. P. Burgess, and R. C. Jeffrey. *Computability and Logic (5th ed.)*. Cambridge University Press, 2007.

[2] B. Felgenhauer. Minsky machines. *Archive of Formal Proofs*, Aug. 2018. http://isa-afp.org/entries/Minsky_Machines.html, Formal proof development.

[3] S. J. Joosten. Finding models through graph saturation. *Journal of Logical and Algebraic Methods in Programming*, 100:98 – 112, 2018.

[4] S. J. C. Joosten. Graph saturation. *Archive of Formal Proofs*, Nov. 2018. http://isa-afp.org/entries/Graph_Saturation.html, Formal proof development.

[5] M. Nedzelsky. Recursion theory i. *Archive of Formal Proofs*, Apr. 2008. http://isa-afp.org/entries/Recursion-Theory-I.html, Formal proof development.

[6] J. Xu, X. Zhang, and C. Urban. Mechanising turing machines and computability theory in isabelle/hol. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, pages 147–162, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.