Universal Turing Machine and Computability Theory in Isabelle/HOL

Jian Xu[2] Xingyuan Zhang[2] Christian Urban[1]
Sebastiaan J. C. Joosten[3]
Franz A. B. Regensburger[4]

[1]King's College London, UK
[2]PLA University of Science and Technology, China
[3]University of Twente, the Netherlands
[4]Technische Hochschule Ingolstadt, Germany

October 13, 2025

Abstract

We formalise results from computability theory: Turing decidability, Turing computability, reduction of decision problems, recursive functions, undecidability of the special and the general halting problem, and the existence of a universal Turing machine. This formalisation extends the original AFP entry of 2014 that corresponded to: Mechanising Turing Machines and Computability Theory in Isabelle/HOL, ITP 2013

The AFP entry and by extension this document is largely written by Xu, Zhang and Urban. The Universal Turing Machine is explained in this document, starting at Figure 6.1. You may want to also consult the original ITP article [6]. If you are interested in results about Turing Machines and Computability theory: the main book used for this formalisation is by Boolos, Burgess and Jeffrey [1].

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 Felgenhauer [2], using a definition of computably enumerable sets formalised by Nedzelsky [5]. This entry establishes the equivalence of these entirely separate notions of computability, but decidability remains future work.

In 2022, Regensburger contributed by adding definitions for concepts like Turing Decidability, Turing Computability and Turing Reducibility for problem reduction. He also enhanced the result about the undecidability of the General Halting Problem given in the original AFP entry by first proving the undecidability of the Special Halting Problem and then proving its reducibility to the general problem. The original version of this AFP entry did only prove a weak form of the undecidability theorem. The main motivation behind this contribution is to make the AFP entry accessible for bachelor and master students.

As a result, the presentation of the first chapter about Turing Machines has been considerably restructured and, in this context some minor changes in the naming of concepts were performed as well. In the rest of the theories the sectioning of the LaTeX document was improved. The overall contribution approximately doubled the size of the code base. Please refer to the CHANGELOG in the AFP entry for more details.

Chapter 1

Turing Machines

theory *Turing*
 imports *Main*
begin

1.1 Some lemmas about natural numbers used for rewriting

lemma *numeral_4_eq_4*: $4 = Suc\ 3$
 $\langle proof \rangle$

lemma *numeral_eqs_upto_12*:
 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 $12 = Suc\ 11$
 $\langle proof \rangle$

1.2 Basic Definitions for Turing Machines

datatype *action* = *WB* | *WO* | *L* | *R* | *Nop*

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

Remark: the constructors *W0* and *W1* were renamed into *WB* and *WO* respectively because this makes a better match with the constructors *Bk* and *Oc* of type *cell*.

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* :
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*⟩

lemma *fetch_imp* [*code*]: *fetch p n b* = (
let len = *length p*
in
if n = *0*
then (*Nop*, *0*)
else if b = *Bk*
then if len ≤ *2*∗*n* −*2*
then (*Nop*,*0*)
else (*p* ! (*2*∗*n*−*2*))
else if len ≤ *2*∗*n*−*1*
then (*Nop*,*0*)

else $(p\ !\ (2*n-1))$

)

⟨*proof*⟩

lemma *even_le_div2_imp_le_times_2*: $m \quad div\ 2 < (Suc\ n) \land ((m::nat)\ mod\ 2 = 0) \Longrightarrow m \leq 2*n$ ⟨*proof*⟩

lemma *odd_le_div2_imp_le_times_2*: $(m+1)\ div\ 2 < (Suc\ n) \land ((m::nat)\ mod\ 2 \neq 0) \Longrightarrow m \leq 2*n$ ⟨*proof*⟩

lemma *odd_div2_plus_1_eq*: $(n::nat)\ mod\ 2 \neq 0 \Longrightarrow (n\ div\ 2) + 1 = (n+1)\ div\ 2$
⟨*proof*⟩

lemma *list_length_tl_neq_Nil*: $1 < length\ (nl::nat\ list) \Longrightarrow tl\ nl \neq []$
⟨*proof*⟩

fun
update :: *action* ⇒ *tape* ⇒ *tape*
where
update WB $(l, r) = (l, Bk \# (tl\ r))$
| *update WO* $(l, r) = (l, Oc \# (tl\ r))$
| *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 *step_0′*: $step\ (0, tap)\ p = (0, tap)$ ⟨*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* $(s, tap) = (s = 0)$
⟨*proof*⟩

lemma *is_finalI* [*intro*]:
shows *is_final* $(0, tap)$
⟨*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*: $n1 \leq n2$
shows *is_final* (*steps c p n2*)
⟨*proof*⟩

lemma *stable_config_after_final_add*:
assumes $steps\ (1, l, r)\ p\ n1 \quad = (0, l', r')$
shows $steps\ (1, l, r)\ p\ (n1+n2) = (0, l', r')$
⟨*proof*⟩

lemma *stable_config_after_final_add_2*:
assumes $steps\ (s, l, r)\ p\ n1 \quad = (0, l', r')$
shows $steps\ (s, l, r)\ p\ (n1+n2) = (0, l', r')$
⟨*proof*⟩

lemma *stable_config_after_final_ge*:
assumes a: $steps\ (1, l, r)\ p\ n1 = (0, l', r')$ **and** b: $n1 \leq n2$
shows $steps\ (1, l, r)\ p\ n2 = (0, l', r')$
⟨*proof*⟩

lemma *stable_config_after_final_ge_2*:
assumes a: $steps\ (s, l, r)\ p\ n1 = (0, l', r')$ **and** b: $n1 \leq n2$
shows $steps\ (s, l, r)\ p\ n2 = (0, l', r')$
⟨*proof*⟩

lemma *stable_config_after_final_ge'*:
assumes $steps0\ (1, l, r)\ p\ n1 = (0, l', r')$ **and** b: $n1 \leq n2$
shows $steps0\ (1, l, r)\ p\ n2 = (0, l', r')$
⟨*proof*⟩

lemma *stable_config_after_final_ge_2'*:
assumes $steps0\ (s, l, r)\ p\ n1 = (0, l', r')$ **and** b: $n1 \leq n2$
shows $steps0\ (s, l, r)\ p\ n2 = (0, l', r')$
⟨*proof*⟩

lemma *not_is_final*:
assumes a: $\neg$ *is_final* $(steps\ c\ p\ n1)$
and b: $n2 \leq n1$
shows $\neg$ *is_final* $(steps\ c\ p\ n2)$
⟨*proof*⟩

lemma *before_final*:
assumes $steps0\ (1, tap)\ A\ n = (0, tap')$
shows $\exists\ n'.\ \neg$ *is_final* $(steps0\ (1, tap)\ A\ n') \wedge steps0\ (1, tap)\ A\ (Suc\ n') = (0, tap')$
⟨*proof*⟩

lemma *least_steps*:
assumes $steps0\ (1, tap)\ A\ n = (0, tap')$

shows $\exists\ n'.\ (\forall n'' < n'.\ \neg\ \mathit{is\_final}\ (\mathit{steps0}\ (1, \mathit{tap})\ A\ n'')) \wedge$
$(\forall n'' \geq n'.\ \mathit{is\_final}\ (\mathit{steps0}\ (1, \mathit{tap})\ A\ n''))$
⟨*proof*⟩

lemma *at_least_one_step*:$\mathit{steps0}\ (1, [], r)\ \mathit{tm}\ n = (0,\mathit{tap}) \Longrightarrow 0 < n$
⟨*proof*⟩

end

1.2.1 Auxiliary theorems about Turing Machines

theory *Turing_aux*
imports *Turing*
begin

fun $\mathit{fetch}' :: \mathit{instr\ list} \Rightarrow \mathit{state} \Rightarrow \mathit{cell} \Rightarrow \mathit{instr}$
where
$\mathit{fetch}'\ []\quad s\quad b = (\mathit{Nop}, 0)$

$|\ \mathit{fetch}'\ [\mathit{iBk}]\quad 0\quad b\ = (\mathit{Nop}, 0)$
$|\ \mathit{fetch}'\ [\mathit{iBk}]\quad (\mathit{Suc}\ 0)\quad \mathit{Bk} = \mathit{iBk}$
$|\ \mathit{fetch}'\ [\mathit{iBk}]\quad (\mathit{Suc}\ 0)\quad \mathit{Oc} = (\mathit{Nop}, 0)$
$|\ \mathit{fetch}'\ [\mathit{iBk}]\quad (\mathit{Suc}\ (\mathit{Suc}\ s'))\ b\ = (\mathit{Nop}, 0)$

$|\ \mathit{fetch}'\ (\mathit{iBk}\ \#\ \mathit{iOc}\ \#\ \mathit{inss})\ 0\quad b\ = (\mathit{Nop}, 0)$
$|\ \mathit{fetch}'\ (\mathit{iBk}\ \#\ \mathit{iOc}\ \#\ \mathit{inss})\ (\mathit{Suc}\ 0)\quad \mathit{Bk} = \mathit{iBk}$
$|\ \mathit{fetch}'\ (\mathit{iBk}\ \#\ \mathit{iOc}\ \#\ \mathit{inss})\ (\mathit{Suc}\ 0)\quad \mathit{Oc} = \mathit{iOc}$
$|\ \mathit{fetch}'\ (\mathit{iBk}\ \#\ \mathit{iOc}\ \#\ \mathit{inss})\ (\mathit{Suc}\ (\mathit{Suc}\ s'))\ b\ = \mathit{fetch}'\ \mathit{inss}\ (\mathit{Suc}\ s')\ b$

lemma *fetch′_Nil*:
shows $\mathit{fetch}'\ []\ s\ b = (\mathit{Nop}, 0)$
⟨*proof*⟩

lemma *fetch′_eq_fetch_app*: $\mathit{fetch}'\ \mathit{tm}\ s\ b = \mathit{fetch}\ \mathit{tm}\ s\ b$
⟨*proof*⟩

corollary *fetch′_eq_fetch*: $\mathit{fetch}' = \mathit{fetch}$
⟨*proof*⟩

definition
tm_step0_rel :: *tprog0* ⇒ ((*config* × *config*) *set*)
where
tm_step0_rel tp = {(*c1*, *c2*) . *step0 c1 tp* = *c2*}

abbreviation *tm_step0_rel_aux* :: [*config*, *tprog0*, *config*] ⇒ *bool* (‹((*1*_)/ ⊨⟨(_)⟩=/ (*1*_))› *50*)
where
tm_step0_rel_aux c1 tp c2 $\stackrel{def}{=}$ (*c1*,*c2*) ∈ *tm_step0_rel tp*

theorem *tm_step0_rel_iff_step0*: (*c1* ⊨⟨*tp*⟩= *c2*) ⟷ *step0 c1 tp* = *c2*
⟨*proof*⟩

definition *tm_steps0_rel* :: *tprog0* ⇒ ((*config* × *config*) *set*)
where
tm_steps0_rel tp = *rtrancl* (*tm_step0_rel tp*)

abbreviation *tm_steps0_rel_aux* :: [*config*, *tprog0*, *config*] ⇒ *bool* (‹((*1*_)/ ⊨⟨(_)⟩=*/ (*1*_))› *50*)
where
tm_steps0_rel_aux c1 tp c2 $\stackrel{def}{=}$ (*c1*,*c2*) ∈ *tm_steps0_rel tp*

lemma *tm_step0_rel_power*: (*tm_step0_rel tp* ^^ *n*) = {(*c1*,*c2*) . *steps0 c1 tp n* = *c2*}
⟨*proof*⟩

theorem *tm_steps0_rel_iff_steps0*: (*c1* ⊨⟨*tp*⟩=* *c2*) ⟷ (∃ *stp*. *steps0 c1 tp stp* = *c2*)
⟨*proof*⟩

end

1.3 Trailing Blanks on the input tape do not matter

theory *BlanksDoNotMatter*
imports *Turing*
begin

sledgehammer-params[*minimize*=*false*,*preplay_timeout*=*10*,*timeout*=*30*,*strict*=*true*,
provers=*e z3 cvc5 vampire*]

1.3.1 Replication of symbols

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

lemma *hd_repeat_cases*:
$P\ (hd\ (a \uparrow m\ @\ r)) \longleftrightarrow (m = 0 \longrightarrow P\ (hd\ r)) \wedge (\forall\, nat.\ m = Suc\ nat \longrightarrow P\ a)$
$\langle proof \rangle$

lemma *hd_repeat_cases'*:
$P\ (hd\ (a \uparrow m\ @\ r)) = (if\ m = 0\ then\ P\ (hd\ r)\ else\ P\ a)$
$\langle proof \rangle$

lemma
$(if\ m = 0\ then\ P\ (hd\ r)\ else\ P\ a) = ((m = 0 \longrightarrow P\ (hd\ r)) \wedge (\forall\, nat.\ m = Suc\ nat \longrightarrow P\ a))$
$\langle proof \rangle$

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

lemma *Bk_no_Oc_repeatE*[*elim*]: $Bk\ \#\ list = Oc \uparrow t \Longrightarrow RR$
$\langle proof \rangle$

lemma *replicate_Suc_1*: $a \uparrow (z1 + Suc\ z2) = (a \uparrow z1)\ @\ (a \uparrow Suc\ z2)$
$\langle proof \rangle$

lemma *replicate_Suc_2*: $a \uparrow (z1 + Suc\ z2) = (a \uparrow Suc\ z1)\ @\ (a \uparrow z2)$
$\langle proof \rangle$

1.3.2 Trailing blanks on the left tape do not matter

In this section we will show that we may add or remove trailing blanks on the initial left and right portions of the tape at will. However, we may not add or remove trailing blanks on the tape resulting from the computation. The resulting tape is completely determined by the contents of the initial tape.

lemma *step_left_tape_ShrinkBkCtx_right_Nil*:
assumes $step0\ (s,CL@Bk{\uparrow}\ z1\ ,\ [])\ tm = (s',l',r')$
and $za < z1$
shows $\exists\, CL'\ zb.\ l' = CL'@Bk{\uparrow}za@Bk{\uparrow}zb\ \wedge$
$(step0\ (s,CL@Bk{\uparrow}za,\ [])\ tm = (s',CL'@Bk{\uparrow}za,r')\ \vee$
$step0\ (s,CL@Bk{\uparrow}za,\ [])\ tm = (s',CL'@Bk{\uparrow}(za-1),r'))$
$\langle proof \rangle$

lemma *step_left_tape_ShrinkBkCtx_right_Bk*:
assumes $step0\ (s,CL@Bk{\uparrow}\ z1\ ,\ Bk\#rs)\ tm = (s',l',r')$
and $za < z1$
shows $\exists\, CL'\ zb.\ l' = CL'@Bk{\uparrow}za@Bk{\uparrow}zb\ \wedge$

$(step0\ (s,CL@Bk{\uparrow}za,\ Bk\#rs)\ tm = (s',CL'@Bk{\uparrow}za,r') \lor$
$step0\ (s,CL@Bk{\uparrow}za,\ Bk\#rs)\ tm = (s',CL'@Bk{\uparrow}(za-1),r'))$

$\langle proof \rangle$

lemma *step_left_tape_ShrinkBkCtx_right_Oc*:
assumes $step0\ (s,CL@Bk{\uparrow}\ z1\ ,\ Oc\#rs)\ tm = (s',l',r')$
and $za < z1$
shows $\exists\, CL'\ zb.\ l' = CL'@Bk{\uparrow}za@Bk{\uparrow}zb\ \land$
$(step0\ (s,CL@Bk{\uparrow}za,\ Oc\#rs)\ tm = (s',CL'@Bk{\uparrow}za,r') \lor$
$step0\ (s,CL@Bk{\uparrow}za,\ Oc\#rs)\ tm = (s',CL'@Bk{\uparrow}(za-1),r'))$

$\langle proof \rangle$

corollary *step_left_tape_ShrinkBkCtx*:
assumes $step0\ (s,CL@Bk{\uparrow}\ z1\ ,\ r)\ tm = (s',l',r')$
and $za < z1$
shows $\exists\, zb\ CL'.\ l' = CL'@Bk{\uparrow}za@Bk{\uparrow}zb\ \land$
$(step0\ (s,CL@Bk{\uparrow}za,\ r)\ tm = (s',CL'@Bk{\uparrow}za,r') \lor$
$step0\ (s,CL@Bk{\uparrow}za,\ r)\ tm = (s',CL'@Bk{\uparrow}(za-1),r'))$

$\langle proof \rangle$

lemma *steps_left_tape_ShrinkBkCtx_arbitrary_CL*:
$[\![\ steps0\ (s,\ CL@Bk{\uparrow}z1\ ,\ r)\ tm\ stp = (s',l',r');\ 0 < z1\ ]\!] \Longrightarrow$
$\exists\, zb\ CL'.\ l' = CL'@Bk{\uparrow}zb\ \land\ steps0\ (s,CL,\ r)\ tm\ stp = (s',CL',r')$

$\langle proof \rangle$

lemma *step_left_tape_EnlargeBkCtx_eq_Bks*:
assumes $step0\ (s,Bk{\uparrow}\ z1,\quad r)\ tm = (s',l',r')$
shows $step0\ (s,Bk{\uparrow}(z1+Suc\ z2),\ r)\ tm = (s',l'@Bk{\uparrow}Suc\ z2,r') \lor$
$step0\ (s,Bk{\uparrow}(z1+Suc\ z2),\ r)\ tm = (s',l'@Bk{\uparrow}z2,r')$

$\langle proof \rangle$

lemma *step_left_tape_EnlargeBkCtx_eq_Bk_C_Bks*:
assumes $step0\ (s,(Bk\#C)@Bk{\uparrow}\ z1,\quad r)\ tm = (s',l',r')$
shows $step0\ (s,(Bk\#C)@Bk{\uparrow}(z1+z2),\ r)\ tm = (s',l'@Bk{\uparrow}z2,r')$

$\langle proof \rangle$

lemma *step_left_tape_EnlargeBkCtx_eq_Oc_C_Bks*:
assumes $step0\ (s,(Oc\#C)@Bk{\uparrow}\ z1,\quad r)\ tm = (s',l',r')$
shows $step0\ (s,(Oc\#C)@Bk{\uparrow}(z1+z2),\ r)\ tm = (s',l'@Bk{\uparrow}z2,r')$

$\langle proof \rangle$

lemma *step_left_tape_EnlargeBkCtx_eq_C_Bks_Suc*:
assumes $step0\ (s,C@Bk{\uparrow}\ z1,\quad r)\ tm = (s',l',r')$
shows $step0\ (s,C@Bk{\uparrow}(z1+\ Suc\ z2),\ r)\ tm = (s',l'@Bk{\uparrow}Suc\ z2,r') \lor$
$step0\ (s,C@Bk{\uparrow}(z1+\ Suc\ z2),\ r)\ tm = (s',l'@Bk{\uparrow}z2,r')$

$\langle proof \rangle$

lemma *step_left_tape_EnlargeBkCtx_eq_C_Bks*:
assumes $step0\ (s,C@Bk\uparrow z1,\quad r)\ tm = (s',l',r')$
shows $step0\ (s,C@Bk\uparrow(z1+z2),\ r)\ tm = (s',l'@Bk\uparrow z2,r') \lor$
$step0\ (s,C@Bk\uparrow(z1+z2),\ r)\ tm = (s',l'@Bk\uparrow(z2-1),r')$
$\langle proof \rangle$

lemma *steps_left_tape_EnlargeBkCtx_arbitrary_CL*:
$steps0\ (s,\ CL\ @\ Bk\uparrow z1,\ r)\ tm\ stp = (s',\ l',r')$
$\Longrightarrow$
$\exists z3.\ z3 \leq z1 + z2 \land$
$steps0\ (s,\ CL\ @\ Bk\uparrow(z1 + z2),\ r)\ tm\ stp = (s',l'\ @\ Bk\uparrow z3\ ,r')$
$\langle proof \rangle$

corollary *steps_left_tape_EnlargeBkCtx*:
$steps0\ (s,\ Bk \uparrow k,\ r)\ tm\ stp = (s',\ Bk \uparrow l,\ r')$
$\Longrightarrow$
$\exists z3.\ z3 \leq k + z2 \land$
$steps0\ (s,\ Bk \uparrow (k + z2),\ r)\ tm\ stp = (s',Bk \uparrow (l + z3),\ r')$

$\langle proof \rangle$

corollary *steps_left_tape_ShrinkBkCtx_to_NIL*:
$steps0\ (s,\ Bk \uparrow k,\ r)\ tm\ stp = (s',\ Bk \uparrow l,\ r')$
$\Longrightarrow$
$\exists z3.\ z3 \leq l \land$
$steps0\ (s,\ [],\ r)\ tm\ stp = (s',\ Bk \uparrow z3,\ r')$
$\langle proof \rangle$

lemma *steps_left_tape_Nil_imp_All*:
$steps0\ (s,\ ([]\ \ ,\ r))\ p\ stp = (s',\ Bk \uparrow k,\ CR\ @\ Bk \uparrow l)$
$\Longrightarrow$
$\forall z.\ \exists stp\ k\ l.\ (steps0\ (s,\ (Bk\uparrow z,\ r))\ p\ stp) = (s',\ Bk \uparrow k,\ CR\ @\ Bk \uparrow l)$
$\langle proof \rangle$

lemma *ex_steps_left_tape_Nil_imp_All*:
$\exists stp\ k\ l.\ (steps0\ (s,\ ([]\ \ ,\ r))\ p\ stp) = (s',\ Bk \uparrow k,\ CR\ @\ Bk \uparrow l)$
$\Longrightarrow$
$\forall z.\ \exists stp\ k\ l.\ (steps0\ (s,\ (Bk\uparrow z,\ r))\ p\ stp) = (s',\ Bk \uparrow k,\ CR\ @\ Bk \uparrow l)$
$\langle proof \rangle$

1.3.3 Trailing blanks on the right tape do not matter

lemma *step_left_tape_Nil_imp_all_trailing_right_Nil*:

assumes *step0* $(s, CL1, []\ )$ *tm* $= (s', CR1, CR2\ )$
shows *step0* $(s, CL1, []\ @\ Bk \uparrow y)$ *tm* $= (s', CR1, CR2\ @\ Bk \uparrow y) \lor$
step0 $(s, CL1, []\ @\ Bk \uparrow y)$ *tm* $= (s', CR1, CR2\ @\ Bk \uparrow (y-1))$
⟨*proof*⟩

lemma *step_left_tape_Nil_imp_all_trailing_right_Cons*:
assumes *step0* $(s, CL1, rx\#rs\ \ )$ *tm* $= (s', CR1, CR2\ )$
shows *step0* $(s, CL1, rx\#rs\ @\ Bk \uparrow y)$ *tm* $= (s', CR1, CR2\ @\ Bk \uparrow y)$
⟨*proof*⟩

lemma *step_left_tape_Nil_imp_all_trailing_right*:
assumes *step0* $(s, CL1, r\ \ )$ *tm* $= (s', CR1, CR2\ )$
shows *step0* $(s, CL1, r\ @\ Bk \uparrow y)$ *tm* $= (s', CR1, CR2\ @\ Bk \uparrow y) \lor$
step0 $(s, CL1, r\ @\ Bk \uparrow y)$ *tm* $= (s', CR1, CR2\ @\ Bk \uparrow (y-1))$
⟨*proof*⟩

lemma *steps_left_tape_Nil_imp_all_trailing_right*:
steps0 $(s, CL1, r\ )$ *tm stp* $= (s', CR1, CR2)$
$\Longrightarrow$
$\exists x1\ x2.\ y = x1 + x2 \land$
steps0 $(s, CL1, r\ @\ Bk \uparrow y\ )$ *tm stp* $= (s', CR1, CR2\ @\ Bk \uparrow x2)$
⟨*proof*⟩

lemma *ex_steps_left_tape_Nil_imp_All_left_and_right*:
$(\exists kr\ lr.\ \ steps0\ (1, ([]\ \ , r\ \ \ ))\ p\ stp = (0, Bk \uparrow kr, r'\ @\ Bk \uparrow lr))$
$\Longrightarrow$
$\forall kl\ ll.\ \exists kr\ lr.\ steps0\ (1, (Bk \uparrow kl, r\ @\ Bk \uparrow ll))\ p\ stp = (0, Bk \uparrow kr, r'\ @\ Bk \uparrow lr)$
⟨*proof*⟩

end

theory *ComposableTMs*
imports *Turing*
begin

1.4 Making Turing Machines composable

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

fun

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

abbreviation
composable_tm0 p $\stackrel{def}{=}$ *composable_tm* (*p*, *0*)

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

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

1.4.1 Definitin of function fix_jumps and mk_composable0

fun *fix_jumps* :: *nat* ⇒ *tprog0* ⇒ *tprog0* **where**
fix_jumps smax [] = [] |
fix_jumps smax (*ins*#*inss*) = (*if* (*snd ins*) ≤ *smax*
then ins # *fix_jumps smax inss*
else ((*fst ins*),*0*)#*fix_jumps smax inss*)

fun *mk_composable0* :: *tprog0* ⇒ *tprog0* **where**
mk_composable0 [] = [(*Nop*,*0*),(*Nop*,*0*)] |
mk_composable0 [*i1*] = *fix_jumps 1* [*i1*,(*Nop*,*0*)] |
mk_composable0 (*i1*#*i2*#*ins*) = (*let l* = *2* + *length ins*
in if is_even l
then fix_jumps (*l div 2*) (*i1*#*i2*#*ins*)
else fix_jumps ((*l div 2*) + *1*) ((*i1*#*i2*#*ins*)@[(*Nop*,*0*)]))

1.4.2 Properties of function fix_jumps

lemma *fix_jumps_len*: *length* (*fix_jumps smax insl*) = *length insl*
⟨*proof*⟩

lemma *fix_jumps_le_smax*: ∀ *x* ∈ *set* (*fix_jumps smax tm*). (*snd x*) ≤ *smax*
⟨*proof*⟩

lemma *fix_jumps_nth_no_fix*:
assumes *n* < *length tm* **and** *tm*!*n* = *ins* **and** (*snd ins*) ≤ *smax*
shows (*fix_jumps smax tm*)!*n* = *ins*
⟨*proof*⟩

lemma *fix_jumps_nth_fix*:

assumes $n < length\ tm$ **and** $tm!n = ins$ **and** $\neg(snd\ ins) \leq smax$
shows $(fix\_jumps\ smax\ tm)!n = ((fst\ ins),0)$
$\langle proof \rangle$

1.4.3 Functions fix_jumps and mk_composable0 generate composable Turing Machines.

lemma *composable_tm0_fix_jumps_pre*:
assumes $length\ tm \geq 2$ **and** $is\_even\ (length\ tm)$
shows $length\ (fix\_jumps\ (length\ tm\ div\ 2)\ tm) \geq 2\ \wedge$
$is\_even\ (length\ (fix\_jumps\ (length\ tm\ div\ 2)\ tm))\ \wedge$
$(\forall x \in set\ (fix\_jumps\ (length\ tm\ div\ 2)\ tm).$
$(snd\ x) \leq length\ (fix\_jumps\ (length\ tm\ div\ 2)\ tm)\ div\ 2 + 0 \wedge (snd\ x) \geq 0)$
$\langle proof \rangle$

lemma *composable_tm0_fix_jumps*:
assumes $length\ tm \geq 2$ **and** $is\_even\ (length\ tm)$
shows $composable\_tm0\ (fix\_jumps\ (length\ tm\ div\ 2)\ tm)$
$\langle proof \rangle$

lemma *fix_jumps_composable0_eq*:
assumes $composable\_tm0\ tm$
shows $(fix\_jumps\ (length\ tm\ div\ 2)\ tm) = tm$
$\langle proof \rangle$

lemma *composable_tm0_mk_composable0*: $composable\_tm0\ (mk\_composable0\ tm)$
$\langle proof \rangle$

1.4.4 Functions mk_composable0 is the identity on composable Turing Machines

lemma *mk_composable0_eq*:
assumes $composable\_tm0\ tm$
shows $mk\_composable0\ tm = tm$
$\langle proof \rangle$

1.4.5 About the length of mk_composable0 tm

lemma *length_mk_composable0_nil*: $length\ (mk\_composable0\ []) = 2$
$\langle proof \rangle$

lemma *length_mk_composable0_singleton*: $length\ (mk\_composable0\ [i1]) = 2$
$\langle proof \rangle$

lemma *length_mk_composable0_gt2_even*: $is\_even\ (length\ (i1\ \#\ i2\ \#\ ins)) \Longrightarrow length\ (mk\_composable0\ (i1\ \#\ i2\ \#\ ins)) = length\ (i1\#i2\#ins)$
$\langle proof \rangle$

lemma *length_mk_composable0_gt2_odd*: $\neg$*is_even* (*length* (*i1* # *i2* # *ins*)) $\Longrightarrow$ *length* (*mk_composable0* (*i1* # *i2* # *ins*)) = *length* (*i1*#*i2*#*ins*)+*1*
⟨*proof*⟩

lemma *length_mk_composable0_even*: ⟦*0* < *length tm* ; *is_even* (*length tm*) ⟧ $\Longrightarrow$ *length* (*mk_composable0 tm*) = *length tm*
⟨*proof*⟩

lemma *length_mk_composable0_odd*: ⟦*0* < *length tm* ; $\neg$*is_even* (*length tm*) ⟧ $\Longrightarrow$ *length* (*mk_composable0 tm*) = *1* + *length tm*
⟨*proof*⟩

lemma *length_tm_le_mk_composable0*: *length tm* ≤ *length* (*mk_composable0 tm*)
⟨*proof*⟩

1.4.6 Properties of function fetch with respect to function mk_composable0

lemma *fetch_mk_composable0_Bk_too_short_Suc*:
  assumes *b* = *Bk* **and** *length tm* ≤ *2*∗*s*
  shows *fetch* (*mk_composable0 tm*) (*Suc s*) *b* = (*Nop*, *0*::*nat*)
⟨*proof*⟩

lemma *fetch_mk_composable0_Oc_too_short_Suc*:
  assumes *b* = *Oc* **and** *length tm* ≤ *2*∗*s*+*1*
  shows *fetch* (*mk_composable0 tm*) (*Suc s*) *b* = (*Nop*, *0*::*nat*)
⟨*proof*⟩

lemma *nth_append'*: *n* < *length xs* $\Longrightarrow$ (*xs* @ *ys*) ! *n* = *xs* ! *n* ⟨*proof*⟩

lemma *fetch_mk_composable0_Bk_Suc_no_fix*:
  assumes *b* = *Bk*
    and *2*∗*s* < *length tm*
    and *fetch tm* (*Suc s*) *b* = (*a*, *s'*)
    and *s'* ≤ *length* (*mk_composable0 tm*) *div 2*
  shows *fetch* (*mk_composable0 tm*) (*Suc s*) *b* = *fetch tm* (*Suc s*) *b*
⟨*proof*⟩

lemma *fetch_mk_composable0_Bk_Suc_fix*:
assumes *b* = *Bk*
    and *2*∗*s* < *length tm*
    and *fetch tm* (*Suc s*) *b* = (*a*, *s'*)
    and *length* (*mk_composable0 tm*) *div 2* < *s'*
shows *fetch* (*mk_composable0 tm*) (*Suc s*) *b* = (*a*, *0*)
⟨*proof*⟩

lemma *fetch_mk_composable0_Oc_Suc_no_fix*:
  assumes *b* = *Oc*
    and *2*∗*s*+*1* < *length tm*
    and *fetch tm* (*Suc s*) *b* = (*a*, *s'*)

and $s' \leq$ *length* (*mk_composable0 tm*) *div 2*
shows *fetch* (*mk_composable0 tm*) (*Suc s*) *b* = *fetch tm* (*Suc s*) *b*
⟨*proof*⟩

lemma *fetch_mk_composable0_Oc_Suc_fix*:
assumes $b = Oc$
and $2*s+1 < length\ tm$
and *fetch tm* (*Suc s*) $b = (a, s')$

and *length* (*mk_composable0 tm*) *div 2* $< s'$
shows *fetch* (*mk_composable0 tm*) (*Suc s*) $b = (a, 0)$
⟨*proof*⟩

1.4.7 Properties of function step0 with respect to function mk_composable0

lemma *length_mk_composable0_div2_lt_imp_length_tm_le_times2*:
assumes *length* (*mk_composable0 tm*) *div 2* $< s'$
and $s' = Suc\ s2$
shows *length tm* $\leq 2 * s2$
⟨*proof*⟩

lemma *jump_out_of_pgm_is_final_next_step*:
assumes *step0* (s, l, r) *tm* $= (s', update\ a1\ (l, r))$
and $s' = Suc\ s2$ **and** *length* (*mk_composable0 tm*) *div 2* $< s'$
shows *step0* (*step0* (s, l, r) *tm*) *tm* = (*0*, *snd* (*step0* (s, l, r) *tm*))
⟨*proof*⟩

lemma *step0_mk_composable0_after_one_step*:
assumes *step0* $(s, (l, r))$ *tm* $\neq$ *step0* (s, l, r) (*mk_composable0 tm*)
shows *step0* (*step0* $(s, (l, r))$ *tm*) *tm* = (*0*, *snd*((*step0* $(s, (l, r))$ *tm*))) ∧
step0 (s, l, r) (*mk_composable0 tm*) = (*0*, *snd*((*step0* $(s, (l, r))$ *tm*)))
⟨*proof*⟩

lemma *step0_mk_composable0_eq_after_two_steps*:
assumes *step0* $(s, (l, r))$ *tm* $\neq$ *step0* (s, l, r) (*mk_composable0 tm*)
shows *step0* (*step0* $(s, (l, r))$ *tm*) *tm* = (*0*, *snd*((*step0* $(s, (l, r))$ *tm*))) ∧
step0 (*step0* $(s, (l, r))$ (*mk_composable0 tm*)) (*mk_composable0 tm*) = *step0* (*step0* (*s*, (*l*,
r)) *tm*) *tm*
⟨*proof*⟩

1.4.8 Properties of function steps0 with respect to function mk_composable0

lemma *steps0* $(s, (l, r))$ *tm 0* = *steps0* (s, l, r) (*mk_composable0 tm*) *0*
⟨*proof*⟩

lemma *mk_composable0_tm_at_most_one_diff_pre*:

assumes *steps0* $(s, (l, r))$ *tm stp* $\neq$ *steps0* (s, l, r) (*mk_composable0 tm*) *stp*
shows $0 < stp \wedge (\exists k.\ k < stp$
$\quad \wedge (\forall i \leq k.\ steps0\ (s, l, r)\ (mk\_composable0\ tm)\ i = steps0\ (s, l, r)\ tm\ i)$
$\quad \wedge (\forall j > k+1.$
$\qquad steps0\ (s, l, r)\ tm\ (j) = (0, snd(steps0\ (s, l, r)\ tm\ (k+1))) \wedge$
$\qquad steps0\ (s, l, r)\ (mk\_composable0\ tm)\ j = steps0\ (s, l, r)\ tm\ j))$
⟨*proof*⟩

lemma *mk_composable0_tm_at_most_one_diff*:
assumes *steps0* (s, l, r) (*mk_composable0 tm*) *stp* $\neq$ *steps0* $(s, (l, r))$ *tm stp*
shows $0 < stp \wedge$
$\quad (\forall i < stp.\ steps0\ (s, l, r)\ (mk\_composable0\ tm)\ i = steps0\ (s, l, r)\ tm\ i) \wedge$
$\quad (\forall j > stp.\ steps0\ (s, l, r)\ tm\ (j) = (0, snd(steps0\ (s, l, r)\ tm\ stp)) \wedge$
$\qquad steps0\ (s, l, r)\ (mk\_composable0\ tm)\ j = steps0\ (s, l, r)\ tm\ j)$
⟨*proof*⟩

lemma *mk_composable0_tm_at_most_one_diff'*:
assumes *steps0* (s, l, r) (*mk_composable0 tm*) *stp* $\neq$ *steps0* $(s, (l, r))$ *tm stp*
shows $0 < stp \wedge (\exists fl\ fr.\ snd(steps0\ (s, l, r)\ tm\ stp) = (fl, fr) \wedge$
$\quad (\forall i < stp.\ steps0\ (s, l, r)\ (mk\_composable0\ tm)\ i = steps0\ (s, l, r)\ tm\ i) \wedge$
$\quad (\forall j > stp.\ steps0\ (s, l, r)\ \ tm\ \quad j = (0, fl, fr) \wedge$
$\qquad steps0\ (s, l, r)\ (mk\_composable0\ tm)\ j = (0, fl, fr)\ ))$
⟨*proof*⟩

end

theory *ComposedTMs*
imports *ComposableTMs*
begin

1.5 Composition of Turing Machines

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

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

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
seq_tm :: *instr list* ⇒ *instr list* ⇒ *instr list* (**infixl** ‹|+|› *60*)
where
seq_tm p1 p2 = ((*adjust0 p1*) @ (*shift p2* (*length p1 div 2*)))

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

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

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

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

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

lemma *seq_tm_next*:
assumes *a_ht*: *steps0* (*1*, *tap*) *A n* = (*0*, *tap*′)

and *a_composable*: *composable_tm* $(A, 0)$
obtains n' **where** *steps0* $(1, tap)$ $(A\ |{+}|\ B)$ $n' = (Suc\ (length\ A\ div\ 2), tap')$
⟨*proof*⟩

lemma *seq_tm_fetch_second_zero*:
assumes *h1*: *fetch B s x* $= (a, 0)$
and *hs*: *composable_tm* $(A, 0)$ $s \neq 0$
shows *fetch* $(A\ |{+}|\ B)$ $(s + (length\ A\ div\ 2))$ $x = (a, 0)$
⟨*proof*⟩

lemma *seq_tm_fetch_second_inst*:
assumes *h1*: *fetch B sa x* $= (a, s)$
and *hs*: *composable_tm* $(A, 0)$ $sa \neq 0$ $s \neq 0$
shows *fetch* $(A\ |{+}|\ B)$ $(sa + length\ A\ div\ 2)$ $x = (a, s + length\ A\ div\ 2)$
⟨*proof*⟩

lemma *seq_tm_second*:
assumes *a_composable*: *composable_tm* $(A, 0)$
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 *seq_tm_final*:
assumes *composable_tm* $(A, 0)$
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

1.6 Encoding of Natural Numbers

theory *Numerals*
imports *ComposedTMs BlanksDoNotMatter*
begin

1.6.1 A class for generating numerals

class *tape* =
fixes *tape_of* :: $'a \Rightarrow cell\ list$ (‹<_>› *100*)

instantiation *nat*::*tape* **begin**
definition *tape_of_nat* **where** *tape_of_nat* $(n::nat) \stackrel{def}{=} Oc \uparrow (Suc\ n)$
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

1.6.2 Some lemmas about numerals used for rewriting

lemma tape_of_list_empty[simp]: <[]> = ([]::cell list) ⟨proof⟩

lemma tape_of_nat_list_cases2: <(nl::nat list)> = [] ∨ (∃ r'. <nl> = Oc # r')
  ⟨proof⟩

1.6.3 Unique decomposition of standard tapes

Some lemmas about unique decomposition of tapes in standard halting configuration.

lemma OcSuc_lemma: Oc # Oc ↑ n1 = Oc ↑ n2 ⟹ Suc n1 = n2
⟨proof⟩

lemma inj_tape_of_list: (<n1::nat>) = (<n2::nat>) ⟹ n1 = n2
  ⟨proof⟩

lemma inj_repl_Bk: Bk ↑ k1 = Bk ↑ k2 ⟹ k1 = k2 ⟨proof⟩

lemma last_of_numeral_is_Oc: last (<n::nat>) = Oc
  ⟨proof⟩

lemma hd_of_numeral_is_Oc: hd (<n::nat>) = Oc
  ⟨proof⟩

lemma rev_replicate: rev (Bk ↑ l1) = (Bk ↑ l1) ⟨proof⟩

lemma rev_numeral: rev (<n::nat>) = <n::nat>
  ⟨proof⟩

lemma *drop_Bk_prefix*: $n < l \Longrightarrow hd\ (drop\ n\ ((Bk \uparrow l)\ @\ xs)) = Bk$
⟨*proof*⟩

lemma *unique_Bk_postfix*: $<n1::nat>\ @\ Bk \uparrow l1 = <n2::nat>\ @\ Bk \uparrow l2 \Longrightarrow l1 = l2$
⟨*proof*⟩

lemma *unique_decomp_tap*:
assumes $(lx, <n1::nat>\ @\ Bk \uparrow l1) = (ly, <n2::nat>\ @\ Bk \uparrow l2)$
shows $lx{=}ly \wedge n1{=}n2 \wedge l1{=}l2$
⟨*proof*⟩

lemma *unique_decomp_std_tap*:
assumes $(Bk \uparrow k1, <n1::nat>\ @\ Bk \uparrow l1) = (Bk \uparrow k2, <n2::nat>\ @\ Bk \uparrow l2)$
shows $k1{=}k2 \wedge n1{=}n2 \wedge l1{=}l2$
⟨*proof*⟩

1.6.4 Lists of numerals never contain two consecutive blanks

definition *noDblBk*:: *cell list* $\Rightarrow$ *bool*
where $noDblBk\ cs \stackrel{def}{=} \forall i.\ Suc\ i < length\ cs \wedge cs!i = Bk \longrightarrow\ cs!(Suc\ i) = Oc$

lemma *noDblBk_Bk_Oc_rep*: $noDblBk\ (Oc \uparrow n1)$
⟨*proof*⟩

lemma *noDblBk_Bk_imp_Oc*: $[\![noDblBk\ cs;\ Suc\ i < length\ cs;\ cs!i = Bk\ ]\!] \Longrightarrow\ cs!(Suc\ i) = Oc$
⟨*proof*⟩

lemma *noDblBk_imp_noDblBk_Oc_cons*: $noDblBk\ cs \Longrightarrow noDblBk\ (Oc\#cs)$
⟨*proof*⟩

lemma *noDblBk_Numeral*: $noDblBk\ (<n::nat>)$
⟨*proof*⟩

lemma *noDblBk_Nil*: $noDblBk\ []$
⟨*proof*⟩

lemma *noDblBk_Singleton*: $noDblBk\ (<[n::nat]>)$
⟨*proof*⟩

lemma *tape_of_nat_list_cons_eq*:$nl \neq [] \Longrightarrow <(a::nat)\ \#\ nl> = <a>\ @\ Bk\ \#\ <nl>$
⟨*proof*⟩

lemma *noDblBk_cons_cons*: $noDblBk(<(x::nat)\ \#\ xs>) \Longrightarrow noDblBk(<a::nat>\ @\ Bk\ \#\ <x\ \#\ xs>)$
⟨*proof*⟩

theorem *noDblBk_tape_of_nat_list*: $noDblBk(<nl::\ nat\ list>)$
⟨*proof*⟩

lemma *hasDblBk_L1*: $\llbracket\ CR = rs\ @\ [Bk]\ @\ Bk\ \#\ rs';\ noDblBk\ CR\ \rrbracket \Longrightarrow False$
 ⟨*proof*⟩

lemma *hasDblBk_L2*: $\llbracket\ C = Bk\ \#\ cls;\ noDblBk\ C\ \rrbracket \Longrightarrow cls = [] \lor (\exists\, cls'.\ cls = Oc\#cls')$
 ⟨*proof*⟩

lemma *hasDblBk_L3*: $\llbracket\ noDblBk\ C\ ;\ C = C1\ @\ (Bk\#C2)\ \rrbracket \Longrightarrow C2 = [] \lor (\exists\, C3.\ C2 = Oc\#C3)$
 ⟨*proof*⟩

lemma *hasDblBk_L4*:
 assumes $noDblBk\ CL$
  and $r = Bk\ \#\ rs$
  and $r = rev\ ls1\ @\ Oc\ \#\ rss$
  and $CL = ls1\ @\ ls2$
 shows $ls2 = [] \lor (\exists\, bs.\ ls2 = Oc\#bs)$
⟨*proof*⟩

lemma *hasDblBk_L5*:
 assumes $noDblBk\ CL$
  and $r = Bk\ \#\ rs$
  and $r = rev\ ls1\ @\ Oc\ \#\ rss$
  and $CL = ls1\ @\ [Bk]$
 shows $False$
 ⟨*proof*⟩

lemma *noDblBk_cases*:
 assumes $noDblBk\ C$
   and $C = C1\ @\ C2$
   and $C2 = []\ \Longrightarrow P$
   and $C2 = [Bk] \Longrightarrow P$
   and $\bigwedge C3.\ C2 = Bk\#Oc\#C3 \Longrightarrow P$
   and $\bigwedge C3.\ C2 = Oc\#C3 \Longrightarrow P$
 shows P
⟨*proof*⟩

1.6.5 Unique decomposition of tapes containing lists of numerals

A lemma about appending lists of numerals.

lemma *append_numeral_list*: $\llbracket\ (nl1{::}nat\ list) \neq [];\ nl2 \neq []\ \rrbracket \Longrightarrow <nl1\ @\ nl2> = <nl1>@[Bk]@<nl2>$
⟨*proof*⟩

A lemma about reverting lists of numerals.

lemma *rev_numeral_list*: $rev(<nl{::}nat\ list>) = <(rev\ nl)>$
⟨*proof*⟩

Some more lemmas about unique decomposition of tapes that contain lists of numerals.

lemma *unique_Bk_postfix_numeral_list_Nil*: $<[]>\ @\ Bk \uparrow l1 = <yl::nat\ list>\ @\ Bk \uparrow l2 \Longrightarrow [] = yl$
⟨*proof*⟩

lemma *nonempty_list_of_numerals_neq_BKs*: $<a\#\ xs::nat\ list> \neq Bk \uparrow l$
⟨*proof*⟩

lemma *unique_Bk_postfix_nonempty_numeral_list*:
$[\![\ xl \neq []; yl \neq [];\ \ <xl::nat\ list>\ @\ Bk \uparrow l1 = <yl::nat\ list>\ \ @\ Bk \uparrow l2\ ]\!] \Longrightarrow xl = yl$
⟨*proof*⟩

corollary *unique_Bk_postfix_numeral_list*: $<xl::nat\ list>\ @\ Bk \uparrow l1 = <yl::nat\ list>\ @\ Bk \uparrow l2 \Longrightarrow xl = yl$
⟨*proof*⟩

Some more lemmas about noDblBks in lists of numerals.

lemma *numeral_list_head_is_Oc*: $(nl::nat\ list) \neq [] \Longrightarrow hd\ (<nl>) = Oc$
⟨*proof*⟩

lemma *numeral_list_last_is_Oc*: $(nl::nat\ list) \neq [] \Longrightarrow last\ (<nl>) = Oc$
⟨*proof*⟩

lemma *noDblBk_tape_of_nat_list_imp_noDblBk_tl*: $noDblBk\ (<nl>) \Longrightarrow noDblBk\ (tl\ (<nl>))$
⟨*proof*⟩

lemma *noDblBk_tape_of_nat_list_cons_imp_noDblBk_tl*: $noDblBk\ (a\ \#\ <nl>) \Longrightarrow noDblBk\ (<nl>)$
⟨*proof*⟩

lemma *noDblBk_tape_of_nat_list_imp_noDblBk_cons_Bk*: $(nl::nat\ list) \neq [] \Longrightarrow noDblBk\ ([Bk]\ @\ <nl>)$
⟨*proof*⟩

end

theory *Numerals_Ex*
imports *Numerals*
begin

1.6.6 About the expansion of the numeral notation

lemma $<[]> == []$ ⟨*proof*⟩
lemma $<[]::(nat\ list)> = ([]::(cell\ list))$ ⟨*proof*⟩

value $<0::nat>$
value $<1::nat>$

value <[]::(*nat list*)>

value <[*1*::*nat*, *2*::*nat*]>

value <(*0*::*nat*)>
value <(*1*::*nat*)>

value <(*1*::*nat*, *2*::*nat*)>

value <[*1*::*nat*, *2*::*nat*, *3*::*nat*]>
value <(*1*::*nat*, *2*::*nat*, *3*::*nat*)>
value <(*1*::*nat*, (*2*::*nat*, *3*::*nat*))>
value <(*1*::*nat*, [*2*::*nat*, *3*::*nat*])>

end

1.7 Hoare Rules for Turing Machines

theory *Turing_Hoare*
 imports *Numerals*
begin

1.7.1 Hoare_halt and Hoare_unhalt for total correctness

1.7.1.1 Definition for Hoare_halt and Hoare_unhalt conditions

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

definition
 assert_imp :: *assert* ⇒ *assert* ⇒ *bool* (‹_ ↦ _› [*0*, *0*] *100*)
 where
  $P \mapsto Q \stackrel{def}{=} \forall\, l\ r.\ P\ (l, r) \longrightarrow Q\ (l, r)$

lemma *refl_assert*[*intro*, *simp*]:
 $P \mapsto P$
 ⟨*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
$\{\!|P|\!\}\ p\ \{\!|Q|\!\} \stackrel{def}{=} (\forall\, tap.\ P\ tap \longrightarrow (\exists\, n.\ is\_final\ (steps0\ (1, tap)\ p\ n) \wedge Q\ holds\_for\ (steps0\ (1, tap)\ p\ n)\ ))$

definition
Hoare_unhalt :: *assert* ⇒ *tprog0* ⇒ *bool* (‹({⦃(1_)⦄/ (_)) ↑› *50*)
where
$\{\!|P|\!\}\ p \uparrow \stackrel{def}{=} \forall\, tap.\ P\ tap \longrightarrow (\forall\ n\ .\ \neg(is\_final\ (steps0\ (1, tap)\ p\ n)))$

lemma *Hoare_haltI*:
assumes $\bigwedge l\ r.\ P\ (l, r) \Longrightarrow \exists\, n.\ is\_final\ (steps0\ (1, (l, r))\ p\ n) \wedge Q\ holds\_for\ (steps0\ (1, (l, r))\ p\ n)$
shows ⦃*P*⦄ *p* ⦃*Q*⦄
⟨*proof*⟩

lemma *Hoare_haltE*:
assumes ⦃*P*⦄ *p* ⦃*Q*⦄
and *P* (*l*, *r*)
shows $\exists\, n.\ is\_final\ (steps0\ (1, (l, r))\ p\ n) \wedge Q\ holds\_for\ (steps0\ (1, (l, r))\ p\ n)$
⟨*proof*⟩

lemma *Hoare_unhaltI*:
assumes $\bigwedge l\ r\ n.\ P\ (l, r) \Longrightarrow \neg\ is\_final\ (steps0\ (1, (l, r))\ p\ n)$
shows ⦃*P*⦄ *p* ↑
⟨*proof*⟩

lemma *Hoare_unhaltE*:
assumes ⦃*P*⦄ *p* ↑
and *P tap*
shows $\neg\ (is\_final\ (steps0\ (1, tap)\ p\ n))$
⟨*proof*⟩

lemma *Hoare_halt_iff*:
$\{\!|P|\!\}$ *tm* $\{\!|Q|\!\}$
$\longleftrightarrow$
$(\forall\, l1\ r1.\ P\ (l1,r1) \longrightarrow (\exists\, stp\ l0\ r0.\ \ steps0\ (1,\ l1,r1)\ tm\ stp = (0,l0,r0) \wedge Q\ (l0,r0)))$
$\langle proof \rangle$

lemma *Hoare_halt_I0*:
assumes $\bigwedge l1\ r1.\ P(l1,\ r1) \Longrightarrow steps0\ (1,\ l1,\ r1)\ tm\ stp = (0,\ l0,\ r0) \wedge Q\ (l0,\ r0)$
shows $\{\!|P|\!\}$ *tm* $\{\!|Q|\!\}$
$\langle proof \rangle$

lemma *Hoare_halt_E0*:
assumes *major*: $\{\!|P|\!\}$ *tm* $\{\!|Q|\!\}$
and $P(l1,\ r1)$
shows $\exists\, stp\ l0\ r0.\ steps0\ (1,\ l1,\ r1)\ tm\ stp = (0,\ l0,\ r0) \wedge Q(l0,\ r0)$
$\langle proof \rangle$

lemma *partial_correctness_and_halts_imp_total_correctness'*:
assumes *partial_corr*: $(\exists\, stp\ l1\ r1.\ P\ (l1,\ r1) \wedge\ is\_final\ (steps0\ (1,\ l1,r1)\ tm\ stp)) \longrightarrow \{\!|P|\!\}$
tm $\{\!|Q|\!\}$
and *halts*: $(\exists\, stp\ l1\ r1.\ P\ (l1,\ r1) \wedge\ is\_final\ (steps0\ (1,\ l1,r1)\ tm\ stp))$
shows $\{\!|P|\!\}$ *tm* $\{\!|Q|\!\}$
$\langle proof \rangle$

lemma *partial_correctness_and_halts_imp_total_correctness*:
assumes *partial_corr*: $\forall\, l1\ r1\ stp.\ P\ (l1,\ r1) \wedge is\_final\ (steps0\ (1,\ l1,r1)\ tm\ stp) \longrightarrow \{\!|P|\!\}\ tm$
$\{\!|Q|\!\}$
and *halts*: $(\exists\, stp\ l1\ r1.\ P\ (l1,\ r1) \wedge\ is\_final\ (steps0\ (1,\ l1,r1)\ tm\ stp))$
shows $\{\!|P|\!\}$ *tm* $\{\!|Q|\!\}$
$\langle proof \rangle$

lemma $(\ (\exists\, stp\ l1\ r1.\ P\ (l1,\ r1) \wedge\ is\_final\ (steps0\ (1,\ l1,r1)\ tm\ stp)) \longrightarrow \{\!|P|\!\}\ tm\ \{\!|Q|\!\}\ )$
$\longleftrightarrow$
$(\ \ \forall\, stp\ l1\ r1.\ (P\ (l1,\ r1) \wedge is\_final\ (steps0\ (1,\ l1,r1)\ tm\ stp)\ \ \longrightarrow \{\!|P|\!\}\ tm\ \{\!|Q|\!\})\ \ )$
$\langle proof \rangle$

lemma *Hoare_consequence*:
assumes $P' \mapsto P\ \{\!|P|\!\}\ p\ \{\!|Q|\!\}\ Q \mapsto Q'$
shows $\{\!|P'|\!\}\ p\ \{\!|Q'|\!\}$

⟨*proof*⟩

1.7.1.2 Relation between Hoare_halt and Hoare_unhalt

lemma *Hoare_halt_impl_not_Hoare_unhalt*:
assumes $\{\!|P|\!\}\ p\ \{\!|Q|\!\}$ **and** *P tap*
shows $\neg(\{\!|P|\!\}\ p \uparrow)$
⟨*proof*⟩

lemma *Hoare_unhalt_impl_not_Hoare_halt*:
assumes $\{\!|P|\!\}\ p \uparrow$ **and** *P tap*
shows $\neg(\{\!|P|\!\}\ p\ \{\!|Q|\!\})$
⟨*proof*⟩

1.7.1.3 Hoare_halt and Hoare_unhalt for composed Turing Machines

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

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

1.7.2 Trailing Blanks on the left tape do not matter for Hoare_halt

The following theorems have major impact on the definition of Turing Computability.

lemma *Hoare_halt_add_Bks_left_tape_L1*:
assumes $\{\!|\ \lambda tap.\ tap = ([], r)|\!\}\ p\ \{\!|\ (\lambda tap.\ \exists\, k\ l.\ tap = (Bk \uparrow k,\ CR\ @\ Bk \uparrow l))\ |\!\}$
shows $\forall z.\ \exists\, stp\ k\ l.\ (steps0\ (1, Bk{\uparrow}z,r)\ p\ stp) = (0, Bk \uparrow k, CR\ @\ Bk \uparrow l)$
⟨*proof*⟩

lemma *Hoare_halt_add_Bks_left_tape_L2*:
assumes $\forall z.\ \exists\, stp\ k\ l.\ (steps0\ (1, Bk{\uparrow}z,r)\ p\ stp) = (0, Bk \uparrow k, CR\ @\ Bk \uparrow l)$
shows $\{\!|(\lambda tap.\ \exists z.\ tap = (Bk{\uparrow}z, r))|\!\}\ p\ \{\!|\ (\lambda tap.\ (\exists\, k\ l.\ tap = (Bk \uparrow k,\ CR\ @\ Bk \uparrow l)))\ |\!\}$
⟨*proof*⟩

theorem *Hoare_halt_add_Bks_left_tape*:
$\{\!|(\lambda tap.\quad tap = ([]\ \ , r))|\!\}\ p\ \{\!|\ (\lambda tap.\ (\exists\, k\ l.\ tap = (Bk \uparrow k,\ CR\ @\ Bk \uparrow l)))\ |\!\}$
$\Longrightarrow$
$\forall z.\ \ \{\!|(\lambda tap.\ tap = (Bk{\uparrow}z, r))|\!\}\ p\ \{\!|\ (\lambda tap.\ (\exists\, k\ l.\ tap = (Bk \uparrow k,\ CR\ @\ Bk \uparrow l)))\ |\!\}$
⟨*proof*⟩

theorem *Hoare_halt_del_Bks_left_tape*:
$\{\!\!\{(\lambda tap.\ \exists z.\ tap = (Bk\uparrow z, r))\}\!\!\}\ p\ \{\!\!\{\ (\lambda tap.\ (\exists k\ l.\ tap = (Bk \uparrow k,\ \ CR\ @\ Bk \uparrow l)))\ \}\!\!\}$
$\Longrightarrow$
$\{\!\!\{(\lambda tap.\quad tap = ([]\ \ , r))\}\!\!\}\ p\ \{\!\!\{\ (\lambda tap.\ (\exists k\ l.\ tap = (Bk \uparrow k,\ \ CR\ @\ Bk \uparrow l)))\ \}\!\!\}$
⟨*proof*⟩

lemma *is_final_del_Bks*: *is_final* (*steps0* (*s*, *Bk* ↑ *k*, *r*) *tm stp*) ⟹ *is_final* (*steps0* (*s*, [], *r*) *tm stp*)
⟨*proof*⟩

lemma *Hoare_unhalt_add_Bks_left_tape_L1*:
assumes $\{\!\!\{\lambda tap.\ tap = ([], r)\}\!\!\}\ p \uparrow$
shows $\forall z.\ \{\!\!\{\lambda tap.\ tap = (Bk \uparrow z, r)\}\!\!\}\ p \uparrow$
⟨*proof*⟩

1.7.3 Halt lemmas with respect to function mk_composable0

theorem *Hoare_halt_tm_impl_Hoare_halt_mk_composable0_cell_list*: $\{\!\!\{\lambda tap.\ tap = ([], cl)\}\!\!\}$ $tm\ \{\!\!\{Q\}\!\!\} \Longrightarrow \{\!\!\{\lambda tap.\ tap = ([], cl)\}\!\!\}$ *mk_composable0 tm* $\{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

theorem *Hoare_halt_tm_impl_Hoare_halt_mk_composable0_cell_list_rev*: $\{\!\!\{\lambda tap.\ tap = ([], cl)\}\!\!\}$ *mk_composable0 tm* $\{\!\!\{Q\}\!\!\} \Longrightarrow \{\!\!\{\lambda tap.\ tap = ([], cl)\}\!\!\}\ tm\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

lemma *Hoare_unhalt_tm_impl_Hoare_unhalt_mk_composable0_cell_list*: $(\{\!\!\{\lambda tap.\ tap = ([], cl)\}\!\!\}\ tm \uparrow) \Longrightarrow (\{\!\!\{\lambda tap.\ tap = ([], cl)\ \}\!\!\}\ (mk\_composable0\ tm) \uparrow)$
⟨*proof*⟩

corollary *Hoare_halt_tm_impl_Hoare_halt_mk_composable0*: $\{\!\!\{\lambda tap.\ tap = ([]::cell\ list, <nl>)\}\!\!\}$ $tm\ \{\!\!\{Q\}\!\!\} \Longrightarrow \{\!\!\{\lambda tap.\ tap = ([], <nl>)\}\!\!\}$ *mk_composable0 tm* $\{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

corollary *Hoare_unhalt_tm_impl_Hoare_unhalt_mk_composable0*: $(\{\!\!\{\lambda tap.\ tap = ([], <nl>)\}\!\!\}$ $tm \uparrow) \Longrightarrow (\{\!\!\{\lambda tap.\ tap = ([], <nl>)\}\!\!\}\ (mk\_composable0\ tm) \uparrow)$
⟨*proof*⟩

corollary *Hoare_halt_tm_impl_Hoare_halt_mk_composable0_pair*:
$\{\!\!\{\lambda tap.\ tap = ([], <(nl1,nl2)>)\}\!\!\}\ tm\ \{\!\!\{Q\}\!\!\} \Longrightarrow \{\!\!\{\lambda tap.\ tap = ([], <(nl1,nl2)>)\}\!\!\}$ *mk_composable0* $tm\ \{\!\!\{Q\}\!\!\}$
⟨*proof*⟩

corollary *Hoare_unhalt_tm_impl_Hoare_unhalt_mk_composable0_pair*: $(\{\!\!\{\lambda tap.\ tap = ([], <(nl1,$

$nl2)>)\}\ tm \uparrow) \Longrightarrow (\{\lambda tap.\ tap = ([], <(nl1,nl2)>)\}\ (mk\_composable0\ tm) \uparrow)$
⟨*proof*⟩

1.8 The Halt Lemma: no infinite descend

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

end

1.9 SemiId: Turing machines acting as partial identity functions

theory *SemiIdTM*
imports *Turing_Hoare*
begin

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

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

1.9.1 The Turing Machine tm_semi_id_eq0

If the input is $Oc \uparrow 1$ the machine *tm_semi_id_eq0* will reach the final state in a standard configuration with output identical to its input. For other inputs $Oc \uparrow n$ with $1 \neq n$ it will loop forever.

Please note that our short notation $<n>$ means $Oc \uparrow (n + 1)$ where $0 \leq n$.

definition *tm_semi_id_eq0* :: *instr list*
where
$tm\_semi\_id\_eq0 \stackrel{def}{=} [(WB, 1), (R, 2), (L, 0), (L, 1)]$

lemma *composable_tm0_tm_semi_id_eq0*[*intro*, *simp*]: *composable_tm0 tm_semi_id_eq0*
⟨*proof*⟩

lemma *tm_semi_id_eq0_loops_aux*:
$(steps0\ (1, [], [Oc, Oc])\ tm\_semi\_id\_eq0\ stp = (1, [], [Oc, Oc])) \lor$
$(steps0\ (1, [], [Oc, Oc])\ tm\_semi\_id\_eq0\ stp = (2, Oc\ \#\ [], [Oc]))$
⟨*proof*⟩

lemma *tm_semi_id_eq0_loops_aux′*:
(*steps0* (*1*, [], [*Oc*, *Oc*] @ (*Bk* ↑ *q*)) *tm_semi_id_eq0 stp* = (*1*, [], [*Oc*,*Oc*] @ *Bk* ↑ *q*)) ∨
(*steps0* (*1*, [], [*Oc*, *Oc*] @ (*Bk* ↑ *q*)) *tm_semi_id_eq0 stp* = (*2*, *Oc* # [], [*Oc*] @ (*Bk* ↑ *q*)))
⟨*proof*⟩

lemma *tm_semi_id_eq0_loops_aux′′*:
(*steps0* (*1*, [], [*Oc*, *Oc*] @ (*Oc* ↑ *q*) @ (*Bk* ↑ *q*)) *tm_semi_id_eq0 stp* = (*1*, [], [*Oc*,*Oc*] @ (*Oc* ↑ *q*) @ *Bk* ↑ *q*)) ∨
(*steps0* (*1*, [], [*Oc*, *Oc*] @ (*Oc* ↑ *q*) @ (*Bk* ↑ *q*)) *tm_semi_id_eq0 stp* = (*2*, *Oc* # [], [*Oc*] @ (*Oc* ↑ *q*) @ (*Bk* ↑ *q*)))
⟨*proof*⟩

lemma *tm_semi_id_eq0_loops_aux′′′*:
(*steps0* (*1*, [], []) *tm_semi_id_eq0 stp* = (*1*, [], [])) ∨
(*steps0* (*1*, [], []) *tm_semi_id_eq0 stp* = (*1*, [], [*Bk*]))
⟨*proof*⟩

lemma <*0*::*nat*> = [*Oc*] ⟨*proof*⟩
lemma *Oc*↑(*0*+*1*) = [*Oc*] ⟨*proof*⟩
lemma <*n*::*nat*> = *Oc*↑(*n*+*1*) ⟨*proof*⟩
lemma <*1*::*nat*> = [*Oc*, *Oc*] ⟨*proof*⟩

1.9.1.1 The machine tm_semi_id_eq0 in action

lemma *steps0* (*1*, [], []) *tm_semi_id_eq0 0* = (*1*, [], []) ⟨*proof*⟩
lemma *steps0* (*1*, [], []) *tm_semi_id_eq0 1* = (*1*, [], [*Bk*]) ⟨*proof*⟩
lemma *steps0* (*1*, [], []) *tm_semi_id_eq0 2* = (*1*, [], [*Bk*]) ⟨*proof*⟩
lemma *steps0* (*1*, [], []) *tm_semi_id_eq0 3* = (*1*, [], [*Bk*]) ⟨*proof*⟩

lemma *steps0* (*1*, [], [*Oc*]) *tm_semi_id_eq0 0* = (*1*, [], [*Oc*]) ⟨*proof*⟩
lemma *steps0* (*1*, [], [*Oc*]) *tm_semi_id_eq0 1* = (*2*, [*Oc*], []) ⟨*proof*⟩
lemma *steps0* (*1*, [], [*Oc*]) *tm_semi_id_eq0 2* = (*0*, [], [*Oc*]) ⟨*proof*⟩

lemma *steps0* (*1*, [], [*Oc*, *Oc*]) *tm_semi_id_eq0 0* = (*1*, [], [*Oc*, *Oc*]) ⟨*proof*⟩
lemma *steps0* (*1*, [], [*Oc*, *Oc*]) *tm_semi_id_eq0 1* = (*2*, [*Oc*], [*Oc*]) ⟨*proof*⟩
lemma *steps0* (*1*, [], [*Oc*, *Oc*]) *tm_semi_id_eq0 2* = (*1*, [], [*Oc*, *Oc*]) ⟨*proof*⟩
lemma *steps0* (*1*, [], [*Oc*, *Oc*]) *tm_semi_id_eq0 3* = (*2*, [*Oc*], [*Oc*]) ⟨*proof*⟩
lemma *steps0* (*1*, [], [*Oc*, *Oc*]) *tm_semi_id_eq0 4* = (*1*, [], [*Oc*, *Oc*]) ⟨*proof*⟩

1.9.2 The Turing Machine tm_semi_id_gt0

If the input is $Oc \uparrow 0$ or $Oc \uparrow 1$ the machine *tm_semi_id_gt0* (aka dither) will loop forever. For other non-blank inputs $Oc \uparrow n$ with $1 < n$ it will reach the final state in a standard configuration with output identical to its input.

Please note that our short notation <*n*> means $Oc \uparrow (n + 1)$ where $0 \leq n$.

definition *tm_semi_id_gt0* :: *instr list*
where
$tm\_semi\_id\_gt0 \stackrel{def}{=} [(WB, 1), (R, 2), (L, 1), (L, 0)]$

lemma *tm_semi_id_gt0*[*intro*, *simp*]: *composable_tm0 tm_semi_id_gt0*
⟨*proof*⟩

lemma *tm_semi_id_gt0_loops_aux*:
$(steps0\ (1, [], [Oc])\ tm\_semi\_id\_gt0\ stp = (1, [], [Oc])) \vee$
$(steps0\ (1, [], [Oc])\ tm\_semi\_id\_gt0\ stp = (2, Oc \# [], []))$
⟨*proof*⟩

lemma *tm_semi_id_gt0_loops_aux*$'$:
$(steps0\ (1, [], [Oc] @ Bk \uparrow n)\ tm\_semi\_id\_gt0\ stp = (1, [], [Oc] @ Bk \uparrow n)) \vee$
$(steps0\ (1, [], [Oc] @ Bk \uparrow n)\ tm\_semi\_id\_gt0\ stp = (2, Oc \# [], Bk \uparrow n))$
⟨*proof*⟩

lemma *tm_semi_id_gt0_loops_aux*$'''$:
$(steps0\ (1, [], [])\ tm\_semi\_id\_gt0\ stp = (1, [], [])) \vee$
$(steps0\ (1, [], [])\ tm\_semi\_id\_gt0\ stp = (1, [], [Bk]))$
⟨*proof*⟩

1.9.2.1 The machine tm_semi_id_gt0 in action

lemma $steps0\ (1, [], [])\ tm\_semi\_id\_gt0\ 0 = (1, [], [])$ ⟨*proof*⟩
lemma $steps0\ (1, [], [])\ tm\_semi\_id\_gt0\ 1 = (1, [], [Bk])$ ⟨*proof*⟩
lemma $steps0\ (1, [], [])\ tm\_semi\_id\_gt0\ 2 = (1, [], [Bk])$ ⟨*proof*⟩
lemma $steps0\ (1, [], [])\ tm\_semi\_id\_gt0\ 3 = (1, [], [Bk])$ ⟨*proof*⟩

lemma $steps0\ (1, [], [Oc])\ tm\_semi\_id\_gt0\ 0 = (1, [], [Oc])$ ⟨*proof*⟩
lemma $steps0\ (1, [], [Oc])\ tm\_semi\_id\_gt0\ 1 = (2, [Oc], [])$ ⟨*proof*⟩
lemma $steps0\ (1, [], [Oc])\ tm\_semi\_id\_gt0\ 2 = (1, [], [Oc])$ ⟨*proof*⟩
lemma $steps0\ (1, [], [Oc])\ tm\_semi\_id\_gt0\ 3 = (2, [Oc], [])$ ⟨*proof*⟩
lemma $steps0\ (1, [], [Oc])\ tm\_semi\_id\_gt0\ 4 = (1, [], [Oc])$ ⟨*proof*⟩

lemma $steps0\ (1, [], [Oc, Oc])\ tm\_semi\_id\_gt0\ 0 = (1, [], [Oc, Oc])$ ⟨*proof*⟩
lemma $steps0\ (1, [], [Oc, Oc])\ tm\_semi\_id\_gt0\ 1 = (2, [Oc], [Oc])$ ⟨*proof*⟩
lemma $steps0\ (1, [], [Oc, Oc])\ tm\_semi\_id\_gt0\ 2 = (0, [], [Oc, Oc])$ ⟨*proof*⟩
lemma $steps0\ (1, [], [Oc, Oc])\ tm\_semi\_id\_gt0\ 3 = (0, [], [Oc, Oc])$ ⟨*proof*⟩

1.9.3 Properties of the SemiId machines

Using Hoare style rules is more elegant since they allow for compositional reasoning. Therefore, its preferable to use them, if the program that we reason about can be decomposed appropriately.

1.9.3.1 Proving properties of tm_semi_id_eq0 with Hoare Rules

lemma *tm_semi_id_eq0_loops_Nil*:
shows $\{\!|\lambda tap.\ tap = ([], [])|\!\}\ tm\_semi\_id\_eq0 \uparrow$
$\langle proof \rangle$

lemma *tm_semi_id_eq0_loops*:
shows $\{\!|\lambda tap.\ tap = ([], <1{::}nat>)|\!\}\ tm\_semi\_id\_eq0 \uparrow$
$\langle proof \rangle$

lemma *tm_semi_id_eq0_loops′*:
shows $\{\!|\lambda tap.\ \exists\, l.\ tap = ([], [Oc, Oc] @ Bk\uparrow l)|\!\}\ tm\_semi\_id\_eq0 \uparrow$
$\langle proof \rangle$

lemma *tm_semi_id_eq0_loops′′*:
shows $\{\!|\lambda tap.\ \exists\, k\ l.\ tap = (Bk\uparrow k, [Oc, Oc] @ Bk\uparrow l)|\!\}\ tm\_semi\_id\_eq0 \uparrow$
$\langle proof \rangle$

lemma *tm_semi_id_eq0_halts_aux*:
shows $steps0\ (1, Bk \uparrow m, [Oc])\ tm\_semi\_id\_eq0\ 2 = (0, Bk \uparrow m, [Oc])$
$\langle proof \rangle$

lemma *tm_semi_id_eq0_halts_aux′*:
shows $steps0\ (1, Bk \uparrow m, [Oc]@Bk \uparrow n)\ tm\_semi\_id\_eq0\ 2 = (0, Bk \uparrow m, [Oc]@Bk \uparrow n)$
$\langle proof \rangle$

lemma *tm_semi_id_eq0_halts*:
shows $\{\!|\lambda tap.\ tap = ([], <0{::}nat>)|\!\}\ tm\_semi\_id\_eq0\ \{\!|\lambda tap.\ tap = ([], <0{::}nat>)|\!\}$
$\langle proof \rangle$

lemma *tm_semi_id_eq0_halts′*:
shows $\{\!|\lambda tap.\ \exists\, l.\ tap = ([], [Oc] @ Bk\uparrow l)|\!\}\ tm\_semi\_id\_eq0\ \{\!|\lambda tap.\ \exists\, l.\ tap = ([], [Oc] @ Bk\uparrow l)|\!\}$
$\langle proof \rangle$

lemma *tm_semi_id_eq0_halts′′*:
shows $\{\!|\ \lambda tap.\ \exists\, k\ l.\ tap = (Bk\uparrow k, [Oc] @ Bk\uparrow l)\ |\!\}\ tm\_semi\_id\_eq0\ \{\!|\ \lambda tap.\ \exists\, k\ l.\ tap = (Bk\uparrow k, [Oc] @ Bk\uparrow l)\ |\!\}$
$\langle proof \rangle$

1.9.3.2 Proving properties of tm_semi_id_gt0 with Hoare Rules

lemma *tm_semi_id_gt0_loops_Nil*:
shows $\{\!|\lambda tap.\ tap = ([], [])|\!\}\ tm\_semi\_id\_gt0 \uparrow$
$\langle proof \rangle$

lemma *tm_semi_id_gt0_loops*:
shows $\{\!|\lambda tap.\ tap = ([], <0{::}nat>)|\!\}\ tm\_semi\_id\_gt0 \uparrow$

⟨*proof*⟩

lemma *tm_semi_id_gt0_loops′*:
 shows $\{\!|\lambda tap.\ \exists\, l.\ tap = ([], [Oc]\ @\ Bk{\uparrow}\ l)|\!\}$ *tm_semi_id_gt0* ↑
 ⟨*proof*⟩

lemma *tm_semi_id_gt0_loops′′*:
 shows $\{\!|\lambda tap.\ \exists\, k\ l.\ tap = (Bk{\uparrow}k, [Oc]\ @\ Bk{\uparrow}\ l)|\!\}$ *tm_semi_id_gt0* ↑
 ⟨*proof*⟩

lemma *tm_semi_id_gt0_halts_aux*:
 shows *steps0* $(1, Bk \uparrow m, [Oc, Oc])$ *tm_semi_id_gt0* $2 = (0, Bk \uparrow m, [Oc, Oc])$
 ⟨*proof*⟩

lemma *tm_semi_id_gt0_halts_aux′*:
 shows *steps0* $(1, Bk \uparrow m, [Oc, Oc]@Bk \uparrow n)$ *tm_semi_id_gt0* $2 = (0, Bk \uparrow m, [Oc, Oc]@Bk \uparrow n)$
 ⟨*proof*⟩

lemma *tm_semi_id_gt0_halts*:
 shows $\{\!|\lambda tap.\ tap = ([], <1::nat>)|\!\}$ *tm_semi_id_gt0* $\{\!|\lambda tap.\ tap = ([], <1::nat>)|\!\}$
 ⟨*proof*⟩

lemma *tm_semi_id_gt0_halts′*:
 shows $\{\!|\lambda tap.\ \exists\, l.\ tap = ([], [Oc, Oc]\ @\ Bk{\uparrow}\ l)|\!\}$ *tm_semi_id_gt0* $\{\!|\lambda tap.\ \exists\, l.\ tap = ([], [Oc, Oc]\ @\ Bk{\uparrow}\ l)|\!\}$
 ⟨*proof*⟩

lemma *tm_semi_id_gt0_halts′′*:
 shows $\{\!|\ \lambda tap.\ \exists\, k\ l.\ tap = (Bk{\uparrow}\ k, [Oc, Oc]\ @\ Bk{\uparrow}\ l)|\!\}$ *tm_semi_id_gt0* $\{\!|\lambda tap.\ \exists\, k\ l.\ tap = (Bk{\uparrow}\ k, [Oc, Oc]\ @\ Bk{\uparrow}\ l)|\!\}$
 ⟨*proof*⟩

end

1.10 Halting Conditions and Standard Halting Configuration

theory *Turing_HaltingConditions*
 imports *Turing_Hoare*
begin

1.10.1 Looping of Turing Machines

definition *TMC_loops* :: *tprog0* ⇒ *nat list* ⇒ *bool*
 where
  TMC_loops p ns $\stackrel{def}{=}$ (∀ *stp*.¬ *is_final* (*steps0* (*1*, [], <*ns*::*nat list*>) *p stp*))

1.10.2 Reaching the Final State

definition *reaches_final* :: *tprog0* ⇒ *nat list* ⇒ *bool*
 where
 reaches_final p ns $\overset{def}{=}$ ⦃(λ*tap*. *tap* = ([], <*ns*>))⦄ *p* ⦃(λ*tap*. *True*)⦄

The definition *reaches_final* and all lemmas about it are only needed for the special halting problem K0.

lemma *True_holds_for_all*: (λ*tap*. *True*) *holds_for c*
 ⟨*proof*⟩

lemma *reaches_final_iff*: *reaches_final p ns* ⟷ (∃ *n*. *is_final* (*steps0* (*1*, ([], <*ns*>)) *p n*))
 ⟨*proof*⟩

Some lemmas about reaching the Final State.

lemma *Hoare_halt_from_init_imp_reaches_final*:
 assumes ⦃λ*tap*. *tap* = ([], <*ns*>)⦄ *p* ⦃*Q*⦄
 shows *reaches_final p ns*
⟨*proof*⟩

lemma *Hoare_unhalt_impl_not_reaches_final*:
 assumes ⦃(λ*tap*. *tap* = ([], <*ns*>))⦄ *p* ↑
 shows ¬(*reaches_final p ns*)
⟨*proof*⟩

1.10.3 What is a Standard Tape

A tape is called *standard*, if the left tape is empty or contains only blanks and the right tape contains a single nonempty block of strokes (occupied cells) followed by zero or more blanks..

Thus, by definition of left and right tape, the head of the machine is always scanning the first cell of this single block of strokes.

We extend the notion of a standard tape to lists of numerals as well.

definition *std_tap* :: *tape* ⇒ *bool*
 where
 std_tap tap $\overset{def}{=}$ (∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*))

definition *std_tap_list* :: *tape* ⇒ *bool*
 where
 std_tap_list tap $\overset{def}{=}$ (∃ *k ml l*. *tap* = (*Bk* ↑ *k*, <*ml*::*nat list*> @ *Bk* ↑ *l*))

lemma *std_tap tap* ⟹ *std_tap_list tap*
 ⟨*proof*⟩

A configuration (*st*, *l*, *r*) of a Turing machine is called a *standard configuration*, if the state *st* is the final state 0 and the (*l*, *r*) is a standard tape.

definition *TSTD*′:: *config* ⇒ *bool*

where
$TSTD'\ c = ((let\ (st, l, r) = c\ in$
$\quad st = 0 \wedge (\exists\ m.\ l = Bk{\uparrow}(m)) \wedge (\exists\ rs\ n.\ r = Oc{\uparrow}(Suc\ rs) @ Bk{\uparrow}(n))))$

lemma $TSTD'\ (st, l, r) = ((st = 0) \wedge std\_tap\ (l,r))$
$\langle proof \rangle$

1.10.4 What does Hoare_halt mean in general?

We say *in general* because the result computed on the right tape is not necessarily a numeral but some arbitrary component r'.

lemma *Hoare_halt2_iff*:
$\{\!|\lambda tap.\ \exists kl\ ll.\ tap = (Bk \uparrow kl, r @ Bk \uparrow ll)|\!\}\ p\ \{\!|\lambda tap.\ \exists kr\ lr.\ tap = (Bk \uparrow kr, r' @ Bk \uparrow lr)|\!\}$
$\longleftrightarrow$
$(\forall kl\ ll.\ \exists n.\ is\_final\ (steps0\ (1, (Bk \uparrow kl, r @ Bk \uparrow ll))\ p\ n) \wedge (\exists kr\ lr.\ steps0\ (1, (Bk \uparrow kl, r @ Bk \uparrow ll))\ p\ n = (0, Bk \uparrow kr, r' @ Bk \uparrow lr)))$
$\langle proof \rangle$

lemma *Hoare_halt_D*:
assumes $\{\!|\lambda tap.\ \exists kl\ ll.\ tap = (Bk \uparrow kl, r @ Bk \uparrow ll)|\!\}\ p\ \{\!|\lambda tap.\ \exists kr\ lr.\ tap = (Bk \uparrow kr, r' @ Bk \uparrow lr)|\!\}$
shows $\exists n.\ is\_final\ (steps0\ (1, (Bk \uparrow kl, r @ Bk \uparrow ll))\ p\ n) \wedge (\exists kr\ lr.\ steps0\ (1, (Bk \uparrow kl, r @ Bk \uparrow ll))\ p\ n = (0, Bk \uparrow kr, r' @ Bk \uparrow lr))$
$\langle proof \rangle$

lemma *Hoare_halt_I2*:
assumes $\bigwedge kl\ ll.\ \exists n.\ is\_final\ (steps0\ (1, (Bk \uparrow kl, r @ Bk \uparrow ll))\ p\ n) \wedge (\exists kr\ lr.\ steps0\ (1, (Bk \uparrow kl, r @ Bk \uparrow ll))\ p\ n = (0, Bk \uparrow kr, r' @ Bk \uparrow lr))$
shows $\{\!|\lambda tap.\ \exists kl\ ll.\ tap = (Bk \uparrow kl, r @ Bk \uparrow ll)|\!\}\ p\ \{\!|\lambda tap.\ \exists kr\ lr.\ tap = (Bk \uparrow kr, r' @ Bk \uparrow lr)|\!\}$
$\langle proof \rangle$

lemma *Hoare_halt_D_Nil*:
assumes $\{\!|\lambda tap.\ tap = ([], r)|\!\}\ p\ \{\!|\lambda tap.\ \exists kr\ lr.\ tap = (Bk \uparrow kr, r' @ Bk \uparrow lr)|\!\}$
shows $\exists n.\ is\_final\ (steps0\ (1, ([], r))\ p\ n) \wedge (\exists kr\ lr.\ steps0\ (1, ([], r))\ p\ n = (0, Bk \uparrow kr, r' @ Bk \uparrow lr))$
$\langle proof \rangle$

lemma *Hoare_halt_I2_Nil*:
assumes $\exists n.\ is\_final\ (steps0\ (1, ([], r\ ))\ p\ n) \wedge (\exists kr\ lr.\ steps0\ (1, ([], r\ ))\ p\ n = (0, Bk \uparrow kr, r' @ Bk \uparrow lr))$
shows $\{\!|\lambda tap.\ tap = ([], r)|\!\}\ p\ \{\!|\lambda tap.\ \exists kr\ lr.\ tap = (Bk \uparrow kr, r' @ Bk \uparrow lr)|\!\}$
$\langle proof \rangle$

lemma *Hoare_halt2_Nil_iff*:
$\{\!|\lambda tap.\ tap = ([], r)|\!\}\ p\ \{\!|\lambda tap.\ \exists kr\ lr.\ tap = (Bk \uparrow kr, r' @ Bk \uparrow lr)|\!\}$

$\longleftrightarrow$
$(\exists n.\ is\_final\ (steps0\ (1, ([], r))\ p\ n) \wedge (\exists kr\ lr.\ steps0\ (1, ([], r))\ p\ n = (0, Bk \uparrow kr, r' @ Bk \uparrow lr)))$
$\langle proof \rangle$

corollary *Hoare_halt_left_tape_Nil_imp_All_left_and_right*:
assumes $\{\!|\lambda tap.\ tap = ([], r)|\!\}\ p\ \{\!|\lambda tap.\ \exists k\ l.\ tap = (Bk \uparrow k, r' @ Bk \uparrow l)|\!\}$
shows $\{\!|\lambda tap.\ \exists x\ y.\ tap = (Bk \uparrow x, r @ Bk \uparrow y)|\!\}\ p\ \{\!|\lambda tap.\ \exists k\ l.\ tap = (Bk \uparrow k, r' @ Bk \uparrow l)|\!\}$
$\langle proof \rangle$

1.10.4.1 What does Hoare_halt with a numeral list result mean?

About computations that result in numeral lists on the final right tape.

lemma *TMC_has_num_res_list_without_initial_Bks_imp_TMC_has_num_res_list_after_adding_Bks_to_initial_right_tape*:
$\{\!|\lambda tap.\ tap = ([], <ns::nat\ list>)|\!\}\ p\ \{\!|\lambda tap.\ \exists ms\ kr\ lr.\ tap = (Bk \uparrow kr, <ms::nat\ list> @ Bk \uparrow lr)|\!\}$
$\Longrightarrow$
$\{\!|\lambda tap.\ \exists ll.\ tap = ([], <ns::nat\ list> @ Bk \uparrow ll)|\!\}\ p\ \{\!|\lambda tap.\ \exists ms\ kr\ lr.\ tap = (Bk \uparrow kr, <ms::nat\ list> @ Bk \uparrow lr)|\!\}$
$\langle proof \rangle$

lemma *TMC_has_num_res_list_without_initial_Bks_iff_TMC_has_num_res_list_after_adding_Bks_to_initial_right_tape*:
$\{\!|\lambda tap.\ tap = ([], <ns::nat\ list>)|\!\}\ p\ \{\!|\lambda tap.\ \exists ms\ kr\ lr.\ tap = (Bk \uparrow kr, <ms::nat\ list> @ Bk \uparrow lr)|\!\}$
$\longleftrightarrow$
$\{\!|\lambda tap.\ \exists ll.\ tap = ([], <ns::nat\ list> @ Bk \uparrow ll)|\!\}\ p\ \{\!|\lambda tap.\ \exists ms\ kr\ lr.\ tap = (Bk \uparrow kr, <ms::nat\ list> @ Bk \uparrow lr)|\!\}$
$\langle proof \rangle$

lemma *TMC_has_num_res_list_without_initial_Bks_imp_TMC_has_num_res_list_after_adding_Bks_to_initial_left_and_right_tape*
$\{\!|\lambda tap.\ tap = ([], <ns::nat\ list>)|\!\}\ p\ \{\!|\lambda tap. \exists kr\ lr.\ tap = (Bk \uparrow kr, <ms::nat\ list> @ Bk \uparrow lr)|\!\}$
$\Longrightarrow$
$\{\!|\lambda tap.\ \exists kl\ ll.\ tap = (Bk \uparrow kl, <ns::nat\ list> @ Bk \uparrow ll)|\!\}\ p\ \{\!|\lambda tap.\ \exists kr\ lr.\ tap = (Bk \uparrow kr, <ms::nat\ list> @ Bk \uparrow lr)|\!\}$
$\langle proof \rangle$

lemma *TMC_has_num_res_list_without_initial_Bks_iff_TMC_has_num_res_list_after_adding_Bks_to_initial_left_and_right_tape*:
$\{\!|\lambda tap.\ tap = ([], <ns::nat\ list>)|\!\}\ p\ \{\!|\lambda tap. \exists kr\ lr.\ tap = (Bk \uparrow kr, <ms::nat\ list> @ Bk \uparrow lr)|\!\}$
$\longleftrightarrow$
$\{\!|\lambda tap.\ \exists kl\ ll.\ tap = (Bk \uparrow kl, <ns::nat\ list> @ Bk \uparrow ll)|\!\}\ p\ \{\!|\lambda tap.\ \exists kr\ lr.\ tap = (Bk \uparrow kr, <ms::nat\ list> @ Bk \uparrow lr)|\!\}$
$\langle proof \rangle$

1.10.5 Halting in a Standard Configuration

1.10.5.1 Definition of Halting in a Standard Configuration

The predicates *TMC_has_num_res p ns* and *TMC_has_num_list_res* describe that a run of the Turing program *p* on input *ns* reaches the final state 0 and the final tape produced thereby is standard. Thus, the computation of the Turing machine *p* produced a result, which is either a single numeral or a list of numerals.

Since trailing blanks on the initial left or right tape do not matter, we may restrict our definitions to the case where the initial left tape is empty and there are no trailing blanks on the initial right tape!

definition *TMC_has_num_res* :: *tprog0* ⇒ *nat list* ⇒ *bool*
 where
  TMC_has_num_res p ns $\stackrel{def}{=}$
  ⦃ λ*tap*. *tap* = ([], <*ns*>) ⦄ *p* ⦃ λ*tap*. (∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)) ⦄

lemma *TMC_has_num_res_iff*: *TMC_has_num_res p ns*
  ⟷
  (∃ *stp*. *is_final* (*steps0* (*1*, [],<*ns*::*nat list*>) *p stp*) ∧
    (∃ *k n l*. *steps0* (*1*, [],<*ns*::*nat list*>) *p stp* = (*0*, *Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)))
 ⟨*proof*⟩

definition *TMC_has_num_list_res* :: *tprog0* ⇒ *nat list* ⇒ *bool*
 where
  TMC_has_num_list_res p ns $\stackrel{def}{=}$
  ⦃λ*tap*. *tap* = ([], <*ns*::*nat list*>)⦄ *p* ⦃λ*tap*. ∃ *kr ms lr*. *tap* = (*Bk* ↑ *kr*, <*ms*::*nat list*> @ *Bk* ↑ *lr*)⦄

lemma *TMC_has_num_list_res_iff*: *TMC_has_num_list_res p ns*
  ⟷
  (∃ *stp*. *is_final* (*steps0* (*1*, [],<*ns*::*nat list*>) *p stp*) ∧
    (∃ *k ms l*. *steps0* (*1*, [],<*ns*::*nat list*>) *p stp* = (*0*, *Bk* ↑ *k*, <*ms*::*nat list*> @ *Bk* ↑ *l*)))
 ⟨*proof*⟩

1.10.5.2 Relation between TMC_has_num_res and TMC_has_num_list_res

A computation of a Turing machine, which started on a list of numerals and halts in a standard configuration with a single numeral result is a special case of a halt in a standard configuration that halts with a list of numerals.

theorem *TMC_has_num_res_imp_TMC_has_num_list_res*:
 ⦃λ*tap*. *tap* = ([], <*ns*::*nat list*>)⦄ *p* ⦃λ*tap*. ∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)⦄
 ⟹
  ⦃λ*tap*. *tap* = ([], <*ns*::*nat list*>)⦄ *p* ⦃λ*tap*. ∃ *kr ms lr*. *tap* = (*Bk* ↑ *kr*, <*ms*::*nat list*> @ *Bk* ↑ *lr*)⦄
⟨*proof*⟩

corollary *TMC_has_num_res_imp_TMC_has_num_list_res'*: *TMC_has_num_res p ns* $\Longrightarrow$ *TMC_has_num_list_res p ns*
⟨*proof*⟩

1.10.5.3 Convenience Lemmas for Halting Problems

lemma *Hoare_halt_with_Oc_imp_std_tap*:
assumes $\{\!\!\{(\lambda tap.\ tap = ([], <ns::nat\ list>))\}\!\!\}\ p\ \{\!\!\{\lambda tap.\ \exists\, k\ l.\ tap = (Bk \uparrow k, [Oc]\ @\ Bk \uparrow l)\}\!\!\}$
shows *TMC_has_num_res p ns*
⟨*proof*⟩

lemma *Hoare_halt_with_OcOc_imp_std_tap*:
assumes $\{\!\!\{(\lambda tap.\ tap = ([], <ns::nat\ list>))\}\!\!\}\ p\ \{\!\!\{\lambda tap.\ \exists\, k\ l.\ tap = (Bk \uparrow k, [Oc, Oc]\ @\ Bk \uparrow l)\}\!\!\}$
shows *TMC_has_num_res p ns*
⟨*proof*⟩

1.10.5.4 Hoare_halt on numeral lists with single numeral result

lemma *Hoare_halt_on_numeral_imp_result*:
assumes $\{\!\!\{(\lambda tap.\ tap = ([], <ns::nat\ list>))\}\!\!\}\ p\ \{\!\!\{(\lambda tap.\ (\exists\, k\ n\ l.\ tap = (Bk \uparrow k, <n::nat>\ @\ Bk \uparrow l)))\}\!\!\}$
shows $\exists\, stp\ k\ n\ l.\ steps0\ (1, [],<ns::nat\ list>)\ p\ stp = (0, Bk \uparrow k, <n::nat>\ @\ Bk\uparrow l)$
⟨*proof*⟩

lemma *Hoare_halt_on_numeral_imp_result_rev*:
assumes $\exists\, stp\ k\ n\ l.\ steps0\ (1, [],<ns::nat\ list>)\ p\ stp = (0, Bk \uparrow k, <n::nat>\ @\ Bk\uparrow l)$
shows $\{\!\!\{(\lambda tap.\ tap = ([], <ns::nat\ list>))\}\!\!\}\ p\ \{\!\!\{(\lambda tap.\ (\exists\, k\ n\ l.\ tap = (Bk \uparrow k, <n::nat>\ @\ Bk \uparrow l)))\}\!\!\}$
⟨*proof*⟩

lemma *Hoare_halt_on_numeral_imp_result_iff*:
$\{\!\!\{(\lambda tap.\ tap = ([], <ns::nat\ list>))\}\!\!\}\ p\ \{\!\!\{(\lambda tap.\ (\exists\, k\ n\ l.\ tap = (Bk \uparrow k, <n::nat>\ @\ Bk \uparrow l)))\}\!\!\}$
$\longleftrightarrow$
$(\exists\, stp\ k\ n\ l.\ steps0\ (1, [],<ns::nat\ list>)\ p\ stp = (0, Bk \uparrow k, <n::nat>\ @\ Bk\uparrow l))$
⟨*proof*⟩

1.10.5.5 Hoare_halt on numeral lists with numeral list result

lemma *Hoare_halt_on_numeral_imp_list_result*:
assumes $\{\!\!\{(\lambda tap.\ tap = ([], <ns::nat\ list>))\}\!\!\}\ p\ \{\!\!\{(\lambda tap.\ (\exists\, k\ ms\ l.\ tap = (Bk \uparrow k, <ms::nat\ list>\ @\ Bk \uparrow l)))\}\!\!\}$
shows $\exists\, stp\ k\ ms\ l.\ steps0\ (1, [],<ns::nat\ list>)\ p\ stp = (0, Bk \uparrow k, <ms::nat\ list>\ @\ Bk\uparrow l)$
⟨*proof*⟩

lemma *Hoare_halt_on_numeral_imp_list_result_rev*:
assumes $\exists\, stp\ k\ ms\ l.\ steps0\ (1, [],<ns::nat\ list>)\ p\ stp = (0, Bk \uparrow k, <ms::nat\ list>\ @\ Bk\uparrow l)$
shows $\{\!\!\{(\lambda tap.\ tap = ([], <ns::nat\ list>))\}\!\!\}\ p\ \{\!\!\{(\lambda tap.\ (\exists\, k\ ms\ l.\ tap = (Bk \uparrow k, <ms::nat\ list>\ @\ Bk \uparrow l)))\}\!\!\}$

$\langle proof \rangle$

lemma *Hoare_halt_on_numeral_imp_list_result_iff*:
$\{\!\{(\lambda tap.\ tap = ([], <ns::nat\ list>))\}\!\}\ p\ \{\!\{(\lambda tap.\ (\exists k\ ms\ l.\ tap = (Bk \uparrow k, <ms::nat\ list> @ Bk \uparrow l)))\}\!\}$
$\longleftrightarrow$
$(\exists stp\ k\ ms\ l.\ steps0\ (1, [], <ns::nat\ list>)\ p\ stp = (0, Bk \uparrow k, <ms::nat\ list> @ Bk \uparrow l))$
$\langle proof \rangle$

1.10.6 Trailing left blanks do not matter for computations with result

lemma *TMC_has_num_res_NIL_impl_TMC_has_num_res_with_left_BKs*:
$\{\!\{(\lambda tap.\ tap = ([], <ns::nat\ list>))\}\!\}\ p\ \{\!\{\ \lambda tap.\ (\exists k\ n\ l.\ tap = (Bk \uparrow k, <n::nat> @ Bk \uparrow l))\ \}\!\}$
$\Longrightarrow$
$\{\!\{(\lambda tap.\ \exists z.\ tap = (Bk \uparrow z, <ns>))\}\!\}\ p\ \{\!\{\ \lambda tap.\ (\exists k\ n\ l.\ tap = (Bk \uparrow k, <n::nat> @ Bk \uparrow l))\ \}\!\}$
$\langle proof \rangle$

corollary *TMC_has_num_res_NIL_iff_TMC_has_num_res_with_left_BKs*:
$\{\!\{(\lambda tap.\ tap = ([], <ns::nat\ list>))\}\!\}\ p\ \{\!\{\ \lambda tap.\ (\exists k\ n\ l.\ tap = (Bk \uparrow k, <n::nat> @ Bk \uparrow l))\ \}\!\}$
$\longleftrightarrow$
$\{\!\{(\lambda tap.\ \exists z.\ tap = (Bk \uparrow z, <ns>))\}\!\}\ p\ \{\!\{\ \lambda tap.\ (\exists k\ n\ l.\ tap = (Bk \uparrow k, <n::nat> @ Bk \uparrow l))\ \}\!\}$
$\langle proof \rangle$

1.10.7 About Turing Computations and the result they yield

definition *TMC_yields_num_res* :: $tprog0 \Rightarrow nat\ list \Rightarrow nat \Rightarrow bool$
where $TMC\_yields\_num\_res\ tm\ ns\ n \stackrel{def}{=} (\exists stp\ k\ l.\ (steps0\ (1, ([], <ns::nat\ list>))\ tm\ stp) = (0, Bk \uparrow k, <n::nat> @ Bk \uparrow l))$

definition *TMC_yields_num_list_res* :: $tprog0 \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
where $TMC\_yields\_num\_list\_res\ tm\ ns\ ms \stackrel{def}{=} (\exists stp\ k\ l.\ (steps0\ (1, ([], <ns::nat\ list>))\ tm\ stp) = (0, Bk \uparrow k, <ms::nat\ list> @ Bk \uparrow l))$

lemma *TMC_yields_num_res_unfolded_into_Hoare_halt*:
$TMC\_yields\_num\_res\ tm\ ns\ n \stackrel{def}{=} \{\!\{(\lambda tap.\ tap = ([], <ns>))\}\!\}\ tm\ \{\!\{(\lambda tap.\ \exists k\ l.\ tap = (Bk \uparrow k, <n::nat> @ Bk \uparrow l))\}\!\}$
$\langle proof \rangle$

lemma *TMC_yields_num_list_res_unfolded_into_Hoare_halt*:
$TMC\_yields\_num\_list\_res\ tm\ ns\ ms \stackrel{def}{=} \{\!\{(\lambda tap.\ tap = ([], <ns>))\}\!\}\ tm\ \{\!\{(\lambda tap.\ \exists k\ l.\ tap = (Bk \uparrow k, <ms::nat\ list> @ Bk \uparrow l))\}\!\}$
$\langle proof \rangle$

lemma *TMC_yields_num_res_Hoare_plus_halt*:
assumes *TMC_yields_num_list_res tm1 nl r1*
and *TMC_yields_num_res tm2 r1 r2*
and *composable_tm0 tm1*
shows *TMC_yields_num_res* (*tm1* |+| *tm2*) *nl r2*
⟨*proof*⟩

end

1.10.8 tm_onestroke: A Machine for deciding the empty set

theory *OneStrokeTM*
imports
Turing_Hoare
begin

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

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

1.10.8.1 Definition of the machine tm_onestroke

We can rely on the fact, that on the initial tape, two consecutive blanks mean end of input (see theorem *noDblBk* (<*?nl*>)).

Thus, the machine can check both ends of the (initial) tape. Note however, that this is just a convention for encoding arguments for functions. Nevertheless, the tape is (potentially) infinite on both sides.

definition *tm_onestroke* :: *instr list*
where
tm_onestroke $\stackrel{def}{=}$ [(*R*, *3*),(*WB*,*2*), (*R*,*1*),(*R*,*1*), (*WO*,*0*),(*WB*,*2*)]

1.10.8.2 The machine tm_onestroke in action

value *steps0* (*1*, [], <([]::*nat list*)>) *tm_onestroke 0*
value *steps0* (*1*, [], <([]::*nat list*)>) *tm_onestroke 1*
value *steps0* (*1*, [], <([]::*nat list*)>) *tm_onestroke 2*

lemma *steps0* (*1*, [], <([]::*nat list*)>) *tm_onestroke 2* = (*0*, [*Bk*], [*Oc*])
⟨*proof*⟩

value *steps0* (*1*, [], <[*0*::*nat*]>) *tm_onestroke 0*
value *steps0* (*1*, [], <[*0*::*nat*]>) *tm_onestroke 1*
value *steps0* (*1*, [], <[*0*::*nat*]>) *tm_onestroke 2*
value *steps0* (*1*, [], <[*0*::*nat*]>) *tm_onestroke 3*
value *steps0* (*1*, [], <[*0*::*nat*]>) *tm_onestroke 4*

lemma *steps0* (*1*, [], <[*0*::*nat*]>) *tm_onestroke 4* = (*0*, [*Bk*, *Bk*], [*Oc*])
⟨*proof*⟩

lemma *steps0* (*1*, [], <[*0*::*nat*,*0*]>) *tm_onestroke 7* = (*0*, [*Bk*, *Bk*, *Bk*, *Bk*], [*Oc*])
⟨*proof*⟩

lemma *steps0* (*1*, [], <[*1*::*nat*,*1*]>) *tm_onestroke 11* = (*0*, [*Bk*, *Bk*, *Bk*, *Bk*, *Bk*, *Bk*], [*Oc*])
⟨*proof*⟩

1.10.8.3 Partial and Total Correctness of machine tm_onestroke

fun
inv_tm_onestroke1 :: *tape* ⇒ *bool* **and**
inv_tm_onestroke2 :: *tape* ⇒ *bool* **and**
inv_tm_onestroke3 :: *tape* ⇒ *bool* **and**
inv_tm_onestroke0 :: *tape* ⇒ *bool*
where
inv_tm_onestroke1 (*l*, *r*) =
(*noDblBk r* ∧ *l* = *Bk*↑ (*length l*))
| *inv_tm_onestroke2* (*l*, *r*) =
(*noDblBk* (*tl r*) ∧ *l* = *Bk*↑ (*length l*) ∧ (∃ *rs*. *r* = *Bk*#*rs*))
| *inv_tm_onestroke3* (*l*, *r*) =
(*noDblBk r* ∧ *l* = *Bk*↑ (*length l*) ∧ (*r*= [] ∨ (∃ *rs*. *r* = *Oc*#*rs*)))
| *inv_tm_onestroke0* (*l*, *r*) =
(*noDblBk r* ∧ *l* = *Bk*↑ (*length l*) ∧ (*r* = [*Oc*]))

fun *inv_tm_onestroke* :: *config* ⇒ *bool*
where
inv_tm_onestroke (*s*, *tap*) =
(*if s = 0 then inv_tm_onestroke0 tap else*
if s = 1 then inv_tm_onestroke1 tap else
if s = 2 then inv_tm_onestroke2 tap else
if s = 3 then inv_tm_onestroke3 tap
else False)

lemma *tm_onestroke_cases*:
fixes *s*::*nat*
assumes *inv_tm_onestroke* (*s*,*l*,*r*)
and *s*=*0* ⟹ *P*
and *s*=*1* ⟹ *P*
and *s*=*2* ⟹ *P*

and *s=3* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *inv_tm_onestroke_step*:
assumes *inv_tm_onestroke cf*
shows *inv_tm_onestroke* (*step0 cf tm_onestroke*)
⟨*proof*⟩

lemma *inv_tm_onestroke_steps*:
assumes *inv_tm_onestroke cf*
shows *inv_tm_onestroke* (*steps0 cf tm_onestroke stp*)
⟨*proof*⟩

lemma *tm_onestroke_partial_correctness*:
assumes ∃ *stp*. *is_final* (*steps0* (*1*, [], <*nl*:: *nat list*>) *tm_onestroke stp*)
shows ⦃ λ*tap*. *tap* = ([], <*nl*:: *nat list*>) ⦄
tm_onestroke
⦃ λ*tap*. ∃ *k l*. *tap* = (*Bk* ↑ *k*, [*Oc*] @ *Bk* ↑*l*) ⦄
⟨*proof*⟩

definition *measure_tm_onestroke* :: (*config* × *config*) *set*
where
measure_tm_onestroke = *measures* [
λ(*s*, *l*, *r*). (*if s* = *0 then 0 else 1*),
λ(*s*, *l*, *r*). *length r*,
λ(*s*, *l*, *r*). *count_list r Oc*,
λ(*s*, *l*, *r*). (*if s* = *3 then 0 else 1*)
]

lemma *wf_measure_tm_onestroke*: *wf measure_tm_onestroke*
⟨*proof*⟩

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

lemma *tm_onestroke_induct_halts*: ∃ *stp*. *is_final* (*steps0* (*1*, [], <*nl*:: *nat list*>) *tm_onestroke stp*)
⟨*proof*⟩

lemma *tm_onestroke_total_correctness*:
⦃ λ*tap*. *tap* = ([], <*nl*:: *nat list*>) ⦄ *tm_onestroke* ⦃ λ*tap*. ∃ *k l*. *tap* = (*Bk* ↑ *k*, [*Oc*] @ *Bk* ↑*l*)

⟧
⟨*proof*⟩

end

1.10.9 Machines that duplicate a single Numeral

1.10.9.1 A Turing machine that duplicates its input if the input is a single numeral

The Machine WeakCopyTM does not check the number of its arguments on the initial tape. If it is provided a single numeral it does a perfect job. However, if it gets no or more than one argument, it does not complain but produces some result.

theory *WeakCopyTM*
 imports
  Turing_HaltingConditions
begin

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

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

fun
 inv_begin0 :: *nat* ⇒ *tape* ⇒ *bool* **and**
 inv_begin1 :: *nat* ⇒ *tape* ⇒ *bool* **and**
 inv_begin2 :: *nat* ⇒ *tape* ⇒ *bool* **and**
 inv_begin3 :: *nat* ⇒ *tape* ⇒ *bool* **and**
 inv_begin4 :: *nat* ⇒ *tape* ⇒ *bool*
 where
  inv_begin0 n (*l*, *r*) = ((*n* > *1* ∧ (*l*, *r*) = (*Oc* ↑ (*n* − *2*), [*Oc*, *Oc*, *Bk*, *Oc*])) ∨
           (*n* = *1* ∧ (*l*, *r*) = ([], [*Bk*, *Oc*, *Bk*, *Oc*])))
 | *inv_begin1 n* (*l*, *r*) = ((*l*, *r*) = ([], *Oc* ↑ *n*))
 | *inv_begin2 n* (*l*, *r*) = (∃ *i j*. *i* > *0* ∧ *i* + *j* = *n* ∧ (*l*, *r*) = (*Oc* ↑ *i*, *Oc* ↑ *j*))
 | *inv_begin3 n* (*l*, *r*) = (*n* > *0* ∧ (*l*, *tl r*) = (*Bk* # *Oc* ↑ *n*, []))
 | *inv_begin4 n* (*l*, *r*) = (*n* > *0* ∧ (*l*, *r*) = (*Oc* ↑ *n*, [*Bk*, *Oc*]) ∨ (*l*, *r*) = (*Oc* ↑ (*n* − *1*), [*Oc*, *Bk*, *Oc*]))

fun *inv_begin* :: *nat* ⇒ *config* ⇒ *bool*
 where
  inv_begin n (*s*, *tap*) =
   (*if s* = *0 then inv_begin0 n tap else*

if s = 1 then inv_begin1 n tap else
if s = 2 then inv_begin2 n tap else
if s = 3 then inv_begin3 n tap else
if s = 4 then inv_begin4 n tap
else False)

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

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

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

lemma *begin_partial_correctness*:
assumes *is_final* (*steps0* (*1*, [], $Oc \uparrow n$) *tm_copy_begin_orig stp*)
shows $0 < n \Longrightarrow$ ⦃*inv_begin1 n*⦄ *tm_copy_begin_orig* ⦃*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 = [] ∨ 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*, [], $Oc \uparrow x$) *tm_copy_begin_orig stp*)
$\langle proof \rangle$

lemma *begin_correct*:
  shows $0 < n \Longrightarrow$ $\{\!|$*inv_begin1 n*$|\!\}$ *tm_copy_begin_orig* $\{\!|$*inv_begin0 n*$|\!\}$
  $\langle proof \rangle$

lemma *begin_correct2*:
  assumes $0 < (n::nat)$
  shows $\{\!|\lambda tap.\ tap = ([]::cell\ list, Oc \uparrow n)|\!\}$
    tm_copy_begin_orig
    $\{\!|\lambda tap.\ (n > 1 \wedge tap = (Oc \uparrow (n - 2), [Oc, Oc, Bk, Oc])) \vee$
      $(n = 1 \wedge tap = ([]::cell\ list, [Bk, Oc, Bk, Oc]))\ |\!\}$
$\langle proof \rangle$

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

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

fun
  inv_loop1_loop :: *nat* $\Rightarrow$ *tape* $\Rightarrow$ *bool* **and**
  inv_loop1_exit :: *nat* $\Rightarrow$ *tape* $\Rightarrow$ *bool* **and**
  inv_loop5_loop :: *nat* $\Rightarrow$ *tape* $\Rightarrow$ *bool* **and**
  inv_loop5_exit :: *nat* $\Rightarrow$ *tape* $\Rightarrow$ *bool* **and**
  inv_loop6_loop :: *nat* $\Rightarrow$ *tape* $\Rightarrow$ *bool* **and**
  inv_loop6_exit :: *nat* $\Rightarrow$ *tape* $\Rightarrow$ *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
$inv\_loop0\ n\ (l, r) = (0 < n \wedge (l, r) = ([Bk], Oc \# Bk\uparrow n\ @\ Oc\uparrow n))$
| $inv\_loop1\ n\ (l, r) = (inv\_loop1\_loop\ n\ (l, r) \vee inv\_loop1\_exit\ n\ (l, r))$
| $inv\_loop2\ n\ (l, r) = (\exists\ i\ j\ any.\ i + j = n \wedge n > 0 \wedge i > 0 \wedge j > 0 \wedge (l, r) = (Oc\uparrow i, any \# Bk\uparrow j @ Oc\uparrow j))$
| $inv\_loop3\ n\ (l, r) =$
$(\exists\ i\ j\ k\ t.\ i + j = n \wedge i > 0 \wedge j > 0 \wedge\ k + t = Suc\ j \wedge (l, r) = (Bk\uparrow k @ Oc\uparrow i, Bk\uparrow t @ Oc\uparrow j))$
| $inv\_loop4\ n\ (l, r) =$
$(\exists\ i\ j\ k\ t.\ i + j = n \wedge i > 0 \wedge j > 0 \wedge\ k + t = j \wedge (l, r) = (Oc\uparrow k\ @\ Bk\uparrow(Suc\ j) @ Oc\uparrow i, Oc\uparrow t))$
| $inv\_loop5\ n\ (l, r) = (inv\_loop5\_loop\ n\ (l, r) \vee inv\_loop5\_exit\ n\ (l, r))$
| $inv\_loop6\ n\ (l, r) = (inv\_loop6\_loop\ n\ (l, r) \vee inv\_loop6\_exit\ n\ (l, r))$

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 *inv_loop3_Bk_empty_via_2*[*elim*]: $[\![0 < x; inv\_loop2\ x\ (b, [\,])]\!] \Longrightarrow inv\_loop3\ x\ (Bk \# b, [\,])$
⟨*proof*⟩

lemma *inv_loop3_Bk_empty*[*elim*]: $[\![0 < x; inv\_loop3\ x\ (b, [])]\!] \Longrightarrow inv\_loop3\ x\ (Bk \# b, [])$
⟨*proof*⟩

lemma *inv_loop5_Oc_empty_via_4*[*elim*]: $[\![0 < x; inv\_loop4\ x\ (b, [])]\!] \Longrightarrow inv\_loop5\ x\ (b, [Oc])$
⟨*proof*⟩

lemma *inv_loop1_Bk*[*elim*]: $[\![0 < x; inv\_loop1\ x\ (b, Bk \# list)]\!] \Longrightarrow list = Oc \# Bk \uparrow x\ @\ Oc \uparrow x$
⟨*proof*⟩

lemma *inv_loop3_Bk_via_2*[*elim*]: $[\![0 < x; inv\_loop2\ x\ (b, Bk \# list)]\!] \Longrightarrow inv\_loop3\ x\ (Bk \# b, list)$
⟨*proof*⟩

lemma *inv_loop3_Bk_move*[*elim*]: $[\![0 < x; inv\_loop3\ x\ (b, Bk \# list)]\!] \Longrightarrow inv\_loop3\ x\ (Bk \# b, list)$
⟨*proof*⟩

lemma *inv_loop5_Oc_via_4_Bk*[*elim*]: $[\![0 < x; inv\_loop4\ x\ (b, Bk \# list)]\!] \Longrightarrow inv\_loop5\ x\ (b, Oc \# list)$
⟨*proof*⟩

lemma *inv_loop6_Bk_via_5*[*elim*]: $[\![0 < x; inv\_loop5\ x\ ([], Bk \# list)]\!] \Longrightarrow inv\_loop6\ x\ ([], Bk \# Bk \# list)$
⟨*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*]: $[\![0 < x; inv\_loop5\_exit\ x\ (b, Bk \# list); b \neq []; hd\ b = Bk]\!]$
$\Longrightarrow inv\_loop6\_loop\ x\ (tl\ b, Bk \# Bk \# list)$
⟨*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*]: $[\![x > 0; inv\_loop5\_exit\ x\ (b, Bk \# list); b \neq []; hd\ b = Oc]\!] \Longrightarrow$
$inv\_loop6\_exit\ x\ (tl\ b, Oc \# Bk \# list)$
⟨*proof*⟩

lemma *inv_loop6_Bk_tail_via_5*[*elim*]: $[\![0 < x; inv\_loop5\ x\ (b, Bk \# list); b \neq []]\!] \Longrightarrow inv\_loop6$

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*]: $[\![0 < x;\ inv\_loop6\ x\ ([], Oc \# list)]\!] \Longrightarrow inv\_loop1\ x\ ([], Bk \# Oc \# list)$
⟨*proof*⟩

lemma *inv_loop1_Oc_via_6*[*elim*]: $[\![0 < x;\ inv\_loop6\ x\ (b, Oc \# list);\ b \neq []]\!]$
$\Longrightarrow inv\_loop1\ x\ (tl\ b, hd\ b \# Oc \# list)$
⟨*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*]:
$[\![inv\_loop5\ x\ (b, [])]\!] \Longrightarrow RR\ inv\_loop6\ x\ (b, []) \Longrightarrow RR$
$[\![inv\_loop1\ x\ (b, Bk \# list)]\!] \Longrightarrow b = []$
⟨*proof*⟩

lemma *inv_loop6_Bk_Bk_drop*[*elim*]: $[\![inv\_loop6\ x\ ([], Bk \# list)]\!] \Longrightarrow inv\_loop6\ x\ ([], Bk \# Bk \# list)$
⟨*proof*⟩

lemma *inv_loop_step*:
$[\![inv\_loop\ x\ cf;\ x > 0]\!] \Longrightarrow inv\_loop\ x\ (step\ cf\ (tm\_copy\_loop\_orig, 0))$
⟨*proof*⟩

lemma *inv_loop_steps*:
$[\![inv\_loop\ x\ cf;\ x > 0]\!] \Longrightarrow inv\_loop\ x\ (steps\ cf\ (tm\_copy\_loop\_orig, 0)\ stp)$
⟨*proof*⟩

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

fun *loop_state* :: *config* ⇒ *nat*
where
$loop\_state\ (s, l, r) = (if\ s = 2 \wedge hd\ r = Oc\ then\ 0$
$else\ if\ s = 1\ then\ 1$
$else\ 10 - s)$

fun *loop_step* :: *config* ⇒ *nat*
where

$loop\_step\ (s, l, r) = (if\ s = 3\ then\ length\ r$
$\quad else\ if\ s = 4\ then\ length\ r$
$\quad else\ if\ s = 5\ then\ length\ l$
$\quad else\ if\ s = 6\ then\ length\ l$
$\quad else\ 0)$

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

lemma *inv_loop4_not_just_Oc*[*elim*]:
$[\![inv\_loop4\ x\ (l', []);$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ [Oc])) \neq$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'))]\!]$
$\Longrightarrow RR$
$[\![inv\_loop4\ x\ (l', Bk\ \#\ list);$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ Oc\ \#\ list)) \neq$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ Bk\ \#\ list))]\!]$
$\Longrightarrow RR$
⟨*proof*⟩

lemma *takeWhile_replicate_append*:
$P\ a \Longrightarrow takeWhile\ P\ (a\uparrow x\ @\ ys) = a\uparrow x\ @\ takeWhile\ P\ ys$
⟨*proof*⟩

lemma *takeWhile_replicate*:
$P\ a \Longrightarrow takeWhile\ P\ (a\uparrow x) = a\uparrow x$
⟨*proof*⟩

lemma *inv_loop5_Bk_E*[*elim*]:
$[\![inv\_loop5\ x\ (l', Bk\ \#\ list);\ l' \neq [];$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ (tl\ l')\ @\ hd\ l'\ \#\ Bk\ \#\ list)) \neq$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ Bk\ \#\ list))]\!]$
$\Longrightarrow RR$
⟨*proof*⟩

lemma *inv_loop1_hd_Oc*[*elim*]: $[\![inv\_loop1\ x\ (l', Oc\ \#\ list)]\!] \Longrightarrow hd\ list = Oc$
⟨*proof*⟩

lemma *inv_loop6_not_just_Bk*[*dest*!]:
$[\![length\ (takeWhile\ P\ (rev\ (tl\ l')\ @\ hd\ l'\ \#\ list)) \neq$
$length\ (takeWhile\ P\ (rev\ l'\ @\ list))]\!]$
$\Longrightarrow l' = []$

⟨*proof*⟩

lemma *inv_loop2_OcE*[*elim*]:
$[\![inv\_loop2\ x\ (l', Oc\ \#\ list);\ l' \neq []]\!] \Longrightarrow$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ Bk\ \#\ list)) <$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ Oc\ \#\ list))$
⟨*proof*⟩

lemma *loop_halts*:
assumes h: $n > 0\ inv\_loop\ n\ (1, l, r)$
shows $\exists\ stp.\ is\_final\ (steps0\ (1, l, r)\ tm\_copy\_loop\_orig\ stp)$
⟨*proof*⟩

lemma *loop_correct*:
assumes $0 < n$
shows $\{\!|inv\_loop1\ n|\!\}\ tm\_copy\_loop\_orig\ \{\!|inv\_loop0\ n|\!\}$
⟨*proof*⟩

definition
tm_copy_end_orig :: *instr list*
where
$tm\_copy\_end\_orig \stackrel{def}{=}$
$[\ (L, 0),(R, 2),\ (WO, 3),(L, 4),\ (R, 2),(R, 2),\ (L, 5),(WB, 4),\ (R, 0),(L, 5)\ ]$

definition
tm_copy_end_new :: *instr list*
where
$tm\_copy\_end\_new \stackrel{def}{=}$
$[\ (R, 0),(R, 2),\ (WO, 3),(L, 4),\ (R, 2),(R, 2),\ (L, 5),(WB, 4),\ (R, 0),(L, 5)\ ]$

fun
$inv\_end5\_loop :: nat \Rightarrow tape \Rightarrow bool$ **and**
$inv\_end5\_exit :: nat \Rightarrow tape \Rightarrow bool$
where
$inv\_end5\_loop\ x\ (l, r) =$
$(\exists\ i\ j.\ i + j = x \wedge x > 0 \wedge j > 0 \wedge l = Oc{\uparrow}i\ @\ [Bk] \wedge r = Oc{\uparrow}j\ @\ Bk\ \#\ Oc{\uparrow}x)$
$|\ inv\_end5\_exit\ x\ (l, r) = (x > 0 \wedge l = [] \wedge r = Bk\ \#\ Oc{\uparrow}x\ @\ Bk\ \#\ Oc{\uparrow}x)$

fun
$inv\_end0 :: nat \Rightarrow tape \Rightarrow bool$ **and**
$inv\_end1 :: nat \Rightarrow tape \Rightarrow bool$ **and**
$inv\_end2 :: nat \Rightarrow tape \Rightarrow bool$ **and**
$inv\_end3 :: nat \Rightarrow tape \Rightarrow bool$ **and**
$inv\_end4 :: nat \Rightarrow tape \Rightarrow bool$ **and**

$inv\_end5 :: nat \Rightarrow tape \Rightarrow bool$
where
$inv\_end0\ n\ (l, r) = (n > 0 \wedge (l, r) = ([Bk], Oc\uparrow n\ @\ Bk\ \#\ Oc\uparrow n))$
$|\ inv\_end1\ n\ (l, r) = (n > 0 \wedge (l, r) = ([Bk], Oc\ \#\ Bk\uparrow n\ @\ Oc\uparrow n))$
$|\ inv\_end2\ n\ (l, r) = (\exists\ i\ j.\ i + j = Suc\ n \wedge n > 0 \wedge l = Oc\uparrow i\ @\ [Bk] \wedge r = Bk\uparrow j\ @\ Oc\uparrow n)$
$|\ inv\_end3\ n\ (l, r) =$
$(\exists\ i\ j.\ n > 0 \wedge i + j = n \wedge l = Oc\uparrow i\ @\ [Bk] \wedge r = Oc\ \#\ Bk\uparrow j @\ Oc\uparrow n)$
$|\ inv\_end4\ n\ (l, r) = (\exists\ any.\ n > 0 \wedge l = Oc\uparrow n\ @\ [Bk] \wedge r = any \# Oc\uparrow n)$
$|\ inv\_end5\ n\ (l, r) = (inv\_end5\_loop\ n\ (l, r) \vee inv\_end5\_exit\ n\ (l, r))$

fun
$inv\_end :: nat \Rightarrow config \Rightarrow bool$
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 $inv\_end\_nonempty[simp]$:
$inv\_end1\ x\ (b, []) = False$
$inv\_end1\ x\ ([], list) = False$
$inv\_end2\ x\ (b, []) = False$
$inv\_end3\ x\ (b, []) = False$
$inv\_end4\ x\ (b, []) = False$
$inv\_end5\ x\ (b, []) = False$
$inv\_end5\ x\ ([], Oc\ \#\ list) = False$
$\langle proof \rangle$

lemma $inv\_end0\_Bk\_via\_1[elim]$: $[\![0 < x; inv\_end1\ x\ (b, Bk\ \#\ list); b \neq []]\!]$
$\Longrightarrow inv\_end0\ x\ (tl\ b, hd\ b\ \#\ Bk\ \#\ list)$
$\langle proof \rangle$

lemma $inv\_end3\_Oc\_via\_2[elim]$: $[\![0 < x; inv\_end2\ x\ (b, Bk\ \#\ list)]\!]$
$\Longrightarrow inv\_end3\ x\ (b, Oc\ \#\ list)$
$\langle proof \rangle$

lemma $inv\_end2\_Bk\_via\_3[elim]$: $[\![0 < x; inv\_end3\ x\ (b, Bk\ \#\ list)]\!] \Longrightarrow inv\_end2\ x\ (Bk\ \#\ b,$
$list)$
$\langle proof \rangle$

lemma $inv\_end5\_Bk\_via\_4[elim]$: $[\![0 < x; inv\_end4\ x\ ([], Bk\ \#\ list)]\!] \Longrightarrow$
$inv\_end5\ x\ ([], Bk\ \#\ Bk\ \#\ list)$

$\langle proof \rangle$

lemma *inv_end5_Bk_tail_via_4*[*elim*]: $[\![0 < x;\ inv\_end4\ x\ (b,\ Bk\ \#\ list);\ b \neq []]\!] \Longrightarrow$
$inv\_end5\ x\ (tl\ b,\ hd\ b\ \#\ Bk\ \#\ list)$
$\langle proof \rangle$

lemma *inv_end0_Bk_via_5*[*elim*]: $[\![0 < x;\ inv\_end5\ x\ (b,\ Bk\ \#\ list)]\!] \Longrightarrow inv\_end0\ x\ (Bk\ \#\ b,\ list)$
$\langle proof \rangle$

lemma *inv_end2_Oc_via_1*[*elim*]: $[\![0 < x;\ inv\_end1\ x\ (b,\ Oc\ \#\ list)]\!] \Longrightarrow inv\_end2\ x\ (Oc\ \#\ b,\ list)$
$\langle proof \rangle$

lemma *inv_end4_Bk_Oc_via_2*[*elim*]: $[\![0 < x;\ inv\_end2\ x\ ([],\ Oc\ \#\ list)]\!] \Longrightarrow$
$inv\_end4\ x\ ([],\ Bk\ \#\ Oc\ \#\ list)$
$\langle proof \rangle$

lemma *inv_end4_Oc_via_2*[*elim*]: $[\![0 < x;\ inv\_end2\ x\ (b,\ Oc\ \#\ list);\ b \neq []]\!] \Longrightarrow$
$inv\_end4\ x\ (tl\ b,\ hd\ b\ \#\ Oc\ \#\ list)$
$\langle proof \rangle$

lemma *inv_end2_Oc_via_3*[*elim*]: $[\![0 < x;\ inv\_end3\ x\ (b,\ Oc\ \#\ list)]\!] \Longrightarrow inv\_end2\ x\ (Oc\ \#\ b,\ list)$
$\langle proof \rangle$

lemma *inv_end4_Bk_via_Oc*[*elim*]: $[\![0 < x;\ inv\_end4\ x\ (b,\ Oc\ \#\ list)]\!] \Longrightarrow inv\_end4\ x\ (b,\ Bk\ \#\ list)$
$\langle proof \rangle$

lemma *inv_end5_Bk_drop_Oc*[*elim*]: $[\![0 < x;\ inv\_end5\ x\ ([],\ Oc\ \#\ list)]\!] \Longrightarrow inv\_end5\ x\ ([],\ Bk\ \#\ Oc\ \#\ list)$
$\langle proof \rangle$

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

lemma *inv_end5_exit_no_Oc*[*simp*]: $inv\_end5\_exit\ x\ (b,\ Oc\ \#\ list) = False$
$\langle proof \rangle$

lemma *inv_end5_loop_no_Bk_Oc*[*simp*]: $inv\_end5\_loop\ x\ (tl\ b,\ Bk\ \#\ Oc\ \#\ list) = False$
$\langle proof \rangle$

lemma *inv_end5_exit_Bk_Oc_via_loop*[*elim*]:
$[\![0 < x;\ inv\_end5\_loop\ x\ (b,\ Oc\ \#\ list);\ b \neq [];\ hd\ b = Bk]\!] \Longrightarrow$
$inv\_end5\_exit\ x\ (tl\ b,\ Bk\ \#\ Oc\ \#\ list)$
$\langle proof \rangle$

lemma *inv_end5_loop_Oc_Oc_drop*[*elim*]:
$[\![0 < x;\ inv\_end5\_loop\ x\ (b,\ Oc\ \#\ list);\ b \neq [];\ hd\ b = Oc]\!] \Longrightarrow$

inv_end5_loop x (*tl b*, *Oc* # *Oc* # *list*)
⟨*proof*⟩

lemma *inv_end5_Oc_tail*[*elim*]: $[\![0 < x;\ inv\_end5\ x\ (b, Oc\ \#\ list);\ b \neq []]\!] \Longrightarrow$
inv_end5 x (*tl b*, *hd b* # *Oc* # *list*)
⟨*proof*⟩

lemma *inv_end_step*:
$[\![x > 0;\ inv\_end\ x\ cf]\!] \Longrightarrow inv\_end\ x\ (step\ cf\ (tm\_copy\_end\_new, 0))$
⟨*proof*⟩

lemma *inv_end_steps*:
$[\![x > 0;\ inv\_end\ x\ cf]\!] \Longrightarrow inv\_end\ x\ (steps\ cf\ (tm\_copy\_end\_new, 0)\ stp)$
⟨*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 *end_halt*:
$[\![x > 0;\ inv\_end\ x\ (Suc\ 0, l, r)]\!] \Longrightarrow$
$\exists\ stp.\ is\_final\ (steps\ (Suc\ 0, l, r)\ (tm\_copy\_end\_new, 0)\ stp)$
⟨*proof*⟩

lemma *end_correct*:
$n > 0 \Longrightarrow$ ⦃*inv_end1 n*⦄ *tm_copy_end_new* ⦃*inv_end0 n*⦄
⟨*proof*⟩

definition
tm_weak_copy :: *instr list*
where
tm_weak_copy $\stackrel{def}{=}$ (*tm_copy_begin_orig* |+| *tm_copy_loop_orig*) |+| *tm_copy_end_new*

lemma [*intro*]:
composable_tm (*tm_copy_begin_orig*, *0*)
composable_tm (*tm_copy_loop_orig*, *0*)
composable_tm (*tm_copy_end_new*, *0*)
⟨*proof*⟩

lemma *composable_tm0_tm_weak_copy*[*intro*, *simp*]: *composable_tm0 tm_weak_copy*
⟨*proof*⟩

lemma *tm_weak_copy_correct_pre*:
assumes $0 < x$
shows ⦃*inv_begin1 x*⦄ *tm_weak_copy* ⦃*inv_end0 x*⦄
⟨*proof*⟩

lemma *tm_weak_copy_correct*:
shows ⦃λ*tap*. *tap* = ([]::*cell list*, *Oc* $\uparrow$ (*Suc n*))⦄ *tm_weak_copy* ⦃λ*tap*. *tap*= ([*Bk*], <(*n*, *n*::*nat*)>)⦄
⟨*proof*⟩

lemma *tm_weak_copy_correct5*: ⦃λ*tap*. *tap* = ([], <[*n*::*nat*]>)⦄ *tm_weak_copy* ⦃λ*tap*. $\exists$ *k l*. *tap* = (*Bk* $\uparrow$ *k*, <[*n*, *n*]> @ *Bk* $\uparrow$ *l*) ⦄
⟨*proof*⟩

lemma *tm_weak_copy_correct6*:
⦃λ*tap*. $\exists$ *z4*. *tap* = (*Bk* $\uparrow$ *z4*, <[*n*::*nat*]> @[*Bk*])⦄ *tm_weak_copy* ⦃λ*tap*. $\exists$ *k l*. *tap* = (*Bk* $\uparrow$ *k*, <[*n*::*nat*, *n*]> @ *Bk* $\uparrow$ *l*) ⦄
⟨*proof*⟩

definition
strong_copy_post :: *instr list*
where
strong_copy_post $\stackrel{def}{=}$ [
(*WB*,*5*),(*R*,*2*), (*R*,*3*),(*R*,*2*), (*WO*,*3*),(*L*,*4*), (*L*,*4*),(*L*,*5*), (*R*,*11*),(*R*,*6*),
(*R*,*7*),(*WB*,*6*), (*R*,*7*),(*R*,*8*), (*WO*,*9*),(*R*,*8*), (*L*,*10*),(*L*,*9*), (*L*,*10*),(*L*,*5*),
(*R*,*0*),(*R*,*12*), (*WO*,*13*),(*L*,*14*), (*R*,*12*),(*R*,*12*), (*L*,*15*),(*WB*,*14*), (*R*,*0*),(*L*,*15*)
]

value *steps0* (*1*, [*Bk*,*Bk*], [*Bk*]) *strong_copy_post 3* = (*0*::*nat*, [*Bk*, *Bk*, *Bk*, *Bk*], [])

lemma *steps0* (*1*, [*Bk*,*Bk*], [*Bk*]) *strong_copy_post 3* = (*0*::*nat*, [*Bk*, *Bk*, *Bk*, *Bk*], [])
⟨*proof*⟩

lemma *tm_weak_copy_eq_strong_copy_post*: *tm_weak_copy* = *strong_copy_post*
⟨*proof*⟩

lemma *tm_weak_copy_correct11*:
⦃λ*tap*. *tap* = ([*Bk*,*Bk*], [*Bk*]) ⦄ *tm_weak_copy* ⦃λ*tap*. *tap* = ([*Bk*,*Bk*,*Bk*,*Bk*],[]) ⦄
⟨*proof*⟩

lemma *tm_weak_copy_correct12*:
⦃λ*tap*. *tap* = ([*Bk*,*Bk*], [*Bk*]) ⦄ *tm_weak_copy* ⦃λ*tap*. ∃ *k l*. *tap* = (*Bk* ↑ *k*, *Bk* ↑ *l*) ⦄
⟨*proof*⟩

lemma *tm_weak_copy_correct13*:
⦃λ*tap*. *tap* = ([], [*Bk*,*Bk*]@*r*) ⦄ *tm_weak_copy* ⦃λ*tap*. *tap* = ([*Bk*,*Bk*], *r*) ⦄
⟨*proof*⟩

lemma *tm_weak_copy_correct11′*:
⦃λ*tap*. *tap* = ([*Bk*,*Bk*], [*Bk*]) ⦄ *tm_weak_copy* ⦃λ*tap*. *tap* = ([*Bk*,*Bk*,*Bk*,*Bk*],[]) ⦄
⟨*proof*⟩

lemma *tm_weak_copy_correct13'*:
$\{\!|\lambda tap.\ tap = ([], [Bk,Bk]@r)\ |\!\}$ *tm_weak_copy* $\{\!|\lambda tap.\ tap = ([Bk,Bk], r)\ |\!\}$
⟨*proof*⟩

end

1.10.9.2 A Turing machine that duplicates its input iff the input is a single numeral

theory *StrongCopyTM*
imports
WeakCopyTM
begin

If we run *tm_strong_copy* on a single numeral, it behaves like the original weak version *tm_weak_copy*. However, if we run the strong machine on an empty list, the result is an empty list. If we run the machine on a list with more than two numerals, this strong variant will just return the first numeral of the list (a singleton list).

Thus, the result will be a list of two numerals only if we run it on a singleton list.

This is exactly the property, we need for the reduction of problem *K1* to problem *H1*.

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

lemma *tm_skip_first_arg_correct_Nil*:
$\{\!|\lambda tap.\ tap = ([], [])|\!\}$ *tm_skip_first_arg* $\{\!|\lambda tap.\ tap = ([], [Bk])\ |\!\}$
⟨*proof*⟩

corollary *tm_skip_first_arg_correct_Nil'*:
length nl = *0*
$\Longrightarrow$ $\{\!|\lambda tap.\ tap = ([],$ <*nl*::*nat list*> $)|\!\}$ *tm_skip_first_arg* $\{\!|\lambda tap.\ tap = ([], [Bk])\ |\!\}$
⟨*proof*⟩

fun
inv_tm_skip_first_arg_len_eq_1_s0 :: *nat* ⇒ *tape* ⇒ *bool* **and**

```
  inv_tm_skip_first_arg_len_eq_1_s1 :: nat ⇒ tape ⇒ bool and
  inv_tm_skip_first_arg_len_eq_1_s2 :: nat ⇒ tape ⇒ bool and
  inv_tm_skip_first_arg_len_eq_1_s3 :: nat ⇒ tape ⇒ bool and
  inv_tm_skip_first_arg_len_eq_1_s4 :: nat ⇒ tape ⇒ bool and
  inv_tm_skip_first_arg_len_eq_1_s5 :: nat ⇒ tape ⇒ bool
  where
   inv_tm_skip_first_arg_len_eq_1_s0 n (l, r) = (
    l = [Bk] ∧ r = Oc ↑ (Suc n) @ [Bk])
  | inv_tm_skip_first_arg_len_eq_1_s1 n (l, r) = (
    l = [] ∧ r = Oc ↑ Suc n)
  | inv_tm_skip_first_arg_len_eq_1_s2 n (l, r) =
    (∃ n1 n2. l = Oc ↑ (Suc n1) ∧ r = Oc ↑ n2 ∧ Suc n1 + n2 = Suc n)
  | inv_tm_skip_first_arg_len_eq_1_s3 n (l, r) = (
    l = Bk # Oc ↑ (Suc n) ∧ r = [])
  | inv_tm_skip_first_arg_len_eq_1_s4 n (l, r) = (
    l = Oc ↑ (Suc n) ∧ r = [Bk])
  | inv_tm_skip_first_arg_len_eq_1_s5 n (l, r) =
     (∃ n1 n2. (l = Oc ↑ Suc n1 ∧ r = Oc ↑ Suc n2 @ [Bk] ∧ Suc n1 + Suc n2 = Suc n) ∨
          (l = [] ∧ r = Oc ↑ Suc n2 @ [Bk] ∧ Suc n2 = Suc n) ∨
          (l = [] ∧ r = Bk # Oc ↑ Suc n2 @ [Bk] ∧ Suc n2 = Suc n) )


fun inv_tm_skip_first_arg_len_eq_1 :: nat ⇒ config ⇒ bool
  where
   inv_tm_skip_first_arg_len_eq_1 n (s, tap) =
     (if s = 0 then inv_tm_skip_first_arg_len_eq_1_s0 n tap else
      if s = 1 then inv_tm_skip_first_arg_len_eq_1_s1 n tap else
      if s = 2 then inv_tm_skip_first_arg_len_eq_1_s2 n tap else
      if s = 3 then inv_tm_skip_first_arg_len_eq_1_s3 n tap else
      if s = 4 then inv_tm_skip_first_arg_len_eq_1_s4 n tap else
      if s = 5 then inv_tm_skip_first_arg_len_eq_1_s5 n tap
      else False)

lemma tm_skip_first_arg_len_eq_1_cases:
  fixes s::nat
  assumes inv_tm_skip_first_arg_len_eq_1 n (s,l,r)
   and s=0 ⟹ P
   and s=1 ⟹ P
   and s=2 ⟹ P
   and s=3 ⟹ P
   and s=4 ⟹ P
   and s=5 ⟹ P
  shows P
⟨proof⟩

lemma inv_tm_skip_first_arg_len_eq_1_step:
  assumes inv_tm_skip_first_arg_len_eq_1 n cf
  shows inv_tm_skip_first_arg_len_eq_1 n (step0 cf tm_skip_first_arg)
⟨proof⟩
```

lemma *inv_tm_skip_first_arg_len_eq_1_steps*:
 assumes *inv_tm_skip_first_arg_len_eq_1 n cf*
 shows *inv_tm_skip_first_arg_len_eq_1 n* (*steps0 cf tm_skip_first_arg stp*)
⟨*proof*⟩

lemma *tm_skip_first_arg_len_eq_1_partial_correctness*:
 assumes ∃ *stp*. *is_final* (*steps0* (*1*, [], <[*n*::*nat*]>) *tm_skip_first_arg stp*)
 shows ⦃ λ*tap*. *tap* = ([], <[*n*::*nat*]>) ⦄
  tm_skip_first_arg
  ⦃ λ*tap*. *tap* = ([*Bk*], <[*n*::*nat*]> @[*Bk*]) ⦄
⟨*proof*⟩

definition *measure_tm_skip_first_arg_len_eq_1* :: (*config* × *config*) *set*
 where
  measure_tm_skip_first_arg_len_eq_1 = *measures* [
   λ(*s*, *l*, *r*). (*if s* = *0 then 0 else 5* − *s*),
   λ(*s*, *l*, *r*). (*if s* = *2 then length r else 0*),
   λ(*s*, *l*, *r*). (*if s* = *5 then length l* + (*if hd r* = *Oc then 2 else 1*) *else 0*)
   ]

lemma *wf_measure_tm_skip_first_arg_len_eq_1*: *wf measure_tm_skip_first_arg_len_eq_1*
 ⟨*proof*⟩

lemma *measure_tm_skip_first_arg_len_eq_1_induct* [*case_names Step*]:
 ⟦⋀*n*. ¬ *P* (*f n*) ⟹ (*f* (*Suc n*), (*f n*)) ∈ *measure_tm_skip_first_arg_len_eq_1*⟧ ⟹ ∃ *n*. *P* (*f n*)
 ⟨*proof*⟩

lemma *tm_skip_first_arg_len_eq_1_halts*:
 ∃ *stp*. *is_final* (*steps0* (*1*, [], <[*n*::*nat*]>) *tm_skip_first_arg stp*)
⟨*proof*⟩

lemma *tm_skip_first_arg_len_eq_1_total_correctness*:
 ⦃ λ*tap*. *tap* = ([], <[*n*::*nat*]>)⦄
  tm_skip_first_arg
 ⦃ λ*tap*. *tap* = ([*Bk*], <[*n*::*nat*]> @[*Bk*])⦄
⟨*proof*⟩

lemma *tm_skip_first_arg_len_eq_1_total_correctness'*:
 length nl = *1*
  ⟹ ⦃λ*tap*. *tap* = ([], <*nl*::*nat list*>)⦄ *tm_skip_first_arg* ⦃ λ*tap*. *tap* = ([*Bk*], <[*hd nl*]> @[*Bk*])⦄
⟨*proof*⟩

```
fun
  inv_tm_skip_first_arg_len_gt_1_s0 :: nat ⇒ nat list ⇒ tape ⇒ bool and
  inv_tm_skip_first_arg_len_gt_1_s1 :: nat ⇒ nat list⇒ tape ⇒ bool and
  inv_tm_skip_first_arg_len_gt_1_s2 :: nat ⇒ nat list⇒ tape ⇒ bool and
  inv_tm_skip_first_arg_len_gt_1_s3 :: nat ⇒ nat list⇒ tape ⇒ bool
  where
   inv_tm_skip_first_arg_len_gt_1_s1 n ns (l, r) = (
     l = [] ∧ r = Oc ↑ Suc n @ [Bk] @ (<ns::nat list>) )
  | inv_tm_skip_first_arg_len_gt_1_s2 n ns (l, r) =
     (∃ n1 n2. l = Oc ↑ (Suc n1) ∧ r = Oc ↑ n2 @ [Bk] @ (<ns::nat list>) ∧
            Suc n1 + n2 = Suc n)
  | inv_tm_skip_first_arg_len_gt_1_s3 n ns (l, r) = (
     l = Bk # Oc ↑ (Suc n) ∧ r = (<ns::nat list>)
     )
  |  inv_tm_skip_first_arg_len_gt_1_s0 n ns (l, r) = (
     l = Bk# Oc ↑ (Suc n) ∧ r = (<ns::nat list>)
     )

fun inv_tm_skip_first_arg_len_gt_1 :: nat ⇒ nat list ⇒ config ⇒ bool
  where
    inv_tm_skip_first_arg_len_gt_1 n ns (s, tap) =
       (if s = 0 then inv_tm_skip_first_arg_len_gt_1_s0 n ns tap else
        if s = 1 then inv_tm_skip_first_arg_len_gt_1_s1 n ns tap else
        if s = 2 then inv_tm_skip_first_arg_len_gt_1_s2 n ns tap else
        if s = 3 then inv_tm_skip_first_arg_len_gt_1_s3 n ns tap
        else False)

lemma tm_skip_first_arg_len_gt_1_cases:
  fixes s::nat
  assumes inv_tm_skip_first_arg_len_gt_1 n ns (s,l,r)
    and s=0 ⟹ P
    and s=1 ⟹ P
    and s=2 ⟹ P
    and s=3 ⟹ P
    and s=4 ⟹ P
    and s=5 ⟹ P
  shows P
⟨proof⟩

lemma inv_tm_skip_first_arg_len_gt_1_step:
  assumes length ns > 0
    and inv_tm_skip_first_arg_len_gt_1 n ns cf
  shows inv_tm_skip_first_arg_len_gt_1 n ns (step0 cf tm_skip_first_arg)
⟨proof⟩
```

lemma *inv_tm_skip_first_arg_len_gt_1_steps*:
assumes *length ns* > *0*
and *inv_tm_skip_first_arg_len_gt_1 n ns cf*
shows *inv_tm_skip_first_arg_len_gt_1 n ns* (*steps0 cf tm_skip_first_arg stp*)
⟨*proof*⟩

lemma *tm_skip_first_arg_len_gt_1_partial_correctness*:
assumes ∃ *stp*. *is_final* (*steps0* (*1*, [], *Oc* ↑ *Suc n* @ [*Bk*] @ (<*ns*::*nat list*>)) *tm_skip_first_arg stp*)
and *0* < *length ns*
shows ⦃λ*tap*. *tap* = ([], *Oc* ↑ *Suc n* @ [*Bk*] @ (<*ns*::*nat list*>)) ⦄
tm_skip_first_arg
⦃ λ*tap*. *tap* = (*Bk*# *Oc* ↑ *Suc n*, (<*ns*::*nat list*>)) ⦄
⟨*proof*⟩

definition *measure_tm_skip_first_arg_len_gt_1* :: (*config* × *config*) *set*
where
measure_tm_skip_first_arg_len_gt_1 = *measures* [
λ(*s*, *l*, *r*). (*if s* = *0 then 0 else 4* − *s*),
λ(*s*, *l*, *r*). (*if s* = *2 then length r else 0*)
]

lemma *wf_measure_tm_skip_first_arg_len_gt_1*: *wf measure_tm_skip_first_arg_len_gt_1*
⟨*proof*⟩

lemma *measure_tm_skip_first_arg_len_gt_1_induct* [*case_names Step*]:
⟦⋀*n*. ¬ *P* (*f n*) ⟹ (*f* (*Suc n*), (*f n*)) ∈ *measure_tm_skip_first_arg_len_gt_1*⟧ ⟹ ∃ *n*. *P* (*f n*)
⟨*proof*⟩

lemma *tm_skip_first_arg_len_gt_1_halts*:
0 < *length ns* ⟹ ∃ *stp*. *is_final* (*steps0* (*1*, [], *Oc* ↑ *Suc n* @ [*Bk*] @ <*ns*::*nat list*>) *tm_skip_first_arg stp*)
⟨*proof*⟩

lemma *tm_skip_first_arg_len_gt_1_total_correctness_pre*:
assumes *0* < *length ns*
shows ⦃λ*tap*. *tap* = ([], *Oc* ↑ *Suc n* @ [*Bk*] @ (<*ns*::*nat list*>)) ⦄
tm_skip_first_arg
⦃ λ*tap*. *tap* = (*Bk*# *Oc* ↑ *Suc n*, (<*ns*::*nat list*>)) ⦄
⟨*proof*⟩

lemma *tm_skip_first_arg_len_gt_1_total_correctness*:
assumes *1* < *length* (*nl*::*nat list*)
shows ⦃λ*tap*. *tap* = ([], <*nl*::*nat list*>)⦄ *tm_skip_first_arg* ⦃ λ*tap*. *tap* = (*Bk*# <*rev* [*hd nl*]>, <*tl nl*>) ⦄

$\langle proof \rangle$

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

$(R, 7)$,$(WB,6)$,
$(R, 9)$,$(WB,8)$,

$(R, 7)$,$(R, 7)$,
$(L,10)$,$(WB,8)$,

$(L,10)$,$(L,11)$,

$(L,12)$,$(L,11)$,
$(WB,0)$,$(L,11)$
]

fun
inv_tm_erase_right_then_dblBk_left_dnp_s0 :: $(cell\ list) \Rightarrow tape \Rightarrow bool$ **and**
inv_tm_erase_right_then_dblBk_left_dnp_s1 :: $(cell\ list) \Rightarrow tape \Rightarrow bool$ **and**
inv_tm_erase_right_then_dblBk_left_dnp_s2 :: $(cell\ list) \Rightarrow tape \Rightarrow bool$ **and**
inv_tm_erase_right_then_dblBk_left_dnp_s3 :: $(cell\ list) \Rightarrow tape \Rightarrow bool$ **and**
inv_tm_erase_right_then_dblBk_left_dnp_s4 :: $(cell\ list) \Rightarrow tape \Rightarrow bool$

```
 where
   inv_tm_erase_right_then_dblBk_left_dnp_s0 CR (l, r) = (l = [Bk, Bk] ∧ CR = r)
 | inv_tm_erase_right_then_dblBk_left_dnp_s1 CR (l, r) = (l = []      ∧ CR = r)
 | inv_tm_erase_right_then_dblBk_left_dnp_s2 CR (l, r) = (l = []      ∧ r = Bk#CR)
 | inv_tm_erase_right_then_dblBk_left_dnp_s3 CR (l, r) = (l = []      ∧ r = Bk#Bk#CR)
 | inv_tm_erase_right_then_dblBk_left_dnp_s4 CR (l, r) = (l = [Bk]    ∧ r = Bk#CR)


fun inv_tm_erase_right_then_dblBk_left_dnp :: (cell list) ⇒ config ⇒ bool
 where
   inv_tm_erase_right_then_dblBk_left_dnp CR (s, tap) =
     (if s = 0 then inv_tm_erase_right_then_dblBk_left_dnp_s0 CR tap else
      if s = 1 then inv_tm_erase_right_then_dblBk_left_dnp_s1 CR tap else
      if s = 2 then inv_tm_erase_right_then_dblBk_left_dnp_s2 CR tap else
      if s = 3 then inv_tm_erase_right_then_dblBk_left_dnp_s3 CR tap else
      if s = 4 then inv_tm_erase_right_then_dblBk_left_dnp_s4 CR tap
      else False)

lemma tm_erase_right_then_dblBk_left_dnp_cases:
 fixes s::nat
 assumes inv_tm_erase_right_then_dblBk_left_dnp CR (s,l,r)
   and s=0 ⟹ P
   and s=1 ⟹ P
   and s=2 ⟹ P
   and s=3 ⟹ P
   and s=4 ⟹ P
 shows P
⟨proof⟩

lemma inv_tm_erase_right_then_dblBk_left_dnp_step:
 assumes inv_tm_erase_right_then_dblBk_left_dnp CR cf
 shows inv_tm_erase_right_then_dblBk_left_dnp CR (step0 cf tm_erase_right_then_dblBk_left)
⟨proof⟩

lemma inv_tm_erase_right_then_dblBk_left_dnp_steps:
 assumes inv_tm_erase_right_then_dblBk_left_dnp CR cf
 shows inv_tm_erase_right_then_dblBk_left_dnp CR (steps0 cf tm_erase_right_then_dblBk_left
stp)
⟨proof⟩


lemma tm_erase_right_then_dblBk_left_dnp_partial_correctness:
 assumes ∃ stp. is_final (steps0 (1, [], r) tm_erase_right_then_dblBk_left stp)
 shows ⦃ λtap. tap = ([], r ) ⦄
         tm_erase_right_then_dblBk_left
       ⦃ λtap. tap = ([Bk,Bk], r ) ⦄
⟨proof⟩
```

definition *measure_tm_erase_right_then_dblBk_left_dnp* :: $(config \times config)$ *set*
where
measure_tm_erase_right_then_dblBk_left_dnp = *measures* [
$\lambda(s, l, r)$. (*if* $s = 0$ *then* 0 *else* $5 - s$)
]

lemma *wf_measure_tm_erase_right_then_dblBk_left_dnp*: *wf measure_tm_erase_right_then_dblBk_left_dnp*
⟨*proof*⟩

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

lemma *tm_erase_right_then_dblBk_left_dnp_halts*:
∃ *stp*. *is_final* (*steps0* (*1*, [], *r*) *tm_erase_right_then_dblBk_left stp*)
⟨*proof*⟩

lemma *tm_erase_right_then_dblBk_left_dnp_total_correctness*:
⦃ λ*tap*. *tap* = ([], *r*) ⦄
tm_erase_right_then_dblBk_left
⦃ λ*tap*. *tap* = ([*Bk*,*Bk*], *r*) ⦄
⟨*proof*⟩

fun *inv_tm_erase_right_then_dblBk_left_erp_s1* :: (*cell list*) ⇒ (*cell list*) ⇒ *tape* ⇒ *bool*
where
inv_tm_erase_right_then_dblBk_left_erp_s1 *CL CR* (*l*, *r*) =
(*l* = [*Bk*,*Oc*] @ *CL* ∧ *r* = *CR*)
fun *inv_tm_erase_right_then_dblBk_left_erp_s2* :: (*cell list*) ⇒ (*cell list*) ⇒ *tape* ⇒ *bool*
where
inv_tm_erase_right_then_dblBk_left_erp_s2 *CL CR* (*l*, *r*) =
(*l* = [*Oc*] @ *CL* ∧ *r* = *Bk*#*CR*)
fun *inv_tm_erase_right_then_dblBk_left_erp_s3* :: (*cell list*) ⇒ (*cell list*) ⇒ *tape* ⇒ *bool*
where
inv_tm_erase_right_then_dblBk_left_erp_s3 *CL CR* (*l*, *r*) =
(*l* = *CL* ∧ *r* = *Oc*#*Bk*#*CR*)

fun *inv_tm_erase_right_then_dblBk_left_erp_s5* :: (*cell list*) ⇒ (*cell list*) ⇒ *tape* ⇒ *bool*
where

$inv\_tm\_erase\_right\_then\_dblBk\_left\_erp\_s5\ \ CL\ CR\ (l, r) =$
$(l = [Oc] \quad @\ CL \wedge r = Bk\#CR)$

fun $inv\_tm\_erase\_right\_then\_dblBk\_left\_erp\_s6\ :: (cell\ list) \Rightarrow (cell\ list) \Rightarrow tape \Rightarrow bool$
where
$inv\_tm\_erase\_right\_then\_dblBk\_left\_erp\_s6\ \ CL\ CR\ (l, r) =$
$(l = [Bk,Oc]\ @\ CL \wedge (\ (CR = [] \wedge r = CR) \vee (CR \neq [] \wedge (r = CR \vee r = Bk\ \#\ tl\ CR))\ )\ )$

fun $inv\_tm\_erase\_right\_then\_dblBk\_left\_erp\_s7\ :: (cell\ list) \Rightarrow (cell\ list) \Rightarrow tape \Rightarrow bool$
where
$inv\_tm\_erase\_right\_then\_dblBk\_left\_erp\_s7\ \ CL\ CR\ (l, r) =$
$((\exists\, lex.\ l = Bk \uparrow Suc\ lex\ @\ [Bk,Oc]\ @\ CL) \wedge (\exists\, rs.\ CR = rs\ @\ r)\ )$

fun $inv\_tm\_erase\_right\_then\_dblBk\_left\_erp\_s8\ :: (cell\ list) \Rightarrow (cell\ list) \Rightarrow tape \Rightarrow bool$
where
$inv\_tm\_erase\_right\_then\_dblBk\_left\_erp\_s8\ \ CL\ CR\ (l, r) =$
$((\exists\, lex.\ l = Bk \uparrow Suc\ lex\ @\ [Bk,Oc]\ @\ CL) \wedge$
$(\exists\, rs1\ rs2.\ CR = rs1\ @\ [Oc]\ @\ rs2 \wedge r = Bk\#rs2)\ )$

fun $inv\_tm\_erase\_right\_then\_dblBk\_left\_erp\_s9\ :: (cell\ list) \Rightarrow (cell\ list) \Rightarrow tape \Rightarrow bool$
where
$inv\_tm\_erase\_right\_then\_dblBk\_left\_erp\_s9\ \ CL\ CR\ (l, r) =$
$((\exists\, lex.\ l = Bk \uparrow Suc\ lex\ @\ [Bk,Oc]\ @\ CL) \wedge (\exists\, rs.\ CR = rs\ @\ [Bk]\ @\ r \vee CR = rs \wedge r =$
$[])\ )$

fun $inv\_tm\_erase\_right\_then\_dblBk\_left\_erp\_s10 :: (cell\ list) \Rightarrow (cell\ list) \Rightarrow tape \Rightarrow bool$
where
$inv\_tm\_erase\_right\_then\_dblBk\_left\_erp\_s10\ \ CL\ CR\ (l, r) =$
$($
$(\exists\, lex\ rex.\ l = Bk \uparrow lex\ @\ [Bk,Oc]\ @\ CL \wedge r = Bk \uparrow Suc\ rex) \vee$
$(\exists\, rex.\ l = [Oc]\ @\ CL \wedge r = Bk \uparrow Suc\ rex) \vee$
$(\exists\, rex.\ l = CL \wedge r = Oc\ \#\ Bk \uparrow Suc\ rex)$
$)$

fun $inv\_tm\_erase\_right\_then\_dblBk\_left\_erp\_s11 :: (cell\ list) \Rightarrow (cell\ list) \Rightarrow tape \Rightarrow bool$
where
$inv\_tm\_erase\_right\_then\_dblBk\_left\_erp\_s11\ \ CL\ CR\ (l, r) =$
$($
$(\exists\, rex. \quad l = [] \quad \wedge r = Bk\#\ rev\ CL \quad @\ Oc\ \#\ Bk \uparrow Suc\ rex \wedge (CL = [] \vee last\ CL =$
$Oc)) \quad \vee$

$(\exists\, rex. \quad l = [] \quad \wedge r = rev\ CL \quad @\ Oc\ \#\ Bk \uparrow Suc\ rex \wedge CL \neq [] \wedge last\ CL = Bk$
$) \vee$
$(\exists\, rex. \quad l = [] \quad \wedge r = rev\ CL \quad @\ Oc\ \#\ Bk \uparrow Suc\ rex \wedge CL \neq [] \wedge last\ CL = Oc$
$) \vee$

$(\exists\, rex. \quad l = [Bk] \quad \wedge r = rev\ [Oc] \quad @\ Oc\ \#\ Bk \uparrow Suc\ rex \wedge CL = [Oc, Bk]) \vee$

$(\exists\, rex\ ls1\ ls2.\ l = Bk\#Oc\#ls2\ \wedge r = rev\ ls1 \quad @\ Oc\ \#\ Bk \uparrow Suc\ rex \wedge CL = ls1\ @\ Bk\#Oc\#ls2 \wedge ls1 = [Oc]\ ) \vee$

$(\exists\, rex\ ls1\ ls2.\ l = \quad Oc\#ls2\ \wedge r = rev\ ls1 \quad @\ Oc\ \#\ Bk \uparrow Suc\ rex \wedge CL = ls1\ @\ Oc\#ls2 \wedge ls1 = [Bk]\ ) \vee$

$(\exists\, rex\ ls1\ ls2.\ l = \quad Oc\#ls2\ \wedge r = rev\ ls1 \quad @\ Oc\ \#\ Bk \uparrow Suc\ rex \wedge CL = ls1\ @\ Oc\#ls2 \wedge ls1 = [Oc]\ ) \vee$

$(\exists\, rex\ ls1\ ls2.\ l = \quad ls2\ \wedge r = rev\ ls1 \quad @\ Oc\ \#\ Bk \uparrow Suc\ rex \wedge CL = ls1\ @\ ls2 \quad \wedge tl\ ls1 \neq [])$

)

fun *inv_tm_erase_right_then_dblBk_left_erp_s12* :: (*cell list*) ⇒ (*cell list*) ⇒ *tape* ⇒ *bool*
where
inv_tm_erase_right_then_dblBk_left_erp_s12 CL CR (*l*, *r*) =
(

$(\exists\, rex\ ls1\ ls2.\ l = ls2 \wedge r = rev\ ls1\ @\ Oc\ \#\ Bk \uparrow Suc\ rex \wedge CL = ls1\ @\ ls2\ \wedge tl\ ls1 \neq [] \wedge last\ ls1 = Oc) \vee$

$(\exists\, rex.\ l = []\ \wedge r = Bk\#rev\ CL\ @\ Oc\ \#\ Bk \uparrow Suc\ rex \wedge CL \neq [] \wedge last\ CL = Bk) \vee$

$(\exists\, rex.\ l = []\ \wedge r = Bk\#Bk\#rev\ CL\ @\ Oc\ \#\ Bk \uparrow Suc\ rex \wedge (CL = [] \vee last\ CL = Oc)\ ) \vee$

False
)

fun *inv_tm_erase_right_then_dblBk_left_erp_s0* :: (*cell list*) ⇒ (*cell list*) ⇒ *tape* ⇒ *bool*
where
inv_tm_erase_right_then_dblBk_left_erp_s0 CL CR (*l*, *r*) =
(

$(\exists\, rex.\ l = [] \wedge\ r = [Bk, Bk]\ @\ (rev\ CL)\ @\ [Oc, Bk]\ @\ Bk \uparrow rex \wedge (CL = [] \vee last\ CL = Oc)\ \ ) \vee$

$(\exists\, rex.\ l = [] \wedge\ r = [Bk] \quad @\ (rev\ CL)\ @\ [Oc, Bk]\ @\ Bk \uparrow rex \wedge\ CL \neq [] \wedge last\ CL = Bk\ )$

)

fun *inv_tm_erase_right_then_dblBk_left_erp* :: (*cell list*) ⇒ (*cell list*) ⇒ *config* ⇒ *bool*
where
inv_tm_erase_right_then_dblBk_left_erp CL CR (*s*, *tap*) =
(*if s = 0 then inv_tm_erase_right_then_dblBk_left_erp_s0 CL CR tap else*
if s = 1 then inv_tm_erase_right_then_dblBk_left_erp_s1 CL CR tap else
if s = 2 then inv_tm_erase_right_then_dblBk_left_erp_s2 CL CR tap else
if s = 3 then inv_tm_erase_right_then_dblBk_left_erp_s3 CL CR tap else
if s = 5 then inv_tm_erase_right_then_dblBk_left_erp_s5 CL CR tap else
if s = 6 then inv_tm_erase_right_then_dblBk_left_erp_s6 CL CR tap else
if s = 7 then inv_tm_erase_right_then_dblBk_left_erp_s7 CL CR tap else
if s = 8 then inv_tm_erase_right_then_dblBk_left_erp_s8 CL CR tap else
if s = 9 then inv_tm_erase_right_then_dblBk_left_erp_s9 CL CR tap else

if s = 10 then inv_tm_erase_right_then_dblBk_left_erp_s10 CL CR tap else
if s = 11 then inv_tm_erase_right_then_dblBk_left_erp_s11 CL CR tap else
if s = 12 then inv_tm_erase_right_then_dblBk_left_erp_s12 CL CR tap
else False)

lemma *tm_erase_right_then_dblBk_left_erp_cases*:
fixes *s*::*nat*
assumes *inv_tm_erase_right_then_dblBk_left_erp CL CR* (*s*,*l*,*r*)
and *s=0* $\Longrightarrow$ *P*
and *s=1* $\Longrightarrow$ *P*
and *s=2* $\Longrightarrow$ *P*
and *s=3* $\Longrightarrow$ *P*
and *s=5* $\Longrightarrow$ *P*
and *s=6* $\Longrightarrow$ *P*
and *s=7* $\Longrightarrow$ *P*
and *s=8* $\Longrightarrow$ *P*
and *s=9* $\Longrightarrow$ *P*
and *s=10* $\Longrightarrow$ *P*
and *s=11* $\Longrightarrow$ *P*
and *s=12* $\Longrightarrow$ *P*
shows *P*
$\langle proof \rangle$

lemma *inv_tm_erase_right_then_dblBk_left_erp_step*:
assumes *inv_tm_erase_right_then_dblBk_left_erp CL CR cf*
and *noDblBk CL*
and *noDblBk CR*
shows *inv_tm_erase_right_then_dblBk_left_erp CL CR* (*step0 cf tm_erase_right_then_dblBk_left*)
$\langle proof \rangle$

lemma *inv_tm_erase_right_then_dblBk_left_erp_steps*:
assumes *inv_tm_erase_right_then_dblBk_left_erp CL CR cf*
and *noDblBk CL* **and** *noDblBk CR*
shows *inv_tm_erase_right_then_dblBk_left_erp CL CR* (*steps0 cf tm_erase_right_then_dblBk_left stp*)
$\langle proof \rangle$

lemma *tm_erase_right_then_dblBk_left_erp_partial_correctness_CL_is_Nil*:
assumes $\exists$ *stp*. *is_final* (*steps0* (*1*, [*Bk*,*Oc*] @ *CL*, *CR*) *tm_erase_right_then_dblBk_left stp*)
and *noDblBk CL*
and *noDblBk CR*
and *CL* = []
shows $\{\!|$ λ*tap*. *tap* = ([*Bk*,*Oc*] @ *CL*, *CR*) $|\!\}$

tm_erase_right_then_dblBk_left
$\{\!|\ \lambda tap.\ \exists\, rex.\ tap = ([], [Bk,Bk] @ (rev\ CL) @ [Oc, Bk] @ Bk \uparrow rex\ )\ |\!\}$
$\langle proof \rangle$

lemma *tm_erase_right_then_dblBk_left_erp_partial_correctness_CL_ew_Bk*:
assumes $\exists\, stp.\ is\_final\ (steps0\ (1, [Bk,Oc] @ CL, CR)\ tm\_erase\_right\_then\_dblBk\_left\ stp)$
and *noDblBk CL*
and *noDblBk CR*
and $CL \neq []$
and $last\ CL = Bk$
shows $\{\!|\ \lambda tap.\ tap = ([Bk,Oc] @ CL, CR)\ |\!\}$
tm_erase_right_then_dblBk_left
$\{\!|\ \lambda tap.\ \exists\, rex.\ tap = ([], [Bk] @ (rev\ CL) @ [Oc, Bk] @ Bk \uparrow rex\ )\ |\!\}$
$\langle proof \rangle$

lemma *tm_erase_right_then_dblBk_left_erp_partial_correctness_CL_ew_Oc*:
assumes $\exists\, stp.\ is\_final\ (steps0\ (1, [Bk,Oc] @ CL, CR)\ tm\_erase\_right\_then\_dblBk\_left\ stp)$
and *noDblBk CL*
and *noDblBk CR*
and $CL \neq []$
and $last\ CL = Oc$
shows $\{\!|\ \lambda tap.\ tap = ([Bk,Oc] @ CL, CR)\ |\!\}$
tm_erase_right_then_dblBk_left
$\{\!|\ \lambda tap.\ \exists\, rex.\ tap = ([], [Bk, Bk] @ (rev\ CL) @ [Oc, Bk] @ Bk \uparrow rex\ )\ |\!\}$
$\langle proof \rangle$

definition *measure_tm_erase_right_then_dblBk_left_erp* :: $(config \times config)\ set$
where
measure_tm_erase_right_then_dblBk_left_erp = *measures* [
$\lambda(s, l, r).$ (
if $s = 0$
then 0
else if $s < 6$
then $13 - s$
else 1),

$\lambda(s, l, r).$ (
if $s = 6$
then if $r = [] \vee (hd\ r) = Bk$
then 1
else 2
else 0),

$\lambda(s, l, r).$ (

if $7 \le s \wedge s \le 9$
then $2+$ *length r*
else 1),

$\lambda(s, l, r)$. (
if $7 \le s \wedge s \le 9$
then
if $r = [] \vee$ *hd* $r = Bk$
then 2
else 3
else 1),

$\lambda(s, l, r)$.(
if $7 \le s \wedge s \le 10$
then $13 - s$
else 1),

$\lambda(s, l, r)$. (
if $10 \le s$
then $2+$ *length l*
else 1),

$\lambda(s, l, r)$. (
if $11 \le s$
then if hd $r = Oc$
then 3
else 2
else 1),

$\lambda(s, l, r)$.(
if $11 \le s$
then $13 - s$
else 1)

]

lemma *wf_measure_tm_erase_right_then_dblBk_left_erp*: *wf measure_tm_erase_right_then_dblBk_left_erp*
⟨*proof*⟩

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

lemma *spike_erp_cases*:
$CL \neq [] \wedge$ *last* $CL = Bk \vee CL \neq [] \wedge$ *last* $CL = Oc \vee CL = []$
⟨*proof*⟩

lemma *tm_erase_right_then_dblBk_left_erp_halts*:

assumes *noDblBk CL*
and *noDblBk CR*
shows
$\exists$ *stp*. *is_final* (*steps0* (*1*, [*Bk*,*Oc*] @ *CL*, *CR*) *tm_erase_right_then_dblBk_left stp*)
⟨*proof*⟩

lemma *tm_erase_right_then_dblBk_left_erp_total_correctness_CL_is_Nil*:
assumes *noDblBk CL*
and *noDblBk CR*
and *CL* = []
shows {| λ*tap*. *tap* = ([*Bk*,*Oc*] @ *CL*, *CR*) |}
tm_erase_right_then_dblBk_left
{| λ*tap*. $\exists$ *rex*. *tap* = ([], [*Bk*,*Bk*] @ (*rev CL*) @ [*Oc*, *Bk*] @ *Bk* ↑ *rex*) |}
⟨*proof*⟩

lemma *tm_erase_right_then_dblBk_left_correctness_CL_ew_Bk*:
assumes *noDblBk CL*
and *noDblBk CR*
and *CL* ≠ []
and *last CL* = *Bk*
shows {| λ*tap*. *tap* = ([*Bk*,*Oc*] @ *CL*, *CR*) |}
tm_erase_right_then_dblBk_left
{| λ*tap*. $\exists$ *rex*. *tap* = ([], [*Bk*] @ (*rev CL*) @ [*Oc*, *Bk*] @ *Bk* ↑ *rex*) |}
⟨*proof*⟩

lemma *tm_erase_right_then_dblBk_left_erp_total_correctness_CL_ew_Oc*:
assumes *noDblBk CL*
and *noDblBk CR*
and *CL* ≠ []
and *last CL* = *Oc*
shows {| λ*tap*. *tap* = ([*Bk*,*Oc*] @ *CL*, *CR*) |}
tm_erase_right_then_dblBk_left
{| λ*tap*. $\exists$ *rex*. *tap* = ([], [*Bk*, *Bk*] @ (*rev CL*) @ [*Oc*, *Bk*] @ *Bk* ↑ *rex*) |}
⟨*proof*⟩

lemma *tm_erase_right_then_dblBk_left_erp_total_correctness_helper_CL_eq_Nil_n_eq_1_last_eq_0*:
assumes (*nl*::*nat list*) ≠ []
and *n*=*1*
and *n* ≤ *length nl*
and *last* (*take n nl*) = *0*
shows $\exists$ *CL CR*.

[Oc] @ CL = rev(<take n nl>) ∧ noDblBk CL ∧ CL = [] ∧
CR = (<drop n nl>) ∧ noDblBk CR
⟨proof⟩

lemma tm_erase_right_then_dblBk_left_erp_total_correctness_helper_CL_neq_Nil_n_eq_1_last_neq_0:
assumes (nl::nat list) ≠ []
and n=1
and n ≤ length nl
and 0 < last (take n nl)
shows ∃ CL CR.
[Oc] @ CL = rev(<take n nl>) ∧ noDblBk CL ∧ CL ≠ [] ∧ last CL = Oc ∧
CR = (<drop n nl>) ∧ noDblBk CR
⟨proof⟩

lemma tm_erase_right_then_dblBk_left_erp_total_correctness_helper_CL_eq_Nil_last_eq_0′:
assumes 1 ≤ length (nl:: nat list)
and hd nl = 0
shows ∃ CL CR.
[Oc] @ CL = rev(<hd nl>) ∧ noDblBk CL ∧ CL = [] ∧
CR = (<tl nl>) ∧ noDblBk CR
⟨proof⟩

lemma tm_erase_right_then_dblBk_left_erp_total_correctness_helper_CL_neq_Nil_last_neq_0′:
assumes 1 ≤ length (nl::nat list)
and 0 < hd nl
shows ∃ CL CR.
[Oc] @ CL = rev(<hd nl>) ∧ noDblBk CL ∧ CL ≠ [] ∧ last CL = Oc ∧
CR = (<tl nl>) ∧ noDblBk CR
⟨proof⟩

lemma tm_erase_right_then_dblBk_left_erp_total_correctness_helper_CL_neq_Nil_n_gt_1_last_eq_0:
assumes (nl::nat list) ≠ []
and 1<n
and n ≤ length nl
and last (take n nl) = 0
shows ∃ CL CR.
[Oc] @ CL = rev(<take n nl>) ∧ noDblBk CL ∧ CL ≠ [] ∧ last CL = Oc ∧
CR = (<drop n nl>) ∧ noDblBk CR
⟨proof⟩

lemma tm_erase_right_then_dblBk_left_erp_total_correctness_helper_CL_neq_Nil_n_gt_1_last_neq_0:
assumes (nl::nat list) ≠ []
and 1 < n
and n ≤ length nl

and $0 < \mathit{last}\ (\mathit{take}\ n\ \mathit{nl})$
shows $\exists\, \mathit{CL}\ \mathit{CR}.$
$[\mathit{Oc}]\ @\ \mathit{CL} = \mathit{rev}(<\mathit{take}\ n\ \mathit{nl}>) \wedge \mathit{noDblBk}\ \mathit{CL} \wedge \mathit{CL} \neq [] \wedge \mathit{last}\ \mathit{CL} = \mathit{Oc}\ \wedge$
$\mathit{CR} = (<\mathit{drop}\ n\ \mathit{nl}>) \quad \wedge \mathit{noDblBk}\ \mathit{CR}$
$\langle \mathit{proof} \rangle$

lemma *tm_erase_right_then_dblBk_left_erp_total_correctness_helper_CL_neq_Nil_n_gt_1*:
assumes $(\mathit{nl}{::}\mathit{nat\ list}) \neq []$
and $1<n$
and $n \leq \mathit{length}\ \mathit{nl}$
shows $\exists\, \mathit{CL}\ \mathit{CR}.$
$[\mathit{Oc}]\ @\ \mathit{CL} = \mathit{rev}(<\mathit{take}\ n\ \mathit{nl}>) \wedge \mathit{noDblBk}\ \mathit{CL} \wedge \mathit{CL} \neq [] \wedge \mathit{last}\ \mathit{CL} = \mathit{Oc}\ \wedge$
$\mathit{CR} = (<\mathit{drop}\ n\ \mathit{nl}>) \wedge \mathit{noDblBk}\ \mathit{CR}$
$\langle \mathit{proof} \rangle$

lemma *tm_erase_right_then_dblBk_left_erp_total_correctness_one_arg*:
assumes $1 \leq \mathit{length}\ (\mathit{nl}{::}\mathit{nat\ list})$
shows $\{\!|\ \lambda \mathit{tap}.\ \mathit{tap} = (\mathit{Bk}\#\ \mathit{rev}(<\mathit{hd}\ \mathit{nl}>), <\mathit{tl}\ \mathit{nl}>)\ |\!\}$
tm_erase_right_then_dblBk_left
$\{\!|\ \lambda \mathit{tap}.\ \exists\, \mathit{rex}.\ \mathit{tap} = ([], [\mathit{Bk},\mathit{Bk}]\ @\ (<\mathit{hd}\ \mathit{nl}>)\ @\ [\mathit{Bk}]\ @\ \mathit{Bk} \uparrow \mathit{rex}\ )\ |\!\}$
$\langle \mathit{proof} \rangle$

definition
tm_check_for_one_arg :: *instr list*
where
tm_check_for_one_arg $\stackrel{\mathit{def}}{=}$ *tm_skip_first_arg* $|{+}|$ *tm_erase_right_then_dblBk_left*

lemma *tm_check_for_one_arg_total_correctness_Nil*:
$\mathit{length}\ \mathit{nl} = 0$
$\Longrightarrow \{\!|\lambda \mathit{tap}.\ \mathit{tap} = ([], <\mathit{nl}{::}\mathit{nat\ list}>)\ |\!\}$ *tm_check_for_one_arg* $\{\!|\lambda \mathit{tap}.\ \mathit{tap} = ([\mathit{Bk},\mathit{Bk}], [\mathit{Bk}]\ )$
$|\!\}$
$\langle \mathit{proof} \rangle$

lemma *tm_check_for_one_arg_total_correctness_len_eq_1*:
length nl = 1
⟹ ⦃λ*tap*. *tap* = ([], <*nl*::*nat list*>) ⦄ *tm_check_for_one_arg* ⦃λ*tap*. ∃ *z4*. *tap* = (*Bk* ↑ *z4*, <*nl*> @ [*Bk*])⦄
⟨*proof*⟩

lemma *tm_check_for_one_arg_total_correctness_len_gt_1*:
length nl > 1
⟹ ⦃λ*tap*. *tap* = ([], <*nl*::*nat list*>)⦄ *tm_check_for_one_arg* ⦃ λ*tap*. ∃ *l*. *tap* = ([], [*Bk*,*Bk*] @ <[*hd nl*]> @ *Bk* ↑ *l*) ⦄
⟨*proof*⟩

definition
tm_strong_copy :: *instr list*
where
tm_strong_copy $\stackrel{def}{=}$ *tm_check_for_one_arg* |+| *tm_weak_copy*

lemma *tm_strong_copy_total_correctness_Nil*:
length nl = 0
⟹ ⦃λ*tap*. *tap* = ([], <*nl*::*nat list*>) ⦄ *tm_strong_copy* ⦃λ*tap*. *tap* = ([*Bk*,*Bk*,*Bk*,*Bk*],[]) ⦄
⟨*proof*⟩

lemma *tm_strong_copy_total_correctness_len_gt_1*:
1 < length nl
⟹ ⦃λ*tap*. *tap* = ([], <*nl*::*nat list*>) ⦄ *tm_strong_copy* ⦃ λ*tap*. ∃ *l*. *tap* = ([*Bk*,*Bk*], <[*hd nl*]> @ *Bk* ↑ *l*) ⦄
⟨*proof*⟩

lemma *tm_strong_copy_total_correctness_len_eq_1*:
1 = length nl
⟹ ⦃λ*tap*. *tap* = ([], <*nl*::*nat list*>) ⦄ *tm_strong_copy* ⦃ λ*tap*. ∃ *k l*. *tap* = (*Bk* ↑ *k*, <[*hd nl*, *hd nl*]> @ *Bk* ↑ *l*) ⦄
⟨*proof*⟩

end

1.11 Turing Decidability

theory *TuringDecidable*
imports
OneStrokeTM

Turing_HaltingConditions
begin

1.11.1 Turing Decidable Sets and Relations of natural numbers

We use lists of natural numbers in order to model tuples of arity k of natural numbers, where $0 \leq k$.

Now, we define the notion of *Turing Decidable Sets and Relations*. In our definition, we directly relate decidability of sets and relations to Turing machines and do not adhere to the formal concept of a characteristic function.

However, the notion of a characteristic function is introduced in the theory about Turing computable functions.

definition *turing_decidable* :: (*nat list*) *set* ⇒ *bool*
where
$turing\_decidable\ nls \stackrel{def}{=} (\exists D.\ (\forall nl.$
$(nl \in nls \longrightarrow \{\!\!|(\lambda tap.\ tap = ([],\ <nl>))|\!\!\}\ D\ \{\!\!|(\lambda tap.\ \exists k\ l.\ tap = (Bk \uparrow k,\ <1::nat>\ @\ Bk\uparrow l))|\!\!\})$
$\wedge\ (nl \notin nls \longrightarrow \{\!\!|(\lambda tap.\ tap = ([],\ <nl>))|\!\!\}\ D\ \{\!\!|(\lambda tap.\ \exists k\ l.\ tap = (Bk \uparrow k,\ <0::nat>\ @\ Bk\uparrow l))|\!\!\})$
$))$

lemma *turing_decidable_unfolded_into_TMC_yields_conditions*:
$turing\_decidable\ nls \stackrel{def}{=} (\exists D.\ (\forall nl.$
$(nl \in nls \longrightarrow TMC\_yields\_num\_res\ D\ nl\ (1::nat))$
$\wedge\ (nl \notin nls \longrightarrow TMC\_yields\_num\_res\ D\ nl\ (0::nat))$
$))$
⟨*proof*⟩

1.11.2 Examples for decidable sets of natural numbers

Using the machine OneStrokeTM as a decider we are able to proof the decidability of the empty set. Moreover, in the theory about Halting Problems, we will show that there are undecidable sets as well. Thus, the notion of Turing Decidability is not a trivial concept.

lemma *turing_decidable_empty_set_iff*:
$turing\_decidable\ \{\} = (\exists D.\ \forall (nl::\ nat\ list).$
$\{\!\!|(\lambda tap.\ tap = ([],\ <nl>))|\!\!\}\ D\ \{\!\!|(\lambda tap.\ \exists k\ l.\ tap = (Bk \uparrow k,\ [Oc]\ @\ Bk \uparrow l))|\!\!\})$
⟨*proof*⟩

theorem *turing_decidable_empty_set*: $turing\_decidable\ \{\}$
⟨*proof*⟩

end

1.12 Turing Reducibility

theory *TuringReducible*
 imports
  TuringDecidable
  StrongCopyTM
begin

1.12.1 Definition of Turing Reducibility of Sets and Relations of Natural Numbers

Let A and B be two sets of lists of natural numbers.

The set A is called many-one reducible to set B, if there is a Turing machine tm such that for all a we have:

1. the Turing machine always computes a list b of natural numbers from the list b of natural numbers

2. $a \in A$ if and only if the value b computed by tm from a is an element of set B.

We generalized our definition to lists, which eliminates the need to encode lists of natural numbers into a single natural number. Compare this to the theory of recursive functions, where all values computed must be a single natural number.

Note however, that our notion of reducibility is not stronger than the one used in recursion theory. Every finite list of natural numbers can be encoded into a single natural number. Our definition is just more convenient for Turing machines, which are capable of producing lists of values.

definition *turing_reducible* :: $(nat\ list)\ set \Rightarrow (nat\ list)\ set\ \Rightarrow bool$
 where

  $turing\_reducible\ A\ B \stackrel{def}{=}$

$(\exists\, tm.\ \forall\, nl{::}nat\ list.\ \exists\, ml{::}nat\ list.$
$\{\!|(\lambda tap.\ tap = ([], <nl>))|\!\}\ tm\ \{\!|(\lambda tap.\ \exists\, k\ l.\ tap = (Bk \uparrow k, <ml>\ @\ Bk\uparrow l))|\!\} \wedge$
$(nl \in A \longleftrightarrow ml \in B)$
$)$

lemma *turing_reducible_unfolded_into_TMC_yields_condition*:
turing_reducible A B $\stackrel{def}{=}$
$(\exists\, tm.\ \forall\, nl{::}nat\ list.\ \exists\, ml{::}nat\ list.$
TMC_yields_num_list_res tm nl ml $\wedge\ (nl \in A \longleftrightarrow ml \in B)$
$)$
⟨*proof*⟩

1.12.2 Theorems about Turing Reducibility of Sets and Relations of Natural Numbers

lemma *turing_reducible_A_B_imp_composable_reducer_ex*: *turing_reducible A B*
$\Longrightarrow$
$\exists$ *Red. composable_tm0 Red* $\wedge$
($\forall$ *nl*::*nat list*. $\exists$ *ml*::*nat list*. *TMC_yields_num_list_res Red nl ml* $\wedge$ $(nl \in A \longleftrightarrow ml \in$
$B))$
⟨*proof*⟩

theorem *turing_reducible_AB_and_decB_imp_decA*:
$[\![$ *turing_reducible A B*; *turing_decidable B* $]\!] \Longrightarrow$ *turing_decidable A*
⟨*proof*⟩

corollary *turing_reducible_AB_and_non_decA_imp_non_decB*:
$[\![$*turing_reducible A B*; $\neg$ *turing_decidable A* $]\!] \Longrightarrow \neg$*turing_decidable B*
⟨*proof*⟩

end

1.13 Halting Problems: do Turing Machines for deciding Termination exist?

In this section we will show that there cannot exist Turing Machines that are able to decide the termination of some other arbitrary Turing Machine.

1.13.1 A simple Gödel Encoding for Turing machines

theory *SimpleGoedelEncoding*

imports
  Turing_HaltingConditions
  HOL−Library.Nat_Bijection
begin

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

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

1.13.1.1 Some general results on injective functions and their inversion

lemma *dec_is_inv_on_A*:
  $dec = (\lambda w.\ (if\ (\exists\, t{\in}A.\ enc\ t = w)\ then\ (THE\ t.\ t{\in}A \wedge enc\ t = w)\ else\ (SOME\ t.\ t \in A)))$
  $\Longrightarrow dec = (\lambda w.\ (if\ (\exists\, t{\in}A.\ enc\ t = w)\ then\ (the\_inv\_into\ A\ enc)\ w\ else\ (SOME\ t.\ t \in A)))$
  ⟨*proof*⟩

lemma *encode_decode_A_eq*:
  $[\![\ inj\_on\ (enc{::}'a \Rightarrow 'b)\ (A{::}'a\ set);$
    $(dec{::}'b \Rightarrow\ 'a) = (\lambda w.\ (if\ (\exists\, t{\in}A.\ enc\ t = w)$
      $then\ (THE\ t.\ t{\in}A \wedge enc\ t = w)$
      $else\ (SOME\ t.\ t \in A)))$
  $]\!] \Longrightarrow \forall M{\in}A.\ dec(enc\ M) = M$
⟨*proof*⟩

lemma *decode_encode_A_eq*:
  $[\![\ inj\_on\ (enc{::}'a \Rightarrow 'b)\ (A{::}'a\ set);$
   $dec = (\lambda w.\ (if\ (\exists\, t{\in}A.\ enc\ t = w)\ then\ (THE\ t.\ t{\in}A \wedge enc\ t = w)\ else\ (SOME\ t.\ t \in A)))]\!]$
  $\Longrightarrow \forall w.\ w \in enc`A \longrightarrow enc(dec(w)) = w$
⟨*proof*⟩

lemma *dec_in_A*:
  $[\![inj\_on\ (enc{::}'a \Rightarrow 'b)\ (A{::}'a\ set);$
   $dec = (\lambda w.\ if\ \exists\, t{\in}A.\ enc\ t = w\ then\ THE\ t.\ t \in A \wedge enc\ t = w\ else\ SOME\ t.\ t \in A);$
   $A \neq \{\}\ ]\!]$
  $\Longrightarrow \forall w.\ dec\ w \in A$
⟨*proof*⟩

1.13.1.2 An injective encoding of Turing Machines into the natural number

We define an injective encoding function from Turing machines to natural numbers. This encoding function is only used for the proof of the undecidability of the special halting problem K where we use a locale that postulates the existence of some injective encoding of the Turing machines into the natural numbers.

fun *tm_to_nat_list* :: *tprog0* ⇒ *nat list*
where
tm_to_nat_list [] = [] |
tm_to_nat_list ((*WB* ,*s*) # *is*) = *0* # *s* # *tm_to_nat_list is* |
tm_to_nat_list ((*WO* ,*s*) # *is*) = *1* # *s* # *tm_to_nat_list is* |
tm_to_nat_list ((*L* ,*s*) # *is*) = *2* # *s* # *tm_to_nat_list is* |
tm_to_nat_list ((*R* ,*s*) # *is*) = *3* # *s* # *tm_to_nat_list is* |
tm_to_nat_list ((*Nop* ,*s*) # *is*) = *4* # *s* # *tm_to_nat_list is*

lemma *prefix_tm_to_nat_list_cons*:
∃ *u v*. *tm_to_nat_list* (*x*#*xs*) = *u* # *v* # *tm_to_nat_list xs*
⟨*proof*⟩

lemma *tm_to_nat_list_cons_is_not_nil*: *tm_to_nat_list* (*x*#*xs*) ≠ *tm_to_nat_list* []
⟨*proof*⟩

lemma *inj_in_fst_arg_tm_to_nat_list*:
tm_to_nat_list (*x* # *xs*) = *tm_to_nat_list* (*y* # *xs*) ⟹ *x* = *y*
⟨*proof*⟩

lemma *inj_tm_to_nat_list*: *tm_to_nat_list xs* = *tm_to_nat_list ys* ⟶ *xs* = *ys*
⟨*proof*⟩

definition *tm_to_nat* :: *tprog0* ⇒ *nat*
where *tm_to_nat* = (*list_encode* ∘ *tm_to_nat_list*)

theorem *inj_tm_to_nat*: *inj tm_to_nat*
⟨*proof*⟩

fun *nat_list_to_tm* :: *nat list* ⇒ *tprog0*
where
nat_list_to_tm [] = []
| *nat_list_to_tm* [*ac*] = [(*Nop*, *0*)]
| *nat_list_to_tm* (*ac* # *s* # *ns*) = (
if ac < *5*
then ([*WB*,*WO*,*L*,*R*,*Nop*]!*ac* ,*s*) # *nat_list_to_tm ns*
else [(*Nop*, *0*)])

lemma *nat_list_to_tm_is_inv_of_tm_to_nat_list*: *nat_list_to_tm* (*tm_to_nat_list ns*) = *ns*
⟨*proof*⟩

definition *nat_to_tm* :: *nat* ⇒ *tprog0*
where *nat_to_tm* = (*nat_list_to_tm* ∘ *list_decode*)

lemma *nat_to_tm_is_inv_of_tm_to_nat*: $nat\_to\_tm\ (tm\_to\_nat\ tm) = tm$
⟨*proof*⟩

end

1.13.2 Undecidability of Halting Problems

theory *HaltingProblems_K_H*
imports
SimpleGoedelEncoding
SemiIdTM
TuringReducible

begin

1.13.2.1 A locale for variations of the Halting Problem

The following locale assumes that there is an injective coding function *t2c* from Turing machines to natural numbers. In this locale, we will show that the Special Halting Problem K1 and the General Halting Problem H1 are not Turing decidable.

locale *hpk* =
fixes $t2c :: tprog0 \Rightarrow nat$
assumes
t2c_inj: *inj t2c*

begin

The function *tm_to_nat* is a witness that the locale hpk is inhabited.

interpretation *tm_to_nat*: $hpk\ tm\_to\_nat :: tprog0 \Rightarrow nat$
⟨*proof*⟩

We define the function *c2t* as the unique inverse of the injective function *t2c*.

definition $c2t :: nat \Rightarrow instr\ list$
where
$c2t = (\lambda n.\ if\ (\exists p.\ t2c\ p = n)$
$then\ (THE\ p.\ t2c\ p = n)$
$else\ (SOME\ p.\ True)\ )$

lemma *t2c_inj'*: $inj\_on\ t2c\ \{x.\ True\}$
⟨*proof*⟩

lemma *c2t_comp_t2c_eq*: $c2t\ (t2c\ p) = p$
⟨*proof*⟩

1.13.2.2 Undecidability of the Special Halting Problem K1

definition *K1* :: (*nat list*) *set*
 where
 $K1 \overset{def}{=} \{nl.\ (\exists\, n.\ nl = [n] \wedge TMC\_has\_num\_res\ (c2t\ n)\ [n])\ \}$

Assuming the existence of a Turing Machine K1D1, which is able to decide the set K1, we derive a contradiction using the machine *tm_semi_id_eq0*. Thus, we show that the *Special Halting Problem K1* is not Turing decidable. The proof uses a diagonal argument.

lemma *mk_composable_decider_K1D1*:
 assumes $\exists\, K1D1'.\ (\forall\, nl.$
 $(nl \in K1 \longrightarrow TMC\_yields\_num\_res\ K1D1'\ nl\ (1{::}nat))$
 $\wedge\ (nl \notin K1 \longrightarrow TMC\_yields\_num\_res\ K1D1'\ nl\ (0{::}nat)\ ))$

 shows $\exists\, K1D1'.\ (\forall\, nl.\ composable\_tm0\ K1D1' \wedge$
 $(nl \in K1 \longrightarrow TMC\_yields\_num\_res\ K1D1'\ nl\ (1{::}nat))$
 $\wedge\ (nl \notin K1 \longrightarrow TMC\_yields\_num\_res\ K1D1'\ nl\ (0{::}nat)\ ))$
⟨*proof*⟩

lemma *res_1_fed_into_tm_semi_id_eq0_loops*:
 assumes *composable_tm0 D*
 and $TMC\_yields\_num\_res\ D\ nl\ (1{::}nat)$
 shows $TMC\_loops\ (D\ |{+}|\ tm\_semi\_id\_eq0)\ nl$
 ⟨*proof*⟩

lemma *loops_imp_has_no_res*: $TMC\_loops\ tm\ [n] \Longrightarrow \neg\ TMC\_has\_num\_res\ tm\ [n]$
⟨*proof*⟩

lemma *yields_res_imp_has_res*: $TMC\_yields\_num\_res\ tm\ [n]\ (m{::}nat) \Longrightarrow TMC\_has\_num\_res\ tm\ [n]$
⟨*proof*⟩

lemma *res_0_fed_into_tm_semi_id_eq0_yields_0*:
 assumes *composable_tm0 D*
 and $TMC\_yields\_num\_res\ D\ nl\ (0{::}nat)$
 shows $TMC\_yields\_num\_res\ (D\ |{+}|\ tm\_semi\_id\_eq0)\ nl\ 0$
 ⟨*proof*⟩

lemma *existence_of_decider_K1D1_for_K1_imp_False*:
 assumes *major*: $\exists\, K1D1'.\ (\forall\, nl.$
 $(nl \in K1 \longrightarrow TMC\_yields\_num\_res\ K1D1'\ nl\ (1{::}nat))$
 $\wedge\ (nl \notin K1 \longrightarrow TMC\_yields\_num\_res\ K1D1'\ nl\ (0{::}nat)\ ))$
 shows *False*
⟨*proof*⟩

1.13.2.3 The Special Halting Problem K1 is reducible to the General Halting Problem H1

The proof is by reduction of *K1* to *H1*.

definition *H1* :: (*nat list*) *set*
where
$H1 \stackrel{def}{=} \{nl.\ (\exists\, n\ m.\ nl = [n,m] \wedge TMC\_has\_num\_res\ (c2t\ n)\ [m])\ \}$

lemma *NilNotIn_K1*: $[] \notin K1$
⟨*proof*⟩

lemma *NilNotIn_H1*: $[] \notin H1$
⟨*proof*⟩

lemma *tm_strong_copy_total_correctness_Nil′*:
$length\ nl = 0 \Longrightarrow TMC\_yields\_num\_list\_res\ tm\_strong\_copy\ nl\ []$
⟨*proof*⟩

lemma *tm_strong_copy_total_correctness_len_eq_1′*:
$length\ nl = 1 \Longrightarrow TMC\_yields\_num\_list\_res\ tm\_strong\_copy\ nl\ [hd\ nl,\ hd\ nl]$
⟨*proof*⟩

lemma *tm_strong_copy_total_correctness_len_gt_1′*:
$1 < length\ nl \Longrightarrow TMC\_yields\_num\_list\_res\ tm\_strong\_copy\ nl\ [hd\ nl]$
⟨*proof*⟩

theorem *turing_reducible_K1_H1*: *turing_reducible K1 H1*
⟨*proof*⟩

1.13.2.4 Corollaries about the undecidable sets K1 and H1

corollary *not_Turing_decidable_K1*: $\neg(turing\_decidable\ K1)$
⟨*proof*⟩

corollary *not_Turing_decidable_H1*: $\neg turing\_decidable\ H1$
⟨*proof*⟩

1.13.2.5 Proof variant: The special Halting Problem K1 is not Turing Decidable

Assuming the existence of a Turing Machine K1D0, which is able to decide the set K1, we derive a contradiction using the machine *tm_semi_id_gt0*. Thus, we show that the *Special Halting Problem K1* is not Turing decidable. The proof uses a diagonal argument.

lemma *existence_of_decider_K1D0_for_K1_imp_False*:
assumes $\exists\, K1D0'.\ (\forall\, nl.$
$(nl \in K1 \longrightarrow TMC\_yields\_num\_res\ K1D0'\ nl\ (0{::}nat))$
$\wedge\ (nl \notin K1 \longrightarrow TMC\_yields\_num\_res\ K1D0'\ nl\ (1{::}nat)\ ))$

shows *False*
⟨*proof*⟩

end

end

1.13.2.6 K0: A Variant of the Special Halting Problem K1

theory *HaltingProblems_K_aux*
imports
HaltingProblems_K_H

begin

context *hpk*
begin

definition *K0* :: (*nat list*) *set*
where
$K0 \stackrel{def}{=} \{nl.\ (\exists\, n.\ nl = [n] \wedge reaches\_final\ (c2t\ n)\ [n])\ \}$

Assuming the existence of a Turing Machine K0D0, which is able to decide the set K0, we derive a contradiction using the machine *tm_semi_id_gt0*. Thus, we show that the *Special Halting Problem K0* is not Turing decidable. The proof uses a diagonal argument.

lemma *existence_of_decider_K0D0_for_K0_imp_False*:
assumes $\exists\, K0D0'.\ (\forall\, nl.$
$(nl \in K0 \longrightarrow TMC\_yields\_num\_res\ K0D0'\ nl\ (0::nat))$
$\wedge\ (nl \notin K0 \longrightarrow TMC\_yields\_num\_res\ K0D0'\ nl\ (1::nat)\ ))$
shows *False*
⟨*proof*⟩

Assuming the existence of a Turing Machine K0D1, which is able to decide the set K0, we derive a contradiction using the machine *tm_semi_id_eq0*. Thus, we show that the *Special Halting Problem K0* is not Turing decidable. The proof uses a diagonal argument.

lemma *existence_of_decider_K0D1_for_K0_imp_False*:
assumes $\exists\, K0D1'.\ (\forall\, nl.$
$(nl \in K0 \longrightarrow TMC\_yields\_num\_res\ K0D1'\ nl\ (1::nat))$
$\wedge\ (nl \notin K0 \longrightarrow TMC\_yields\_num\_res\ K0D1'\ nl\ (0::nat)\ ))$
shows *False*
⟨*proof*⟩

corollary *not_Turing_decidable_K0*: $\neg(turing\_decidable\ K0)$

⟨*proof*⟩

end

end

1.14 Turing Computable Functions

theory *TuringComputable*
 imports
  HaltingProblems_K_H
begin

1.14.1 Definition of Partial Turing Computability

We present two variants for a definition of Partial Turing Computability, which we prove to be equivalent, later on.

1.14.1.1 Definition Variant 1

definition *turing_computable_partial* :: (*nat list* ⇒ *nat option*) ⇒ *bool*
 where *turing_computable_partial f* $\stackrel{def}{=}$ (∃ *tm*. ∀ *ns n*.
  (*f ns* = *Some n* ⟶ (∃ *stp k l*. (*steps0* (*1*, ([], <*ns*::*nat list*>)) *tm stp*) = (*0*, *Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*))) ∧
  (*f ns* = *None* ⟶ ¬⦃ λ*tap*. *tap* = ([], <*ns*>) ⦄ *tm* ⦃ λ*tap*. (∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)) ⦄))

lemma *turing_computable_partial_unfolded_into_TMC_yields_TMC_has_conditions*:
 turing_computable_partial f $\stackrel{def}{=}$ (∃ *tm*. ∀ *ns n*.
  (*f ns* = *Some n* ⟶ *TMC_yields_num_res tm ns n*) ∧
  (*f ns* = *None* ⟶ ¬ *TMC_has_num_res tm ns*))
 ⟨*proof*⟩

lemma *turing_computable_partial_unfolded_into_Hoare_halt_conditions*:
 turing_computable_partial f ⟷ (∃ *tm*. ∀ *ns n*.
  (*f ns* = *Some n* ⟶ ⦃ λ*tap*. *tap* = ([], <*ns*::*nat list*>) ⦄ *tm* ⦃ λ*tap*. ∃ *k l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk*↑ *l*) ⦄) ∧
  (*f ns* = *None* ⟶ ¬⦃ λ*tap*. *tap* = ([], <*ns*::*nat list*>) ⦄ *tm* ⦃ λ*tap*. ∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*) ⦄))
 ⟨*proof*⟩

1.14.1.2 Characteristic Functions of Sets

definition *chi_fun* :: (*nat list*) *set* ⇒ (*nat list* ⇒ *nat option*)
where
chi_fun nls = (λ*nl*. *if nl* ∈ *nls then Some 1 else Some 0*)

lemma *chi_fun_0_iff*: *nl* ∉ *nls* ⟷ *chi_fun nls nl* = *Some 0*
⟨*proof*⟩

lemma *chi_fun_1_iff*: *nl* ∈ *nls* ⟷ *chi_fun nls nl* = *Some 1*
⟨*proof*⟩

lemma *chi_fun_0_I*: *nl* ∉ *nls* ⟹ *chi_fun nls nl* = *Some 0*
⟨*proof*⟩

lemma *chi_fun_0_E*: (*chi_fun nls nl* = *Some 0* ⟹ *P*) ⟹ *nl* ∉ *nls* ⟹ *P*
⟨*proof*⟩

lemma *chi_fun_1_I*: *nl* ∈ *nls* ⟹ *chi_fun nls nl* = *Some 1*
⟨*proof*⟩

lemma *chi_fun_1_E*: (*chi_fun nls nl* = *Some 1* ⟹ *P*) ⟹ *nl* ∈ *nls* ⟹ *P*
⟨*proof*⟩

1.14.1.3 Relation between Partial Turing Computability and Turing Decidability

If a set A is Turing Decidable its characteristic function is Turing Computable partial and vice versa. Please note, that although the characteristic function has an option type it will always yield Some value.

theorem *turing_decidable_imp_turing_computable_partial*:
turing_decidable A ⟹ *turing_computable_partial* (*chi_fun A*)
⟨*proof*⟩

theorem *turing_computable_partial_imp_turing_decidable*:
turing_computable_partial (*chi_fun A*) ⟹ *turing_decidable A*
⟨*proof*⟩

corollary *turing_computable_partial_iff_turing_decidable*:
turing_decidable A ⟷ *turing_computable_partial* (*chi_fun A*)
⟨*proof*⟩

1.14.1.4 Examples for uncomputable functions

Now, we prove that the characteristic functions of the undecidable sets K1 and H1 are both uncomputable.

context *hpk*
begin

theorem ¬(*turing_computable_partial* (*chi_fun K1*))

⟨*proof*⟩

theorem ¬(*turing_computable_partial* (*chi_fun H1*))
⟨*proof*⟩

end

1.14.1.5 The Function associated with a Turing Machine

With every Turing machine, we can associate a function.

definition *fun_of_tm* :: *tprog0* ⇒ (*nat list* ⇒ *nat option*)
where *fun_of_tm tm ns* $\stackrel{def}{=}$
(*if* ⦃ λ*tap*. *tap* = ([], <*ns*>) ⦄ *tm* ⦃ λ*tap*. (∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)) ⦄
then
let result =
(*THE n*. ∃ *stp k l*. (*steps0* (*1*, ([], <*ns*>)) *tm stp*) = (*0*, *Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*))
in Some result
else None)

Some immediate consequences of the definition.

lemma *fun_of_tm_unfolded_into_TMC_yields_TMC_has_conditions*:
fun_of_tm tm $\stackrel{def}{=}$
(λ*ns*. (*if TMC_has_num_res tm ns*
then
let result = (*THE n*. *TMC_yields_num_res tm ns n*)
in Some result
else None)
)
⟨*proof*⟩

lemma *fun_of_tm_is_None*:
assumes ¬(⦃ λ*tap*. *tap* = ([], <*ns*>) ⦄ *tm* ⦃ λ*tap*. (∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)) ⦄)
shows *fun_of_tm tm ns* = *None*
⟨*proof*⟩

lemma *fun_of_tm_is_None_rev*:
assumes *fun_of_tm tm ns* = *None*
shows ¬(⦃ λ*tap*. *tap* = ([], <*ns*>) ⦄ *tm* ⦃ λ*tap*. (∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)) ⦄)
⟨*proof*⟩

corollary *fun_of_tm_is_None_iff*: *fun_of_tm tm ns* = *None* ⟷ ¬(⦃ λ*tap*. *tap* = ([], <*ns*>) ⦄ *tm* ⦃ λ*tap*. (∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)) ⦄)
⟨*proof*⟩

corollary *fun_of_tm_is_None_iff'*: *fun_of_tm tm ns* = *None* ⟷ ¬ *TMC_has_num_res tm ns*
⟨*proof*⟩

lemma *fun_of_tm_ex_Some_n'*:
assumes ⦃ λ*tap*. *tap* = ([], <*ns*>) ⦄ *tm* ⦃ λ*tap*. (∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)) ⦄
shows ∃ *n*. *fun_of_tm tm ns* = *Some n*
⟨*proof*⟩

lemma *fun_of_tm_ex_Some_n'_rev*:
assumes ∃ *n*. *fun_of_tm tm ns* = *Some n*
shows ⦃ λ*tap*. *tap* = ([], <*ns*>) ⦄ *tm* ⦃ λ*tap*. (∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)) ⦄
⟨*proof*⟩

corollary *fun_of_tm_ex_Some_n'_iff*:
(∃ *n*. *fun_of_tm tm ns* = *Some n*)
⟷
⦃ λ*tap*. *tap* = ([], <*ns*>) ⦄ *tm* ⦃ λ*tap*. (∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)) ⦄
⟨*proof*⟩

1.14.1.6 Stronger results about uniqueness of results

corollary *Hoare_halt_on_numeral_list_yields_unique_list_result_iff*:
⦃λ*tap*. *tap* = ([], <*nl*::*nat list*>)⦄ *p* ⦃λ*tap*. ∃ *kr ml lr*. *tap* = (*Bk* ↑ *kr*, <*ml*::*nat list*> @ *Bk* ↑ *lr*)⦄
⟷
(∃!*ml*. ∃ *stp k l*. *steps0* (*1*,[], <*nl*::*nat list*>) *p stp* = (*0*, *Bk* ↑ *k*, <*ml*::*nat list*> @ *Bk* ↑ *l*))
⟨*proof*⟩

corollary *Hoare_halt_on_numeral_yields_unique_result_iff*:
⦃(λ*tap*. *tap* = ([], <*ns*::*nat list*>))⦄ *p* ⦃(λ*tap*. (∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)))⦄
⟷
(∃!*n*. ∃ *stp k l*. *steps0* (*1*,[], <*ns*::*nat list*>) *p stp* = (*0*, *Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*))
⟨*proof*⟩

lemma *fun_of_tm_is_Some_unique_value*:
assumes *steps0* (*1*, ([], <*ns*>)) *tm stp* = (*0*, *Bk* ↑ *k1*, <*n*::*nat*> @ *Bk* ↑ *l1*)
shows *fun_of_tm tm ns* = *Some n*
⟨*proof*⟩

lemma *fun_of_tm_ex_Some_n*:
assumes ⦃ λ*tap*. *tap* = ([], <*ns*::*nat list*>) ⦄ *tm* ⦃ λ*tap*. (∃ *k n l*. *tap* = (*Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)) ⦄
shows ∃ *stp k n l*. (*steps0* (*1*, ([], <*ns*::*nat list*>)) *tm stp*) = (*0*, *Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*) ∧
fun_of_tm tm ns = *Some* (*n*::*nat*)
⟨*proof*⟩

lemma *fun_of_tm_ex_Some_n_rev*:
assumes ∃ *stp k n l*. (*steps0* (*1*, ([], <*ns*::*nat list*>)) *tm stp*) = (*0*, *Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)

$\wedge$
$\quad fun\_of\_tm\ tm\ ns = Some\ n$
shows $\{\!|\ \lambda tap.\ tap = ([], <ns::nat\ list>)\ |\!\}\ tm\ \{\!|\ \lambda tap.\ (\exists\, k\ n\ l.\ tap = (Bk \uparrow k, <n::nat> @\ Bk \uparrow l))\ |\!\}$
$\langle proof \rangle$

corollary *fun_of_tm_ex_Some_n_iff*:
$(\exists\, stp\ k\ n\ l.\ (steps0\ (1, ([], <ns>))\ tm\ stp) = (0, Bk \uparrow k, <n::nat> @\ Bk \uparrow l) \wedge$
$\quad fun\_of\_tm\ tm\ ns = Some\ n)$
$\longleftrightarrow$
$\{\!|\ \lambda tap.\ tap = ([], <ns::nat\ list>)\ |\!\}\ tm\ \{\!|\ \lambda tap.\ (\exists\, k\ n\ l.\ tap = (Bk \uparrow k, <n::nat> @\ Bk \uparrow l))\ |\!\}$
$\langle proof \rangle$

lemma *fun_of_tm_eq_Some_n_imp_same_numeral_result*:
assumes $fun\_of\_tm\ tm\ ns = Some\ n$
shows $\exists\, stp\ k\ l.\ steps0\ (1, [], <ns::nat\ list>)\ tm\ stp = (0, Bk \uparrow k, <n::nat> @\ Bk \uparrow l)$
$\langle proof \rangle$

lemma *numeral_result_n_imp_fun_of_tm_eq_n*:
assumes $\exists\, stp\ k\ l.\ steps0\ (1, [], <ns::nat\ list>)\ tm\ stp = (0, Bk \uparrow k, <n::nat> @\ Bk \uparrow l)$
shows $fun\_of\_tm\ tm\ ns = Some\ n$
$\langle proof \rangle$

corollary *numeral_result_n_iff_fun_of_tm_eq_n*:
$fun\_of\_tm\ tm\ ns = Some\ n$
$\longleftrightarrow$
$(\exists\, stp\ k\ l.\ steps0\ (1, [], <ns::nat\ list>)\ tm\ stp = (0, Bk \uparrow k, <n::nat> @\ Bk \uparrow l))$
$\langle proof \rangle$

corollary *numeral_result_n_iff_fun_of_tm_eq_n'*:
$fun\_of\_tm\ tm\ ns = Some\ n \longleftrightarrow TMC\_yields\_num\_res\ tm\ ns\ n$
$\langle proof \rangle$

1.14.1.7 Definition of Turing computability Variant 2

definition $turing\_computable\_partial' :: (nat\ list \Rightarrow nat\ option) \Rightarrow bool$
where $turing\_computable\_partial'\ f \stackrel{def}{=} \exists\, tm.\ fun\_of\_tm\ tm = f$

lemma *turing_computable_partial'I*:
$(\bigwedge ns.\ fun\_of\_tm\ tm\ ns = f\ ns) \Longrightarrow turing\_computable\_partial'\ f$
$\langle proof \rangle$

1.14.1.8 Definitional Variants 1 and 2 of Partial Turing Computability are equivalent

Now, we prove the equivalence of the two definitions of Partial Turing Computability.

lemma *turing_computable_partial′_imp_turing_computable_partial*:
$\mathit{turing\_computable\_partial}'\ f \longrightarrow \mathit{turing\_computable\_partial}\ f$
$\langle \mathit{proof} \rangle$

lemma *turing_computable_partial_imp_turing_computable_partial′*:
$\mathit{turing\_computable\_partial}\ f \longrightarrow \mathit{turing\_computable\_partial}'\ f$
$\langle \mathit{proof} \rangle$

corollary *turing_computable_partial′_turing_computable_partial_iff*:
$\mathit{turing\_computable\_partial}'\ f \longleftrightarrow \mathit{turing\_computable\_partial}\ f$
$\langle \mathit{proof} \rangle$

As a now trivial consequence we obtain:

corollary $\mathit{turing\_computable\_partial}\ f \stackrel{\mathit{def}}{=} \exists\, \mathit{tm}.\ \mathit{fun\_of\_tm}\ \mathit{tm} = f$
$\langle \mathit{proof} \rangle$

1.14.2 Definition of Total Turing Computability

definition $\mathit{turing\_computable\_total} :: (\mathit{nat\ list} \Rightarrow \mathit{nat\ option}) \Rightarrow \mathit{bool}$
where $\mathit{turing\_computable\_total}\ f \stackrel{\mathit{def}}{=} (\exists\, \mathit{tm}.\ \forall\, \mathit{ns}.\ \exists\, n.$
$f\ \mathit{ns} = \mathit{Some}\ n\ \wedge$
$(\exists\, \mathit{stp}\ k\ l.\ (\mathit{steps0}\ (1,\ ([],\ <\mathit{ns}::\mathit{nat\ list}>))\ \mathit{tm}\ \mathit{stp}) = (0,\ \mathit{Bk} \uparrow k,\ <n::\mathit{nat}>\ @\ \mathit{Bk} \uparrow l))\ )$

lemma *turing_computable_total_unfolded_into_TMC_yields_condition*:
$\mathit{turing\_computable\_total}\ f \stackrel{\mathit{def}}{=} (\exists\, \mathit{tm}.\ \forall\, \mathit{ns}.\ \exists\, n.\ f\ \mathit{ns} = \mathit{Some}\ n \wedge \mathit{TMC\_yields\_num\_res}\ \mathit{tm}\ \mathit{ns}\ n$
$)$
$\langle \mathit{proof} \rangle$

lemma *turing_computable_total_imp_turing_computable_partial*:
$\mathit{turing\_computable\_total}\ f \Longrightarrow \mathit{turing\_computable\_partial}\ f$
$\langle \mathit{proof} \rangle$

corollary *turing_decidable_imp_turing_computable_total_chi_fun*:
$\mathit{turing\_decidable}\ A \Longrightarrow \mathit{turing\_computable\_total}\ (\mathit{chi\_fun}\ A)$
$\langle \mathit{proof} \rangle$

definition $\mathit{turing\_computable\_total}' :: (\mathit{nat\ list} \Rightarrow \mathit{nat\ option}) \Rightarrow \mathit{bool}$
where $\mathit{turing\_computable\_total}'\ f \stackrel{\mathit{def}}{=} (\exists\, \mathit{tm}.\ \forall\, \mathit{ns}.\ \exists\, n.\ f\ \mathit{ns} = \mathit{Some}\ n \wedge \mathit{fun\_of\_tm}\ \mathit{tm} = f)$

theorem *turing_computable_total′_eq_turing_computable_total*:
$\mathit{turing\_computable\_total}'\ f = \mathit{turing\_computable\_total}\ f$
$\langle \mathit{proof} \rangle$

definition *turing_computable_total''* :: (*nat list* ⇒ *nat option*) ⇒ *bool*
where *turing_computable_total'' f* $\stackrel{def}{=}$ (∃ *tm*. *fun_of_tm tm* = *f* ∧ (∀ *ns*. ∃ *n*. *f ns* = *Some n*))

theorem *turing_computable_total''_eq_turing_computable_total*:
turing_computable_total'' f = *turing_computable_total f*
⟨*proof*⟩

definition *turing_computable_total_on* :: (*nat list* ⇒ *nat option*) ⇒ (*nat list*) *set* ⇒ *bool*
where *turing_computable_total_on f A* $\stackrel{def}{=}$ (∃ *tm*. ∀ *ns*.
ns ∈ *A* ⟶
(∃ *n*. *f ns* = *Some n* ∧
(∃ *stp k l*. (*steps0* (*1*, ([], <*ns*::*nat list*>)) *tm stp*) = (*0*, *Bk* ↑ *k*, <*n*::*nat*> @ *Bk* ↑ *l*)))
)

lemma *turing_computable_total_on_unfolded_into_TMC_yields_condition*:
turing_computable_total_on f A $\stackrel{def}{=}$ (∃ *tm*. ∀ *ns*. *ns* ∈ *A* ⟶ (∃ *n*. *f ns* = *Some n* ∧ *TMC_yields_num_res tm ns n*))
⟨*proof*⟩

lemma *turing_computable_total_on_UNIV_imp_turing_computable_total*:
turing_computable_total_on f UNIV ⟹ *turing_computable_total f*
⟨*proof*⟩

end

1.15 A Variation of the theme due to Boolos, Burgess and, Jeffrey

In sections 1.13.2.2 and 1.13.2.5 we discussed two variants of the proof of the undecidability of the Sepcial Halting Problem. There, we used the Turing Machines *tm_semi_id_eq0* and *tm_semi_id_gt0* for the construction a contradiction.

The machine *tm_semi_id_gt0* is identical to the machine *dither*, which is discussed in length together with the Turing Machine *copy* in the book by Boolos, Burgess, and Jeffrey [1].

For backwards compatibility with the original AFP entry, we again present the formalization of the machines*dither* and *copy* here in this section. This allows for reuse of theory CopyTM, which in turn is referenced in the original proof about the existence of an uncomputable function in theory TuringUnComputable_H2_original.

In addition we present an enhanced version in theory TuringUnComputable_H2, which is in line with the principles of Conservative Extension.

1.15.1 The Dithering Turing Machine

If the input is empty or the numeral $< 0 >$, the *Dithering* TM will loop forever, otherwise it will terminate.

theory *DitherTM*
 imports *Turing_Hoare*
begin

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

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

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

lemma *composable_tm0_tm_dither*[*intro*, *simp*]: *composable_tm0 tm_dither*
 ⟨*proof*⟩

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

lemma *tm_dither_loops_aux′*:
 (*steps0* (*1*, *Bk* ↑ *m*, [*Oc*] @ *Bk* ↑ *n*) *tm_dither stp* = (*1*, *Bk* ↑ *m*, [*Oc*] @ *Bk* ↑ *n*)) ∨
  (*steps0* (*1*, *Bk* ↑ *m*, [*Oc*] @ *Bk* ↑ *n*) *tm_dither stp* = (*2*, *Oc* # *Bk* ↑ *m*, *Bk* ↑ *n*))
 ⟨*proof*⟩

If the input is *Oc* ↑ *1* the *Dithering* TM will loop forever, for other non-blank inputs *Oc* ↑ (*n* + *1*) with *1* < *n* it will reach the final state in a standard configuration.

Please note that our short notation $<n>$ means *Oc* ↑ (*n* + *1*) where *0* ≤ *n*.

lemma $<0::nat>$ = [*Oc*] ⟨*proof*⟩

lemma $Oc\uparrow(0{+}1) = [Oc]$ ⟨*proof*⟩
lemma $<n::nat> = Oc\uparrow(n{+}1)$ ⟨*proof*⟩

lemma $<1::nat> = [Oc, Oc]$ ⟨*proof*⟩

1.15.1.1 Dither in action.

lemma *steps0* $(1, [], [Oc])$ *tm_dither* $0 = (1, [], [Oc])$ ⟨*proof*⟩
lemma *steps0* $(1, [], [Oc])$ *tm_dither* $1 = (2, [Oc], [])$ ⟨*proof*⟩
lemma *steps0* $(1, [], [Oc])$ *tm_dither* $2 = (1, [], [Oc])$ ⟨*proof*⟩
lemma *steps0* $(1, [], [Oc])$ *tm_dither* $3 = (2, [Oc], [])$ ⟨*proof*⟩
lemma *steps0* $(1, [], [Oc])$ *tm_dither* $4 = (1, [], [Oc])$ ⟨*proof*⟩

lemma *steps0* $(1, [], [Oc, Oc])$ *tm_dither* $0 = (1, [], [Oc, Oc])$ ⟨*proof*⟩
lemma *steps0* $(1, [], [Oc, Oc])$ *tm_dither* $1 = (2, [Oc], [Oc])$ ⟨*proof*⟩
lemma *steps0* $(1, [], [Oc, Oc])$ *tm_dither* $2 = (0, [], [Oc, Oc])$ ⟨*proof*⟩
lemma *steps0* $(1, [], [Oc, Oc])$ *tm_dither* $3 = (0, [], [Oc, Oc])$ ⟨*proof*⟩

lemma *steps0* $(1, [], [Oc, Oc, Oc])$ *tm_dither* $0 = (1, [], [Oc, Oc, Oc])$ ⟨*proof*⟩
lemma *steps0* $(1, [], [Oc, Oc, Oc])$ *tm_dither* $1 = (2, [Oc], [Oc, Oc])$ ⟨*proof*⟩
lemma *steps0* $(1, [], [Oc, Oc, Oc])$ *tm_dither* $2 = (0, [], [Oc, Oc, Oc])$ ⟨*proof*⟩
lemma *steps0* $(1, [], [Oc, Oc, Oc])$ *tm_dither* $3 = (0, [], [Oc, Oc, Oc])$ ⟨*proof*⟩

1.15.1.2 Proving properties of tm_dither with Hoare rules

Using Hoare style rules is more elegant since they allow for compositional reasoning. Therefore, its preferable to use them, if the program that we reason about can be decomposed appropriately.

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

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

lemma $<0::nat> = [Oc]$ ⟨*proof*⟩

lemma *tm_dither_loops*:
shows $\{\!|tm\_dither\_unhalt\_ass|\!\}\ tm\_dither \uparrow$
⟨*proof*⟩

lemma *tm_dither_loops''*:
shows $\{\!|\lambda tap.\ \exists k\ l.\ tap = (Bk\uparrow k, [Oc]\ @\ Bk\uparrow l)|\!\}\ tm\_dither \uparrow$
⟨*proof*⟩

lemma *tm_dither_halts_aux*:
shows *steps0* $(1, Bk \uparrow m, [Oc, Oc])$ *tm_dither* $2 = (0, Bk \uparrow m, [Oc, Oc])$

⟨*proof*⟩

lemma *tm_dither_halts_aux′*:
shows *steps0* $(1, Bk \uparrow m, [Oc, Oc]@Bk \uparrow n)$ *tm_dither* $2 = (0, Bk \uparrow m, [Oc, Oc]@Bk \uparrow n)$
⟨*proof*⟩

lemma *tm_dither_halts*:
shows $\{\!|$*tm_dither_halt_inv*$|\!\}$ *tm_dither* $\{\!|$*tm_dither_halt_inv*$|\!\}$
⟨*proof*⟩

lemma *tm_dither_halts″*:
shows $\{\!|\ \lambda tap.\ \exists k\ l.\ tap = (Bk\uparrow k, [Oc, Oc] @ Bk\uparrow l)|\!\}$ *tm_dither* $\{\!|\lambda tap.\ \exists k\ l.\ tap = (Bk\uparrow k, [Oc, Oc] @ Bk\uparrow l)|\!\}$
⟨*proof*⟩

end

1.15.2 A Turing machine that just duplicates its input if the input is a single numeral

The machine tm_copy is almost identical to the machine tm_weak_copy that we presented in theory WeakCopyTM. They only differ in the first instruction of component tm_copy_end (compare tm_copy_end_orig and tm_copy_end_new in theory WeakCopyTM).

As for machine tm_dither, we keep the entire theory CopyTM for backwards compatibility with the original AFP entry.

theory *CopyTM*
imports
Turing_Hoare
Turing_HaltingConditions
begin

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

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

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

definition
tm_copy_end :: *instr list*
where
$tm\_copy\_end \stackrel{def}{=} [(L, 0), (R, 2), (WO, 3), (L, 4),$
$(R, 2), (R, 2), (L, 5), (WB, 4),$
$(R, 0), (L, 5)]$

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

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
$inv\_begin\ n\ (s, tap) =$
$(if\ s = 0\ then\ inv\_begin0\ n\ tap\ else$
$if\ s = 1\ then\ inv\_begin1\ n\ tap\ else$
$if\ s = 2\ then\ inv\_begin2\ n\ tap\ else$
$if\ s = 3\ then\ inv\_begin3\ n\ tap\ else$
$if\ s = 4\ then\ inv\_begin4\ n\ tap$
$else\ False)$

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 tm_copy_begin)*
⟨*proof*⟩

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

lemma *begin_partial_correctness*:
  assumes *is_final* (*steps0* (*1*, [], *Oc* ↑ *n*) *tm_copy_begin stp*)
  shows $0 < n \Longrightarrow$ ⦃*inv_begin1 n*⦄ *tm_copy_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* = [] ∨ *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*]:
  ⟦⋀*n*. ¬ *P* (*f n*) ⟹ (*f* (*Suc n*), (*f n*)) ∈ *measure_begin*⟧ ⟹ ∃ *n*. *P* (*f n*)
  ⟨*proof*⟩

lemma *begin_halts*:
  assumes *h*: $x > 0$
  shows ∃ *stp*. *is_final* (*steps0* (*1*, [], *Oc* ↑ *x*) *tm_copy_begin stp*)
⟨*proof*⟩

lemma *begin_correct*:
  shows $0 < n \Longrightarrow$ ⦃*inv_begin1 n*⦄ *tm_copy_begin* ⦃*inv_begin0 n*⦄
  ⟨*proof*⟩

declare *seq_tm*.*simps* [*simp del*]
declare *shift*.*simps*[*simp del*]
declare *composable_tm*.*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*) = (∃ *i j*. *i* + *j* + *1* = *n* ∧ (*l*, *r*) = (*Oc*↑*i*, *Oc*#*Oc*#*Bk*↑*j* @ *Oc*↑*j*) ∧ *j* > *0*)
| *inv_loop1_exit n* (*l*, *r*) = (*0* < *n* ∧ (*l*, *r*) = ([], *Bk*#*Oc*#*Bk*↑*n* @ *Oc*↑*n*))
| *inv_loop5_loop x* (*l*, *r*) =
(∃ *i j k t*. *i* + *j* = *Suc x* ∧ *i* > *0* ∧ *j* > *0* ∧ *k* + *t* = *j* ∧ *t* > *0* ∧ (*l*, *r*) = (*Oc*↑*k*@*Bk*↑*j*@*Oc*↑*i*, *Oc*↑*t*))
| *inv_loop5_exit x* (*l*, *r*) =
(∃ *i j*. *i* + *j* = *Suc x* ∧ *i* > *0* ∧ *j* > *0* ∧ (*l*, *r*) = (*Bk*↑(*j* − *1*)@*Oc*↑*i*, *Bk* # *Oc*↑*j*))
| *inv_loop6_loop x* (*l*, *r*) =
(∃ *i j k t*. *i* + *j* = *Suc x* ∧ *i* > *0* ∧ *k* + *t* + *1* = *j* ∧ (*l*, *r*) = (*Bk*↑*k* @ *Oc*↑*i*, *Bk*↑(*Suc t*) @ *Oc*↑*j*))
| *inv_loop6_exit x* (*l*, *r*) =
(∃ *i j*. *i* + *j* = *x* ∧ *j* > *0* ∧ (*l*, *r*) = (*Oc*↑*i*, *Oc*#*Bk*↑*j* @ *Oc*↑*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* ∧ (*l*, *r*) = ([*Bk*], *Oc* # *Bk*↑*n* @ *Oc*↑*n*))
| *inv_loop1 n* (*l*, *r*) = (*inv_loop1_loop n* (*l*, *r*) ∨ *inv_loop1_exit n* (*l*, *r*))
| *inv_loop2 n* (*l*, *r*) = (∃ *i j any*. *i* + *j* = *n* ∧ *n* > *0* ∧ *i* > *0* ∧ *j* > *0* ∧ (*l*, *r*) = (*Oc*↑*i*, *any*#*Bk*↑*j*@*Oc*↑*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 *inv_loop3_Bk_empty_via_2*[*elim*]: $[\![0 < x;$ *inv_loop2 x* $(b, [\,])]\!] \Longrightarrow$ *inv_loop3 x* $(Bk \# b, [\,])$
⟨*proof*⟩

lemma *inv_loop3_Bk_empty*[*elim*]: $[\![0 < x;$ *inv_loop3 x* $(b, [\,])]\!] \Longrightarrow$ *inv_loop3 x* $(Bk \# b, [\,])$
⟨*proof*⟩

lemma *inv_loop5_Oc_empty_via_4*[*elim*]: $[\![0 < x;$ *inv_loop4 x* $(b, [\,])]\!] \Longrightarrow$ *inv_loop5 x* $(b, [Oc])$
⟨*proof*⟩

lemma *inv_loop1_Bk*[*elim*]: $[\![0 < x;$ *inv_loop1 x* $(b, Bk \# list)]\!] \Longrightarrow list = Oc \# Bk \uparrow x \mathbin{@} Oc \uparrow x$
⟨*proof*⟩

lemma *inv_loop3_Bk_via_2*[*elim*]: $[\![0 < x;$ *inv_loop2 x* $(b, Bk \# list)]\!] \Longrightarrow$ *inv_loop3 x* $(Bk \# b, list)$
⟨*proof*⟩

lemma *inv_loop3_Bk_move*[*elim*]: $[\![0 < x;$ *inv_loop3 x* $(b, Bk \# list)]\!] \Longrightarrow$ *inv_loop3 x* $(Bk \# b, list)$
⟨*proof*⟩

lemma *inv_loop5_Oc_via_4_Bk*[*elim*]: $[\![0 < x;$ *inv_loop4 x* $(b, Bk \# list)]\!] \Longrightarrow$ *inv_loop5 x* $(b, Oc \# list)$
⟨*proof*⟩

lemma *inv_loop6_Bk_via_5*[*elim*]: $[\![0 < x;$ *inv_loop5 x* $([\,], Bk \# list)]\!] \Longrightarrow$ *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]*: $[\![0 < x;\ inv\_loop5\_exit\ x\ (b, Bk \# list);\ b \neq [\,];\ hd\ b = Bk]\!]$
$\Longrightarrow inv\_loop6\_loop\ x\ (tl\ b, Bk \# Bk \# list)$
⟨*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]*: $[\![x > 0;\ inv\_loop5\_exit\ x\ (b, Bk \# list);\ b \neq [\,];\ hd\ b = Oc]\!] \Longrightarrow$
$inv\_loop6\_exit\ x\ (tl\ b, Oc \# Bk \# list)$
⟨*proof*⟩

lemma *inv_loop6_Bk_tail_via_5[elim]*: $[\![0 < x;\ inv\_loop5\ x\ (b, Bk \# list);\ b \neq [\,]]\!] \Longrightarrow inv\_loop6\ x\ (tl\ b, hd\ b \# Bk \# list)$
⟨*proof*⟩

lemma *inv_loop6_loop_Bk_Bk_drop[elim]*: $[\![0 < x;\ inv\_loop6\_loop\ x\ (b, Bk \# list);\ b \neq [\,];\ hd\ b = Bk]\!]$
$\Longrightarrow inv\_loop6\_loop\ x\ (tl\ b, Bk \# Bk \# list)$
⟨*proof*⟩

lemma *inv_loop6_exit_Oc_Bk_via_loop6[elim]*: $[\![0 < x;\ inv\_loop6\_loop\ x\ (b, Bk \# list);\ b \neq [\,];\ hd\ b = Oc]\!]$
$\Longrightarrow inv\_loop6\_exit\ x\ (tl\ b, Oc \# Bk \# list)$
⟨*proof*⟩

lemma *inv_loop6_Bk_tail[elim]*: $[\![0 < x;\ inv\_loop6\ x\ (b, Bk \# list);\ b \neq [\,]]\!] \Longrightarrow inv\_loop6\ x\ (tl\ b, hd\ b \# Bk \# list)$
⟨*proof*⟩

lemma *inv_loop2_Oc_via_1[elim]*: $[\![0 < x;\ inv\_loop1\ x\ (b, Oc \# list)]\!] \Longrightarrow inv\_loop2\ x\ (Oc \# b, list)$
⟨*proof*⟩

lemma *inv_loop2_Bk_via_Oc[elim]*: $[\![0 < x;\ inv\_loop2\ x\ (b, Oc \# list)]\!] \Longrightarrow inv\_loop2\ x\ (b, Bk \# list)$
⟨*proof*⟩

lemma *inv_loop4_Oc_via_3[elim]*: $[\![0 < x;\ inv\_loop3\ x\ (b, Oc \# list)]\!] \Longrightarrow inv\_loop4\ x\ (Oc \# b, list)$
⟨*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* (*tm_copy_loop*, *0*))
⟨*proof*⟩

lemma *inv_loop_steps*:
$[\![inv\_loop\ x\ cf;\ x > 0]\!] \Longrightarrow inv\_loop\ x\ (steps\ cf\ (tm\_copy\_loop, 0)\ stp)$
$\langle proof \rangle$

fun *loop_stage* :: *config* ⇒ *nat*
where
$loop\_stage\ (s, l, r) = (if\ s = 0\ then\ 0$
$\quad else\ (Suc\ (length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l\ @\ r)))))$

fun *loop_state* :: *config* ⇒ *nat*
where
$loop\_state\ (s, l, r) = (if\ s = 2 \wedge hd\ r = Oc\ then\ 0$
$\quad else\ if\ s = 1\ then\ 1$
$\quad else\ 10 - s)$

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

definition *measure_loop* :: (*config* × *config*) *set*
where
$measure\_loop = measures\ [loop\_stage, loop\_state, loop\_step]$

lemma *wf_measure_loop*: *wf measure_loop*
$\langle proof \rangle$

lemma *measure_loop_induct* [*case_names Step*]:
$[\![\bigwedge n.\ \neg\ P\ (f\ n) \Longrightarrow (f\ (Suc\ n), (f\ n)) \in measure\_loop]\!] \Longrightarrow \exists\, n.\ P\ (f\ n)$
$\langle proof \rangle$

lemma *inv_loop4_not_just_Oc*[*elim*]:
$[\![inv\_loop4\ x\ (l', []);$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ [Oc])) \neq$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'))]\!]$
$\Longrightarrow RR$
$[\![inv\_loop4\ x\ (l', Bk\ \#\ list);$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ Oc\ \#\ list)) \neq$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ Bk\ \#\ list))]\!]$
$\Longrightarrow RR$
$\langle proof \rangle$

lemma *takeWhile_replicate_append*:
$P\ a \Longrightarrow takeWhile\ P\ (a\uparrow x\ @\ ys) = a\uparrow x\ @\ takeWhile\ P\ ys$
$\langle proof \rangle$

lemma *takeWhile_replicate*:

$P\ a \Longrightarrow takeWhile\ P\ (a\uparrow x) = a\uparrow x$
⟨*proof*⟩

lemma *inv_loop5_Bk_E*[*elim*]:
$[\![inv_loop5\ x\ (l', Bk\ \#\ list); l' \neq []; $
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ (tl\ l')\ @\ hd\ l'\ \#\ Bk\ \#\ list)) \neq$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ Bk\ \#\ list))]\!]$
$\Longrightarrow RR$
⟨*proof*⟩

lemma *inv_loop1_hd_Oc*[*elim*]: $[\![inv\_loop1\ x\ (l', Oc\ \#\ list)]\!] \Longrightarrow hd\ list = Oc$
⟨*proof*⟩

lemma *inv_loop6_not_just_Bk*[*dest*!]:
$[\![length\ (takeWhile\ P\ (rev\ (tl\ l')\ @\ hd\ l'\ \#\ list)) \neq$
$length\ (takeWhile\ P\ (rev\ l'\ @\ list))]\!]$
$\Longrightarrow l' = []$
⟨*proof*⟩

lemma *inv_loop2_OcE*[*elim*]:
$[\![inv\_loop2\ x\ (l', Oc\ \#\ list); l' \neq []]\!] \Longrightarrow$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ Bk\ \#\ list)) <$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ Oc\ \#\ list))$
⟨*proof*⟩

lemma *loop_halts*:
assumes h: $n > 0\ inv\_loop\ n\ (1, l, r)$
shows $\exists\ stp.\ is\_final\ (steps0\ (1, l, r)\ tm\_copy\_loop\ stp)$
⟨*proof*⟩

lemma *loop_correct*:
assumes $0 < n$
shows $\{\!|inv\_loop1\ n|\!\}\ tm\_copy\_loop\ \{\!|inv\_loop0\ n|\!\}$
⟨*proof*⟩

fun
inv_end5_loop :: $nat \Rightarrow tape \Rightarrow bool$ **and**
inv_end5_exit :: $nat \Rightarrow tape \Rightarrow bool$
where
$inv\_end5\_loop\ x\ (l, r) =$
$(\exists\ i\ j.\ i + j = x \wedge x > 0 \wedge j > 0 \wedge l = Oc\uparrow i\ @\ [Bk] \wedge r = Oc\uparrow j\ @\ Bk\ \#\ Oc\uparrow x)$
$|\ inv\_end5\_exit\ x\ (l, r) = (x > 0 \wedge l = [] \wedge r = Bk\ \#\ Oc\uparrow x\ @\ Bk\ \#\ Oc\uparrow x)$

fun
inv_end0 :: $nat \Rightarrow tape \Rightarrow bool$ **and**
inv_end1 :: $nat \Rightarrow tape \Rightarrow bool$ **and**
inv_end2 :: $nat \Rightarrow tape \Rightarrow bool$ **and**
inv_end3 :: $nat \Rightarrow tape \Rightarrow bool$ **and**

$inv\_end4 :: nat \Rightarrow tape \Rightarrow bool$ **and**
$inv\_end5 :: nat \Rightarrow tape \Rightarrow bool$
where
$inv\_end0\ n\ (l, r) = (n > 0 \wedge (l, r) = ([Bk], Oc\uparrow n\ @\ Bk\ \#\ Oc\uparrow n))$
$|\ inv\_end1\ n\ (l, r) = (n > 0 \wedge (l, r) = ([Bk], Oc\ \#\ Bk\uparrow n\ @\ Oc\uparrow n))$
$|\ inv\_end2\ n\ (l, r) = (\exists\ i\ j.\ i + j = Suc\ n \wedge n > 0 \wedge l = Oc\uparrow i\ @\ [Bk] \wedge r = Bk\uparrow j\ @\ Oc\uparrow n)$
$|\ inv\_end3\ n\ (l, r) =$
$(\exists\ i\ j.\ n > 0 \wedge i + j = n \wedge l = Oc\uparrow i\ @\ [Bk] \wedge r = Oc\ \#\ Bk\uparrow j @\ Oc\uparrow n)$
$|\ inv\_end4\ n\ (l, r) = (\exists\ any.\ n > 0 \wedge l = Oc\uparrow n\ @\ [Bk] \wedge r = any\#Oc\uparrow n)$
$|\ inv\_end5\ n\ (l, r) = (inv\_end5\_loop\ n\ (l, r) \vee inv\_end5\_exit\ n\ (l, r))$

fun
$inv\_end :: nat \Rightarrow config \Rightarrow bool$
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*]:
$inv\_end1\ x\ (b, []) = False$
$inv\_end1\ x\ ([], list) = False$
$inv\_end2\ x\ (b, []) = False$
$inv\_end3\ x\ (b, []) = False$
$inv\_end4\ x\ (b, []) = False$
$inv\_end5\ x\ (b, []) = False$
$inv\_end5\ x\ ([], Oc\ \#\ list) = False$
$\langle proof \rangle$

lemma *inv_end0_Bk_via_1*[*elim*]: $[\![0 < x; inv\_end1\ x\ (b, Bk\ \#\ list); b \neq []]\!]$
$\Longrightarrow inv\_end0\ x\ (tl\ b, hd\ b\ \#\ Bk\ \#\ list)$
$\langle proof \rangle$

lemma *inv_end3_Oc_via_2*[*elim*]: $[\![0 < x; inv\_end2\ x\ (b, Bk\ \#\ list)]\!]$
$\Longrightarrow inv\_end3\ x\ (b, Oc\ \#\ list)$
$\langle proof \rangle$

lemma *inv_end2_Bk_via_3*[*elim*]: $[\![0 < x; inv\_end3\ x\ (b, Bk\ \#\ list)]\!] \Longrightarrow inv\_end2\ x\ (Bk\ \#\ b,$
$list)$
$\langle proof \rangle$

lemma *inv_end5_Bk_via_4*[*elim*]: $[\![0 < x; inv\_end4\ x\ ([], Bk\ \#\ list)]\!] \Longrightarrow$

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*]:

$[\![0 < x;\ inv\_end5\_loop\ x\ (b, Oc \# list);\ b \neq [];\ hd\ b = Oc]\!] \Longrightarrow$
$inv\_end5\_loop\ x\ (tl\ b, Oc \# Oc \# list)$
$\langle proof \rangle$

lemma *inv_end5_Oc_tail*[*elim*]: $[\![0 < x;\ inv\_end5\ x\ (b, Oc \# list);\ b \neq []]\!] \Longrightarrow$
$inv\_end5\ x\ (tl\ b, hd\ b \# Oc \# list)$
$\langle proof \rangle$

lemma *inv_end_step*:
$[\![x > 0;\ inv\_end\ x\ cf]\!] \Longrightarrow inv\_end\ x\ (step\ cf\ (tm\_copy\_end, 0))$
$\langle proof \rangle$

lemma *inv_end_steps*:
$[\![x > 0;\ inv\_end\ x\ cf]\!] \Longrightarrow inv\_end\ x\ (steps\ cf\ (tm\_copy\_end, 0)\ stp)$
$\langle proof \rangle$

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

fun *end_stage* :: *config* ⇒ *nat*
where
$end\_stage\ (s, l, r) =$
$(if\ s = 2 \lor s = 3\ then\ (length\ r)\ else\ 0)$

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*
$\langle proof \rangle$

lemma *end_halt*:
$[\![x > 0;\ inv\_end\ x\ (Suc\ 0, l, r)]\!] \Longrightarrow$
$\exists\ stp.\ is\_final\ (steps\ (Suc\ 0, l, r)\ (tm\_copy\_end, 0)\ stp)$

⟨*proof*⟩

lemma *end_correct*:
  $n > 0 \Longrightarrow$ ⦃*inv_end1 n*⦄ *tm_copy_end* ⦃*inv_end0 n*⦄
⟨*proof*⟩

lemma [*intro*]:
  composable_tm (*tm_copy_begin*, *0*)
  composable_tm (*tm_copy_loop*, *0*)
  composable_tm (*tm_copy_end*, *0*)
  ⟨*proof*⟩

lemma *composable_tm0_tm_copy*[*intro*, *simp*]: *composable_tm0 tm_copy*
  ⟨*proof*⟩

lemma *tm_copy_correct1*:
  assumes $0 < x$
  shows ⦃*inv_begin1 x*⦄ *tm_copy* ⦃*inv_end0 x*⦄
⟨*proof*⟩

abbreviation (*input*)
  pre_tm_copy n $\stackrel{def}{=}$ λ*tap*. *tap* = ([]::*cell list*, *Oc* ↑ (*Suc n*))
abbreviation (*input*)
  post_tm_copy n $\stackrel{def}{=}$ λ*tap*. *tap*= ([*Bk*], <(*n*, *n*::*nat*)>)

lemma *tm_copy_correct*:
  shows ⦃*pre_tm_copy n*⦄ *tm_copy* ⦃*post_tm_copy n*⦄
⟨*proof*⟩

end

1.15.3 Existence of an uncomputable Function

theory *TuringUnComputable_H2*
  imports
    CopyTM
    DitherTM

begin

1.15.3.1 Undecidability of the General Halting Problem H, Variant 2, revised version

This variant of the decision problem H is discussed in the book Computability and Logic by Boolos, Burgess and Jeffrey [1] in chapter 4.

The proof makes use of the TMs *tm_copy* and *tm_dither*. In [1], the machines are called *copy* and *dither*.

fun *dummy_code* :: *tprog0* $\Rightarrow$ *nat*
where *dummy_code tp* = *0*

locale *hph2* =

fixes *code* :: *instr list* $\Rightarrow$ *nat*

begin

The function *dummy_code* is a witness that the locale hph2 is inhabited.

Note: there just has to be some function with the correct type since we did not specify any axioms for the locale. The behaviour of the instance of the locale function code does not matter at all.

This detail differs from the locale hpk, where a locale axiom specifies that the coding function has to be injective.

Obviously, the entire logical argument of the undecidability proof H2 relies on the combination of the machines *tm_copy* and *tm_dither*.

interpretation *dummy_code*: *hph2 dummy_code* :: *tprog0* $\Rightarrow$ *nat*
$\langle proof \rangle$

The next lemma plays a crucial role in the proof by contradiction. Due to our general results about trailing blanks on the left tape, we are able to compensate for the additional blank, which is a mandatory by-product of the *tm_copy*.

lemma *add_single_BK_to_left_tape*:
$\{\!|\lambda tap.\ tap = ([]\ ,\ <(m{::}nat,\ m)>\ )\ |\!\}\ p\ \{\!|\lambda tap.\ \exists\, k\ l.\ tap = (Bk \uparrow k,\ r'\ @Bk{\uparrow}l\ )|\!\}$
$\Longrightarrow$
$\{\!|\lambda tap.\ tap = ([Bk],\ <(m\quad ,\ m)>\ )\ |\!\}\ p\ \{\!|\lambda tap.\ \exists\, k\ l.\ tap = (Bk \uparrow k,\ r'\ @Bk{\uparrow}l\ )|\!\}$
$\langle proof \rangle$

Definition of the General Halting Problem H2.

definition *H2* :: ((*instr list*) $\times$ (*nat list*)) *set*
where
$H2 \stackrel{def}{=} \{(tm,nl).\ TMC\_has\_num\_res\ tm\ nl\ \}$

No Turing Machine is able to decide the General Halting Problem H2.

lemma *existence_of_decider_H2D0_for_H2_imp_False*:
assumes $\exists\, H2D0'.\ (\forall\, nl\ (tm{::}instr\ list).$

$((tm,nl) \in H2 \longrightarrow \{\!|\lambda tap.\ tap = ([], <(code\ tm,\ nl)>)|\!\}\ H2D0'\ \{\!|\lambda tap.\ \exists\, k\ l.\ tap = (Bk \uparrow k,\ [Oc]\ @\ Bk\uparrow l)|\!\})$
$\wedge\ ((tm,nl) \notin H2 \longrightarrow \{\!|\lambda tap.\ tap = ([], <(code\ tm,\ nl)>)|\!\}\ H2D0'\ \{\!|\lambda tap.\ \exists\, k\ l.\ tap = (Bk \uparrow k,\ [Oc,\ Oc]\ @Bk\uparrow l)|\!\})\ )$
shows *False*
⟨*proof*⟩

Note: since we did not formalize the concept of Turing Computable Functions and Characteristic Functions of sets yet, we are (at the moment) not able to formalize the existence of an uncomputable function, namely the characteristic function of the set H2.

Another caveat is the fact that the set H2 has type (*instr list* × *nat list*) *set*. This is in contrast to the classical formalization of decision problems, where the sets discussed only contain tuples respectively lists of natural numbers.

end

end

1.15.3.2 Undecidability of the General Halting Problem H, Variant 2, original version

theory *TuringUnComputable_H2_original*
imports
DitherTM
CopyTM

begin

The diagonal argument below shows the undecidability of a variant of the General Halting Problem. Implicitly, we thus show that the General Halting Function (the characteristic function of the Halting Problem) is not Turing computable.

The following locale specifies that some TM *H* can be used to decide the *General Halting Problem* and *False* is going to be derived under this locale. Therefore, the undecidability of the *General Halting Problem* is established.

The proof makes use of the TMs *tm_copy* and *tm_dither*.

locale *uncomputable* =

fixes *code* :: *instr list* ⇒ *nat*

and *H* :: *instr list*

assumes *h_composable*[*intro*]: *composable_tm0 H*

and *h_case*:
$\bigwedge M\ ns.\ \ TMC\_has\_num\_res\ M\ ns \Longrightarrow \{\!\!\{(\lambda tap.\ tap = ([Bk],\ <(code\ M,\ ns)>))\}\!\!\}\ H\ \{\!\!\{(\lambda tap.\ \exists k.\ tap = (Bk \uparrow k,\ <0{::}nat>))\}\!\!\}$
and *nh_case*:
$\bigwedge M\ ns.\ \neg\ \ TMC\_has\_num\_res\ M\ ns \Longrightarrow \{\!\!\{(\lambda tap.\ tap = ([Bk],\ <(code\ M,\ ns)>))\}\!\!\}\ H\ \{\!\!\{(\lambda tap.\ \exists k.\ tap = (Bk \uparrow k,\ <1{::}nat>))\}\!\!\}$
begin

abbreviation (*input*)
$pre\_H\_ass\ M\ ns \stackrel{def}{=} \lambda tap.\ tap = ([Bk],\ <(code\ M,\ ns{::}nat\ list)>)$

abbreviation (*input*)
$post\_H\_halt\_ass \stackrel{def}{=} \lambda tap.\ \exists k.\ tap = (Bk \uparrow k,\ <1{::}nat>)$

abbreviation (*input*)
$post\_H\_unhalt\_ass \stackrel{def}{=} \lambda tap.\ \exists k.\ tap = (Bk \uparrow k,\ <0{::}nat>)$

lemma *H_halt*:
assumes $\neg\ \ TMC\_has\_num\_res\ M\ ns$
shows $\{\!\!\{pre\_H\_ass\ M\ ns\}\!\!\}\ H\ \{\!\!\{post\_H\_halt\_ass\}\!\!\}$
$\langle proof \rangle$

lemma *H_unhalt*:
assumes $TMC\_has\_num\_res\ M\ ns$
shows $\{\!\!\{pre\_H\_ass\ M\ ns\}\!\!\}\ H\ \{\!\!\{post\_H\_unhalt\_ass\}\!\!\}$
$\langle proof \rangle$

definition
$tcontra \stackrel{def}{=} (tm\_copy\ |{+}|\ H)\ |{+}|\ tm\_dither$
abbreviation
$code\_tcontra \stackrel{def}{=} code\ tcontra$

lemma *tcontra_unhalt*:
assumes $\neg\ \ TMC\_has\_num\_res\ tcontra\ [code\ tcontra]$
shows *False*
$\langle proof \rangle$

lemma *tcontra_halt*:
assumes $TMC\_has\_num\_res\ tcontra\ [code\ tcontra]$
shows *False*
$\langle proof \rangle$

Thus *False* is derivable.

lemma *false*: *False*
$\langle proof \rangle$

end

end

Chapter 2

Abacus Programs

Abacus Machines (aka Counter Machines) and their programs are discussed in [1]. They serve as an intermediate computation model in the course of the translation of Recursive Functions into Turing Machines.

2.1 A Mopup Turing Machine that deletes all "registers" on the tape, except one

In this section we define the higher order function mopup_n_tm that generates a mopup Turing Machine for every argument n. The generated mopup function deletes all numerals from the right tape except the n-th one. Such mopup machines will be used in order to tidy up the result computed by Turing Machines that were compiled from Abacus programs. Refer to [1] for more details.

theory *Abacus_Mopup*
 imports
  Turing_Hoare
begin

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

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

declare *replicate_Suc*[*simp del*]

fun *mopup_a* :: *nat* ⇒ *instr list*

where
$mopup\_a\ 0 = []\ |$
$mopup\_a\ (Suc\ n) = mopup\_a\ n\ @$
$[(R, 2{*}n + 3), (WB, 2{*}n + 2), (R, 2{*}n + 1), (WO, 2{*}n + 2)]$

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

fun $mopup\_n\_tm :: nat \Rightarrow instr\ list$
where
$mopup\_n\_tm\ n = mopup\_a\ n\ @\ shift\ mopup\_b\ (2{*}n)$

type-synonym $mopup\_type = config \Rightarrow nat\ list \Rightarrow nat \Rightarrow cell\ list \Rightarrow bool$

fun $mopup\_stop :: mopup\_type$
where
$mopup\_stop\ (s, l, r)\ lm\ n\ ires=$
$(\exists\ ln\ rn.\ l = Bk{\uparrow}ln\ @\ Bk\ \#\ Bk\ \#\ ires \wedge r = <lm\ !\ n>\ @\ Bk{\uparrow}rn)$

fun $mopup\_bef\_erase\_a :: mopup\_type$
where
$mopup\_bef\_erase\_a\ (s, l, r)\ lm\ n\ ires=$
$(\exists\ ln\ m\ rn.\ l = Bk{\uparrow}ln\ @\ Bk\ \#\ Bk\ \#\ ires \wedge$
$r = Oc{\uparrow}m@\ Bk\ \#\ <(drop\ ((s + 1)\ div\ 2)\ lm)>\ @\ Bk{\uparrow}rn)$

fun $mopup\_bef\_erase\_b :: mopup\_type$
where
$mopup\_bef\_erase\_b\ (s, l, r)\ lm\ n\ ires =$
$(\exists\ ln\ m\ rn.\ l = Bk{\uparrow}ln\ @\ Bk\ \#\ Bk\ \#\ ires \wedge r = Bk\ \#\ Oc{\uparrow}m\ @\ Bk\ \#$
$<(drop\ (s\ div\ 2)\ lm)>\ @\ Bk{\uparrow}rn)$

fun $mopup\_jump\_over1 :: mopup\_type$
where
$mopup\_jump\_over1\ (s, l, r)\ lm\ n\ ires =$
$(\exists\ ln\ m1\ m2\ rn.\ m1 + m2 = Suc\ (lm\ !\ n) \wedge$
$l = Oc{\uparrow}m1\ @\ Bk{\uparrow}ln\ @\ Bk\ \#\ Bk\ \#\ ires \wedge$
$(r = Oc{\uparrow}m2\ @\ Bk\ \#\ <(drop\ (Suc\ n)\ lm)>\ @\ Bk{\uparrow}rn \vee$
$(r = Oc{\uparrow}m2 \wedge (drop\ (Suc\ n)\ lm) = [])))$

fun $mopup\_aft\_erase\_a :: mopup\_type$
where
$mopup\_aft\_erase\_a\ (s, l, r)\ lm\ n\ ires =$
$(\exists\ lnl\ lnr\ rn\ (ml{::}nat\ list)\ m.$
$m = Suc\ (lm\ !\ n) \wedge l = Bk{\uparrow}lnr\ @\ Oc{\uparrow}m\ @\ Bk{\uparrow}lnl\ @\ Bk\ \#\ Bk\ \#\ ires \wedge$
$(r = <ml>\ @\ Bk{\uparrow}rn))$

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↑(m − 1) @ Bk↑lnl @ Bk # Bk # ires ∧ r = Oc # Bk↑rn)))

fun mopup_jump_over2 :: mopup_type
 where
  mopup_jump_over2 (s, l, r) lm n ires =
   (∃ ln rn m1 m2.
      m1 + m2 = Suc (lm ! n)
     ∧ r ≠ []
     ∧ (hd r = Oc ⟶ (l = Oc↑m1 @ Bk↑ln @ Bk # Bk # ires ∧ r = Oc↑m2 @ Bk↑rn))
     ∧ (hd r = Bk ⟶ (l = Bk↑ln @ Bk # ires ∧ r = Bk # Oc↑(m1+m2)@ Bk↑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 ≤ 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*]:
$[\![0 < s; s \leq 2 * n; s\ mod\ 2 = Suc\ 0]\!]$
$\Longrightarrow (fetch\ (mopup\_a\ n\ @\ shift\ mopup\_b\ (2 * n))\ s\ Oc) = (WB, s + 1)$
⟨*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 $n < length\ lm\ 0 < s\ s \leq 2 * n\ s\ mod\ 2 = 0$
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*:
$fetch\ (mopup\_a\ n\ @\ shift\ mopup\_b\ (2 * n))\ (Suc\ (Suc\ (2 * n)))\ Oc$
$= (WB, Suc\ (2 * n + 2))$
⟨*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*:
$fetch\ (mopup\_a\ n\ @\ shift\ mopup\_b\ (2 * n))\ (2 * n + 4)\ Oc$

$= (WB, Suc\ (2 * n + 2))$
⟨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*:
$[\![0 < s; s \leq 2 * n; s\ mod\ 2 = Suc\ 0;\ \neg\ Suc\ s \leq 2 * n]\!]$
$\Longrightarrow RR$

$\langle proof \rangle$

lemma *mopup_bef_erase_a_implies_two*[*simp*]:
$[\![n < length\ lm;\ 0 < s;\ s \leq 2 * n;\ s\ mod\ 2 = Suc\ 0;$
$mopup\_bef\_erase\_a\ (s,\ l,\ Oc\ \#\ xs)\ lm\ n\ ires;\ r = Oc\ \#\ xs]\!]$
$\Longrightarrow (Suc\ s \leq 2 * n \longrightarrow mopup\_bef\_erase\_b\ (Suc\ s,\ l,\ Bk\ \#\ xs)\ lm\ n\ ires)\ \wedge$
$(\neg\ Suc\ s \leq 2 * n \longrightarrow mopup\_jump\_over1\ (Suc\ s,\ l,\ Bk\ \#\ xs)\ lm\ n\ ires)$
$\langle proof \rangle$

lemma *tape_of_nl_cons*: $<m\ \#\ lm> = (if\ lm = []\ then\ Oc\uparrow(Suc\ m)$
$else\ Oc\uparrow(Suc\ m)\ @\ Bk\ \#\ <lm>)$
$\langle proof \rangle$

lemma *drop_tape_of_cons*:
$[\![Suc\ q < length\ lm;\ x = lm\ !\ q]\!] \Longrightarrow <drop\ q\ lm> = Oc\ \#\ Oc \uparrow x\ @\ Bk\ \#\ <drop\ (Suc\ q)\ lm>$
$\langle proof \rangle$

lemma *erase2jumpover1*:
$[\![q < length\ list;$
$\forall\ rn.\ <drop\ q\ list> \neq Oc\ \#\ Oc \uparrow (list\ !\ q)\ @\ Bk\ \#\ <drop\ (Suc\ q)\ list>\ @\ Bk \uparrow rn]\!]$
$\Longrightarrow <drop\ q\ list> = Oc\ \#\ Oc \uparrow (list\ !\ q)$
$\langle proof \rangle$

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

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

declare *replicate_Suc*[*simp*]

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

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

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

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

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

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

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

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

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

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

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 ⟹ l ≠ []
 ⟨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*
$\langle proof \rangle$

lemma *mopup_aft_erase_c_via_b*[*simp*]:
$[\![$*mopup_aft_erase_b* $(2 * n + 3, l, [])$ *lm n ires*$]\!]$
$\Longrightarrow$ *mopup_aft_erase_c* $(4 + 2 * n, Bk \# l, [])$ *lm n ires*
$\langle proof \rangle$

lemma *mopup_left_moving_nonempty*[*simp*]:
mopup_left_moving $(2 * n + 5, l, Oc \# xs)$ *lm n ires* $\Longrightarrow l \neq []$
$\langle proof \rangle$

lemma *exp_ind*: $a\uparrow(Suc\ x) = a\uparrow x$ @ $[a]$
$\langle proof \rangle$

lemma *mopup_jump_over2_via_left_moving*[*simp*]:
$[\![$*mopup_left_moving* $(2 * n + 5, l, Oc \# xs)$ *lm n ires*$]\!]$
$\Longrightarrow$ *mopup_jump_over2* $(2 * n + 6, tl\ l, hd\ l \# Oc \# xs)$ *lm n ires*
$\langle proof \rangle$

lemma *mopup_left_moving_nonempty_snd*[*simp*]: *mopup_left_moving* $(2 * n + 5, l, xs)$ *lm n ires* $\Longrightarrow l \neq []$
$\langle proof \rangle$

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*
$\langle proof \rangle$

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*
$\langle proof \rangle$

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

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*
$\langle proof \rangle$

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*
$\langle proof \rangle$

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*: $x \; mod \; 2 \; = 0 \lor \; x \; mod \; 2 = Suc \; 0$
 ⟨*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 $(s, l, r) \; n =$
 $(if \; s > 0 \land s \leq 2{*}n \; then \; 6$
 $else \; if \; s = 2{*}n + 1 \; then \; 4$
 $else \; if \; s \geq 2{*}n + 2 \land s \leq 2{*}n + 4 \; then \; 3$
 $else \; if \; s = 2{*}n + 5 \; then \; 2$
 $else \; if \; s = 2{*}n + 6 \; then \; 1$
 $else \; 0)$

fun *abc_mopup_stage2* :: *config* ⇒ *nat* ⇒ *nat*
 where
 abc_mopup_stage2 $(s, l, r) \; n =$
 $(if \; s > 0 \land s \leq 2{*}n \; then \; length \; r$
 $else \; if \; s = 2{*}n + 1 \; then \; length \; r$
 $else \; if \; s = 2{*}n + 5 \; then \; length \; l$
 $else \; if \; s = 2{*}n + 6 \; then \; length \; l$
 $else \; if \; s \geq 2{*}n + 2 \land s \leq 2{*}n + 4 \; then \; length \; r$
 $else \; 0)$

fun *abc_mopup_stage3* :: *config* ⇒ *nat* ⇒ *nat*
 where
 abc_mopup_stage3 $(s, l, r) \; n =$
 $(if \; s > 0 \land s \leq 2{*}n \; then$
 $if \; hd \; r = Bk \; then \; 0$
 $else \; 1$
 $else \; if \; s = 2{*}n + 2 \; then \; 1$

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
$\langle$proof$\rangle$

lemma abc_mopup_measure_induct [case_names Step]:
$[\![\bigwedge$n. $\neg$ P (f n) $\Longrightarrow$ (f (Suc n), (f n)) $\in$ abc_mopup_measure$]\!]$ $\Longrightarrow$ $\exists$ n. P (f n)
$\langle$proof$\rangle$

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
$\langle$proof$\rangle$

declare mopup_inv.simps[simp del]

lemma fetch_mopup_a_shift[simp]:
assumes 0 $<$ q q $\le$ n
shows fetch (mopup_a n @ shift mopup_b (2 * n)) (2*q) Bk = (R, 2*q $-$ 1)
$\langle$proof$\rangle$

lemma mopup_halt:
assumes
less: n $<$ length lm
and inv: mopup_inv (Suc 0, l, r) lm n ires
and f: f = (λ stp. (steps (Suc 0, l, r) (mopup_a n @ shift mopup_b (2 * n), 0) stp, n))
and P: P = (λ (c, n). is_final c)
shows $\exists$ stp. P (f stp)
$\langle$proof$\rangle$

lemma mopup_inv_start:
n $<$ length am $\Longrightarrow$ mopup_inv (Suc 0, Bk # Bk # ires, <am> @ Bk $\uparrow$ k) am n ires
$\langle$proof$\rangle$

lemma mopup_correct:
assumes less: n $<$ length (am::nat list)
and rs: am ! n = rs
shows $\exists$ stp i j. (steps (Suc 0, Bk # Bk # ires, <am> @ Bk $\uparrow$ k) (mopup_a n @ shift mopup_b (2 * n), 0) stp)
= (0, Bk$\uparrow$i @ Bk # Bk # ires, Oc # Oc$\uparrow$ rs @ Bk$\uparrow$j)
$\langle$proof$\rangle$

lemma *composable_mopup_n_tm*[*intro*]: *composable_tm* (*mopup_n_tm n*, *0*)
⟨*proof*⟩

end

2.2 Definition of Abacus Machines

theory *Abacus*
imports *Turing_Hoare Abacus_Mopup Turing_HaltingConditions*
begin

declare *adjust*.*simps*[*simp del*]
declare *seq_tm*.*simps* [*simp del*]
declare *shift*.*simps*[*simp del*]
declare *composable_tm*.*simps*[*simp del*]
declare *step*.*simps*[*simp del*]
declare *steps*.*simps*[*simp del*]
declare *fetch*.*simps*[*simp del*]

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 Abacus machines consists of its current state and its current memory:

type-synonym $abc\_conf = abc\_state \times abc\_lm$

Fetch instruction out of Abacus program:

fun $abc\_fetch :: nat \Rightarrow abc\_prog \Rightarrow abc\_inst\ option$
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 fetched, configuration does not change.

fun $abc\_step\_l :: abc\_conf \Rightarrow abc\_inst\ option \Rightarrow abc\_conf$
where
$abc\_step\_l\ (s, lm)\ a = (case\ a\ of$
$None \Rightarrow (s, lm)\ |$
$Some\ (Inc\ n) \Rightarrow (let\ nv = abc\_lm\_v\ lm\ n\ in$
$(s + 1, abc\_lm\_s\ lm\ n\ (nv + 1)))\ |$
$Some\ (Dec\ n\ e) \Rightarrow (let\ nv = abc\_lm\_v\ lm\ n\ in$
$if\ (nv = 0)\ then\ (e, abc\_lm\_s\ lm\ n\ 0)$
$else\ (s + 1,\ abc\_lm\_s\ lm\ n\ (nv - 1)))\ |$
$Some\ (Goto\ n) \Rightarrow (n, lm)$
$)$

Multi-step execution of Abacus Machines.

fun $abc\_steps\_l :: abc\_conf \Rightarrow abc\_prog \Rightarrow nat \Rightarrow abc\_conf$
where
$abc\_steps\_l\ (s, lm)\ p\ 0 = (s, lm)\ |$
$abc\_steps\_l\ (s, lm)\ p\ (Suc\ n) = abc\_steps\_l\ (abc\_step\_l\ (s, lm)\ (abc\_fetch\ s\ p))\ p\ n$

2.3 Compiling Abacus Machines into Turing Machines

2.3.1 Functions used for compilation

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 $findnth :: nat \Rightarrow instr\ list$
where
$findnth\ 0 = []\ |$
$findnth\ (Suc\ n) = (findnth\ n\ @\ [(WO, 2 * n + 1),$
$(R, 2 * n + 2), (R, 2 * n + 3), (R, 2 * n + 2)])$

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

$$
\begin{aligned}
tinc\_b \stackrel{def}{=} [&(WO, 1), (R, 2), (WO, 3), (R, 2), (WO, 3), (R, 4),\\
&(L, 7), (WB, 5), (R, 6), (WB, 5), (WO, 3), (R, 6),\\
&(L, 8), (L, 7), (R, 9), (L, 7), (R, 10), (WB, 9)]
\end{aligned}
$$

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
$tinc\ ss\ n = shift\ (findnth\ n\ @\ shift\ tinc\_b\ (2 * n))\ (ss - 1)$

tdec_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

$$
\begin{aligned}
tdec\_b \stackrel{def}{=} [&(WO, 1), (R, 2), (L, 14), (R, 3), (L, 4), (R, 3),\\
&(R, 5), (WB, 4), (R, 6), (WB, 5), (L, 7), (L, 8),\\
&(L, 11), (WB, 7), (WO, 8), (R, 9), (L, 10), (R, 9),\\
&(R, 5), (WB, 10), (L, 12), (L, 11), (R, 13), (L, 11),\\
&(R, 17), (WB, 13), (L, 15), (L, 14), (R, 16), (L, 14),\\
&(R, 0), (WB, 16)]
\end{aligned}
$$

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
$length\_of\ i = (case\ i\ of$
$\quad Inc\ n \Rightarrow 2 * n + 9\ |$
$\quad Dec\ n\ e \Rightarrow 2 * n + 16\ |$

Goto n $\Rightarrow$ *1*)

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

fun *layout_of* :: *abc_prog* $\Rightarrow$ *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* $\Rightarrow$ *nat* $\Rightarrow$ *nat*
where
start_of ly x = (*Suc* (*sum_list* (*take x ly*)))

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

fun *ci* :: *layout* $\Rightarrow$ *nat* $\Rightarrow$ *abc_inst* $\Rightarrow$ *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* $\Rightarrow$ (*nat* $\times$ *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* $\Rightarrow$ (*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* $\Rightarrow$ *instr list*
where *tm_of ap* = *concat* (*tms_of ap*)

lemma *length_findnth*:
length (*findnth n*) = *4* $*$ *n*
$\langle$*proof*$\rangle$

lemma *ci_length* : *length* (*ci ns n ai*) *div 2* = *length_of ai*
$\langle$*proof*$\rangle$

2.3.2 Representation of Abacus Memory by TM tapes

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

fun *crsp* :: *layout* $\Rightarrow$ *abc_conf* $\Rightarrow$ *config* $\Rightarrow$ *cell list* $\Rightarrow$ *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 invariants 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*]
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*⟩

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*: $[\![ly = layout\_of\ ap;$
$abc\_fetch\ as\ ap = Some\ (Inc\ n)]\!]$
$\Longrightarrow start\_of\ ly\ (Suc\ as) = start\_of\ ly\ as + 2 * n + 9$
⟨*proof*⟩

lemma *start_of_Suc2*:
$[\![ly = layout\_of\ ap;$
$abc\_fetch\ as\ ap = Some\ (Dec\ n\ e)]\!] \Longrightarrow$
$start\_of\ ly\ (Suc\ as) =$
$start\_of\ ly\ as + 2 * n + 16$
⟨*proof*⟩

lemma *start_of_Suc3*:
$[\![ly = layout\_of\ ap;$
$abc\_fetch\ as\ ap = Some\ (Goto\ n)]\!] \Longrightarrow$
$start\_of\ ly\ (Suc\ as) = start\_of\ ly\ as + 1$
⟨*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*]: $Suc\ as \leq length\ xs \Longrightarrow$
$take\ (Suc\ as)\ xs = take\ as\ xs\ @\ [xs\ !\ as]$
⟨*proof*⟩

lemma *concat_suc*: $Suc\ as \leq length\ xs \Longrightarrow$
$concat\ (take\ (Suc\ as)\ xs) = concat\ (take\ as\ xs)\ @\ xs!\ as$
⟨*proof*⟩

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

declare *append_assoc*[*simp del*]

lemma *tm_append*:
$[\![n < length\ tps;\ tp = tps\ !\ n]\!] \Longrightarrow$
$\exists\ tp1\ tp2.\ concat\ tps = tp1\ @\ tp\ @\ tp2 \wedge tp1 =$
$concat\ (take\ n\ tps) \wedge tp2 = concat\ (drop\ (Suc\ n)\ tps)$

⟨*proof*⟩

declare *append_assoc*[*simp*]

lemma *length_tms_of*[*simp*]: $length\ (tms\_of\ aprog) = length\ aprog$
⟨*proof*⟩

lemma *ci_nth*:
$[\![ly = layout\_of\ aprog;$
$abc\_fetch\ as\ aprog = Some\ ins]\!]$
$\Longrightarrow ci\ ly\ (start\_of\ ly\ as)\ ins = tms\_of\ aprog\ !\ as$
⟨*proof*⟩

lemma *t_split*:$[\![$
$ly = layout\_of\ aprog;$
$abc\_fetch\ as\ aprog = Some\ ins]\!]$
$\Longrightarrow \exists\ tp1\ tp2.\ concat\ (tms\_of\ aprog) =$
$tp1\ @\ (ci\ ly\ (start\_of\ ly\ as)\ ins)\ @\ tp2$
$\wedge\ tp1 = concat\ (take\ as\ (tms\_of\ aprog))\ \wedge$
$tp2 = concat\ (drop\ (Suc\ as)\ (tms\_of\ aprog))$
⟨*proof*⟩

lemma *div_apart*: $[\![x\ mod\ (2::nat) = 0;\ y\ mod\ 2 = 0]\!]$
$\Longrightarrow (x + y)\ div\ 2 = x\ div\ 2 + y\ div\ 2$
⟨*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*:
$[\![tp = concat\ (take\ as\ (tms\_of\ ap));$
$as \le length\ ap]\!] \Longrightarrow$
$start\_of\ (layout\_of\ ap)\ as = Suc\ (length\ tp\ div\ 2)$
⟨*proof*⟩

declare *fetch.simps*[*simp*]
lemma *append_append_fetch*:
$[\![length\ tp1\ mod\ 2 = 0;\ length\ tp\ mod\ 2 = 0;$
$length\ tp1\ div\ 2 < a \wedge a \le length\ tp1\ div\ 2 + length\ tp\ div\ 2]\!]$
$\Longrightarrow fetch\ (tp1\ @\ tp\ @\ tp2)\ a\ b = fetch\ tp\ (a - length\ tp1\ div\ 2)\ b$
⟨*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 $fetch\ tp\ s\ b = fetch\ (ci\ ly\ (start\_of\ ly\ as)\ ins)$
$(Suc\ s - start\_of\ ly\ as)\ b$
$\langle proof \rangle$

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 $fetch\ tp\ s\ b = (ac, ns)$
$\langle proof \rangle$

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 $step\ (s, l, r)\ (tp, 0) = (s', l', r')$
$\langle proof \rangle$

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*: $s' \neq 0$
shows $steps\ (s, l, r)\ (tp, 0)\ stp = (s', l', r')$
$\langle proof \rangle$

lemma *tm_append_fetch_first*:
$[\![fetch\ A\ s\ b = (ac, ns); ns \neq 0]\!] \Longrightarrow$
$fetch\ (A\ @\ B)\ s\ b = (ac, ns)$
$\langle proof \rangle$

lemma *tm_append_first_step_eq*:
assumes $step\ (s, l, r)\ (A, off) = (s', l', r')$
and $s' \neq 0$
shows $step\ (s, l, r)\ (A\ @\ B, off) = (s', l', r')$
$\langle proof \rangle$

lemma *tm_append_first_steps_eq*:
assumes *steps* (s, l, r) (A, off) *stp* $= (s', l', r')$
and $s' \neq 0$
shows *steps* (s, l, r) $(A\ @\ B, \mathit{off})$ *stp* $= (s', l', r')$
$\langle \mathit{proof} \rangle$

lemma *tm_append_second_fetch_eq*:
assumes
even: *length A mod 2* $= 0$
and *off*: $\mathit{off} =$ *length A div 2*
and *fetch*: *fetch B s b* $= (ac, ns)$
and *notfinal*: $ns \neq 0$
shows *fetch* $(A\ @\ \mathit{shift}\ B\ \mathit{off})$ $(s + \mathit{off})$ $b = (ac, ns + \mathit{off})$
$\langle \mathit{proof} \rangle$

lemma *tm_append_second_step_eq*:
assumes
exec: *step0* (s, l, r) $B = (s', l', r')$
and *notfinal*: $s' \neq 0$
and *off*: $\mathit{off} =$ *length A div 2*
and *even*: *length A mod 2* $= 0$
shows *step0* $(s + \mathit{off}, l, r)$ $(A\ @\ \mathit{shift}\ B\ \mathit{off}) = (s' + \mathit{off}, l', r')$
$\langle \mathit{proof} \rangle$

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*: $\mathit{off} =$ *length A div 2*
and *even*: *length A mod 2* $= 0$
shows *steps* $(s + \mathit{off}, l, r)$ $(A\ @\ \mathit{shift}\ B\ \mathit{off}, 0)$ *stp* $= (s' + \mathit{off}, l', r')$
$\langle \mathit{proof} \rangle$

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

lemma *tm_append_second_halt_eq*:
assumes
exec: *steps* $(\mathit{Suc}\ 0, l, r)$ $(B, 0)$ *stp* $= (0, l', r')$
and *composable_tm* $(B, 0)$
and *off*: $\mathit{off} =$ *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 ≠ 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⟩
```

2.3.3 Compilation of instruction Inc

```
fun at_begin_fst_bwtn :: inc_inv_t
  where
    at_begin_fst_bwtn (as, lm) (s, l, r) ires =
     (∃ lm1 tn rn. lm1 = (lm @ 0↑tn) ∧ length lm1 = s ∧
        (if lm1 = [] then l = Bk # Bk # ires
         else l = [Bk]@<rev lm1>@Bk#Bk#ires) ∧ r = Bk↑rn)


fun at_begin_fst_awtn :: inc_inv_t
  where
    at_begin_fst_awtn (as, lm) (s, l, r) ires =
     (∃ lm1 tn rn. lm1 = (lm @ 0↑tn) ∧ length lm1 = s ∧
       (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=
(∃ rn. l = Bk # ires ∧ r = Bk # <lm> @ Bk↑rn)

fun *inv_stop* :: *inc_inv_t*
where *inv_stop* (as, lm) (s, l, r) ires=
(∃ rn. l = Bk # Bk # ires ∧ r = <lm> @ Bk↑rn)

lemma *halt_lemma2′*:
⟦wf LE; ∀ n. ((¬ P (f n) ∧ Q (f n)) ⟶
(Q (f (Suc n)) ∧ (f (Suc n), (f n)) ∈ LE)); Q (f 0)⟧
⟹ ∃ n. P (f n)
⟨proof⟩

lemma *halt_lemma2″*:
⟦P (f n); ¬ P (f (0::nat))⟧ ⟹
∃ n. (P (f n) ∧ (∀ i < n. ¬ P (f i)))
⟨proof⟩

lemma *halt_lemma2‴*:
⟦∀ n. ¬ P (f n) ∧ Q (f n) ⟶ Q (f (Suc n)) ∧ (f (Suc n), f n) ∈ LE;
Q (f 0); ∀ i<na. ¬ P (f i)⟧ ⟹ Q (f na)
⟨proof⟩

```
lemma halt_lemma2:
 ⟦wf LE;
  Q (f 0); ¬ P (f 0);
  ∀ n. ((¬ P (f n) ∧ Q (f n)) ⟶ (Q (f (Suc n)) ∧ (f (Suc n), (f n)) ∈ LE))⟧
 ⟹ ∃ n. P (f n) ∧ 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 ≠ [] ∧ 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 ≝ less_than <∗lex∗> less_than

definition findnth_LE :: ((config × nat) × (config × nat)) set
 where
  findnth_LE ≝ (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]:
```

$[\![x < Suc\ (Suc\ (2 * n)); Suc\ x\ mod\ 2 = Suc\ 0; \neg\ x < 2 * n]\!]$
$\Longrightarrow x = 2{*}n$
$\langle proof \rangle$

lemma *between_sucs*:$x < Suc\ n \Longrightarrow \neg\ x < n \Longrightarrow x = n$ $\langle proof \rangle$

lemma *fetch_findnth*[*simp*]:
$[\![0 < a; a < Suc\ (2 * n); a\ mod\ 2 = Suc\ 0]\!] \Longrightarrow fetch\ (findnth\ n)\ a\ Oc = (R, Suc\ a)$
$[\![0 < a; a < Suc\ (2 * n); a\ mod\ 2 \neq Suc\ 0]\!] \Longrightarrow fetch\ (findnth\ n)\ a\ Oc = (R, a)$
$[\![0 < a; a < Suc\ (2 * n); a\ mod\ 2 \neq Suc\ 0]\!] \Longrightarrow fetch\ (findnth\ n)\ a\ Bk = (R, Suc\ a)$
$[\![0 < a; a < Suc\ (2 * n); a\ mod\ 2 = Suc\ 0]\!] \Longrightarrow fetch\ (findnth\ n)\ a\ Bk = (WO, a)$
$\langle proof \rangle$

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*]
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*]: $\exists\ rn.\ [Bk] = Bk \uparrow rn$
$\langle proof \rangle$

lemma *at_begin_norm_Bk*[*intro*]: $at\_begin\_norm\ (as, am)\ (q, aaa, [])\ ires$
$\Longrightarrow at\_begin\_norm\ (as, am)\ (q, aaa, [Bk])\ ires$
$\langle proof \rangle$

lemma *at_begin_fst_bwtn_Bk*[*intro*]: $at\_begin\_fst\_bwtn\ (as, am)\ (q, aaa, [])\ ires$
$\Longrightarrow at\_begin\_fst\_bwtn\ (as, am)\ (q, aaa, [Bk])\ ires$
$\langle proof \rangle$

lemma *at_begin_fst_awtn_Bk*[*intro*]: $at\_begin\_fst\_awtn\ (as, am)\ (q, aaa, [])\ ires$
$\Longrightarrow at\_begin\_fst\_awtn\ (as, am)\ (q, aaa, [Bk])\ ires$
$\langle proof \rangle$

lemma *inv_locate_a_Bk*[*intro*]: $inv\_locate\_a\ (as, am)\ (q, aaa, [])\ ires$
$\Longrightarrow inv\_locate\_a\ (as, am)\ (q, aaa, [Bk])\ ires$
$\langle proof \rangle$

lemma *locate_a_2_locate_a*[*simp*]: $inv\_locate\_a\ (as, am)\ (q, aaa, Bk\ \#\ xs)\ ires$

⟹ *inv_locate_a* (*as*, *am*) (*q*, *aaa*, *Oc* # *xs*) *ires*
⟨*proof*⟩

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

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

lemma *inv_locate*[*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 *repeat_Bk_no_Oc*[*simp*]: (*Oc* # *r* = *Bk* ↑ *rn*) = *False*
⟨*proof*⟩

lemma *repeat_Bk*[*simp*]: (∃ *rna*. *Bk* ↑ *rn* = *Bk* # *Bk* ↑ *rna*) ∨ *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*]: $x\ mod\ 2 = 0 \Longrightarrow Suc\ x\ mod\ 2 = Suc\ 0$
⟨*proof*⟩

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

lemma *locate_b_2_locate_a*[*simp*]:
$[\![q > 0;\ inv\_locate\_b\ (as, am)\ (q - Suc\ 0, aaa, Bk\ \#\ xs)\ ires]\!]$
$\Longrightarrow\ inv\_locate\_a\ (as, am)\ (q, Bk\ \#\ aaa, xs)\ ires$
⟨*proof*⟩

lemma *findnth_inv_layout_of_via_crsp*[*simp*]:
$crsp\ (layout\_of\ ap)\ (as, lm)\ (s, l, r)\ ires$
$\Longrightarrow findnth\_inv\ (layout\_of\ ap)\ n\ (as, lm)\ (Suc\ 0, l, r)\ ires$
⟨*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*: $f = (\lambda\ stp.\ (steps\ (Suc\ 0, l, r)\ (findnth\ n, 0)\ stp, n))$
and *P*: $P = (\lambda\ ((s, l, r), n).\ s = Suc\ (2 * n))$
and *Q*: $Q = (\lambda\ ((s, l, r), n).\ findnth\_inv\ ly\ n\ (as, lm)\ (s, l, r)\ ires)$
shows $\exists\ stp.\ P\ (f\ stp) \wedge Q\ (f\ stp)$
⟨*proof*⟩

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

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

fun *inc_inv* :: $nat \Rightarrow inc\_inv\_t$
where
$inc\_inv\ n\ (as, lm)\ (s, l, r)\ ires =$
$(let\ lm' = abc\_lm\_s\ lm\ n\ (Suc\ (abc\_lm\_v\ lm\ n))\ in$
$if\ s = 0\ then\ False$
$else\ if\ s = 1\ then$
$inv\_locate\_a\ (as, lm)\ (n, l, r)\ ires$
$else\ if\ s = 2\ then$

```
      inv_locate_b (as, lm) (n, l, r) ires
    else if s = 3 then
      inv_after_write (as, lm') (s, l, r) ires
    else if s = Suc 3 then
      inv_after_move (as, lm') (s, l, r) ires
    else if s = Suc 4 then
      inv_after_clear (as, lm') (s, l, r) ires
    else if s = Suc (Suc 4) then
      inv_on_right_moving (as, lm') (s, l, r) ires
    else if s = Suc (Suc 5) then
      inv_on_left_moving (as, lm') (s, l, r) ires
    else if s = Suc (Suc (Suc 5)) then
      inv_check_left_moving (as, lm') (s, l, r) ires
    else if s = Suc (Suc (Suc (Suc 5))) then
      inv_after_left_moving (as, lm') (s, l, r) ires
    else if s = Suc (Suc (Suc (Suc (Suc 5)))) then
      inv_stop (as, lm') (s, l, r) ires
    else False)
```

```
fun abc_inc_stage1 :: config ⇒ nat
  where
  abc_inc_stage1 (s, l, r) =
      (if s = 0 then 0
       else if s ≤ 2 then 5
       else if s ≤ 6 then 4
       else if s ≤ 8 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* $\stackrel{def}{=}$ *less_than* <∗*lex*∗> *lex_pair*

definition *inc_LE* :: (*config* × *config*) *set*
 where
  inc_LE $\stackrel{def}{=}$ (*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 ($l = [] \longrightarrow$
inv_on_left_moving (*as*, *lm*) (s', [], *Bk* # *Bk* # *r*) *ires*) $\wedge$
($l \neq [] \longrightarrow$
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*
$\Longrightarrow$ ($l = [] \longrightarrow$ *inv_on_left_moving* (*as*, *lm*) (s', [], [*Bk*]) *ires*) $\wedge$
($l \neq [] \longrightarrow$ *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*
$\Longrightarrow$ *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*
$\Longrightarrow$ *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*
$\Longrightarrow$ *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*$\Longrightarrow$

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*;

$hd\ l = Bk;\ l \neq []\rrbracket \Longrightarrow$
$inv\_on\_left\_moving\_in\_middle\_B\ (as, lm)\ (s, tl\ l, Bk \# Oc \# r)\ ires$
$\langle proof \rangle$

lemma $inv\_on\_left\_moving\_norm\_Oc\_Oc[simp]$: $\llbracket inv\_on\_left\_moving\_norm\ (as, lm)\ (s, l, Oc \# r)\ ires;$
$hd\ l = Oc;\ l \neq []\rrbracket$
$\Longrightarrow inv\_on\_left\_moving\_norm\ (as, lm)$
$(s, tl\ l, Oc \# Oc \# r)\ ires$
$\langle proof \rangle$

lemma $inv\_on\_left\_moving\_in\_middle\_B\_Bk\_Oc[simp]$: $inv\_on\_left\_moving\_norm\ (as, lm)\ (s, [], Oc \# r)\ ires$
$\Longrightarrow inv\_on\_left\_moving\_in\_middle\_B\ (as, lm)\ (s, [], Bk \# Oc \# r)\ ires$
$\langle proof \rangle$

lemma $inv\_on\_left\_moving\_Oc\_cases[simp]$:$inv\_on\_left\_moving\ (as, lm)\ (s, l, Oc \# r)\ ires$
$\Longrightarrow (l = [] \longrightarrow inv\_on\_left\_moving\ (as, lm)\ (s, [], Bk \# Oc \# r)\ ires)$
$\wedge\ (l \neq [] \longrightarrow inv\_on\_left\_moving\ (as, lm)\ (s, tl\ l, hd\ l \# Oc \# r)\ ires)$
$\langle proof \rangle$

lemma $from\_on\_left\_moving\_to\_check\_left\_moving[simp]$: $inv\_on\_left\_moving\_in\_middle\_B\ (as, lm)$
$(s, Bk \# list, Bk \# r)\ ires$
$\Longrightarrow inv\_check\_left\_moving\_on\_leftmost\ (as, lm)$
$(s', list, Bk \# Bk \# r)\ ires$
$\langle proof \rangle$

lemma $inv\_check\_left\_moving\_in\_middle\_no\_Bk[simp]$:
$inv\_check\_left\_moving\_in\_middle\ (as, lm)\ (s, l, Bk \# r)\ ires = False$
$\langle proof \rangle$

lemma $inv\_check\_left\_moving\_on\_leftmost\_Bk\_Bk[simp]$:
$inv\_on\_left\_moving\_in\_middle\_B\ (as, lm)\ (s, [], Bk \# r)\ ires \Longrightarrow$
$inv\_check\_left\_moving\_on\_leftmost\ (as, lm)\ (s', [], Bk \# Bk \# r)\ ires$
$\langle proof \rangle$

lemma $inv\_check\_left\_moving\_on\_leftmost\_no\_Oc[simp]$: $inv\_check\_left\_moving\_on\_leftmost\ (as, lm)$
$(s, list, Oc \# r)\ ires = False$
$\langle proof \rangle$

lemma $inv\_check\_left\_moving\_in\_middle\_Oc\_Bk[simp]$: $inv\_on\_left\_moving\_in\_middle\_B\ (as, lm)$
$(s, Oc \# list, Bk \# r)\ ires$
$\Longrightarrow inv\_check\_left\_moving\_in\_middle\ (as, lm)\ (s', list, Oc \# Bk \# r)\ ires$
$\langle proof \rangle$

lemma $inv\_on\_left\_moving\_2\_check\_left\_moving[simp]$:
$inv\_on\_left\_moving\ (as, lm)\ (s, l, Bk \# r)\ ires$

$\Longrightarrow (l = [] \longrightarrow inv\_check\_left\_moving\ (as, lm)\ (s', [], Bk \# Bk \# r)\ ires)$
$\wedge\ (l \neq [] \longrightarrow$
$inv\_check\_left\_moving\ (as, lm)\ (s', tl\ l, hd\ l \# Bk \# r)\ ires)$
⟨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*]:
$inv\_check\_left\_moving\_in\_middle\ (as, lm)\ (s, [], Oc \# r)\ ires \Longrightarrow$
$inv\_check\_left\_moving\_in\_middle\ (as, lm)\ (s', [Bk], Oc \# r)\ ires$
⟨proof⟩

lemma *inv_on_left_moving_norm_Oc_Oc_via_middle*[*simp*]: $inv\_check\_left\_moving\_in\_middle$
(as, lm)
$(s, Oc \# list, Oc \# r)\ ires$
$\Longrightarrow inv\_on\_left\_moving\_norm\ (as, lm)\ (s', list, Oc \# Oc \# r)\ ires$
⟨proof⟩

lemma *inv_check_left_moving_Oc_cases*[*simp*]: $inv\_check\_left\_moving\ (as, lm)\ (s, l, Oc \# r)$
$ires$
$\Longrightarrow (l = [] \longrightarrow inv\_on\_left\_moving\ (as, lm)\ (s', [], Bk \# Oc \# r)\ ires)\ \wedge$
$(l \neq [] \longrightarrow inv\_on\_left\_moving\ (as, lm)\ (s', tl\ l, hd\ l \# Oc \# r)\ ires)$
⟨proof⟩

lemma *inv_after_left_moving_Bk_via_check*[*simp*]: $inv\_check\_left\_moving\ (as, lm)\ (s, l, Bk \#$
$r)\ ires$
$\Longrightarrow inv\_after\_left\_moving\ (as, lm)\ (s', Bk \# l, r)\ ires$
⟨proof⟩

lemma *inv_after_left_moving_Bk_empty_via_check*[*simp*]:$inv\_check\_left\_moving\ (as, lm)\ (s, l,$
$[])\ ires$
$\Longrightarrow inv\_after\_left\_moving\ (as, lm)\ (s', Bk \# l, [])\ ires$
⟨proof⟩

lemma *inv_stop_Bk_move*[*simp*]: $inv\_after\_left\_moving\ (as, lm)\ (s, l, Bk \# r)\ ires$
$\Longrightarrow inv\_stop\ (as, lm)\ (s', Bk \# l, r)\ ires$

$\langle proof \rangle$

lemma *inv_stop_Bk_empty*[*simp*]: $inv\_after\_left\_moving\ (as, lm)\ (s, l, [])\ ires$
$\Longrightarrow inv\_stop\ (as, lm)\ (s', Bk\ \#\ l, [])\ ires$
$\langle proof \rangle$

lemma *inv_stop_indep_fst*[*simp*]: $inv\_stop\ (as, lm)\ (x, l, r)\ ires \Longrightarrow$
$inv\_stop\ (as, lm)\ (y, l, r)\ ires$
$\langle proof \rangle$

lemma *inv_after_clear_no_Oc*[*simp*]: $inv\_after\_clear\ (as, lm)\ (s, aaa, Oc\ \#\ xs)\ ires = False$
$\langle proof \rangle$

lemma *inv_after_left_moving_no_Oc*[*simp*]:
$inv\_after\_left\_moving\ (as, lm)\ (s, aaa, Oc\ \#\ xs)\ ires = False$
$\langle proof \rangle$

lemma *inv_after_clear_Suc_nonempty*[*simp*]:
$inv\_after\_clear\ (as, abc\_lm\_s\ lm\ n\ (Suc\ (abc\_lm\_v\ lm\ n)))\ (s, b, [])\ ires = False$
$\langle proof \rangle$

lemma *inv_on_left_moving_Suc_nonempty*[*simp*]: $inv\_on\_left\_moving\ (as, abc\_lm\_s\ lm\ n\ (Suc$
$(abc\_lm\_v\ lm\ n)))$
$(s, b, Oc\ \#\ list)\ ires \Longrightarrow b \neq []$
$\langle proof \rangle$

lemma *inv_check_left_moving_Suc_nonempty*[*simp*]:
$inv\_check\_left\_moving\ (as, abc\_lm\_s\ lm\ n\ (Suc\ (abc\_lm\_v\ lm\ n)))\ (s, b, Oc\ \#\ list)\ ires \Longrightarrow b$
$\neq []$
$\langle proof \rangle$

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: $P = (\lambda\ (s, l, r).\ s = 10)$
and Q: $Q = (\lambda\ (s, l, r).\ inc\_inv\ n\ (as, lm)\ (s, l, r)\ ires)$
shows $\exists\ stp.\ P\ (f\ stp) \wedge Q\ (f\ stp)$
$\langle proof \rangle$

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 $\exists\ stp\ l'\ r'.\ steps\ (Suc\ 0, l, r)\ (tinc\_b, 0)\ stp = (10, l', r')$
$\wedge\ inv\_stop\ (as, lm')\ (10, l', r')\ ires$
$\langle proof \rangle$

```
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⟩
```

2.3.4 Compilation of instruction 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*)
⟨*proof*⟩

lemma *fetch_Dec*[*simp*]:
fetch (*ci ly* (*start_of ly as*) (*Dec n e*)) (*Suc* (*2* ∗ *n*)) *Bk* = (*WO*, *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 = (WB, 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 = (WO, 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 = (WB, 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)
⟨proof⟩

lemma steps_start_of_invb_inv_locate_a1[simp]:
⟦r = [] ∨ hd r = Bk; inv_locate_a (as, lm) (n, l, r) ires⟧
⟹ ∃ 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

$abc\_dec\_1\_stage2\ (s, l, r)\ ss\ n =$
$(if\ s \le ss + 2 * n + 1\ then\ (ss + 2 * n + 16 - s)$
$else\ if\ s = ss + 2{*}n + 13\ then\ length\ l$
$else\ if\ s = ss + 2{*}n + 14\ then\ length\ l$
$else\ 0)$

fun $abc\_dec\_1\_stage3 :: config \Rightarrow nat \Rightarrow nat \Rightarrow nat$
where
$abc\_dec\_1\_stage3\ (s, l, r)\ ss\ n\ =$
$(if\ s \le ss + 2{*}n + 1\ then$
$if\ (s - ss)\ mod\ 2 = 0\ then$
$if\ r \ne []\ \wedge\ hd\ r = Oc\ then\ 0\ else\ 1$
$else\ length\ r$
$else\ if\ s = ss + 2 * n + 13\ then$
$if\ r \ne []\ \wedge\ hd\ r = Oc\ then\ 2$
$else\ 1$
$else\ if\ s = ss + 2 * n + 14\ then$
$if\ r \ne []\ \wedge\ hd\ r = Oc\ then\ 3\ else\ 0$
$else\ 0)$

fun $abc\_dec\_1\_measure :: (config \times nat \times nat) \Rightarrow (nat \times nat \times nat)$
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 $abc\_dec\_1\_LE ::$
$((config \times nat \times$
$nat) \times (config \times nat \times nat))\ set$
where $abc\_dec\_1\_LE \overset{def}{=} (inv\_image\ lex\_triple\ abc\_dec\_1\_measure)$

lemma wf_dec_le: $wf\ abc\_dec\_1\_LE$
$\langle proof \rangle$

lemma $startof\_Suc2$:
$abc\_fetch\ as\ ap = Some\ (Dec\ n\ e) \Longrightarrow$
$start\_of\ (layout\_of\ ap)\ (Suc\ as) =$
$start\_of\ (layout\_of\ ap)\ as + 2 * n + 16$
$\langle proof \rangle$

lemma $start\_of\_less\_2$:
$start\_of\ ly\ e \le start\_of\ ly\ (Suc\ e)$
$\langle proof \rangle$

lemma $start\_of\_less\_1$: $start\_of\ ly\ e \le start\_of\ ly\ (e + d)$
$\langle proof \rangle$

lemma $start\_of\_less$:
assumes $e < as$
shows $start\_of\ ly\ e \le start\_of\ ly\ as$

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

lemma *dec_after_clear_Bk_tail*[*simp*]: *dec_on_right_moving* (*as*, *am*) (*s*, *l*, *Bk* # *r*) *ires*
$\Longrightarrow$ *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*
$\Longrightarrow$ *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*⟧
$\Longrightarrow$ *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*⟧
$\Longrightarrow$ *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*$]\!]$
$\Longrightarrow$ *inv_on_left_moving* (*as*, *abc_lm_s am n 0*) (*s*, *tl aaa*, [*hd aaa*]) *ires*
⟨*proof*⟩

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*$]\!]$
$\Longrightarrow$ *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*)$]\!]$
$\Longrightarrow$ *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*),

$start\_of\ ly\ as - Suc\ 0)\ stp = (start\_of\ ly\ e, lb, rb) \wedge$
$dec\_inv\_1\ ly\ n\ e\ (as, lm)\ (start\_of\ ly\ e, lb, rb)\ ires$
$\langle proof \rangle$

lemma *crsp_abc_step_via_stop*[*simp*]:
$[\![abc\_lm\_v\ lm\ n = 0;$
$inv\_stop\ (as, abc\_lm\_s\ lm\ n\ (abc\_lm\_v\ lm\ n))\ (start\_of\ ly\ e, lb, rb)\ ires]\!]$
$\Longrightarrow crsp\ ly\ (abc\_step\_l\ (as, lm)\ (Some\ (Dec\ n\ e)))\ (start\_of\ ly\ e, lb, rb)\ ires$
$\langle proof \rangle$

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 $\exists stp > 0.\ crsp\ ly\ (abc\_step\_l\ (as, lm)\ (Some\ (Dec\ n\ e)))$
$(steps\ (start\_of\ ly\ as + 2 * n, l, r)\ (ci\ ly\ (start\_of\ ly\ as)\ (Dec\ n\ e), start\_of\ ly\ as - Suc\ 0)\ stp)$
ires
$\langle proof \rangle$

fun $dec\_inv\_2 :: layout \Rightarrow nat \Rightarrow nat \Rightarrow dec\_inv\_t$
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 $abc\_dec\_2\_stage1 :: config \Rightarrow nat \Rightarrow nat \Rightarrow nat$
where
$abc\_dec\_2\_stage1\ (s, l, r)\ ss\ n =$
$(if\ s \le ss + 2{*}n + 1\ then\ 7$
$else\ if\ s = ss + 2{*}n + 2\ then\ 6$
$else\ if\ s = ss + 2{*}n + 3\ then\ 5$
$else\ if\ s \ge ss + 2{*}n + 4 \wedge s \le ss + 2{*}n + 9\ then\ 4$
$else\ if\ s = ss + 2{*}n + 6\ then\ 3$
$else\ if\ s = ss + 2{*}n + 10 \vee s = ss + 2{*}n + 11\ then\ 2$
$else\ if\ s = ss + 2{*}n + 12\ then\ 1$
$else\ 0)$

fun $abc\_dec\_2\_stage2 :: config \Rightarrow nat \Rightarrow nat \Rightarrow nat$
where
$abc\_dec\_2\_stage2\ (s, l, r)\ ss\ n =$
$(if\ s \le ss + 2 * n + 1\ then\ (ss + 2 * n + 16 - s)$
$else\ if\ s = ss + 2{*}n + 10\ then\ length\ l$
$else\ if\ s = ss + 2{*}n + 11\ then\ length\ l$
$else\ if\ s = ss + 2{*}n + 4\ then\ length\ r - 1$
$else\ if\ s = ss + 2{*}n + 5\ then\ length\ r$
$else\ if\ s = ss + 2{*}n + 7\ then\ length\ r - 1$
$else\ if\ s = ss + 2{*}n + 8\ then$
$length\ r + length\ (takeWhile\ (\lambda\ a.\ a = Oc)\ l) - 1$
$else\ if\ s = ss + 2{*}n + 9\ then$
$length\ r + length\ (takeWhile\ (\lambda\ a.\ a = Oc)\ l) - 1$
$else\ 0)$

fun $abc\_dec\_2\_stage3 :: config \Rightarrow nat \Rightarrow nat \Rightarrow nat$
where
$abc\_dec\_2\_stage3\ (s, l, r)\ ss\ n\ =$
$(if\ s \le ss + 2{*}n + 1\ then$
$if\ (s - ss)\ mod\ 2 = 0\ then\ if\ r \ne [] \wedge$
$hd\ r = Oc\ then\ 0\ else\ 1$
$else\ length\ r$
$else\ if\ s = ss + 2 * n + 10\ then$
$if\ \ r \ne [] \wedge hd\ r = Oc\ then\ 2$
$else\ 1$
$else\ if\ s = ss + 2 * n + 11\ then$
$if\ r \ne [] \wedge hd\ r = Oc\ then\ 3$
$else\ 0$
$else\ (ss + 2 * n + 16 - s))$

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* $\stackrel{def}{=}$ *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* = []

$\langle proof \rangle$

lemma *fewer_Ocs_dec_on_right_moving*[*simp*]:
$[\![dec\_on\_right\_moving\ (as, lm)\ (s, aa, [])\ ires;$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ aa))$
$\neq length\ (takeWhile\ (\lambda a.\ a = Oc)\ aa) - Suc\ 0]\!]$
$\Longrightarrow length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ aa)) <$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ aa) - Suc\ 0$
$\langle proof \rangle$

lemma *more_Ocs_dec_after_clear*[*simp*]:
$dec\_after\_clear\ (as, abc\_lm\_s\ am\ n\ (abc\_lm\_v\ am\ n - Suc\ 0))$
$(start\_of\ (layout\_of\ aprog)\ as + 2 * n + 9, aa, Bk \# xs)\ ires$
$\Longrightarrow length\ xs - Suc\ 0 < length\ xs +$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ aa)$
$\langle proof \rangle$

lemma *more_Ocs_dec_after_clear2*[*simp*]:
$[\![dec\_after\_clear\ (as, abc\_lm\_s\ am\ n\ (abc\_lm\_v\ am\ n - Suc\ 0))$
$(start\_of\ (layout\_of\ aprog)\ as + 2 * n + 9, aa, [])\ ires]\!]$
$\Longrightarrow Suc\ 0 < length\ (takeWhile\ (\lambda a.\ a = Oc)\ aa)$
$\langle proof \rangle$

lemma *inv_check_left_moving_nonemptyE*[*elim*]:
$inv\_check\_left\_moving\ (as, lm)\ (s, [], Oc \# xs)\ ires$
$\Longrightarrow RR$
$\langle proof \rangle$

lemma *inv_locate_n_b_Oc_via_at_begin_norm*[*simp*]:
$[\![0 < abc\_lm\_v\ am\ n;$
$at\_begin\_norm\ (as, am)\ (n, aaa, Oc \# xs)\ ires]\!]$
$\Longrightarrow inv\_locate\_n\_b\ (as, am)\ (n, Oc \# aaa, xs)\ ires$
$\langle proof \rangle$

lemma *inv_locate_n_b_Oc_via_at_begin_fst_awtn*[*simp*]:
$[\![0 < abc\_lm\_v\ am\ n;$
$at\_begin\_fst\_awtn\ (as, am)\ (n, aaa, Oc \# xs)\ ires]\!]$
$\Longrightarrow inv\_locate\_n\_b\ (as, am)\ (n, Oc \# aaa, xs)\ ires$
$\langle proof \rangle$

lemma *inv_locate_n_b_Oc_via_inv_locate_n_a*[*simp*]:
$[\![0 < abc\_lm\_v\ am\ n; inv\_locate\_a\ (as, am)\ (n, aaa, Oc \# xs)\ ires]\!]$
$\Longrightarrow inv\_locate\_n\_b\ (as, am)\ (n, Oc\#aaa, xs)\ ires$
$\langle proof \rangle$

lemma *more_Oc_dec_on_right_moving*[*simp*]:
$[\![dec\_on\_right\_moving\ (as, am)\ (s, aa, Bk \# xs)\ ires;$
$Suc\ (length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ aa)))$
$\neq length\ (takeWhile\ (\lambda a.\ a = Oc)\ aa)]\!]$
$\Longrightarrow Suc\ (length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ aa)))$

$< length\ (takeWhile\ (\lambda a.\ a = Oc)\ aa)$
⟨*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*: $f = (\lambda\ stp.\ (steps\ (start\_of\ ly\ as + 2 * n, la, ra)\ (ci\ ly\ (start\_of\ ly\ as)\ (Dec\ n\ e),$
$start\_of\ ly\ as - Suc\ 0)\ stp, start\_of\ ly\ as, n))$
and *P*: $P = (\lambda\ ((s, l, r), ss, x).\ s = start\_of\ ly\ as + 2{*}n + 16)$
and *Q*: $Q = (\lambda\ ((s, l, r), ss, x).\ dec\_inv\_2\ ly\ x\ e\ (as, lm)\ (s, l, r)\ ires)$
shows $\exists\ stp.\ P\ (f\ stp) \wedge Q(f\ stp)$
⟨*proof*⟩

lemma *crsp_abc_step_l_start_of*[*simp*]:
$[\![inv\_stop\ (as, abc\_lm\_s\ lm\ n\ (abc\_lm\_v\ lm\ n - Suc\ 0))$
$(start\_of\ (layout\_of\ ap)\ as + 2 * n + 16, a, b)\ ires;$
$abc\_lm\_v\ lm\ n > 0;$
$abc\_fetch\ as\ ap = Some\ (Dec\ n\ e)]\!]$
$\Longrightarrow crsp\ (layout\_of\ ap)\ (abc\_step\_l\ (as, lm)\ (Some\ (Dec\ n\ e)))$
$(start\_of\ (layout\_of\ ap)\ as + 2 * n + 16, a, b)\ ires$
⟨*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*⟩

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

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 ∃ 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⟩

2.3.5 Compilation of instruction Goto

lemma crsp_step_goto:
assumes layout: ly = layout_of ap
and crsp: crsp ly (as, lm) (s, l, r) ires
shows ∃ 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 ∃ 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*:
$(map\ (length \circ (\lambda(xa, y).\ ci\ (layout\_of\ xs\ @\ [length\_of\ x])\ xa\ y))\ (tpairs\_of\ xs)) =$
$(map\ (length \circ (\lambda(x, y).\ ci\ (layout\_of\ xs)\ x\ y))\ (tpairs\_of\ xs))$
⟨*proof*⟩

lemma *length_tp'*[*simp*]:
$[\![ly = layout\_of\ ap; tp = tm\_of\ ap]\!] \Longrightarrow$
$length\ tp = 2 * sum\_list\ (take\ (length\ ap)\ (layout\_of\ ap))$
⟨*proof*⟩

lemma *length_tp*:
$[\![ly = layout\_of\ ap; tp = tm\_of\ ap]\!] \Longrightarrow$
$start\_of\ ly\ (length\ ap) = Suc\ (length\ tp\ div\ 2)$
⟨*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 $\exists\ stp\ i\ j.\ steps\ (Suc\ 0, l, r)\ (tp\ @\ shift\ (mopup\_n\_tm\ n)\ off, 0)\ stp = (0, Bk{\uparrow}i\ @\ Bk\ \#\ Bk\ \#\ ires, Oc{\uparrow}Suc\ rs\ @\ Bk{\uparrow}j)$
⟨*proof*⟩

declare *mopup_n_tm.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 $\exists\, stpa \geq stp.\ crsp\ ly\ (as, am)\ (steps\ (Suc\ 0, l, r)\ (tp, 0)\ stpa)\ ires$
⟨*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*: $\forall\ stp.\ (\lambda\ (as, am).\ as < length\ ap)\ (abc\_steps\_l\ (0, lm)\ ap\ stp)$
shows $\forall\ stp. \neg\ is\_final\ (steps\ (1, l, r)\ (tp\ @\ shift\ (mopup\_n\_tm\ n)\ off, 0)\ stp)$
⟨*proof*⟩

end

2.3.6 Alternative Definitions for Translating Abacus Machines to TMs

theory *Abacus_alt_Compile*
imports *Abacus*
begin

abbreviation
$layout \stackrel{def}{=} layout\_of$

fun $address :: abc\_prog \Rightarrow nat \Rightarrow nat$
where
$address\ p\ x = (Suc\ (sum\_list\ (take\ x\ (layout\ p))))$

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

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

abbreviation
$TMDec \stackrel{def}{=} [(WO, 1), (R, 2), (L, 14), (R, 3), (L, 4), (R, 3),$
$(R, 5), (WB, 4), (R, 6), (WB, 5), (L, 7), (L, 8),$
$(L, 11), (WB, 7), (WO, 8), (R, 9), (L, 10), (R, 9),$

$(R, 5), (WB, 10), (L, 12), (L, 11), (R, 13), (L, 11),$
$(R, 17), (WB, 13), (L, 15), (L, 14), (R, 16), (L, 14),$
$(R, 0), (WB, 16)]$

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

2.4 Hoare Rules for Abacus Programs

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* $\times$ *nat list*$) \Rightarrow$ *abc_prog* $\Rightarrow$ *bool*
where
abc_final (s, lm) *p* $= (s =$ *length p*$)$

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

fun *abc_out_of_prog* :: *abc_conf* $\Rightarrow$ *abc_prog* $\Rightarrow$ *bool*
where
abc_out_of_prog (s, lm) *p* $= ($*length p* $< s)$

definition *abcP_out_of_pgm_ex* :: *abc_prog*
where
abcP_out_of_pgm_ex = [*Dec 0 41*, *Inc 1*, *Goto 0*]

lemma *abc_steps_l* (*0*,[*5*,*3*]) *abcP_out_of_pgm_ex* (*10* +*6*) = (*41*, [*0*, *8*])
⟨*proof*⟩

lemma *abc_out_of_prog* (*abc_steps_l* (*0*,[*5*,*3*]) *abcP_out_of_pgm_ex* (*10* +*6*)) *abcP_out_of_pgm_ex*
⟨*proof*⟩

lemma *abc_notfinal cf p* $\vee$ *abc_final cf p* $\vee$ *abc_out_of_prog cf p*
⟨*proof*⟩

lemma $[\![$ *length p* $\neq 0$; *abc_notfinal cf p* $]\!] \Longrightarrow \neg$ *abc_final cf p* $\wedge \neg$ *abc_out_of_prog cf p*
⟨*proof*⟩

lemma $[\![$ *length p* $\neq 0$; *abc_final cf p* $]\!] \Longrightarrow \neg$ *abc_notfinal cf p* $\wedge \neg$ *abc_out_of_prog cf p*
⟨*proof*⟩

lemma $[\![$ *length p* $\neq 0$; *abc_out_of_prog cf p* $]\!] \Longrightarrow \neg$ *abc_notfinal cf p* $\wedge \neg$ *abc_final cf p*
⟨*proof*⟩

definition
abc_Hoare_halt :: *abc_assert* $\Rightarrow$ *abc_prog* $\Rightarrow$ *abc_assert* $\Rightarrow$ *bool* (‹({⦃}(*1*_){⦄}/ (_)/ {⦃}(*1*_){⦄})›

50)
where
abc_Hoare_halt P p Q $\stackrel{def}{=}$ $\forall lm.\ P\ lm \longrightarrow (\exists n.\ abc\_final\ (abc\_steps\_l\ (0, lm)\ p\ n)\ p \wedge Q\ abc\_holds\_for\ (abc\_steps\_l\ (0, lm)\ p\ n))$

lemma *abc_Hoare_haltI*:
assumes $\bigwedge lm.\ P\ lm \Longrightarrow \exists n.\ abc\_final\ (abc\_steps\_l\ (0,\ lm)\ p\ n)\ p \wedge Q\ abc\_holds\_for\ (abc\_steps\_l\ (0, lm)\ p\ n)$
shows $\{\!|P|\!\}\ (p{::}abc\_prog)\ \{\!|Q|\!\}$
$\langle proof \rangle$

fun *app_mopup* :: $tprog0 \Rightarrow nat \Rightarrow tprog0$
where
$app\_mopup\ tp\ n = tp\ @\ shift\ (mopup\_n\_tm\ n)\ (length\ tp\ div\ 2)$

lemma *compile_correct_halt_2*:
assumes *compile*: $tp = tm\_of\ ap$
and *abc_halt*: $abc\_steps\_l\ (0, ns)\ ap\ stp = (length\ ap, am)$
and *rs_loc*: $n < length\ am$
shows $\exists stp\ i\ j.\ steps0\ (Suc\ 0,\ [Bk,Bk],\ <ns{::}nat\ list>)\ (app\_mopup\ tp\ n)\ stp = (0,\ Bk\uparrow i,\ <abc\_lm\_v\ am\ n>\ @\ Bk\uparrow j)$
$\langle proof \rangle$

lemma *compile_correct_halt_3*:
assumes *compile*: $tp = tm\_of\ ap$
and *abc_halt*: $abc\_steps\_l\ (0, ns)\ ap\ stp = (length\ ap, am)$
and *rs_loc*: $n < length\ am$
shows $\exists stp\ i\ j.\ steps0\ (Suc\ 0,\ [],\ <ns{::}nat\ list>)\ (app\_mopup\ tp\ n)\ stp = (0,\ Bk\uparrow i,\ <abc\_lm\_v\ am\ n>\ @\ Bk\uparrow j)$
$\langle proof \rangle$

lemma *compile_correct_halt_4*:
assumes *compile*: $tp = tm\_of\ ap$
and *abc_halt*: $abc\_steps\_l\ (0, ns)\ ap\ stp = (length\ ap, am)$
and *rs_loc*: $n < length\ am$
shows $TMC\_yields\_num\_res\ (app\_mopup\ tp\ n)\ ns\ (abc\_lm\_v\ am\ n)$
$\langle proof \rangle$

definition *ABC_yields_res* :: $abc\_prog \Rightarrow nat\ list \Rightarrow nat \Rightarrow nat \Rightarrow bool$
where *ABC_yields_res ap ns n r* $\stackrel{def}{=}$
$(\exists stp\ am.\ \ abc\_steps\_l\ (0, ns)\ ap\ stp = (length\ ap, am) \wedge$
$r < length\ am \wedge (abc\_lm\_v\ am\ r = n))$

definition *ABC_loops_on* :: *abc_prog* ⇒ *nat list* ⇒ *bool*
where *ABC_loops_on ap ns* $\stackrel{def}{=}$
∀ *stp*. *abc_notfinal* (*abc_steps_l* (*0*, *ns*) *ap stp*) *ap*

theorem *ABC_yields_res_imp_TMC_yields_num_res*:
assumes *tp* = *tm_of ap*
and *ABC_yields_res ap ns n r*
shows *TMC_yields_num_res* (*app_mopup tp r*) *ns n*
⟨*proof*⟩

lemma *abc_unhalt_2*:
assumes *compile*: *tp* = *tm_of ap*
and *notfinal*: ∀ *stp*. *abc_notfinal* (*abc_steps_l* (*0*, *ns*) *ap stp*) *ap*
shows ∀ *stp*.¬ *is_final* (*steps0* (*Suc 0*, [*Bk*,*Bk*], <*ns*::*nat list*>) (*app_mopup tp r*) *stp*)
⟨*proof*⟩

theorem *ABC_loops_imp_TMC_loops*:
assumes *tp* = *tm_of ap*
and *ABC_loops_on ap ns*
shows *TMC_loops* (*app_mopup tp r*) *ns*
⟨*proof*⟩

definition
abc_Hoare_unhalt :: *abc_assert* ⇒ *abc_prog* ⇒ *bool* (‹({|(*1*_)|}/ (_)) ↑› *50*)
where
abc_Hoare_unhalt P p $\stackrel{def}{=}$ ∀ *args*. *P args* ⟶ (∀ *n*. *abc_notfinal* (*abc_steps_l* (*0*, *args*) *p n*) *p*)

lemma *abc_Hoare_unhaltI*:
assumes ⋀*args n*. *P args* ⟹ *abc_notfinal* (*abc_steps_l* (*0*, *args*) *p n*) *p*
shows {|*P*|} (*p*::*abc_prog*) ↑
⟨*proof*⟩

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

fun *abc_comp* :: *abc_inst list* ⇒ *abc_inst list* ⇒
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*:
$[\![abc\_final\ (abc\_steps\_l\ (0, lm)\ p\ n)\ p;\ p \neq []]\!]$
$\Longrightarrow \exists\ n'.\ abc\_notfinal\ (abc\_steps\_l\ (0, lm)\ p\ n')\ p \wedge$
$abc\_final\ (abc\_steps\_l\ (0, lm)\ p\ (Suc\ n'))\ p$
⟨*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_first_steps_eq_pre*:
assumes *notfinal*: $abc\_notfinal\ (abc\_steps\_l\ (0, lm)\ A\ n)\ A$
and *notnull*: $A \neq []$
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_first_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_first_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 *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*) ↑
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*) ↑ ⟹ ⦃*P*⦄ (*A* [+] *B*) ↑
⟨*proof*⟩

lemma *notfinal_all_before*:
⟦*abc_notfinal* (*abc_steps_l* (*0*, *args*) *A x*) *A*; *y*≤*x* ⟧
⟹ *abc_notfinal* (*abc_steps_l* (*0*, *args*) *A y*) *A*
⟨*proof*⟩

lemma *abc_Hoare_plus_unhalt2′*:
assumes *unhalt*: ⦃*Q*⦄ (*B*::*abc_prog*) ↑
and *halt*: ⦃*P*⦄ (*A*::*abc_prog*) ⦃*Q*⦄
and *notnull*: *A* ≠ []
and *P*: *P args*
shows *abc_notfinal* (*abc_steps_l* (*0*, *args*) (*A* [+] *B*) *n*) (*A* [+] *B*)

$\langle proof \rangle$

lemma *abc_comp_null_left*[*simp*]: $[]\ [+]\ A = A$
$\langle proof \rangle$

lemma *abc_comp_null_right*[*simp*]: $A\ [+]\ [] = A$
$\langle proof \rangle$

lemma *abc_Hoare_plus_unhalt2*:
$[\![\{\!|Q|\!\}\ (B{::}abc\_prog)\uparrow;\ \{\!|P|\!\}\ (A{::}abc\_prog)\ \{\!|Q|\!\}]\!] \Longrightarrow \{\!|P|\!\}\ (A\ [+]\ B) \uparrow$
$\langle proof \rangle$

end

Chapter 3

Recursive Function and their compilation into Turing Machines

theory *Rec_Def*
 imports *Main*
begin

3.1 Definition of a recursive datatype for Recursive Functions

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

3.2 Definition of an interpreter for Recursive Functions

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* =

$rec\_exec\ f\ (map\ (\lambda\ a.\ rec\_exec\ a\ xs)\ gs)\ |$
$rec\_exec\ (Pr\ n\ f\ g)\ xs =$
$(if\ last\ xs = 0\ then\ rec\_exec\ f\ (butlast\ xs)$
$else\ rec\_exec\ g\ (butlast\ xs\ @\ (last\ xs - 1)\ \#\ [rec\_exec\ (Pr\ n\ f\ g)\ (butlast\ xs\ @\ [last\ xs - 1])]))\ |$
$rec\_exec\ (Mn\ n\ f)\ xs = (LEAST\ x.\ rec\_exec\ f\ (xs\ @\ [x]) = 0)$
$\langle proof \rangle$

termination
$\langle proof \rangle$

inductive *terminate* :: *recf* ⇒ *nat list* ⇒ *bool*
where
$termi\_z$: $terminate\ z\ [n]$
$|\ termi\_s$: $terminate\ s\ [n]$
$|\ termi\_id$: $[\![n < m;\ length\ xs = m]\!] \Longrightarrow terminate\ (id\ m\ n)\ xs$
$|\ termi\_cn$: $[\![terminate\ f\ (map\ (\lambda g.\ rec\_exec\ g\ xs)\ gs);$
$\forall\ g \in set\ gs.\ terminate\ g\ xs;\ length\ xs = n]\!] \Longrightarrow terminate\ (Cn\ n\ f\ gs)\ xs$
$|\ termi\_pr$: $[\![\forall\ y < x.\ terminate\ g\ (xs\ @\ y\ \#\ [rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])]);$
$terminate\ f\ xs;$
$length\ xs = n]\!]$
$\Longrightarrow terminate\ (Pr\ n\ f\ g)\ (xs\ @\ [x])$
$|\ termi\_mn$: $[\![length\ xs = n;\ terminate\ f\ (xs\ @\ [r]);$
$rec\_exec\ f\ (xs\ @\ [r]) = 0;$
$\forall\ i < r.\ terminate\ f\ (xs\ @\ [i]) \wedge rec\_exec\ f\ (xs\ @\ [i]) > 0]\!] \Longrightarrow terminate\ (Mn\ n\ f)\ xs$

end

3.3 Examples for Recursive Functions based on Rec_def

theory *Rec_Ex*
imports *Rec_Def*
begin

definition *plus_2* :: *recf*
where
$plus\_2 = (Cn\ 1\ s\ [s])$

lemma $rec\_exec\ plus\_2\ [0] = Suc\ (Suc\ 0)$
$\langle proof \rangle$

lemma $rec\_exec\ plus\_2\ [2] = 4$
$\langle proof \rangle$

lemma $rec\_exec\ plus\_2\ [0] = 2$
$\langle proof \rangle$

The arity parameter given to the constructors of recursive functions is not checked during execution by the interpreter.

See the next example where we run *pls_2* with two arguments instead of only one.

lemma *rec_exec plus_2* [*2*,*3*] = *4*
⟨*proof*⟩

lemma *rec_exec plus_2* [*2*,*3*] = *4*
⟨*proof*⟩

What is the purpose of the arity parameter?

The argument 1 of the constructors, which is supposed to be the arity, is completely ignored by *rec_exec*. However, for proofing termination, we need a correct arity specification.

lemma *terminate plus_2* [*2*]
⟨*proof*⟩

lemma *terminate plus_2* [*2*]
⟨*proof*⟩

If we try to proof termination of a run with superfluous arguments, we are stuck. We need the correct arity for proving the predicate termination.

lemma *terminate plus_2* [*2*,*3*]
⟨*proof*⟩

end

3.4 Compilation of Recursive Functions into Abacus Programs

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 ≝ (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))
```

3.4.1 Definition of the compiler rec_ci

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))

3.4.2 Correctness of the compiler rec_ci

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* :: $nat \times nat\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow$
$nat\ list \Rightarrow bool$
where
addition_inv $(as, lm')\ m\ n\ p\ lm =$
$(let\ sn = lm\ !\ n\ in$
$let\ sm = lm\ !\ m\ in$
$lm\ !\ p = 0 \wedge$
$(if\ as = 0\ then\ \exists\ x.\ x \le lm\ !\ m \wedge lm' = lm[m := x,$
$n := (sn + sm - x), p := (sm - x)]$
$else\ if\ as = 1\ then\ \exists\ x.\ x < lm\ !\ m \wedge lm' = lm[m := x,$
$n := (sn + sm - x - 1), p := (sm - x - 1)]$
$else\ if\ as = 2\ then\ \exists\ x.\ x < lm\ !\ m \wedge lm' = lm[m := x,$
$n := (sn + sm - x), p := (sm - x - 1)]$
$else\ if\ as = 3\ then\ \exists\ x.\ x < lm\ !\ m \wedge lm' = lm[m := x,$
$n := (sn + sm - x), p := (sm - x)]$

$else\ if\ as = 4\ then\ \exists\ x.\ x \leq lm\ !\ m \wedge lm' = lm[m := x,$
$n := (sn + sm), p := (sm - x)]$
$else\ if\ as = 5\ then\ \exists\ x.\ x < lm\ !\ m \wedge lm' = lm[m := x,$
$n := (sn + sm), p := (sm - x - 1)]$
$else\ if\ as = 6\ then\ \exists\ x.\ x < lm\ !\ m \wedge lm' =$
$lm[m := Suc\ x, n := (sn + sm), p := (sm - x - 1)]$
$else\ if\ as = 7\ then\ lm' = lm[m := sm, n := (sn + sm)]$
$else\ False))$

fun *addition_stage1* :: $nat \times nat\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat$
where
$addition\_stage1\ (as, lm)\ m\ p =$
$(if\ as = 0 \vee as = 1 \vee as = 2 \vee as = 3\ then\ 2$
$else\ if\ as = 4 \vee as = 5 \vee as = 6\ then\ 1$
$else\ 0)$

fun *addition_stage2* :: $nat \times nat\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat$
where
$addition\_stage2\ (as, lm)\ m\ p =$
$(if\ 0 \leq as \wedge as \leq 3\ then\ lm\ !\ m$
$else\ if\ 4 \leq as \wedge as \leq 6\ then\ lm\ !\ p$
$else\ 0)$

fun *addition_stage3* :: $nat \times nat\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat$
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* :: $((nat \times nat\ list) \times nat \times nat) \Rightarrow$
$(nat \times nat \times nat)$
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* :: $(((nat \times nat\ list) \times nat \times nat) \times$
$((nat \times nat\ list) \times nat \times nat))\ set$
where $addition\_LE \stackrel{def}{=} (inv\_image\ lex\_triple\ addition\_measure)$

lemma *wf_additon_LE[simp]*: *wf addition_LE*
$\langle proof \rangle$

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

lemma *update_zero_to_zero*[*simp*]: $[\![am\ !\ n = (0{::}nat);\ n < length\ am]\!] \Longrightarrow am[n := 0] = am$
⟨*proof*⟩

lemma *addition_inv_init*:
$[\![m \neq n;\ max\ m\ n < p;\ length\ lm > p;\ lm\ !\ p = 0]\!] \Longrightarrow$
$addition\_inv\ (0, lm)\ m\ n\ p\ lm$
⟨*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*]:
$[\![m \neq n;\ p < length\ lm;\ lm\ !\ p = 0;\ m < p;\ n < p;\ x \leq lm\ !\ m;\ 0 < x]\!]$
$\Longrightarrow \exists xa{<}lm\ !\ m.\ lm[m := x, n := lm\ !\ n + lm\ !\ m - x,$
$p := lm\ !\ m - x, m := x - Suc\ 0] =$
$lm[m := xa, n := lm\ !\ n + lm\ !\ m - Suc\ xa,$
$p := lm\ !\ m - Suc\ xa]$
⟨*proof*⟩

lemma *exists_small_list_elem2*[*simp*]:
$[\![m \neq n;\ p < length\ lm;\ lm\ !\ p = 0;\ m < p;\ n < p;\ x < lm\ !\ m]\!]$
$\Longrightarrow \exists xa{<}lm\ !\ m.\ lm[m := x, n := lm\ !\ n + lm\ !\ m - Suc\ x,$
$p := lm\ !\ m - Suc\ x, n := lm\ !\ n + lm\ !\ m - x]$
$= lm[m := xa, n := lm\ !\ n + lm\ !\ m - xa,$
$p := lm\ !\ m - Suc\ xa]$
⟨*proof*⟩

lemma *exists_small_list_elem3*[*simp*]:
$[\![m \neq n;\ p < length\ lm;\ lm\ !\ p = 0;\ m < p;\ n < p;\ x < lm\ !\ m]\!]$
$\Longrightarrow \exists xa{<}lm\ !\ m.\ lm[m := x, n := lm\ !\ n + lm\ !\ m - x,$
$p := lm\ !\ m - Suc\ x, p := lm\ !\ m - x]$
$= lm[m := xa, n := lm\ !\ n + lm\ !\ m - xa,$
$p := lm\ !\ m - xa]$
⟨*proof*⟩

lemma *exists_small_list_elem4*[*simp*]:
$[\![m \neq n;\ p < length\ lm;\ lm\ !\ p = (0{::}nat);\ m < p;\ n < p;\ x < lm\ !\ m]\!]$
$\Longrightarrow \exists xa{\leq}lm\ !\ m.\ lm[m := x, n := lm\ !\ n + lm\ !\ m - x,$
$p := lm\ !\ m - x] =$
$lm[m := xa, n := lm\ !\ n + lm\ !\ m - xa,$

$p := lm ! m - xa]$
⟨*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'*:
$[\![m \neq n; max\ m\ n < p; length\ lm > p; lm ! p = 0]\!] \Longrightarrow$
$\exists\ stp.\ (\lambda\ (as, lm').\ as = 7 \wedge addition\_inv\ (as, lm')\ m\ n\ p\ lm)$
$(abc\_steps\_l\ (0, lm)\ (addition\ m\ n\ p)\ stp)$

$\langle proof \rangle$

lemma *length_addition*[*simp*]: $length\ (addition\ a\ b\ c) = 7$
$\langle proof \rangle$

lemma *addition_correct*:
assumes $m \neq n$ $max\ m\ n < p$ $length\ lm > p$ $lm\ !\ p = 0$
shows $\{\!|\lambda\ a.\ a = lm|\!\}\ (addition\ m\ n\ p)\ \{\!|\lambda\ nl.\ addition\_inv\ (7, nl)\ m\ n\ p\ lm|\!\}$
$\langle proof \rangle$

3.4.2.1 Correctness of compilation for constructor s

lemma *compile_s_correct′*:
$\{\!|\lambda nl.\ nl = n\ \#\ 0 \uparrow 2\ @\ anything|\!\}\ addition\ 0\ (Suc\ 0)\ 2\ [+]\ [Inc\ (Suc\ 0)]\ \{\!|\lambda nl.\ nl = n\ \#\ Suc\ n\ \#\ 0\ \#\ anything|\!\}$
$\langle proof \rangle$

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

lemma *abc_comp_commute*: $(A\ [+]\ B)\ [+]\ C = A\ [+]\ (B\ [+]\ C)$
$\langle proof \rangle$

lemma *compile_s_correct*:
$[\![rec\_ci\ s = (ap, arity, fp);\ rec\_exec\ s\ [n] = r]\!] \Longrightarrow$
$\{\!|\lambda nl.\ nl = n\ \#\ 0 \uparrow (fp - arity)\ @\ anything|\!\}\ ap\ \{\!|\lambda nl.\ nl = n\ \#\ r\ \#\ 0 \uparrow (fp - Suc\ arity)\ @\ anything|\!\}$
$\langle proof \rangle$

3.4.2.2 Correctness of compilation for constructor z

lemma *compile_z_correct*:
$[\![rec\_ci\ z = (ap, arity, fp);\ rec\_exec\ z\ [n] = r]\!] \Longrightarrow$
$\{\!|\lambda nl.\ nl = n\ \#\ 0 \uparrow (fp - arity)\ @\ anything|\!\}\ ap\ \{\!|\lambda nl.\ nl = n\ \#\ r\ \#\ 0 \uparrow (fp - Suc\ arity)\ @\ anything|\!\}$
$\langle proof \rangle$

3.4.2.3 Correctness of compilation for constructor id

lemma *compile_id_correct′*:
assumes $n < length\ args$
shows $\{\!|\lambda nl.\ nl = args\ @\ 0 \uparrow 2\ @\ anything|\!\}\ addition\ n\ (length\ args)\ (Suc\ (length\ args))$
$\{\!|\lambda nl.\ nl = args\ @\ rec\_exec\ (recf.id\ (length\ args)\ n)\ args\ \#\ 0\ \#\ anything|\!\}$
$\langle proof \rangle$

lemma *compile_id_correct*:
$[\![n < m;\ length\ xs = m;\ rec\_ci\ (recf.id\ m\ n) = (ap, arity, fp);\ rec\_exec\ (recf.id\ m\ n)\ xs = r]\!]$
$\Longrightarrow \{\!|\lambda nl.\ nl = xs\ @\ 0 \uparrow (fp - arity)\ @\ anything|\!\}\ ap\ \{\!|\lambda nl.\ nl = xs\ @\ r\ \#\ 0 \uparrow (fp - Suc\ arity)\ @\ anything|\!\}$
$\langle proof \rangle$

3.4.2.4 Correctness of compilation for constructor Cn

lemma *cn_merge_gs_tl_app*:
$cn\_merge\_gs\ (gs\ @\ [g])\ pstr =$
$cn\_merge\_gs\ gs\ pstr\ [+]\ cn\_merge\_gs\ [g]\ (pstr + length\ gs)$
$\langle proof \rangle$

lemma *footprint_ge*:
$rec\_ci\ a = (p, arity, fp) \Longrightarrow arity < fp$
$\langle proof \rangle$

lemma *param_pattern*:
$[\![terminate\ f\ xs;\ rec\_ci\ f = (p, arity, fp)]\!] \Longrightarrow length\ xs = arity$
$\langle proof \rangle$

lemma *replicate_merge_anywhere*:
$x{\uparrow}a\ @\ x{\uparrow}b\ @\ ys = x{\uparrow}(a{+}b)\ @\ ys$
$\langle proof \rangle$

fun *mv_box_inv* :: $nat \times nat\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat\ list \Rightarrow bool$
where
$mv\_box\_inv\ (as, lm)\ m\ n\ initlm =$
$(let\ plus = initlm\ !\ m + initlm\ !\ n\ in$
$length\ initlm > max\ m\ n \wedge m \neq n \wedge$
$(if\ as = 0\ then\ \exists\ k\ l.\ lm = initlm[m := k, n := l] \wedge$
$k + l = plus \wedge k \leq initlm\ !\ m$
$else\ if\ as = 1\ then\ \exists\ k\ l.\ lm = initlm[m := k, n := l]$
$\wedge\ k + l + 1 = plus \wedge k < initlm\ !\ m$
$else\ if\ as = 2\ then\ \exists\ k\ l.\ lm = initlm[m := k, n := l]$
$\wedge\ k + l = plus \wedge k \leq initlm\ !\ m$
$else\ if\ as = 3\ then\ lm = initlm[m := 0, n := plus]$
$else\ False))$

fun *mv_box_stage1* :: $nat \times nat\ list \Rightarrow nat \Rightarrow nat$
where
$mv\_box\_stage1\ (as, lm)\ m\ =$
$(if\ as = 3\ then\ 0$
$else\ 1)$

fun *mv_box_stage2* :: $nat \times nat\ list \Rightarrow nat \Rightarrow nat$
where
$mv\_box\_stage2\ (as, lm)\ m = (lm\ !\ m)$

fun *mv_box_stage3* :: $nat \times nat\ list \Rightarrow nat \Rightarrow nat$
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* :: $((nat \times nat\ list) \times nat) \Rightarrow (nat \times nat \times nat)$
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* :: $((nat \times nat) \times nat \times nat)\ set$
where
lex_pair = *less_than* <∗*lex*∗> *less_than*

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

definition *mv_box_LE* ::
$(((nat \times nat\ list) \times nat) \times ((nat \times nat\ list) \times nat))\ set$
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*:
$[\![m < length\ initlm;\ n < length\ initlm;\ m \neq n]\!] \Longrightarrow$
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*: $x\ \#\ x\uparrow b\ @\ ys = x\uparrow(Suc\ b)\ @\ ys$
⟨*proof*⟩

lemma *exists_smaller_in_list0*[*simp*]:
$[\![m \neq n;\ m < length\ initlm;\ n < length\ initlm;$
$k + l = initlm\ !\ m + initlm\ !\ n;\ k \leq initlm\ !\ m;\ 0 < k]\!]$
$\Longrightarrow \exists\, ka\ la.\ initlm[m := k, n := l, m := k - Suc\ 0] =$
$initlm[m := ka, n := la] \wedge$
$Suc\ (ka + la) = initlm\ !\ m + initlm\ !\ n \wedge$

$ka < initlm \,!\, m$
$\langle proof \rangle$

lemma *exists_smaller_in_list1*[*simp*]:
$[\![m \neq n;\ m < length\ initlm;\ n < length\ initlm;$
$Suc\ (k + l) = initlm \,!\, m + initlm \,!\, n;$
$k < initlm \,!\, m]\!]$
$\Longrightarrow \exists\, ka\ la.\ initlm[m := k,\ n := l,\ n := Suc\ l] =$
$initlm[m := ka,\ n := la] \wedge$
$ka + la = initlm \,!\, m + initlm \,!\, n \wedge$
$ka \leq initlm \,!\, m$
$\langle proof \rangle$

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

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

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

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

lemma *length_mvbox*[*simp*]: $length\ (mv\_box\ m\ n) = 3$
$\langle proof \rangle$

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

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* ↑ (*ft* − *Suc* (*length xs*)) @ *map* (λ*i*. *rec_exec i xs*) (*take n gs*) @ *0* ↑ (*length gs* − *n*) @ *0* # *0* ↑ *length xs* @ *anything*)
[*ft* + *n* − *length xs* := *rec_exec* (*gs* ! *n*) *xs*, *0* := *0*] =
0 ↑ (*ft* − *length xs*) @ *map* (λ*i*. *rec_exec i xs*) (*take n gs*) @ *rec_exec* (*gs* ! *n*) *xs* # *0* ↑ (*length gs* − *Suc n*) @ *0* # *0* ↑ *length xs* @ *anything*
⟨*proof*⟩

lemma *compile_cn_gs_correct′*:
assumes
g_cond: ∀ *g*∈*set* (*take n 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 *ft*: *ft* = *max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ‘ *rec_ci* ‘ *set gs*)))
shows
{|λ*nl*. *nl* = *xs* @ *0* # *0* ↑ (*ft* + *length gs*) @ *anything*|}
cn_merge_gs (*map rec_ci* (*take n gs*)) *ft*
{|λ*nl*. *nl* = *xs* @ *0* ↑ (*ft* − *length xs*) @
map (λ*i*. *rec_exec i xs*) (*take n gs*) @ *0*↑(*length gs* − *n*) @ *0* ↑ *Suc* (*length xs*) @ *anything*|}
⟨*proof*⟩

lemma *compile_cn_gs_correct*:
assumes
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 *ft*: *ft* = *max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ‘ *rec_ci* ‘ *set gs*)))
shows
{|λ*nl*. *nl* = *xs* @ *0* # *0* ↑ (*ft* + *length gs*) @ *anything*|}
cn_merge_gs (*map rec_ci gs*) *ft*
{|λ*nl*. *nl* = *xs* @ *0* ↑ (*ft* − *length xs*) @
map (λ*i*. *rec_exec i xs*) *gs* @ *0* ↑ *Suc* (*length xs*) @ *anything*|}
⟨*proof*⟩

lemma *length_mvboxes*[*simp*]: *length* (*mv_boxes aa ba n*) = *3*∗*n*
⟨*proof*⟩

lemma *exp_suc*: *a*↑*Suc b* = *a*↑*b* @ [*a*]
⟨*proof*⟩

lemma *last_0*[*simp*]:
$[\![$*Suc n* ≤ *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*]: $length\ lm1 = aa \Longrightarrow$
$(lm1\ @\ 0{\uparrow}n\ @\ last\ lm2\ \#\ lm3\ @\ butlast\ lm2\ @\ 0\ \#\ lm4)\ !\ (aa + n) = last\ lm2$
$\langle proof \rangle$

lemma *arith_as_simp*[*simp*]: $[\![Suc\ n \le ba - aa;\ aa < ba]\!] \Longrightarrow$
$(ba < Suc\ (aa + (ba - Suc\ (aa + n) + n))) = False$
$\langle proof \rangle$

lemma *butlast_elem*[*simp*]: $[\![Suc\ n \le ba - aa;\ aa < ba;\ length\ lm1 = aa;$
$length\ lm2 = Suc\ n;\ length\ lm3 = ba - Suc\ (aa + n)]\!]$
$\Longrightarrow (lm1\ @\ 0{\uparrow}n\ @\ last\ lm2\ \#\ lm3\ @\ butlast\ lm2\ @\ 0\ \#\ lm4)\ !\ (ba + n) = 0$
$\langle proof \rangle$

lemma *update_butlast_eq0*[*simp*]:
$[\![Suc\ n \le ba - aa;\ aa < ba;\ length\ lm1 = aa;\ length\ lm2 = Suc\ n;$
$length\ lm3 = ba - Suc\ (aa + n)]\!]$
$\Longrightarrow (lm1\ @\ 0{\uparrow}n\ @\ last\ lm2\ \#\ lm3\ @\ butlast\ lm2\ @\ (0{::}nat)\ \#\ lm4)$
$[ba + n := last\ lm2,\ aa + n := 0] =$
$lm1\ @\ 0\ \#\ 0{\uparrow}n\ @\ lm3\ @\ lm2\ @\ lm4$
$\langle proof \rangle$

lemma *update_butlast_eq1*[*simp*]:
$[\![Suc\ (length\ lm1 + n) \le ba;\ length\ lm2 = Suc\ n;\ length\ lm3 = ba - Suc\ (length\ lm1 + n);$
$\neg\ ba - Suc\ (length\ lm1) < ba - Suc\ (length\ lm1 + n);\ \neg\ ba + n - length\ lm1 < n]\!]$
$\Longrightarrow (0{::}nat) \uparrow n\ @\ (last\ lm2\ \#\ lm3\ @\ butlast\ lm2\ @\ 0\ \#\ lm4)[ba - length\ lm1 := last\ lm2,$
$0 := 0] =$
$0\ \#\ 0 \uparrow n\ @\ lm3\ @\ lm2\ @\ lm4$
$\langle proof \rangle$

lemma *mv_boxes_correct*:
$[\![aa + n \le ba;\ ba > aa;\ length\ lm1 = aa;\ length\ lm2 = n;\ length\ lm3 = ba - aa - n]\!]$
$\Longrightarrow \{\!|\lambda\ nl.\ nl = lm1\ @\ lm2\ @\ lm3\ @\ 0{\uparrow}n\ @\ lm4|\!\}\ (mv\_boxes\ aa\ ba\ n)$
$\{\!|\lambda\ nl.\ nl = lm1\ @\ 0{\uparrow}n\ @\ lm3\ @\ lm2\ @\ lm4|\!\}$
$\langle proof \rangle$

lemma *update_butlast_eq2*[*simp*]:
$[\![Suc\ n \le aa - length\ lm1;\ length\ lm1 < aa;$
$length\ lm2 = aa - Suc\ (length\ lm1 + n);$
$length\ lm3 = Suc\ n;$
$\neg\ aa - Suc\ (length\ lm1) < aa - Suc\ (length\ lm1 + n);$
$\neg\ aa + n - length\ lm1 < n]\!]$
$\Longrightarrow butlast\ lm3\ @\ ((0{::}nat)\ \#\ lm2\ @\ 0 \uparrow n\ @\ last\ lm3\ \#\ lm4)[0 := last\ lm3,\ aa - length\ lm1$
$:= 0] = lm3\ @\ lm2\ @\ 0\ \#\ 0 \uparrow n\ @\ lm4$
$\langle proof \rangle$

lemma *mv_boxes_correct2*:
$[\![n \le aa - ba;$
$ba < aa;$
$length\ (lm1{::}nat\ list) = ba;$

$\quad length\ (lm2{::}nat\ list) = aa - ba - n;$
$\quad length\ (lm3{::}nat\ list) = n]\!]$
$\Longrightarrow \{\!|\lambda\ nl.\ nl = lm1\ @\ 0{\uparrow}n\ @\ lm2\ @\ lm3\ @\ lm4|\!\}$
$\qquad (mv\_boxes\ aa\ ba\ n)$
$\quad \{\!|\lambda\ nl.\ nl = lm1\ @\ lm3\ @\ lm2\ @\ 0{\uparrow}n\ @\ lm4|\!\}$
$\langle proof \rangle$

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

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

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

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

declare *max_less_iff_conj*[*simp del*]

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

lemma *length_empty_boxes*[*simp*]: *length* (*empty_boxes n*) = *2*∗*n*
⟨*proof*⟩

lemma *empty_one_box_correct*:
{|λ*nl*. *nl* = *0* ↑ *n* @ *x* # *lm*|} [*Dec n 2*, *Goto 0*] {|λ*nl*. *nl* = *0* # *0* ↑ *n* @ *lm*|}
⟨*proof*⟩

lemma *empty_boxes_correct*:
length lm ≥ *n* ⟹
{|λ *nl*. *nl* = *lm*|} *empty_boxes n* {|λ *nl*. *nl* = *0*↑*n* @ *drop n lm*|}
⟨*proof*⟩

lemma *insert_dominated*[*simp*]: *length gs* ≤ *ffp* ⟹
length gs + (*max xs* (*Max* (*insert ffp* (*x1* ' *x2* ' *set gs*))) − *length gs*) =
max xs (*Max* (*insert ffp* (*x1* ' *x2* ' *set gs*)))
⟨*proof*⟩

lemma *clean_paras*:
ffp ≥ *length gs* ⟹
{|λ*nl*. *nl* = *map* (λ*i*. *rec_exec i xs*) *gs* @
0 ↑ (*max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ' *rec_ci* ' *set gs*))) − *length gs*) @
rec_exec (*Cn* (*length xs*) *f gs*) *xs* # *0* ↑ *length gs* @ *xs* @ *anything*|}
empty_boxes (*length gs*)
{|λ*nl*. *nl* = *0* ↑ *max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ' *rec_ci* ' *set gs*))) @
rec_exec (*Cn* (*length xs*) *f gs*) *xs* # *0* ↑ *length gs* @ *xs* @ *anything*|}
⟨*proof*⟩

lemma *restore_rs*:
{|λ*nl*. *nl* = *0* ↑ *max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ' *rec_ci* ' *set gs*))) @
rec_exec (*Cn* (*length xs*) *f gs*) *xs* # *0* ↑ *length gs* @ *xs* @ *anything*|}
mv_box (*max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ' *rec_ci* ' *set gs*)))) (*length xs*)
{|λ*nl*. *nl* = *0* ↑ *length 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 xs*)) @
0 ↑ *length gs* @ *xs* @ *anything*|}
⟨*proof*⟩

lemma *restore_orgin_paras*:
{|λ*nl*. *nl* = *0* ↑ *length 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 xs*) @ *0* ↑ *length gs* @ *xs* @ *anything*|}
mv_boxes (*Suc* (*max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ' *rec_ci* ' *set gs*)))

+ length gs)) 0 (length xs)
⦃λnl. 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 *f_ind*:
⋀ anything r. rec_exec f (map (λg. rec_exec g xs) gs) = rec_exec (Cn (length xs) f gs) xs ⟹
⦃λnl. nl = map (λg. rec_exec g xs) gs @ 0 ↑ (ffp − far) @ anything⦄ fap
⦃λnl. nl = map (λg. 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 (λg. rec_exec g xs) gs)
and *g_cond*: ∀ g∈set gs. terminate g xs ∧
(∀ x xa xb. rec_ci g = (x, xa, xb) ⟶
(∀ xc. ⦃λnl. nl = xs @ 0 ↑ (xb − xa) @ xc⦄ x ⦃λnl. nl = xs @ rec_exec g xs # 0 ↑ (xb − Suc xa) @ xc⦄))
shows
⦃λnl. 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)))))))
⦃λnl. 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 (λg. rec_exec g xs) gs)
and *f_ind*: ⋀ap arity fp anything.
rec_ci f = (ap, arity, fp)
⟹ ⦃λnl. nl = map (λg. rec_exec g xs) gs @ 0 ↑ (fp − arity) @ anything⦄ ap
⦃λnl. nl = map (λg. rec_exec g xs) gs @ rec_exec f (map (λg. 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. ⦃λnl. nl = xs @ 0 ↑ (xb − xa) @ xc⦄ x ⦃λnl. nl

$= xs\ @\ rec\_exec\ g\ xs\ \#\ 0 \uparrow (xb - Suc\ xa)\ @\ xc \}))$
  and *compile*: $rec\_ci\ (Cn\ n\ f\ gs) = (ap, arity, fp)$
  and *len*: $length\ xs = n$
 shows $\{\lambda nl.\ nl = xs\ @\ 0 \uparrow (fp - arity)\ @\ anything\}\ ap\ \{\lambda nl.\ nl = xs\ @\ rec\_exec\ (Cn\ n\ f\ gs)$
$xs\ \#\ 0 \uparrow (fp - Suc\ arity)\ @\ anything\}$
$\langle proof \rangle$

3.4.2.5 Correctness of compilation for constructor Pr

lemma *mv_box_correct_simp*[*simp*]:
 $[\![length\ xs = n; ft = max\ (n{+}3)\ (max\ fft\ gft)]\!]$
 $\Longrightarrow \{\lambda nl.\ nl = xs\ @\ 0\ \#\ 0 \uparrow (ft - n)\ @\ anything\}\ mv\_box\ n\ ft$
 $\{\lambda nl.\ nl = xs\ @\ 0\ \#\ 0 \uparrow (ft - n)\ @\ anything\}$
 $\langle proof \rangle$

lemma *length_under_max*[*simp*]: $length\ xs < max\ (length\ xs + 3)\ fft$
 $\langle proof \rangle$

lemma *save_init_rs*:
 $[\![length\ xs = n; ft = max\ (n{+}3)\ (max\ fft\ gft)]\!]$
 $\Longrightarrow \{\lambda nl.\ nl = xs\ @\ rec\_exec\ f\ xs\ \#\ 0 \uparrow (ft - n)\ @\ anything\}\ mv\_box\ n\ (Suc\ n)$
 $\{\lambda nl.\ nl = xs\ @\ 0\ \#\ rec\_exec\ f\ xs\ \#\ 0 \uparrow (ft - Suc\ n)\ @\ anything\}$
 $\langle proof \rangle$

lemma *less_then_max_plus2*[*simp*]: $n + (2{::}nat) < max\ (n + 3)\ x$
 $\langle proof \rangle$

lemma *less_then_max_plus3*[*simp*]: $n < max\ (n + (3{::}nat))\ x$
 $\langle proof \rangle$

lemma *mv_box_max_plus_3_correct*[*simp*]:
 $length\ xs = n \Longrightarrow$
 $\{\lambda nl.\ nl = xs\ @\ x\ \#\ 0 \uparrow (max\ (n + (3{::}nat))\ (max\ fft\ gft) - n)\ @\ anything\}\ mv\_box\ n\ (max$
$(n + 3)\ (max\ fft\ gft))$
 $\{\lambda nl.\ nl = xs\ @\ 0 \uparrow (max\ (n + 3)\ (max\ fft\ gft) - n)\ @\ x\ \#\ anything\}$
$\langle proof \rangle$

lemma *max_less_suc_suc*[*simp*]: $max\ n\ (Suc\ n) < Suc\ (Suc\ (max\ (n + 3)\ x + anything - Suc$
$0))$
 $\langle proof \rangle$

lemma *suc_less_plus_3*[*simp*]: $Suc\ n < max\ (n + 3)\ x$
 $\langle proof \rangle$

lemma *mv_box_ok_suc_simp*[*simp*]:
 $length\ xs = n$
 $\Longrightarrow \{\lambda nl.\ nl = xs\ @\ rec\_exec\ f\ xs\ \#\ 0 \uparrow (max\ (n + 3)\ (max\ fft\ gft) - Suc\ n)\ @\ x\ \#\ anything\}$
$mv\_box\ n\ (Suc\ n)$
 $\{\lambda nl.\ nl = xs\ @\ 0\ \#\ rec\_exec\ f\ xs\ \#\ 0 \uparrow (max\ (n + 3)\ (max\ fft\ gft) - Suc\ (Suc\ n))\ @\ x\ \#$
$anything\}$

$\langle proof \rangle$

lemma *abc_append_first_steps_eq_pre*:
assumes *notfinal*: $abc\_notfinal\ (abc\_steps\_l\ (0, lm)\ \ A\ n)\ A$
and *notnull*: $A \neq []$
shows $abc\_steps\_l\ (0, lm)\ (A\ @\ B)\ n = abc\_steps\_l\ (0, lm)\ A\ n$
$\langle proof \rangle$

lemma *abc_append_first_step_eq_pre*:
$st < length\ A$
$\Longrightarrow abc\_step\_l\ (st, lm)\ (abc\_fetch\ st\ (A\ @\ B)) =$
$abc\_step\_l\ (st, lm)\ (abc\_fetch\ st\ A)$
$\langle proof \rangle$

lemma *abc_append_first_steps_halt_eq'*:
assumes *final*: $abc\_steps\_l\ (0, lm)\ A\ n = (length\ A, lm')$
and *notnull*: $A \neq []$
shows $\exists\ n'.\ abc\_steps\_l\ (0, lm)\ (A\ @\ B)\ n' = (length\ A, lm')$
$\langle proof \rangle$

lemma *abc_append_first_steps_halt_eq*:
assumes *final*: $abc\_steps\_l\ (0, lm)\ A\ n = (length\ A, lm')$
shows $\exists\ n'.\ abc\_steps\_l\ (0, lm)\ (A\ @\ B)\ n' = (length\ A, lm')$
$\langle proof \rangle$

lemma *suc_suc_max_simp*[*simp*]: $Suc\ (Suc\ (max\ (xs + 3)\ fft - Suc\ (Suc\ (\ xs))))$
$= max\ (\ xs + 3)\ fft - (\ xs)$
$\langle proof \rangle$

lemma *contract_dec_ft_length_plus_7*[*simp*]: $[\![ft = max\ (n + 3)\ (max\ fft\ gft);\ length\ xs = n]\!]$
$\Longrightarrow$
$\{\!|\lambda nl.\ nl = xs\ @\ (x - Suc\ y)\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [x - Suc\ y])\ \#\ 0 \uparrow (ft - Suc\ (Suc\ n))\ @\ Suc\ y\ \#\ anything|\!\}$
$[Dec\ ft\ (length\ gap + 7)]$
$\{\!|\lambda nl.\ nl = xs\ @\ (x - Suc\ y)\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [x - Suc\ y])\ \#\ 0 \uparrow (ft - Suc\ (Suc\ n))\ @\ y\ \#\ anything|\!\}$
$\langle proof \rangle$

lemma *adjust_paras'*:
$length\ xs = n \Longrightarrow \{\!|\lambda nl.\ nl = xs\ @\ x\ \#\ y\ \#\ anything|\!\}\ \ [Inc\ n]\ [+]\ [Dec\ (Suc\ n)\ 2, Goto\ 0]$
$\{\!|\lambda nl.\ nl = xs\ @\ Suc\ x\ \#\ 0\ \#\ anything|\!\}$
$\langle proof \rangle$

lemma *adjust_paras*:
$length\ xs = n \Longrightarrow \{\!|\lambda nl.\ nl = xs\ @\ x\ \#\ y\ \#\ anything|\!\}\ \ [Inc\ n, Dec\ (Suc\ n)\ 3, Goto\ (Suc\ 0)]$
$\{\!|\lambda nl.\ nl = xs\ @\ Suc\ x\ \#\ 0\ \#\ anything|\!\}$
$\langle proof \rangle$

lemma *rec_ci_SucSuc_n*[*simp*]: $[\![rec\_ci\ g = (gap, gar, gft);\ \forall\ y{<}x.\ terminate\ g\ (xs\ @\ [y, rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])]);$

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 ∃ *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 ∃ *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*]:
$length\ xs = n \Longrightarrow abc\_lm\_s\ (xs\ @\ x\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [x])\ \#\ 0 \uparrow (max\ (n + 3)$
$(max\ fft\ gft) - Suc\ (Suc\ n))\ @\ 0\ \#\ anything)\ (max\ (n + 3)\ (max\ fft\ gft))\ 0 =$
$xs\ @\ x\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [x])\ \#\ 0 \uparrow (max\ (n + 3)\ (max\ fft\ gft) - Suc\ n)\ @\ anything$
$\langle proof \rangle$

lemma *index_at_zero_elem*[*simp*]:
$(xs\ @\ x\ \#\ re\ \#\ 0 \uparrow (max\ (length\ xs + 3)$
$(max\ fft\ gft) - Suc\ (Suc\ (length\ xs)))\ @\ 0\ \#\ anything)\ !$
$max\ (length\ xs + 3)\ (max\ fft\ gft) = 0$
$\langle proof \rangle$

lemma *pr_loop_correct*:
assumes *less*: $y \le x$
and *len*: $length\ xs = n$
and *compile_g*: $rec\_ci\ g = (gap, gar, gft)$
and *termi_g*: $\forall\ y{<}x.\ terminate\ g\ (xs\ @\ [y, rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])])$
and *g_ind*: $\forall\ y{<}x.\ (\forall anything.\ \{\!|\lambda nl.\ nl = xs\ @\ y\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])\ \#\ 0 \uparrow (gft - gar)\ @\ anything|\!\}\ gap$
$\{\!|\lambda nl.\ nl = xs\ @\ y\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])\ \#\ rec\_exec\ g\ (xs\ @\ [y, rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])])\ \#\ 0 \uparrow (gft - Suc\ gar)\ @\ anything|\!\})$
shows $\{\!|\lambda nl.\ nl = xs\ @\ (x - y)\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [x - y])\ \#\ 0 \uparrow (max\ (n + 3)\ (max\ fft\ gft) - Suc\ (Suc\ n))\ @\ y\ \#\ anything|\!\}$
$([Dec\ (max\ (n + 3)\ (max\ fft\ gft))\ (length\ gap + 7)]\ [+]\ (gap\ [+]\ [Inc\ n, Dec\ (Suc\ n)\ 3, Goto\ (Suc\ 0)]))\ @\ [Dec\ (Suc\ (Suc\ n))\ 0, Inc\ (Suc\ n), Goto\ (length\ gap + 4)]$
$\{\!|\lambda nl.\ nl = xs\ @\ x\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [x])\ \#\ 0 \uparrow (max\ (n + 3)\ (max\ fft\ gft) - Suc\ n)\ @\ anything|\!\}$
$\langle proof \rangle$

lemma *compile_pr_correct′*:
assumes *termi_g*: $\forall\ y{<}x.\ terminate\ g\ (xs\ @\ [y, rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])])$
and *g_ind*:
$\forall\ y{<}x.\ (\forall anything.\ \{\!|\lambda nl.\ nl = xs\ @\ y\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])\ \#\ 0 \uparrow (gft - gar)\ @\ anything|\!\}\ gap$
$\{\!|\lambda nl.\ nl = xs\ @\ y\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])\ \#\ rec\_exec\ g\ (xs\ @\ [y, rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])])\ \#\ 0 \uparrow (gft - Suc\ gar)\ @\ anything|\!\})$
and *termi_f*: $terminate\ f\ xs$
and *f_ind*: $\bigwedge anything.\ \{\!|\lambda nl.\ nl = xs\ @\ 0 \uparrow (fft - far)\ @\ anything|\!\}\ fap\ \{\!|\lambda nl.\ nl = xs\ @\ rec\_exec\ f\ xs\ \#\ 0 \uparrow (fft - Suc\ far)\ @\ anything|\!\}$
and *len*: $length\ xs = n$
and *compile1*: $rec\_ci\ f = (fap, far, fft)$
and *compile2*: $rec\_ci\ g = (gap, gar, gft)$
shows
$\{\!|\lambda nl.\ nl = xs\ @\ x\ \#\ 0 \uparrow (max\ (n + 3)\ (max\ fft\ gft) - n)\ @\ anything|\!\}$
$mv\_box\ n\ (max\ (n + 3)\ (max\ fft\ gft))\ [+]$
$(fap\ [+]\ (mv\_box\ n\ (Suc\ n)\ [+]$
$([Dec\ (max\ (n + 3)\ (max\ fft\ gft))\ (length\ gap + 7)]\ [+]\ (gap\ [+]\ [Inc\ n, Dec\ (Suc\ n)\ 3, Goto\ (Suc\ 0)])\ @$
$[Dec\ (Suc\ (Suc\ n))\ 0, Inc\ (Suc\ n), Goto\ (length\ gap + 4)])))$
$\{\!|\lambda nl.\ nl = xs\ @\ x\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [x])\ \#\ 0 \uparrow (max\ (n + 3)\ (max\ fft\ gft) - Suc\ n)$

$@$ *anything*$\}$
$\langle proof \rangle$

lemma *compile_pr_correct*:
assumes *g_ind*: $\forall y<x.$ *terminate g* $(xs\ @\ [y,\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])])\ \wedge$
$(\forall x\ xa\ xb.\ rec\_ci\ g = (x, xa, xb) \longrightarrow$
$(\forall xc.\ \{\lambda nl.\ nl = xs\ @\ y\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])\ \#\ 0 \uparrow (xb - xa)\ @\ xc\}\ x$
$\{\lambda nl.\ nl = xs\ @\ y\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])\ \#\ rec\_exec\ g\ (xs\ @\ [y,\ rec\_exec\ (Pr\ n\ f\ g)$
$(xs\ @\ [y])])\ \#\ 0 \uparrow (xb - Suc\ xa)\ @\ xc\}))$
and *termi_f*: *terminate f xs*
and *f_ind*:
$\bigwedge ap\ arity\ fp\ anything.$
$rec\_ci\ f = (ap, arity, fp) \Longrightarrow \{\lambda nl.\ nl = xs\ @\ 0 \uparrow (fp - arity)\ @\ anything\}\ ap\ \{\lambda nl.\ nl = xs$
$@\ rec\_exec\ f\ xs\ \#\ 0 \uparrow (fp - Suc\ arity)\ @\ anything\}$
and *len*: *length* $xs = n$
and *compile*: $rec\_ci\ (Pr\ n\ f\ g) = (ap, arity, fp)$
shows $\{\lambda nl.\ nl = xs\ @\ x\ \#\ 0 \uparrow (fp - arity)\ @\ anything\}\ ap\ \{\lambda nl.\ nl = xs\ @\ x\ \#\ rec\_exec\ (Pr$
$n\ f\ g)\ (xs\ @\ [x])\ \#\ 0 \uparrow (fp - Suc\ arity)\ @\ anything\}$
$\langle proof \rangle$

3.4.2.6 Correctness of compilation for constructor Mn

fun *mn_ind_inv* ::
$nat \times nat\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
$mn\_ind\_inv\ (as, lm')\ ss\ x\ rsx\ suf\_lm\ lm =$
$(if\ as = ss\ then\ lm' = lm\ @\ x\ \#\ rsx\ \#\ suf\_lm$
$else\ if\ as = ss + 1\ then$
$\exists y.\ (lm' = lm\ @\ x\ \#\ y\ \#\ suf\_lm) \wedge y \leq rsx$
$else\ if\ as = ss + 2\ then$
$\exists y.\ (lm' = lm\ @\ x\ \#\ y\ \#\ suf\_lm) \wedge y \leq rsx$
$else\ if\ as = ss + 3\ then\ lm' = lm\ @\ x\ \#\ 0\ \#\ suf\_lm$
$else\ if\ as = ss + 4\ then\ lm' = lm\ @\ Suc\ x\ \#\ 0\ \#\ suf\_lm$
$else\ if\ as = 0\ then\ lm' = lm\ @\ Suc\ x\ \#\ 0\ \#\ suf\_lm$
$else\ False$
)

fun *mn_stage1* :: $nat \times nat\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat$
where
$mn\_stage1\ (as, lm)\ ss\ n =$
$(if\ as = 0\ then\ 0$
$else\ if\ as = ss + 4\ then\ 1$
$else\ if\ as = ss + 3\ then\ 2$
$else\ if\ as = ss + 2 \vee as = ss + 1\ then\ 3$
$else\ if\ as = ss\ then\ 4$
$else\ 0$
)

fun *mn_stage2* :: $nat \times nat\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat$
where

$mn\_stage2\ (as, lm)\ ss\ n =$
$\quad (if\ as = ss + 1 \lor as = ss + 2\ then\ (lm\ !\ (Suc\ n))$
$\quad else\ 0)$

fun $mn\_stage3 :: nat \times nat\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat$
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 $wf\_mn\_le[intro]$: $wf\ mn\_LE$
$\langle proof \rangle$

declare $mn\_ind\_inv.simps[simp\ del]$

lemma $put\_in\_tape\_small\_enough0[simp]$:
$0 < rsx \Longrightarrow$
$\exists y.\ (xs\ @\ x\ \#\ rsx\ \#\ anything)[Suc\ (length\ xs) := rsx - Suc\ 0] = xs\ @\ x\ \#\ y\ \#\ anything \land y \leq rsx$
$\langle proof \rangle$

lemma $put\_in\_tape\_small\_enough1[simp]$:
$[\![y \leq rsx; 0 < y]\!]$
$\Longrightarrow \exists ya.\ (xs\ @\ x\ \#\ y\ \#\ anything)[Suc\ (length\ xs) := y - Suc\ 0] = xs\ @\ x\ \#\ ya\ \#\ anything \land ya \leq rsx$
$\langle proof \rangle$

lemma $abc\_comp\_null[simp]$: $(A\ [+]\ B = []) = (A = [] \land B = [])$
$\langle proof \rangle$

lemma $rec\_ci\_not\_null[simp]$: $(rec\_ci\ f \neq ([], a, b))$
$\langle proof \rangle$

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 $\exists\ stp.\ P\ (f\ stp) \wedge Q\ (f\ stp)$
$\langle proof \rangle$

lemma *abc_Hoare_haltE*:
$\{\!|\lambda\ nl.\ nl = lm1|\!\}\ p\ \{\!|\lambda\ nl.\ nl = lm2|\!\}$
$\Longrightarrow \exists\ stp.\ abc\_steps\_l\ (0, lm1)\ p\ stp = (length\ p, lm2)$
$\langle proof \rangle$

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 $\exists\ stp > 0.\ abc\_steps\_l\ (0, xs\ @\ x\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)\ (fap\ @\ B)\ stp =$
$(0, xs\ @\ Suc\ x\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)$
$\langle proof \rangle$

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: $\forall i \le x.\ (\forall xc.\ (\{\!|\lambda nl.\ nl = xs\ @\ i\ \#\ 0 \uparrow (fft - far)\ @\ xc|\!\}\ fap$
$\{\!|\lambda nl.\ nl = xs\ @\ i\ \#\ rec\_exec\ f\ (xs\ @\ [i])\ \#\ 0 \uparrow (fft - Suc\ far)\ @\ xc|\!\}))$
and $exec\_ge$: $\forall\ i \le x.\ rec\_exec\ f\ (xs\ @\ [i]) > 0$
and $compile$: $rec\_ci\ f = (fap, far, fft)$
and far: $far = Suc\ arity$
shows $\exists\ stp > x.\ abc\_steps\_l\ (0, xs\ @\ 0\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)\ (fap\ @\ B)\ stp =$
$(0, xs\ @\ Suc\ x\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)$
$\langle proof \rangle$

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: $\forall i \le x.\ (\forall xc.\ (\{\!|\lambda nl.\ nl = xs\ @\ i\ \#\ 0 \uparrow (fft - far)\ @\ xc|\!\}\ fap$
$\{\!|\lambda nl.\ nl = xs\ @\ i\ \#\ rec\_exec\ f\ (xs\ @\ [i])\ \#\ 0 \uparrow (fft - Suc\ far)\ @\ xc|\!\}))$
and $exec\_ge$: $\forall\ i \le x.\ rec\_exec\ f\ (xs\ @\ [i]) > 0$
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*⟩

3.4.2.7 Correctness of entire compilation process rec_ci

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* ⇒ *nat list* ⇒ *bool*
  where
  abc_list_crsp xs ys = (∃ *n*. *xs* = *ys* @ *0*↑*n* ∨ *ys* = *xs* @ *0*↑*n*)

lemma *abc_list_crsp_simp1*[*intro*]: *abc_list_crsp* (*lm* @ *0*↑*m*) *lm*
  ⟨*proof*⟩

lemma *abc_list_crsp_lm_v*:
  abc_list_crsp lma lmb ⟹ *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* ⟹ *P* ⟧ ⟹ *P*
  ⟨*proof*⟩

lemma *abc_list_crsp_simp*[*simp*]:
  ⟦*abc_list_crsp lma lmb*; *m* < *length lma*; *m* < *length lmb*⟧ ⟹
      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*⟧ ⟹
  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*⟧ ⟹
  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*⟧ ⟹
  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 ⟹
    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*′)⟧
   ⟹ *a*′ = *a* ∧ *abc_list_crsp lma*′ *lmb*′
  ⟨*proof*⟩

lemma *abc_list_crsp_steps*:
$[\![abc\_steps\_l\ (0,\ lm\ @\ 0{\uparrow}m)\ aprog\ stp = (a,\ lm');\ aprog \neq []]\!]$
$\Longrightarrow \exists\ lma.\ abc\_steps\_l\ (0,\ lm)\ aprog\ stp = (a,\ lma) \wedge$
$abc\_list\_crsp\ lm'\ lma$
$\langle proof \rangle$

lemma *list_crsp_simp2*: $abc\_list\_crsp\ (lm1\ @\ 0{\uparrow}n)\ lm2 \Longrightarrow abc\_list\_crsp\ lm1\ lm2$
$\langle proof \rangle$

lemma *recursive_compile_correct_norm′*:
$[\![rec\_ci\ f = (ap,\ arity,\ ft);$
$terminate\ f\ args]\!]$
$\Longrightarrow \exists\ stp\ rl.\ (abc\_steps\_l\ (0,\ args)\ ap\ stp) = (length\ ap,\ rl) \wedge abc\_list\_crsp\ (args\ @\ [rec\_exec$
$f\ args])\ rl$
$\langle proof \rangle$

lemma *find_exponent_rec_exec*[*simp*]:
assumes *a*: $args\ @\ [rec\_exec\ f\ args] = lm\ @\ 0 \uparrow n$
and *b*: $length\ args < length\ lm$
shows $\exists\, m.\ lm = args\ @\ rec\_exec\ f\ args\ \#\ 0 \uparrow m$
$\langle proof \rangle$

lemma *find_exponent_complex*[*simp*]:
$[\![args\ @\ [rec\_exec\ f\ args] = lm\ @\ 0 \uparrow n;\ \neg\ length\ args < length\ lm]\!]$
$\Longrightarrow \exists\, m.\ (lm\ @\ 0 \uparrow (length\ args - length\ lm)\ @\ [Suc\ 0])[length\ args := 0] =$
$args\ @\ rec\_exec\ f\ args\ \#\ 0 \uparrow m$
$\langle proof \rangle$

lemma *compile_append_dummy_correct*:
assumes *compile*: $rec\_ci\ f = (ap,\ ary,\ fp)$
and *termi*: $terminate\ f\ args$
shows $\{\!|\lambda\ nl.\ nl = args|\!\}\ (ap\ [+]\ dummy\_abc\ (length\ args))\ \{\!|\lambda\ nl.\ (\exists\ m.\ nl = args\ @\ rec\_exec$
$f\ args\ \#\ 0{\uparrow}m)|\!\}$
$\langle proof \rangle$

lemma *cn_merge_gs_split*:
$[\![i < length\ gs;\ rec\_ci\ (gs!i) = (ga,\ gb,\ gc)]\!] \Longrightarrow$
$cn\_merge\_gs\ (map\ rec\_ci\ gs)\ p = cn\_merge\_gs\ (map\ rec\_ci\ (take\ i\ gs))\ p\ [+]\ (ga\ [+]$
$mv\_box\ gb\ (p + i))\ [+]\ cn\_merge\_gs\ (map\ rec\_ci\ (drop\ (Suc\ i)\ gs))\ (p + Suc\ i)$
$\langle proof \rangle$

lemma *cn_unhalt_case*:
assumes *compile1*: $rec\_ci\ (Cn\ n\ f\ gs) = (ap,\ ar,\ ft) \wedge length\ args = ar$
and *g*: $i < length\ gs$
and *compile2*: $rec\_ci\ (gs!i) = (gap,\ gar,\ gft) \wedge gar = length\ args$
and *g_unhalt*: $\bigwedge anything.\ \{\!|\lambda\ nl.\ nl = args\ @\ 0{\uparrow}(gft - gar)\ @\ anything|\!\}\ gap \uparrow$
and *g_ind*: $\bigwedge apj\ arj\ ftj\ j\ anything.\ [\![j < i;\ rec\_ci\ (gs!j) = (apj,\ arj,\ ftj)]\!]$
$\Longrightarrow \{\!|\lambda\ nl.\ nl = args\ @\ 0{\uparrow}(ftj - arj)\ @\ anything|\!\}\ apj\ \{\!|\lambda\ nl.\ nl = args\ @\ rec\_exec\ (gs!j)\ args$
$\#\ 0{\uparrow}(ftj - Suc\ arj)\ @\ anything|\!\}$

and *all_termi*: $\forall$ $j<i$. *terminate* (*gs*!*j*) *args*
shows $\{\!|\lambda$ *nl*. *nl* = *args* @ $0\uparrow(ft - ar)$ @ *anything*$|\!\}$ *ap* $\uparrow$
$\langle$*proof*$\rangle$

lemma *mn_unhalt_case'*:
assumes *compile*: *rec_ci f* = (*a*, *b*, *c*)
and *all_termi*: $\forall$ *i*. *terminate f* (*args* @ [*i*]) $\wedge$ $0 <$ *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 $\{\!|\lambda$*nl*. *nl* = *args* @ $0\uparrow$ (*max* (*Suc* (*length args*)) *c* $-$ *length args*) @ *anything*$|\!\}$
a @ *B* $\uparrow$
$\langle$*proof*$\rangle$

lemma *mn_unhalt_case*:
assumes *compile*: *rec_ci* (*Mn n f*) = (*ap*, *ar*, *ft*) $\wedge$ *length args* = *ar*
and *all_term*: $\forall$ *i*. *terminate f* (*args* @ [*i*]) $\wedge$ *rec_exec f* (*args* @ [*i*]) > 0
shows $\{\!|$ (λ *nl*. *nl* = *args* @ $0\uparrow(ft - ar)$ @ *anything*) $|\!\}$ *ap* $\uparrow$
$\langle$*proof*$\rangle$

3.5 Compilers composed: Compiling Recursive Functions into Turing Machines

fun *tm_of_rec* :: *recf* $\Rightarrow$ *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_n_tm 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 $\exists$ *stp m l*. *steps0* (*Suc 0*, *Bk* # *Bk* # *ires*, <*args*> @ *Bk*$\uparrow$*rn*)
(*tp* @ *shift* (*mopup_n_tm* (*length args*)) (*length tp div 2*)) *stp* = (*0*, *Bk*$\uparrow$*m* @ *Bk* # *Bk* # *ires*,
Oc$\uparrow$*Suc* (*rec_exec recf args*) @ *Bk*$\uparrow$*l*)
$\langle$*proof*$\rangle$

lemma *recursive_compile_to_tm_correct2*:
assumes *termi*: *terminate recf args*
shows $\exists$ *stp m l*. *steps0* (*Suc 0*, [*Bk*, *Bk*], <*args*>) (*tm_of_rec recf*) *stp* =
(*0*, *Bk*$\uparrow$*Suc* (*Suc m*), *Oc*$\uparrow$*Suc* (*rec_exec recf args*) @ *Bk*$\uparrow$*l*)
$\langle$*proof*$\rangle$

lemma *recursive_compile_to_tm_correct3*:
assumes *termi*: *terminate recf args*
shows $\{\!|\lambda$ *tp*. *tp* =([*Bk*, *Bk*], <*args*>)$|\!\}$ (*tm_of_rec recf*)
$\{\!|\lambda$ *tp*. $\exists$ *k l*. *tp* = (*Bk*$\uparrow$ *k*, <*rec_exec recf args*> @ *Bk* $\uparrow$ *l*)$|\!\}$

$\langle proof \rangle$

3.5.1 Appending the mopup TM

lemma *list_all_suc_many*[*simp*]:
$list\_all\ (\lambda(acn, s).\ s \leq Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (2 * n))))))) \ xs \Longrightarrow$
$list\_all\ (\lambda(acn, s).\ s \leq Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (2 * n))))))))) \ xs$
$\langle proof \rangle$

lemma *shift_append*: $shift\ (xs\ @\ ys)\ n = shift\ xs\ n\ @\ shift\ ys\ n$
$\langle proof \rangle$

lemma *length_shift_mopup*[*simp*]: $length\ (shift\ (mopup\_n\_tm\ n)\ ss) = 4 * n + 12$
$\langle proof \rangle$

lemma *length_tm_even*[*intro*]: $length\ (tm\_of\ ap)\ mod\ 2 = 0$
$\langle proof \rangle$

lemma *tms_of_at_index*[*simp*]: $k < length\ ap \Longrightarrow tms\_of\ ap\ !\ k\ =$
$ci\ (layout\_of\ ap)\ (start\_of\ (layout\_of\ ap)\ k)\ (ap\ !\ k)$
$\langle proof \rangle$

lemma *start_of_suc_inc*:
$[\![k < length\ ap;\ ap\ !\ k = Inc\ n]\!] \Longrightarrow start\_of\ (layout\_of\ ap)\ (Suc\ k) =$
$start\_of\ (layout\_of\ ap)\ k + 2 * n + 9$
$\langle proof \rangle$

lemma *start_of_suc_dec*:
$[\![k < length\ ap;\ ap\ !\ k = (Dec\ n\ e)]\!] \Longrightarrow start\_of\ (layout\_of\ ap)\ (Suc\ k) =$
$start\_of\ (layout\_of\ ap)\ k + 2 * n + 16$
$\langle proof \rangle$

lemma *inc_state_all_le*:
$[\![k < length\ ap;\ ap\ !\ k = Inc\ n;$
$(a, b) \in set\ (shift\ (shift\ tinc\_b\ (2 * n))$
$(start\_of\ (layout\_of\ ap)\ k - Suc\ 0))]\!]$
$\Longrightarrow b \leq start\_of\ (layout\_of\ ap)\ (length\ ap)$
$\langle proof \rangle$

lemma *findnth_le*[*elim*]:
$(a, b) \in set\ (shift\ (findnth\ n)\ (start\_of\ (layout\_of\ ap)\ k - Suc\ 0))$
$\Longrightarrow b \leq Suc\ (start\_of\ (layout\_of\ ap)\ k + 2 * n)$
$\langle proof \rangle$

lemma *findnth_state_all_le1*:
$[\![k < length\ ap;\ ap\ !\ k = Inc\ n;$
$(a, b) \in$
$set\ (shift\ (findnth\ n)\ (start\_of\ (layout\_of\ ap)\ k - Suc\ 0))]\!]$
$\Longrightarrow b \leq start\_of\ (layout\_of\ ap)\ (length\ ap)$

⟨*proof*⟩

lemma *start_of_eq*: $\textit{length ap} < \textit{as} \Longrightarrow \textit{start\_of}\ (\textit{layout\_of ap})\ \textit{as} = \textit{start\_of}\ (\textit{layout\_of ap})\ (\textit{length ap})$
⟨*proof*⟩

lemma *start_of_all_le*: $\textit{start\_of}\ (\textit{layout\_of ap})\ \textit{as} \le \textit{start\_of}\ (\textit{layout\_of ap})\ (\textit{length ap})$
⟨*proof*⟩

lemma *findnth_state_all_le2*:
$[\![k < \textit{length ap};$
$\textit{ap} ! k = \textit{Dec n e};$
$(a, b) \in \textit{set}\ (\textit{shift}\ (\textit{findnth n})\ (\textit{start\_of}\ (\textit{layout\_of ap})\ k - \textit{Suc 0}))]\!]$
$\Longrightarrow b \le \textit{start\_of}\ (\textit{layout\_of ap})\ (\textit{length ap})$
⟨*proof*⟩

lemma *dec_state_all_le*[*simp*]:
$[\![k < \textit{length ap}; \textit{ap} ! k = \textit{Dec n e};$
$(a, b) \in \textit{set}\ (\textit{shift}\ (\textit{shift tdec\_b}\ (2 * n))$
$(\textit{start\_of}\ (\textit{layout\_of ap})\ k - \textit{Suc 0}))]\!]$
$\Longrightarrow b \le \textit{start\_of}\ (\textit{layout\_of ap})\ (\textit{length ap})$
⟨*proof*⟩

lemma *tms_any_less*:
$[\![k < \textit{length ap}; (a, b) \in \textit{set}\ (\textit{tms\_of ap} ! k)]\!] \Longrightarrow$
$b \le \textit{start\_of}\ (\textit{layout\_of ap})\ (\textit{length ap})$
⟨*proof*⟩

lemma *concat_in*: $i < \textit{length}\ (\textit{concat xs}) \Longrightarrow$
$\exists k < \textit{length xs}.\ \textit{concat xs} ! i \in \textit{set}\ (\textit{xs} ! k)$
⟨*proof*⟩

declare *length_concat*[*simp*]

lemma *in_tms*: $i < \textit{length}\ (\textit{tm\_of ap}) \Longrightarrow \exists\ k < \textit{length ap}.\ (\textit{tm\_of ap} ! i) \in \textit{set}\ (\textit{tms\_of ap} ! k)$
⟨*proof*⟩

lemma *all_le_start_of*: $\textit{list\_all}\ (\lambda(\textit{acn}, s).$
$s \le \textit{start\_of}\ (\textit{layout\_of ap})\ (\textit{length ap}))\ (\textit{tm\_of ap})$
⟨*proof*⟩

lemma *length_ci*:
$[\![k < \textit{length ap}; \textit{length}\ (\textit{ci ly y}\ (\textit{ap} ! k)) = 2 * \textit{qa}]\!]$
$\Longrightarrow \textit{layout\_of ap} ! k = \textit{qa}$
⟨*proof*⟩

lemma *ci_even*[*intro*]: $\textit{length}\ (\textit{ci ly y i})\ \textit{mod}\ 2 = 0$
⟨*proof*⟩

lemma *sum_list_ci_even*[*intro*]: $\textit{sum\_list}\ (\textit{map}\ (\textit{length} \circ (\lambda(x, y).\ \textit{ci ly x y}))\ \textit{zs})\ \textit{mod}\ 2 = 0$

$\langle proof \rangle$

lemma *zip_pre*:
$(length\ ys) \le length\ ap \Longrightarrow$
$zip\ ys\ ap = zip\ ys\ (take\ (length\ ys)\ (ap::'a\ list))$
$\langle proof \rangle$

lemma *length_start_of_tm*: $start\_of\ (layout\_of\ ap)\ (length\ ap) = Suc\ (length\ (tm\_of\ ap)\ div\ 2)$
$\langle proof \rangle$

lemma *list_all_add_6E*[*elim*]: $list\_all\ (\lambda(acn, s).\ s \le Suc\ q)\ xs$
$\Longrightarrow list\_all\ (\lambda(acn, s).\ s \le q + (2 * n + 6))\ xs$
$\langle proof \rangle$

lemma *mopup_b_12*[*simp*]: $length\ mopup\_b = 12$
$\langle proof \rangle$

lemma *mp_up_all_le*: $list\_all\ (\lambda(acn, s).\ s \le q + (2 * n + 6))$
$[(R, Suc\ (Suc\ (2 * n + q))), (R, Suc\ (2 * n + q)),$
$(L, 5 + 2 * n + q), (WB, Suc\ (Suc\ (Suc\ (2 * n + q)))), (R, 4 + 2 * n + q),$
$(WB, Suc\ (Suc\ (Suc\ (2 * n + q)))), (R, Suc\ (Suc\ (2 * n + q))),$
$(WB, Suc\ (Suc\ (Suc\ (2 * n + q)))), (L, 5 + 2 * n + q),$
$(L, 6 + 2 * n + q), (R, 0), (L, 6 + 2 * n + q)]$
$\langle proof \rangle$

lemma *mopup_le6*[*simp*]: $(a, b) \in set\ (mopup\_a\ n) \Longrightarrow b \le 2 * n + 6$
$\langle proof \rangle$

lemma *shift_le2*[*simp*]: $(a, b) \in set\ (shift\ (mopup\_n\_tm\ n)\ x)$
$\Longrightarrow b \le (2 * x + length\ (mopup\_n\_tm\ n))\ div\ 2$
$\langle proof \rangle$

lemma *mopup_ge2*[*intro*]: $2 \le x + length\ (mopup\_n\_tm\ n)$
$\langle proof \rangle$

lemma *mopup_even*[*intro*]: $(2 * x + length\ (mopup\_n\_tm\ n))\ mod\ 2 = 0$
$\langle proof \rangle$

lemma *mopup_div_2*[*simp*]: $b \le Suc\ x$
$\Longrightarrow b \le (2 * x + length\ (mopup\_n\_tm\ n))\ div\ 2$
$\langle proof \rangle$

3.5.2 A Turing Machine compiled from an Abacus program with mopup code appended is composable

lemma *composable_tm_from_abacus*: **assumes** $tp = tm\_of\ ap$
shows $composable\_tm0\ (tp\ @\ shift\ (mopup\_n\_tm\ n)\ (length\ tp\ div\ 2))$
$\langle proof \rangle$

3.5.3 A Turing Machine compiled from a recursive function is composable

lemma *composable_tm_from_recf*:
assumes *compile*: $tp = tm\_of\_rec\ recf$
shows *composable_tm0 tp*
$\langle proof \rangle$

end

Chapter 4

An alternative modelling of Recursive Functions

theory *Recs_alt_Def*
  imports *Main*
    HOL−Library.Nat_Bijection
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*⟩

4.1 Some auxiliary lemmas about the Recursive Functions Sigma and Pi

lemma *setprod_atMost_Suc*[*simp*]:

$(\prod i \leq Suc\ n.\ f\ i) = (\prod i \leq n.\ f\ i) * f(Suc\ n)$
$\langle proof \rangle$

lemma *setprod_lessThan_Suc*[*simp*]:
$(\prod i < Suc\ n.\ f\ i) = (\prod i < n.\ f\ i) * f\ n$
$\langle proof \rangle$

lemma *setsum_add_nat_ivl2*: $n \leq p \implies$
$sum\ f\ \{..<n\} + sum\ f\ \{n..p\} = sum\ f\ \{..p::nat\}$
$\langle proof \rangle$

lemma *setsum_eq_zero* [*simp*]:
fixes $f::nat \Rightarrow nat$
shows $(\sum i < n.\ f\ i) = 0 \longleftrightarrow (\forall i < n.\ f\ i = 0)$
$(\sum i \leq n.\ f\ i) = 0 \longleftrightarrow (\forall i \leq n.\ f\ i = 0)$
$\langle proof \rangle$

lemma *setprod_eq_zero* [*simp*]:
fixes $f::nat \Rightarrow nat$
shows $(\prod i < n.\ f\ i) = 0 \longleftrightarrow (\exists i < n.\ f\ i = 0)$
$(\prod i \leq n.\ f\ i) = 0 \longleftrightarrow (\exists i \leq n.\ f\ i = 0)$
$\langle proof \rangle$

lemma *setsum_one_less*:
fixes $n::nat$
assumes $\forall i < n.\ f\ i \leq 1$
shows $(\sum i < n.\ f\ i) \leq n$
$\langle proof \rangle$

lemma *setsum_one_le*:
fixes $n::nat$
assumes $\forall i \leq n.\ f\ i \leq 1$
shows $(\sum i \leq n.\ f\ i) \leq Suc\ n$
$\langle proof \rangle$

lemma *setsum_eq_one_le*:
fixes $n::nat$
assumes $\forall i \leq n.\ f\ i = 1$
shows $(\sum i \leq n.\ f\ i) = Suc\ n$
$\langle proof \rangle$

lemma *setsum_least_eq*:
fixes $f::nat \Rightarrow nat$
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 $(\sum i \leq n.\ f\ i) = p$
$\langle proof \rangle$

lemma *nat_mult_le_one*:

fixes $m\ n{::}nat$
assumes $m \le 1\ n \le 1$
shows $m * n \le 1$
$\langle proof \rangle$

lemma *setprod_one_le*:
fixes $f{::}nat \Rightarrow nat$
assumes $\forall\, i \le n.\ f\ i \le 1$
shows $(\prod i \le n.\ f\ i) \le 1$
$\langle proof \rangle$

lemma *setprod_greater_zero*:
fixes $f{::}nat \Rightarrow nat$
assumes $\forall\, i \le n.\ f\ i \ge 0$
shows $(\prod i \le n.\ f\ i) \ge 0$
$\langle proof \rangle$

lemma *setprod_eq_one*:
fixes $f{::}nat \Rightarrow nat$
assumes $\forall\, i \le n.\ f\ i = Suc\ 0$
shows $(\prod i \le n.\ f\ i) = Suc\ 0$
$\langle proof \rangle$

lemma *setsum_cut_off_less*:
fixes $f{::}nat \Rightarrow nat$
assumes $h1$: $m \le n$
and $h2$: $\forall\, i \in \{m..{<}n\}.\ f\ i = 0$
shows $(\sum i < n.\ f\ i) = (\sum i < m.\ f\ i)$
$\langle proof \rangle$

lemma *setsum_cut_off_le*:
fixes $f{::}nat \Rightarrow nat$
assumes $h1$: $m \le n$
and $h2$: $\forall\, i \in \{m..n\}.\ f\ i = 0$
shows $(\sum i \le n.\ f\ i) = (\sum i < m.\ f\ i)$
$\langle proof \rangle$

lemma *setprod_one* [*simp*]:
fixes $n{::}nat$
shows $(\prod i < n.\ Suc\ 0) = Suc\ 0$
$(\prod i \le n.\ Suc\ 0) = Suc\ 0$
$\langle proof \rangle$

4.2 Recursive Functions

datatype *recf* = *Z*
| *S*
| *Id nat nat*
| *Cn nat recf recf list*

$|$ *Pr nat recf recf*
$|$ *Mn nat recf*

fun *arity* :: *recf* $\Rightarrow$ *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* $\Rightarrow$ *nat list* $\Rightarrow$ *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*) [] = *undefined*
$|$ *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* $\Rightarrow$ *nat list* $\Rightarrow$ *bool*
where
termi_z: *terminates Z* [*n*]
$|$ *termi_s*: *terminates S* [*n*]
$|$ *termi_id*: $[\![$*n* < *m*; *length xs* = *m*$]\!]$ $\Longrightarrow$ *terminates* (*Id m n*) *xs*
$|$ *termi_cn*: $[\![$*terminates f* (*map* (λ*g*. *rec_eval g xs*) *gs*);
$\forall$ *g* $\in$ *set gs*. *terminates g xs*; *length xs* = *n*$]\!]$ $\Longrightarrow$ *terminates* (*Cn n f gs*) *xs*
$|$ *termi_pr*: $[\![\forall$ *y* < *x*. *terminates g* (*y* # (*rec_eval* (*Pr n f g*) (*y* # *xs*) # *xs*));
terminates f xs;
length xs = *n*$]\!]$
$\Longrightarrow$ *terminates* (*Pr n f g*) (*x* # *xs*)
$|$ *termi_mn*: $[\![$*length xs* = *n*; *terminates f* (*r* # *xs*);
rec_eval f (*r* # *xs*) = *0*;
$\forall$ *i* < *r*. *terminates f* (*i* # *xs*) $\wedge$ *rec_eval f* (*i* # *xs*) > *0*$]\!]$ $\Longrightarrow$ *terminates* (*Mn n f*) *xs*

4.3 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 $[x, y] = x \hat{}\ y$
⟨*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*⟩

4.4 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* $x \neq y$ *then 1 else 0*)
⟨*proof*⟩

lemma *conj_lemma* [*simp*]:
rec_eval rec_conj [*x*, *y*] $=$ (*if* $x = 0 \vee y = 0$ *then 0 else 1*)
⟨*proof*⟩

lemma *disj_lemma* [*simp*]:
rec_eval rec_disj [*x*, *y*] $=$ (*if* $x = 0 \wedge y = 0$ *then 0 else 1*)
⟨*proof*⟩

lemma *imp_lemma* [*simp*]:
rec_eval rec_imp [*x*, *y*] $=$ (*if* $0 < x \wedge y = 0$ *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*⟩

4.5 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* ($x \leq y$) *then 1 else 0*)
⟨*proof*⟩

4.6 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*] = ($\sum z \leq x$. *rec_eval f* [*z*, *y*])
⟨*proof*⟩

lemma *sigma2_lemma* [*simp*]:
shows *rec_eval* (*rec_sigma2 f*) [*x*, *y1*, *y2*] = ($\sum z \leq x$. *rec_eval f* [*z*, *y1*, *y2*])
⟨*proof*⟩

lemma *accum1_lemma* [*simp*]:
shows *rec_eval* (*rec_accum1 f*) [*x*, *y*] = ($\prod z \leq x$. *rec_eval f* [*z*, *y*])
⟨*proof*⟩

lemma *accum2_lemma* [*simp*]:
shows *rec_eval* (*rec_accum2 f*) [*x*, *y1*, *y2*] = ($\prod z \leq x$. *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*⟩

4.7 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* ($\forall z \leq x$. *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* ($\forall z < x$. *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* ($\forall z < x$. *0* < *rec_eval f* [*z*, *y1*, *y2*]) *then 1 else 0*)
⟨*proof*⟩

4.8 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*⟩

4.9 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* ∘ (*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* (λ*x*. *rec_eval f* [*x*]) *n x*
⟨*proof*⟩

4.10 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* ⇒ *bool*) ⇒ *nat* ⇒ *nat*
where
BMax_set R x = *Max* ({*z*. $z \le x \land R\ z$} ∪ {*0*})

lemma *BMax_rec_eq1*:
BMax_rec R x = (*GREATEST z*. ($R\ z \land z \le x$) $\lor z = 0$)
⟨*proof*⟩

lemma *BMax_rec_eq2*:
BMax_rec R x = *Max* ({*z*. $z \le x \land R\ z$} ∪ {*0*})
⟨*proof*⟩

lemma *BMax_rec_eq3*:
BMax_rec R x = *Max* (*Set.filter* (λ*z*. *R z*) {..*x*} ∪ {*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*⟩

4.11 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*:
$m \leq triangle\ (n + m)$
⟨*proof*⟩

lemma *Max_triangle_triangle_le*:
$triangle\ (Max\_triangle\ z) \leq z$
$\langle proof \rangle$

lemma *Max_triangle_le*:
$Max\_triangle\ z \leq z$
$\langle proof \rangle$

lemma *w_aux*:
$Max\_triangle\ (triangle\ k + m) = Max\_triangle\_aux\ k\ m$
$\langle proof \rangle$

lemma *y_aux*: $y \leq Max\_triangle\_aux\ y\ k$
$\langle proof \rangle$

lemma *Max_triangle_greatest*:
$Max\_triangle\ z = (GREATEST\ k.\ (triangle\ k \leq z \wedge k \leq z) \vee k = 0)$
$\langle proof \rangle$

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*]:
$rec\_eval\ rec\_triangle\ [x] = triangle\ x$
$\langle proof \rangle$

lemma *max_triangle_lemma* [*simp*]:
$rec\_eval\ rec\_max\_triangle\ [x] = Max\_triangle\ x$
$\langle proof \rangle$

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*]:
$rec\_eval\ rec\_pdec1\ [z] = pdec1\ z$
$\langle proof \rangle$

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

4.12 Examples for recursive functions using the alternative definitions

theory *Recs_alt_Ex*
imports *Recs_alt_Def*
begin

definition *plus_2* :: *recf*
where
plus_2 = (*CN S* [*S*])

lemma *rec_eval S* [*0*] = *Suc 0*
⟨*proof*⟩

lemma *rec_eval plus_2* [*0*] = *rec_eval* (*Cn 8 S* [*S*]) [*0*]
⟨*proof*⟩

lemma *Cn 1 S* [*S*] = *CN S* [*S*]
⟨*proof*⟩

lemma *rec_eval plus_2* [*0*] = *2*
⟨*proof*⟩

lemma *rec_eval plus_2* [*2*] = *4*
⟨*proof*⟩

lemma *rec_eval plus_2* [*0*,*4*] = *2*
⟨*proof*⟩

lemma *add_lemma_more_args*:
rec_eval rec_add $([x, y] @ z) = x + y$
$\langle proof \rangle$

lemma *rec_eval* $(Pr\ v\ va\ vb)$ $[] =$ *undefined*
$\langle proof \rangle$

lemma *add_lemma_no_args*:
rec_eval rec_add $[] =$ *undefined*
$\langle proof \rangle$

lemma *add_lemma_one_arg*:
rec_eval rec_add $[x] =$ *undefined*
$\langle proof \rangle$

lemma $[]!0 =$ *undefined*
$\langle proof \rangle$

end

Chapter 5

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 for Turing Machines. Once the correctness of *rec_F* is established, UTM can easily be obtained by compiling *rec_F* into the corresponding Turing Machine.

5.1 Building blocks of the Universal Function rec_F

5.1.1 Some helper functions: Recursive Functions for arithmetic and logic

The recursive function used to do arithmetic addition.

definition *rec_add* :: *recf*
 where
 $rec\_add \overset{def}{=} Pr\ 1\ (id\ 1\ 0)\ (Cn\ 3\ s\ [id\ 3\ 2])$

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 natural 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 Recursive Functions. The function is defined such that it always returns meaningful results for primitive recursive functions.

5.1.1.1 Correctness of the helper 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*: ⋀ *x y*. *rec_exec rec_mult* [*x*, *y*] = *x* ∗ *y*
⟨*proof*⟩

Correctness of *rec_pred*.

lemma *pred_lemma*: ⋀ *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*: $\bigwedge x.\ rec\_exec\ rec\_sg\ [x] = (\mathit{if}\ x = 0\ \mathit{then}\ 0\ \mathit{else}\ 1)$
⟨*proof*⟩

Correctness of *constn*.

lemma *constn_lemma*: $rec\_exec\ (constn\ n)\ [x] = n$
⟨*proof*⟩

Correctness of *rec_less*.

lemma *less_lemma*: $\bigwedge x\ y.\ rec\_exec\ rec\_less\ [x, y] =$
$(\mathit{if}\ x < y\ \mathit{then}\ 1\ \mathit{else}\ 0)$
⟨*proof*⟩

Correctness of *rec_not*.

lemma *not_lemma*:
$\bigwedge x.\ rec\_exec\ rec\_not\ [x] = (\mathit{if}\ x = 0\ \mathit{then}\ 1\ \mathit{else}\ 0)$
⟨*proof*⟩

Correctness of *rec_eq*.

lemma *eq_lemma*: $\bigwedge x\ y.\ rec\_exec\ rec\_eq\ [x, y] = (\mathit{if}\ x = y\ \mathit{then}\ 1\ \mathit{else}\ 0)$
⟨*proof*⟩

Correctness of *rec_conj*.

lemma *conj_lemma*: $\bigwedge x\ y.\ rec\_exec\ rec\_conj\ [x, y] = (\mathit{if}\ x = 0 \vee y = 0\ \mathit{then}\ 0$
$\mathit{else}\ 1)$
⟨*proof*⟩

Correctness of *rec_disj*.

lemma *disj_lemma*: $\bigwedge x\ y.\ rec\_exec\ rec\_disj\ [x, y] = (\mathit{if}\ x = 0 \wedge y = 0\ \mathit{then}\ 0$
$\mathit{else}\ 1)$
⟨*proof*⟩

5.1.2 The characteristic function primerec for the set of Primitive Recursive Functions

primerec recf n is true iff *recf* is a primitive recursive function with arity *n*.

inductive $primerec :: recf \Rightarrow nat \Rightarrow bool$
where
$prime\_z[intro]:\ primerec\ z\ (Suc\ 0)\ |$
$prime\_s[intro]:\ primerec\ s\ (Suc\ 0)\ |$
$prime\_id[intro!]:\ [\![n < m]\!] \Longrightarrow primerec\ (id\ m\ n)\ m\ |$
$prime\_cn[intro!]:\ [\![primerec\ f\ k;\ length\ gs = k;$
$\forall\ i < length\ gs.\ primerec\ (gs\ !\ i)\ m;\ m = n]\!]$
$\Longrightarrow primerec\ (Cn\ n\ f\ gs)\ m\ |$
$prime\_pr[intro!]:\ [\![primerec\ f\ n;$

primerec g (Suc (Suc n)); m = Suc n]]
$\Longrightarrow$ *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*

5.1.3 The Recursive Function rec_sigma

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*]] $\Longrightarrow$

$map\ (\lambda f.\ rec\_exec\ f\ xs)\ (get\_fstn\_args\ m\ n) = take\ n\ xs$
⟨*proof*⟩

lemma *arity_primerec*[*simp*]: $primerec\ f\ n \Longrightarrow arity\ f = n$
⟨*proof*⟩

lemma *rec_sigma_Suc_simp_rewrite*:
$primerec\ f\ (Suc\ (length\ xs))$
$\Longrightarrow rec\_exec\ (rec\_sigma\ f)\ (xs\ @\ [Suc\ x]) =$
$rec\_exec\ (rec\_sigma\ f)\ (xs\ @\ [x]) + rec\_exec\ f\ (xs\ @\ [Suc\ x])$
⟨*proof*⟩

The correctness of *rec_sigma* with respect to its specification.

lemma *sigma_lemma*:
$primerec\ rg\ (Suc\ (length\ xs))$
$\Longrightarrow rec\_exec\ (rec\_sigma\ rg)\ (xs\ @\ [x]) = Sigma\ (rec\_exec\ rg)\ (xs\ @\ [x])$
⟨*proof*⟩

5.1.4 The Recursive Function rec_accum

$rec\_accum\ f\ (x1, x2, \ldots, xn, k) = f(x1, x2, \ldots, xn, 0) * f(x1, x2, \ldots, xn, 1) * \ldots$
$f(x1, x2, \ldots, xn, k)$

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*:
$primerec\ f\ (Suc\ (length\ xs))$
$\Longrightarrow rec\_exec\ (rec\_accum\ f)\ (xs\ @\ [Suc\ x]) =$
$rec\_exec\ (rec\_accum\ f)\ (xs\ @\ [x]) * rec\_exec\ f\ (xs\ @\ [Suc\ x])$
⟨*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*]

5.1.5 The Recursive Function rec_all

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*) =
($\exists$ *t* $\leq$ *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*)⟧
$\Longrightarrow$ *rec_exec* (*rec_all rt rf*) *xs* = (*if* ($\forall$ *x* $\leq$ (*rec_exec rt xs*). *0* < *rec_exec rf* (*xs* @ [*x*])) *then*
1
else 0)
⟨*proof*⟩

5.1.6 The Recursive Function rec_ex

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*) =
($\forall$ *t* $\leq$ *x*. *rec_exec rf* (*xs* @ [*t*]) = *0*)

⟨*proof*⟩

The correctness of *rec_ex*.

lemma *ex_lemma*:
 $[\![$*primerec rf* (*Suc* (*length xs*));
 primerec rt (*length xs*)$]\!]$
$\Longrightarrow$ (*rec_exec* (*rec_ex rt rf*) *xs* =
 (*if* ($\exists$ $x \leq$ (*rec_exec rt xs*). *0* <*rec_exec rf* (*xs* @ [*x*])) *then 1*
 else 0))
 ⟨*proof*⟩

5.1.7 The Recursive Function rec_Minr

Definition of *Min*[*R*] on page 77 of Boolos's book [1].

fun *Minr* :: (*nat list* $\Rightarrow$ *bool*) $\Rightarrow$ *nat list* $\Rightarrow$ *nat* $\Rightarrow$ *nat*
 where *Minr Rr xs w* = (*let setx* = {*y* | *y*. (*y* $\leq$ *w*) $\wedge$ *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 auxiliary lemmas about *Minr*.

lemma *Minr_range*: *Minr Rr xs w* $\leq$ *w* $\vee$ *Minr Rr xs w* = *Suc w*
 ⟨*proof*⟩

lemma *expand_conj_in_set*: {*x*. *x* $\leq$ *Suc w* $\wedge$ *Rr* (*xs* @ [*x*])}
 = (*if Rr* (*xs* @ [*Suc w*]) *then insert* (*Suc w*)
 {*x*. *x* $\leq$ *w* $\wedge$ *Rr* (*xs* @ [*x*])}
 else {*x*. *x* $\leq$ *w* $\wedge$ *Rr* (*xs* @ [*x*])})
 ⟨*proof*⟩

lemma *Minr_strip_Suc*[*simp*]: *Minr Rr xs w* $\leq$ *w* $\Longrightarrow$ *Minr Rr xs* (*Suc w*) = *Minr Rr xs w*
 ⟨*proof*⟩

lemma *x_empty_set*[*simp*]: $\forall$ *x*$\leq$*w*. $\neg$ *Rr* (*xs* @ [*x*]) $\Longrightarrow$
 {*x*. *x* $\leq$ *w* $\wedge$ *Rr* (*xs* @ [*x*])} = {}
 ⟨*proof*⟩

lemma *Minr_is_Suc*[*simp*]: $[\![$*Minr Rr xs w* = *Suc w*; *Rr* (*xs* @ [*Suc w*])$]\!]$ $\Longrightarrow$
 Minr Rr xs (*Suc w*) = *Suc w*
 ⟨*proof*⟩

lemma *Minr_is_Suc_Suc*[*simp*]: $[\![$*Minr Rr xs w* = *Suc w*; $\neg$ *Rr* (*xs* @ [*Suc w*])$]\!]$ $\Longrightarrow$
 Minr Rr xs (*Suc w*) = *Suc* (*Suc w*)
 ⟨*proof*⟩

lemma *Minr_Suc_simp*:
 Minr Rr xs (*Suc w*) =
 (*if Minr Rr xs w* $\leq$ *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*

$\langle proof \rangle$

lemma *primerec_rec_add_2*[*intro*]: *primerec rec_add* (*Suc* (*Suc 0*))
$\langle proof \rangle$

lemma *primerec_rec_mult_2*[*intro*]:*primerec rec_mult* (*Suc* (*Suc 0*))
$\langle proof \rangle$

lemma *primerec_ge_2_elim*[*elim*]: $[\![$*primerec rf n*; $n \geq$ *Suc* (*Suc 0*)$]\!] \Longrightarrow$
primerec (*rec_accum rf*) *n*
$\langle proof \rangle$

lemma *primerec_all_iff*:
$[\![$*primerec rt n*; *primerec rf* (*Suc n*); $n > 0]\!] \Longrightarrow$
primerec (*rec_all rt rf*) *n*
$\langle proof \rangle$

lemma *primerec_rec_not_1*[*intro*]: *primerec rec_not* (*Suc 0*)
$\langle proof \rangle$

lemma *Min_false1*[*simp*]: $[\![\neg$ *Min* {*uu*. $uu \leq w \wedge 0 <$ *rec_exec rf* (*xs* @ [*uu*])} $\leq w$;
$x \leq w$; $0 <$ *rec_exec rf* (*xs* @ [*x*])$]\!]$
$\Longrightarrow$ *False*
$\langle proof \rangle$

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*
$\langle proof \rangle$

The correctness of *rec_Minr*.

lemma *Minr_lemma*:
$[\![$*primerec rf* (*Suc* (*length xs*))$]\!]$
$\Longrightarrow$ *rec_exec* (*rec_Minr rf*) (*xs* @ [*w*]) =
Minr (λ *args*. ($0 <$ *rec_exec rf args*)) *xs w*
$\langle proof \rangle$

5.1.8 The Recursive Function rec_le

rec_le is the comparison 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*:
$\bigwedge x\ y.\ rec\_exec\ rec\_le\ [x, y] = (if\ (x \leq y)\ then\ 1\ else\ 0)$
⟨*proof*⟩

5.1.9 The Recursive Function rec_maxr

Definition of *Max*[*Rr*] on page 77 of Boolos's book [1].

```
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 implement *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*: $\forall\ i \leq n.\ (f\ (xs\ @\ [i]) = 0) \Longrightarrow$
$\quad Sigma\ f\ (xs\ @\ [n]) = 0$
⟨*proof*⟩

lemma *Sigma_Suc*[*elim*]: $\forall k{<}Suc\ w.\ f\ (xs\ @\ [k]) = Suc\ 0$
$\Longrightarrow Sigma\ f\ (xs\ @\ [w]) = Suc\ w$
⟨*proof*⟩

lemma *Sigma_max_point*: $[\![\forall\ k < ma.\ f\ (xs\ @\ [k]) = 1;$
$\forall\ k \geq ma.\ f\ (xs\ @\ [k]) = 0;\ ma \leq w]\!]$
$\Longrightarrow Sigma\ f\ (xs\ @\ [w]) = ma$
⟨*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*⟩

5.1.10 The Recursive Function rec_noteq

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*]

5.1.11 The Recursive Function rec_quo

quo is the formal specification of division.

fun *quo* :: *nat list* ⇒ *nat*
where
$quo\ [x, y] = (let\ Rr =$
$(\lambda\ zs.\ ((zs\ !\ (Suc\ 0) * zs\ !\ (Suc\ (Suc\ 0))$
$\le zs\ !\ 0) \wedge zs\ !\ Suc\ 0 \neq (0::nat)))$
$in\ Maxr\ Rr\ [x, y]\ x)$

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

The following lemmas shows more directly the meaning of *quo*:

lemma *quo_is_div*: $y > 0 \Longrightarrow quo\ [x, y] = x\ div\ y$
$\langle proof \rangle$

lemma *quo_zero*[*intro*]: $quo\ [x, 0] = 0$
$\langle proof \rangle$

lemma *quo_div*: $quo\ [x, y] = x\ div\ y$
$\langle proof \rangle$

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* $[x, y]$
⟨*proof*⟩

The correctness of *quo*.

lemma *quo_lemma2*: *rec_exec rec_quo* $[x, y] = x$ *div* y
⟨*proof*⟩

5.1.12 The Recursive Function rec_mod

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*⟩

5.1.13 The Recursive Function rec_embranch

lemmas for embranch function

type-synonym *ftype* = *nat list* ⇒ *nat*
type-synonym *rtype* = *nat list* ⇒ *bool*

The specifcation of the multi-way branching statement (definition by cases). See page 74 of Boolos's book [1].

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*:
$[\![\forall\ j < length\ rcs.\ rec\_exec\ (rcs\ !\ j)\ xs = 0;$
$length\ rgs = length\ rcs;$
$rcs \neq [\,];$
$list\_all\ (\lambda\ rf.\ primerec\ rf\ (length\ xs))\ (rgs\ @\ rcs)]\!] \Longrightarrow$
$rec\_exec\ (rec\_embranch\ (zip\ rgs\ rcs))\ xs = 0$
⟨*proof*⟩

lemma *embranch_exec_0*: $[\![rec\_exec\ aa\ xs = 0;\ zip\ rgs\ list \neq [\,];$
$list\_all\ (\lambda\ rf.\ primerec\ rf\ (length\ xs))\ ([a, aa]\ @\ rgs\ @\ list)]\!]$
$\Longrightarrow rec\_exec\ (rec\_embranch\ ((a, aa)\ \#\ zip\ rgs\ list))\ xs$
$= rec\_exec\ (rec\_embranch\ (zip\ rgs\ list))\ xs$
⟨*proof*⟩

lemma *zip_null_iff*: $[\![length\ xs = k;\ length\ ys = k;\ zip\ xs\ ys = [\,]]\!] \Longrightarrow xs = [\,] \wedge ys = [\,]$
⟨*proof*⟩

lemma *zip_null_gr*: $[\![length\ xs = k;\ length\ ys = k;\ zip\ xs\ ys \neq [\,]]\!] \Longrightarrow 0 < k$
⟨*proof*⟩

lemma *Embranch_0*:
$[\![length\ rgs = k;\ length\ rcs = k;\ k > 0;$
$\forall\ j < k.\ rec\_exec\ (rcs\ !\ j)\ xs = 0]\!] \Longrightarrow$
$Embranch\ (zip\ (map\ rec\_exec\ rgs)\ (map\ (\lambda r\ args.\ 0 < rec\_exec\ r\ args)\ rcs))\ xs = 0$
⟨*proof*⟩

The correctness of *rec_embranch*.

lemma *embranch_lemma*:
assumes *branch_num*:
$length\ rgs = n\ length\ rcs = n\ n > 0$
and *partition*:
$(\exists\ i < n.\ (rec\_exec\ (rcs\ !\ i)\ xs = 1 \wedge (\forall\ j < n.\ j \neq i \longrightarrow$
$rec\_exec\ (rcs\ !\ j)\ xs = 0)))$
and *prime_all*: $list\_all\ (\lambda\ rf.\ primerec\ rf\ (length\ xs))\ (rgs\ @\ rcs)$
shows $rec\_exec\ (rec\_embranch\ (zip\ rgs\ rcs))\ xs =$
$Embranch\ (zip\ (map\ rec\_exec\ rgs)$
$(map\ (\lambda\ r\ args.\ 0 < rec\_exec\ r\ args)\ rcs))\ xs$
⟨*proof*⟩

5.1.14 The Recursive Function rec_prime

prime n means *n* is a prime number.

fun *Prime* :: $nat \Rightarrow bool$
where
$Prime\ x = (1 < x \wedge (\forall\ u < x.\ (\forall\ v < x.\ u * v \neq x)))$

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*))
⟨*proof*⟩

The correctness of *Prime*.

lemma *prime_lemma*: *rec_exec rec_prime* [*x*] = (*if Prime x then 1 else 0*)
⟨*proof*⟩

5.1.15 The Recursive Function rec_fac for factorization

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 implement 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* ⇒ *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 $fac.simps[simp\ del]$

5.1.16 The Recursive Function rec_np for finding the next prime

$Np\ x$ 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 $divsor\_ex$:
$[\![\neg\ Prime\ x; x > Suc\ 0]\!] \Longrightarrow (\exists\ u > Suc\ 0.\ (\exists\ v > Suc\ 0.\ u * v = x))$
$\langle proof \rangle$

lemma $divsor\_prime\_ex$: $[\![\neg\ Prime\ x; x > Suc\ 0]\!] \Longrightarrow$
$\exists\ p.\ Prime\ p \wedge p\ dvd\ x$
$\langle proof \rangle$

lemma $fact\_pos[intro]$: $0 < n!$
$\langle proof \rangle$

lemma fac_Suc: $Suc\ n! = (Suc\ n) * (n!)$ $\langle proof \rangle$

lemma fac_dvd: $[\![0 < q; q \leq n]\!] \Longrightarrow q\ dvd\ n!$
$\langle proof \rangle$

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*⟩

5.1.17 The Recursive Function rec_power

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*^*y*
⟨*proof*⟩

5.1.18 The Recursive Function rec_pi

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*⟩

5.1.19 The Recursive Function rec_lo

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 [1]. 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*⟩

5.1.20 The Recursive Function rec_lg

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 [1]. It is one of the two notions of integral 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''*: ¬ *Suc 0* < *x* ⟹ *rec_exec rec_lg* [*x*, *y*] = *lg x y*
⟨*proof*⟩

lemma *lg_lemma'''*: ¬ *Suc 0* < *y* ⟹ *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*⟩

5.1.21 The Recursive Function rec_entry

Entry sr i returns the *i*-th entry of a list of natural numbers encoded by number *sr* using Godel's coding. This function is called *ent* on page 80 of Boolos's book [1].

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*⟩

5.2 Main components of rec_F

Using the auxiliary functions obtained in last section, we are going to construct the function *F*, which is an interpreter for 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*⟧ ⟹
   rec_exec (*rec_listsum2 vl n*) *xs* = *listsum2 xs n*
 ⟨*proof*⟩

5.2.1 The Recursive Function rec_strt

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* $\leq$ *vl*$]\!] \Longrightarrow$
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*⟩

5.2.2 The Recursive Function rec_scan

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*: $rec\_exec\ rec\_scan\ [r] = r\ mod\ 2$
$\langle proof \rangle$

5.2.3 The Recursive Function rec_newleft

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 ⟹  P i;
  i = 1 ⟹ P i; i =2 ⟹ P i;
  i =3 ⟹ P i⟧ ⟹ 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⟩
```

5.2.4 The Recursive Function rec_newrght

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 *Suc_5_induct*:
$[\![i < Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0)))); i = 0 \Longrightarrow P\ 0;$
$i = 1 \Longrightarrow P\ 1; i = 2 \Longrightarrow P\ 2; i = 3 \Longrightarrow P\ 3; i = 4 \Longrightarrow P\ 4]\!] \Longrightarrow P\ i$
⟨*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]*

5.2.5 The Recursive Function rec_actn

The *actn* function given on page 92 of B book, which is used to fetch Turing Machine instructions. In *actn m q r*, *m* is the Gödel 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*⟩

5.2.6 The Recursive Function rec_newstat

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*]

5.2.7 The Recursive Function rec_trpl

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*⟩

5.2.8 The Recursive Functions rec_left, rec_right, rec_stat, rec_inpt

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 $rght :: nat \Rightarrow nat$
 where
  $rght\ c = lo\ c\ (Pi\ 2)$

fun $inpt :: nat \Rightarrow nat\ list \Rightarrow nat$
 where
  $inpt\ m\ xs = trpl\ 0\ 1\ (strt\ xs)$

fun $newconf :: nat \Rightarrow nat \Rightarrow nat$
 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 $rec\_inpt :: nat \Rightarrow recf$
 where
  $rec\_inpt\ vl = Cn\ vl\ rec\_trpl$
   $[Cn\ vl\ (constn\ 0)\ [id\ vl\ 0],$
   $Cn\ vl\ (constn\ 1)\ [id\ vl\ 0],$
   $Cn\ vl\ (rec\_strt\ (vl - 1))$
    $(map\ (\lambda\ i.\ id\ vl\ (i))\ [1..<vl])]$

lemma $left\_lemma$: $rec\_exec\ rec\_left\ [c] = left\ c$
 $\langle proof \rangle$

lemma $right\_lemma$: $rec\_exec\ rec\_right\ [c] = rght\ c$
 $\langle proof \rangle$

lemma $stat\_lemma$: $rec\_exec\ rec\_stat\ [c] = stat\ c$
 $\langle proof \rangle$

declare *rec_strt.simps[simp del] strt.simps[simp del]*

lemma *map_cons_eq*:
$(map\ ((\lambda a.\ rec\_exec\ a\ (m\ \#\ xs)) \circ$
$(\lambda i.\ recf.id\ (Suc\ (length\ xs))\ (i)))$
$[Suc\ 0..<Suc\ (length\ xs)])$
$= map\ (\lambda\ i.\ xs\ !\ (i - 1))\ [Suc\ 0..<Suc\ (length\ xs)]$
⟨*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*⟩

5.2.9 The Recursive Function rec_newconf

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*]

5.2.10 The Recursive Function rec_conf

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*⟩

5.2.11 The Recursive Function rec_NSTD

NSTD c returns true if the configuration 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*: $rec\_exec\ rec\_NSTD\ [c] = Suc\ 0 \lor rec\_exec\ rec\_NSTD\ [c] = 0$
⟨*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*⟩

5.2.12 The Recursive Function rec_nonstop

nonstop 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*⟩

5.2.13 The Recursive Function rec_halt

rec_halt is the recursive function calculating the steps a TM needs to execute before to reach a standard final configuration. This recursive function is the only one using the *Mn* combinator. So it is the only non-primitive recursive function that 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.

5.2.14 Execution of Primitive Recursive Functions always terminates

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* $= x]\!] \Longrightarrow$ *terminate f xs*
⟨*proof*⟩

5.2.15 The Recursive Function rec_valu

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]*

5.3 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*⟩

5.4 Correctness of rec_F with respect to rec_halt

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.

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*⟩

5.5 A Gödel-Encoding for TMs: the function code

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

$action\_map\ WB = 0$
$|\ action\_map\ WO = 1$
$|\ action\_map\ L = 2$
$|\ action\_map\ R = 3$
$|\ action\_map\ Nop = 4$

fun $action\_map\_iff :: nat \Rightarrow action$
where
$action\_map\_iff\ (0::nat) = WB$
$|\ action\_map\_iff\ (Suc\ 0) = WO$
$|\ action\_map\_iff\ (Suc\ (Suc\ 0)) = L$
$|\ action\_map\_iff\ (Suc\ (Suc\ (Suc\ 0))) = R$
$|\ action\_map\_iff\ n = Nop$

fun $block\_map :: cell \Rightarrow nat$
where
$block\_map\ Bk = 0$
$|\ block\_map\ Oc = 1$

fun $godel\_code' :: nat\ list \Rightarrow nat \Rightarrow nat$
where
$godel\_code'\ []\ n = 1$
$|\ godel\_code'\ (x\#xs)\ n = (Pi\ n){}^\wedge x * godel\_code'\ xs\ (Suc\ n)$

fun $godel\_code :: nat\ list \Rightarrow nat$
where
$godel\_code\ xs = (let\ lh = length\ xs\ in$
$2^\wedge lh * (godel\_code'\ xs\ (Suc\ 0)))$

fun $modify\_tprog :: instr\ list \Rightarrow nat\ list$
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 $code :: instr\ list \Rightarrow nat$
where
$code\ tp = (let\ nl = modify\_tprog\ tp\ in$
$godel\_code\ nl)$

5.6 Relating interpreter functions to the execution of TMs

lemma $bl2wc\_0[simp]$: $bl2wc\ [] = 0\ \langle proof \rangle$

lemma $fetch\_action\_map\_4[simp]$: $[\![fetch\ tp\ 0\ b = (nact, ns)]\!] \Longrightarrow action\_map\ nact = 4$
$\langle proof \rangle$

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* ≠ *j* ⟹ *Pi i* ≠ *Pi j*
⟨*proof*⟩

lemma *prime_2*[*intro*]: *Prime* (*Suc* (*Suc 0*))
⟨*proof*⟩

lemma *Prime_Pi*[*intro*]: *Prime* (*Pi n*)
⟨*proof*⟩

lemma *Pi_coprime*: *i* ≠ *j* ⟹ *coprime* (*Pi i*) (*Pi j*)
⟨*proof*⟩

lemma *Pi_power_coprime*: *i* ≠ *j* ⟹ *coprime* ((*Pi i*)^*m*) ((*Pi j*)^*n*)
⟨*proof*⟩

lemma *coprime_dvd_mult_nat2*: ⟦*coprime* (*k*::*nat*) *n*; *k dvd n* ∗ *m*⟧ ⟹ *k dvd m*
⟨*proof*⟩

declare *godel_code′.simps*[*simp del*]

lemma *godel_code′_butlast_last_id′* :
$godel\_code'\ (ys\ @\ [y])\ (Suc\ j) = godel\_code'\ ys\ (Suc\ j) *$
$Pi\ (Suc\ (length\ ys + j))\ \hat{}\ y$
⟨*proof*⟩

lemma *godel_code′_butlast_last_id*:
$xs \neq [] \Longrightarrow godel\_code'\ xs\ (Suc\ j) =$
$godel\_code'\ (butlast\ xs)\ (Suc\ j) * Pi\ (length\ xs + j)\hat{}(last\ xs)$
⟨*proof*⟩

lemma *godel_code′_not0*: $godel\_code'\ xs\ n \neq 0$
⟨*proof*⟩

lemma *godel_code_append_cons*:
$length\ xs = i \Longrightarrow godel\_code'\ (xs@y\#ys)\ (Suc\ 0)$
$= godel\_code'\ xs\ (Suc\ 0) * Pi\ (Suc\ i)\hat{}y * godel\_code'\ ys\ (i + 2)$
⟨*proof*⟩

lemma *Pi_coprime_pre*:
$length\ ps \leq i \Longrightarrow coprime\ (Pi\ (Suc\ i))\ (godel\_code'\ ps\ (Suc\ 0))$
⟨*proof*⟩

lemma *Pi_coprime_suf*: $i < j \Longrightarrow coprime\ (Pi\ i)\ (godel\_code'\ ps\ j)$
⟨*proof*⟩

lemma *godel_finite*:
$finite\ \{u.\ Pi\ (Suc\ i)\ \hat{}\ u\ dvd\ godel\_code'\ nl\ (Suc\ 0)\}$
⟨*proof*⟩

lemma *godel_code_in*:
$i < length\ nl \Longrightarrow\ nl\ !\ i\ \in \{u.\ Pi\ (Suc\ i)\ \hat{}\ u\ dvd$
$godel\_code'\ nl\ (Suc\ 0)\}$
⟨*proof*⟩

lemma *godel_code′_get_nth*:
$i < length\ nl \Longrightarrow Max\ \{u.\ Pi\ (Suc\ i)\ \hat{}\ u\ dvd$
$godel\_code'\ nl\ (Suc\ 0)\} = nl\ !\ i$
⟨*proof*⟩

lemma *godel_code′_set*[*simp*]:
$\{u.\ Pi\ (Suc\ i)\ \hat{}\ u\ dvd\ (Suc\ (Suc\ 0))\ \hat{}\ length\ nl *$
$godel\_code'\ nl\ (Suc\ 0)\} =$
$\{u.\ Pi\ (Suc\ i)\ \hat{}\ u\ dvd\ \ godel\_code'\ nl\ (Suc\ 0)\}$
⟨*proof*⟩

lemma *godel_code_get_nth*:
$i < length\ nl \Longrightarrow$
$Max\ \{u.\ Pi\ (Suc\ i)\ \hat{}\ u\ dvd\ godel\_code\ nl\} = nl\ !\ i$
⟨*proof*⟩

lemma *mod_dvd_simp*: $(x\ mod\ y = (0::nat)) = (y\ dvd\ x)$
⟨*proof*⟩

lemma *dvd_power_le*: $[\![a > Suc\ 0;\ a\ \hat{}\ y\ dvd\ a\ \hat{}\ l]\!] \Longrightarrow y \le l$
⟨*proof*⟩

lemma *Pi_nonzeroE*[*elim*]: $Pi\ n = 0 \Longrightarrow RR$
⟨*proof*⟩

lemma *Pi_not_oneE*[*elim*]: $Pi\ n = Suc\ 0 \Longrightarrow RR$
⟨*proof*⟩

lemma *finite_power_dvd*:
$[\![(a::nat) > Suc\ 0;\ y \ne 0]\!] \Longrightarrow finite\ \{u.\ a\hat{}u\ dvd\ y\}$
⟨*proof*⟩

lemma *conf_decode1*: $[\![m \ne n;\ m \ne k;\ k \ne n]\!] \Longrightarrow$
$Max\ \{u.\ Pi\ m\ \hat{}\ u\ dvd\ Pi\ m\ \hat{}\ l * Pi\ n\ \hat{}\ st * Pi\ k\ \hat{}\ r\} = l$
⟨*proof*⟩

lemma *left_trpl_fst*[*simp*]: $left\ (trpl\ l\ st\ r) = l$
⟨*proof*⟩

lemma *stat_trpl_snd*[*simp*]: $stat\ (trpl\ l\ st\ r) = st$
⟨*proof*⟩

lemma *rght_trpl_trd*[*simp*]: $rght\ (trpl\ l\ st\ r) = r$
⟨*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*:
$[\![st \le length\ tp\ div\ 2;\ st > 0]\!] \Longrightarrow$
$modify\_tprog\ tp\ !\ (4 * (st - Suc\ 0)\ ) =$
$action\_map\ (fst\ (tp\ !\ (2 * (st - Suc\ 0))))$
⟨*proof*⟩

lemma *modify_tprog_fetch_odd*:
$[\![st \leq length\ tp\ div\ 2;\ st > 0]\!] \Longrightarrow$
$modify\_tprog\ tp\ !\ (Suc\ (Suc\ (4 * (st - Suc\ 0)))) =$
$action\_map\ (fst\ (tp\ !\ (Suc\ (2 * (st - Suc\ 0)))))$
$\langle proof \rangle$

lemma *modify_tprog_fetch_action*:
$[\![st \leq length\ tp\ div\ 2;\ st > 0;\ b = 1 \vee b = 0]\!] \Longrightarrow$
$modify\_tprog\ tp\ !\ (4 * (st - Suc\ 0) + 2 * b) =$
$action\_map\ (fst\ (tp\ !\ ((2 * (st - Suc\ 0)) + b)))$
$\langle proof \rangle$

lemma *length_modify*: $length\ (modify\_tprog\ tp) = 2 * length\ tp$
$\langle proof \rangle$

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

lemma *fetch_action_eq*:
$[\![block\_map\ b = scan\ r;\ fetch\ tp\ st\ b = (nact, ns);$
$st \leq length\ tp\ div\ 2]\!] \Longrightarrow actn\ (code\ tp)\ st\ r = action\_map\ nact$
$\langle proof \rangle$

lemma *fetch_zero_zero*[*simp*]: $fetch\ tp\ 0\ b = (nact, ns) \Longrightarrow ns = 0$
$\langle proof \rangle$

lemma *modify_tprog_fetch_state*:
$[\![st \leq length\ tp\ div\ 2;\ st > 0;\ b = 1 \vee b = 0]\!] \Longrightarrow$
$modify\_tprog\ tp\ !\ Suc\ (4 * (st - Suc\ 0) + 2 * b) =$
$(snd\ (tp\ !\ (2 * (st - Suc\ 0) + b)))$
$\langle proof \rangle$

lemma *fetch_state_eq*:
$[\![block\_map\ b = scan\ r;$
$fetch\ tp\ st\ b = (nact, ns);$
$st \leq length\ tp\ div\ 2]\!] \Longrightarrow newstat\ (code\ tp)\ st\ r = ns$
$\langle proof \rangle$

lemma *tpl_eqI*[*intro*!]:
$[\![a = a';\ b = b';\ c = c']\!] \Longrightarrow trpl\ a\ b\ c = trpl\ a'\ b'\ c'$
$\langle proof \rangle$

lemma *bl2nat_double*: $bl2nat\ xs\ (Suc\ n) = 2 * bl2nat\ xs\ n$
$\langle proof \rangle$

lemma *bl2wc_simps*[*simp*]:
$bl2wc\ (Oc\ \#\ tl\ c) = Suc\ (bl2wc\ c) - bl2wc\ c\ mod\ 2$
$bl2wc\ (Bk\ \#\ c) = 2 * bl2wc\ (c)$
$2 * bl2wc\ (tl\ c) = bl2wc\ c - bl2wc\ c\ mod\ 2$

$bl2wc\ [Oc] = Suc\ 0$
$c \neq [] \Longrightarrow bl2wc\ (tl\ c) = bl2wc\ c\ div\ 2$
$c \neq [] \Longrightarrow bl2wc\ [hd\ c] = bl2wc\ c\ mod\ 2$
$c \neq [] \Longrightarrow bl2wc\ (hd\ c\ \#\ d) = 2 * bl2wc\ d + bl2wc\ c\ mod\ 2$
$2 * (bl2wc\ c\ div\ 2) = bl2wc\ c - bl2wc\ c\ mod\ 2$
$bl2wc\ (Oc\ \#\ list)\ mod\ 2 = Suc\ 0$
$\langle proof \rangle$

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*:
$(\lambda\ (s, l, r).\ s \leq length\ tp\ div\ 2)\ c \Longrightarrow$
$trpl\_code\ (step0\ c\ tp) =$
$rec\_exec\ rec\_newconf\ [code\ tp, trpl\_code\ c]$
$\langle proof \rangle$

lemma *bl2nat_simps*[*simp*]: $bl2nat\ (Oc\ \#\ Oc{\uparrow}x)\ 0 = (2 * 2\ \hat{}\ x - Suc\ 0)$
$bl2nat\ (Bk{\uparrow}x)\ n = 0$
$\langle proof \rangle$

lemma *bl2nat_exp_zero*[*simp*]: $bl2nat\ (Oc{\uparrow}y)\ 0 = 2\hat{}y - Suc\ 0$
$\langle proof \rangle$

lemma *bl2nat_cons_bk*: $bl2nat\ (ks\ @\ [Bk])\ 0 = bl2nat\ ks\ 0$
$\langle proof \rangle$

lemma *bl2nat_cons_oc*:
$bl2nat\ (ks\ @\ [Oc])\ 0 = bl2nat\ ks\ 0 + 2\ \hat{}\ length\ ks$
$\langle proof \rangle$

lemma *bl2nat_append*:
$bl2nat\ (xs\ @\ ys)\ 0 = bl2nat\ xs\ 0 + bl2nat\ ys\ (length\ xs)$
$\langle proof \rangle$

lemma *trpl_code_simp*[*simp*]:
$trpl\_code\ (steps0\ (Suc\ 0, Bk{\uparrow}l, <lm>)\ tp\ 0) =$
$rec\_exec\ rec\_conf\ [code\ tp, bl2wc\ (<lm>), 0]$
$\langle proof \rangle$

The following lemma relates the multi-step interpreter function *rec_conf* with the multi-step execution of TMs.

lemma *state_in_range_step*
$: [\![a \leq length\ A\ div\ 2; step0\ (a, b, c)\ A = (st, l, r); composable\_tm\ (A,0)]\!]$
$\Longrightarrow st \leq length\ A\ div\ 2$
$\langle proof \rangle$

lemma *state_in_range*: $[\![steps0\ (Suc\ 0, tp)\ A\ stp = (st, l, r); composable\_tm\ (A, 0)]\!]$

```
  ⟹ st ≤ length A div 2
⟨proof⟩

lemma rec_t_eq_steps:
  composable_tm (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 composable_tm.simps[simp del]

lemma nonstop_t_eq:
  ⟦steps0 (Suc 0, Bk↑l, <lm>) tp stp = (0, Bk↑ m, Oc↑ rs @ Bk↑ n);
   composable_tm (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);
    composable_tm (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⟩

5.7 Correctness of rec_F with respect to execution of TMs compiled as Recursive Functions

The correctness of *rec_F*, which relates the interpreter function *rec_F* with the execution of TMs.

lemma *terminate_halt*:
⟦steps0 (Suc 0, Bk↑l, <lm>) tp stp = (0, Bk↑m, Oc↑rs@Bk↑n);
composable_tm (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);
composable_tm (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);
composable_tm (tp,0); 0<rs⟧
⟹ rec_exec rec_F [code tp, (bl2wc (<lm>))] = (rs − Suc 0)
⟨proof⟩

end

Chapter 6

Construction of a Universal Turing Machine

theory *UTM*
 imports *Recursive Abacus UF HOL.GCD Turing_Hoare*
begin

6.1 Wang coding of input arguments

The direct compilation of the universal function *rec_F* can not give us the *utm*, because *rec_F* is of arity 2, where the first argument represents the Gödel coding of the TM being simulated and the second argument represents the right number (in Wang's coding) of the TM tape. (Notice, the left number is always *0* at the very beginning). However, the *utm* needs to simulate the execution of any TM which may take many input arguments.

Therefore, an 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 *wcode_tm*) can not be constructed by compiling from any recursive function, because every recursive function takes a fixed number of input arguments, while *wcode_tm* needs to take varying number of arguments and tranform them into Wang's coding. Therefore, this section gives a direct construction of *wcode_tm* 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 6.1 and 6.2.

As shown in Figure 6.1, the input of $prepare$ is the same as the the input of *utm*,

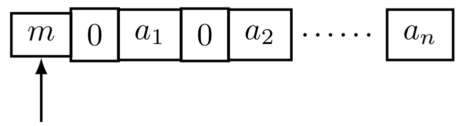

Figure 6.1: The input of TM $prepare$

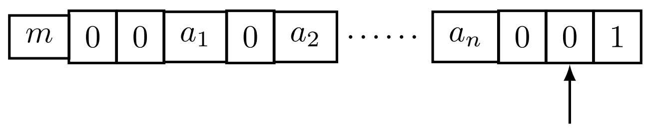

Figure 6.2: The output of TM $prepare$

where m is the Gödel 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 6.2, which is convenient for the generation of Wang's coding 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 6.2). In Figure 6.2, 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 6.3.

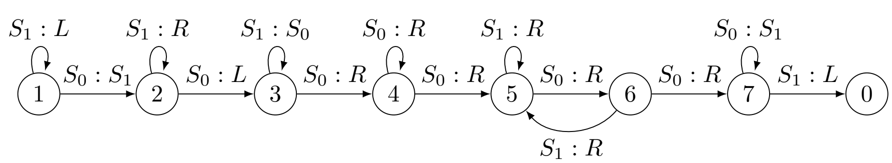

Figure 6.3: The diagram of TM $prepare$

The purpose of TM $mainwork$ is to compute 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 6.4, 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 6.5, 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 6.6, 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 6.7. 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 6.8 and 6.9 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 6.10. 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.

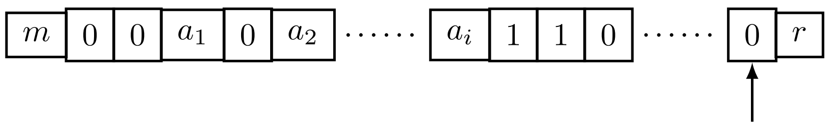

Figure 6.4: The first situation for TM $mainwork$ to consider

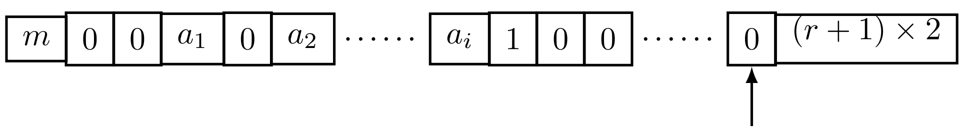

Figure 6.5: The output for the first case of TM $mainwork$'s processing

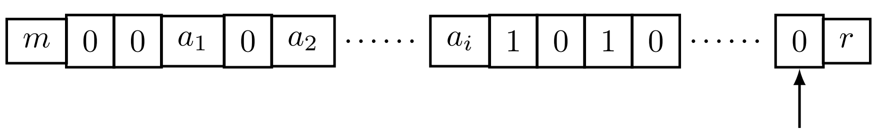

Figure 6.6: The second situation for TM $mainwork$ to consider

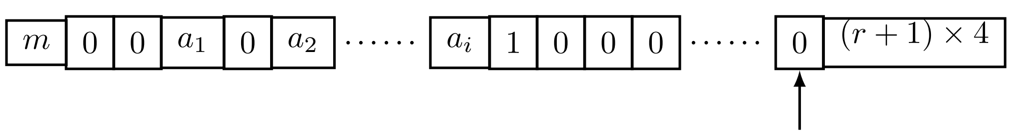

Figure 6.7: The output for the second case of TM $mainwork$'s processing

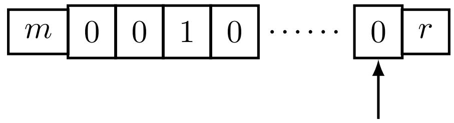

Figure 6.8: The third situation for TM $mainwork$ to consider

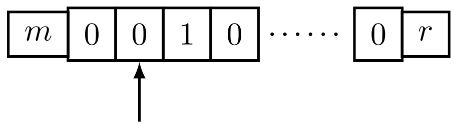

Figure 6.9: The output for the third case of TM $mainwork$'s processing

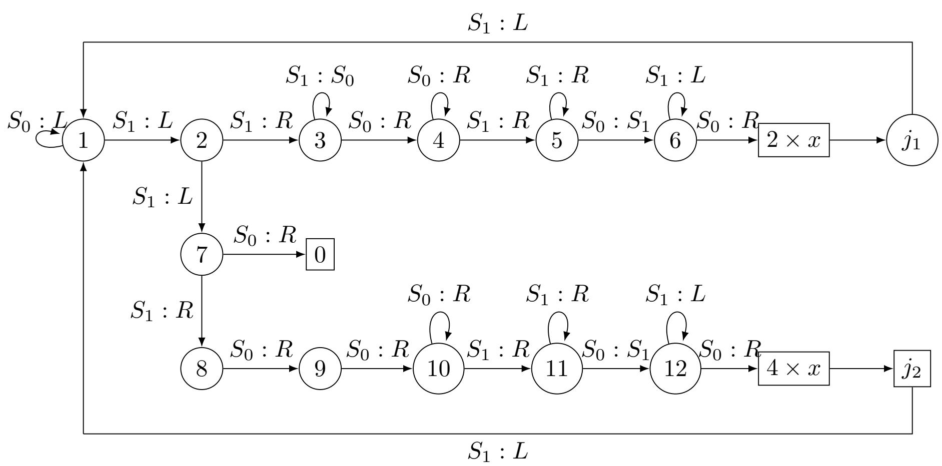

Figure 6.10: The diagram of TM $mainwork$

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 6.11 and 6.12 respectively. The diagram of TM $adjust$ is shown in Figure 6.13.

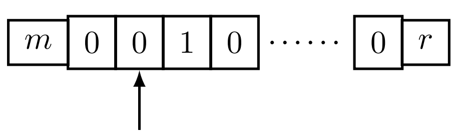

Figure 6.11: Initial configuration of TM $adjust$

definition *rec_twice* :: *recf*
where
rec_twice = *Cn 1 rec_mult* [*id 1 0*, *constn 2*]

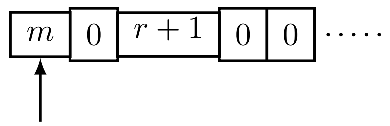

Figure 6.12: Final configuration of TM $adjust$

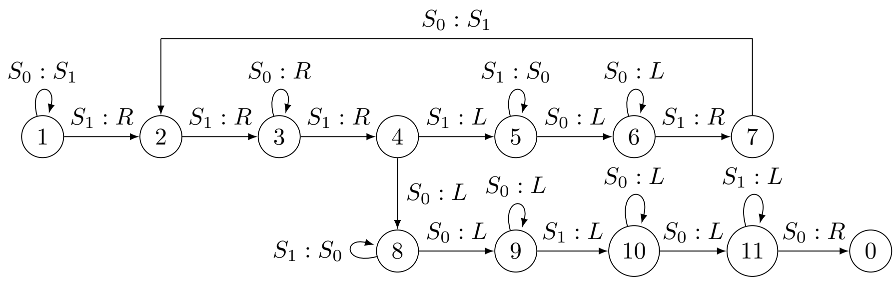

Figure 6.13: Diagram of TM $adjust$

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 *twice_compile_tm* :: *instr list*
where
twice_compile_tm= (*tm_of abc_twice* @ (*shift* (*mopup_n_tm 1*) (*length* (*tm_of abc_twice*) *div 2*)))

definition *twice_tm* :: *instr list*
where

$twice\_tm = adjust0\ twice\_compile\_tm$

definition $fourtimes\_compile\_tm$ :: $instr\ list$
where
$fourtimes\_compile\_tm = (tm\_of\ abc\_fourtimes\ @\ (shift\ (mopup\_n\_tm\ 1)\ (length\ (tm\_of\ abc\_fourtimes)\ div\ 2)))$

definition $fourtimes\_tm$ :: $instr\ list$
where
$fourtimes\_tm = adjust0\ fourtimes\_compile\_tm$

definition $twice\_tm\_len$ :: nat
where
$twice\_tm\_len = length\ twice\_tm\ div\ 2$

definition $wcode\_main\_first\_part\_tm$:: $instr\ list$
where

$$
\begin{aligned}
wcode\_main\_first\_part\_tm \stackrel{def}{=}\ & [(L, 1), (L, 2), (L, 7), (R, 3),\\
& (R, 4), (WB, 3), (R, 4), (R, 5),\\
& (WO, 6), (R, 5), (R, 13), (L, 6),\\
& (R, 0), (R, 8), (R, 9), (Nop, 8),\\
& (R, 10), (WB, 9), (R, 10), (R, 11),\\
& (WO, 12), (R, 11), (R, twice\_tm\_len + 14), (L, 12)]
\end{aligned}
$$

definition $wcode\_main\_tm$ :: $instr\ list$
where

$$
\begin{aligned}
wcode\_main\_tm = (& wcode\_main\_first\_part\_tm\ @\ shift\ twice\_tm\ 12\ @\ [(L, 1), (L, 1)]\\
& @\ shift\ fourtimes\_tm\ (twice\_tm\_len + 13)\ @\ [(L, 1), (L, 1)])
\end{aligned}
$$

fun bl_bin :: $cell\ list \Rightarrow nat$
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 $bin\_inv\_t = cell\ list \Rightarrow nat \Rightarrow tape \Rightarrow bool$

fun $wcode\_before\_double$ :: bin_inv_t
where

$$
\begin{aligned}
& wcode\_before\_double\ ires\ rs\ (l, r) =\\
& \quad (\exists\ ln\ rn.\ l = Bk\ \#\ Bk\ \#\ Bk\uparrow(ln)\ @\ Oc\ \#\ ires\ \wedge\\
& \qquad r = Oc\uparrow((Suc\ (Suc\ rs)))\ @\ Bk\uparrow(rn\ ))
\end{aligned}
$$

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*↑(*mr*) @ *Oc*↑(*Suc rs*) @ *Bk*↑(*rn*) ∧
ml + *mr* > *Suc 0*)

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*) =
(∃ *ml mr ln rn*.
l = *Oc*↑(*ml*) @ *Bk* # *Bk* # *Bk*↑(*ln*) @ *Oc* # *ires* ∧
r = *Oc*↑(*mr*) @ *Bk*↑(*rn*) ∧
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*) =
(∃ *ln rn*. *l* = *Bk* # *Bk*↑(*ln*) @ *Oc* # *ires* ∧
r = *Bk* # *Oc*↑((*Suc* (*Suc rs*))) @ *Bk*↑(*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*) =
(∃ *ml mr ln rn*.
l = *Oc*↑(*ml*) @ *Bk* # *Bk* # *Bk*↑(*ln*) @ *Oc* # *ires* ∧
r = *Oc*↑(*mr*) @ *Bk*↑(*rn*) ∧
ml + *mr* = *Suc* (*Suc rs*) ∧ *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*) ∨
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* ≠ [] ⟹ <*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* $\Rightarrow$ *nat*
where
wcode_double_case_state $(st, l, r) =$
13 $-$ *st*

fun *wcode_double_case_step* :: *config* $\Rightarrow$ *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* $\Rightarrow$ *nat* $\times$ *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* $\times$ *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*
$\langle proof \rangle$

lemma *wf_wcode_double_case_le*[*intro*]: *wf wcode_double_case_le*
$\langle proof \rangle$

lemma *fetch_wcode_main_tm*[*simp*]:
fetch wcode_main_tm $($*Suc 0*$)$ *Bk* $= (L,$ *Suc 0*$)$
fetch wcode_main_tm $($*Suc 0*$)$ *Oc* $= (L,$ *Suc* $($*Suc 0*$))$
fetch wcode_main_tm $($*Suc* $($*Suc 0*$))$ *Oc* $= (R, 3)$

fetch wcode_main_tm (*Suc* (*Suc 0*)) *Bk* = (*L*, *7*)
fetch wcode_main_tm (*Suc* (*Suc* (*Suc 0*))) *Bk* = (*R*, *4*)
fetch wcode_main_tm (*Suc* (*Suc* (*Suc 0*))) *Oc* = (*WB*, *3*)
fetch wcode_main_tm 4 Bk = (*R*, *4*)
fetch wcode_main_tm 4 Oc = (*R*, *5*)
fetch wcode_main_tm 5 Oc = (*R*, *5*)
fetch wcode_main_tm 5 Bk = (*WO*, *6*)
fetch wcode_main_tm 6 Bk = (*R*, *13*)
fetch wcode_main_tm 6 Oc = (*L*, *6*)
fetch wcode_main_tm 7 Oc = (*R*, *8*)
fetch wcode_main_tm 7 Bk = (*R*, *0*)
fetch wcode_main_tm 8 Bk = (*R*, *9*)
fetch wcode_main_tm 9 Bk = (*R*, *10*)
fetch wcode_main_tm 9 Oc = (*WB*, *9*)
fetch wcode_main_tm 10 Bk = (*R*, *10*)
fetch wcode_main_tm 10 Oc = (*R*, *11*)
fetch wcode_main_tm 11 Bk = (*WO*, *12*)
fetch wcode_main_tm 11 Oc = (*R*, *11*)
fetch wcode_main_tm 12 Oc = (*L*, *12*)
fetch wcode_main_tm 12 Bk = (*R*, *twice_tm_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* ∧ *list* = *ba*⟧

$\Longrightarrow$ *wcode_erase1 ires rs* (aa, ba)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *wcode_erase1_nonempty_snd*[*simp*]: *wcode_erase1 ires rs* $(b, [])$ = *False*
⟨*proof*⟩

lemma *wcode_on_right_moving_1_nonempty_snd*[*simp*]: *wcode_on_right_moving_1 ires rs* $(b, [])$ = *False*
⟨*proof*⟩

lemma *wcode_on_right_moving_1_BkE*[*elim*]:
⟦*wcode_on_right_moving_1 ires rs* $(b, Bk \# ba)$; $Bk \# b = aa \wedge list = b$⟧ $\Longrightarrow$
wcode_on_right_moving_1 ires rs (aa, ba)
⟨*proof*⟩

lemma *wcode_on_right_moving_1_OcE*[*elim*]:
⟦*wcode_on_right_moving_1 ires rs* $(b, Oc \# ba)$; $Oc \# b = aa \wedge list = ba$⟧
$\Longrightarrow$ *wcode_goon_right_moving_1 ires rs* (aa, ba)
⟨*proof*⟩

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)
⟨*proof*⟩

lemma *wcode_erase1_OcE*[*elim*]: ⟦*wcode_erase1 ires rs* $(aa, Oc \# list)$; $b = aa \wedge Bk \# list = ba$⟧ $\Longrightarrow$
wcode_erase1 ires rs (aa, ba)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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)
⟨*proof*⟩

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*]

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*)) *wcode_main_tm 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*⟩

lemma *twice_lemma*: *rec_exec rec_twice* [*rs*] = *2*∗*rs*
⟨*proof*⟩

lemma *twice_tm_correct*:
∃ *stp ln rn*. *steps0* (*Suc 0*, *Bk* # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))
(*tm_of abc_twice* @ *shift* (*mopup_n_tm* (*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*:

$[\![0 < a;\ a \leq \mathit{length\ ap\ div\ 2};\ \mathit{fetch\ ap\ a\ b} = (\mathit{aa}, 0)]\!]$
$\Longrightarrow \mathit{fetch}\ (\mathit{adjust0\ ap})\ a\ b = (\mathit{aa}, \mathit{Suc}\ (\mathit{length\ ap\ div\ 2}))$
$\langle \mathit{proof} \rangle$

lemma *adjust_fetch_norm*:
$[\![\mathit{st} > 0;\ \mathit{st} \leq \mathit{length\ tp\ div\ 2};\ \mathit{fetch\ ap\ st\ b} = (\mathit{aa}, \mathit{ns});\ \mathit{ns} \neq 0]\!]$
$\Longrightarrow \mathit{fetch}\ (\mathit{adjust0\ ap})\ \mathit{st}\ b = (\mathit{aa}, \mathit{ns})$
$\langle \mathit{proof} \rangle$

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

lemma *adjust_step_eq*:
assumes *exec*: $\mathit{step0}\ (\mathit{st},l,r)\ \mathit{ap} = (\mathit{st}', l', r')$
and *composable_tm* $(\mathit{ap}, 0)$
and *notfinal*: $\mathit{st}' > 0$
shows $\mathit{step0}\ (\mathit{st}, l, r)\ (\mathit{adjust0\ ap}) = (\mathit{st}', l', r')$
$\langle \mathit{proof} \rangle$

lemma *adjust_steps_eq*:
assumes *exec*: $\mathit{steps0}\ (\mathit{st},l,r)\ \mathit{ap\ stp} = (\mathit{st}', l', r')$
and *composable_tm* $(\mathit{ap}, 0)$
and *notfinal*: $\mathit{st}' > 0$
shows $\mathit{steps0}\ (\mathit{st}, l, r)\ (\mathit{adjust0\ ap})\ \mathit{stp} = (\mathit{st}', l', r')$
$\langle \mathit{proof} \rangle$

lemma *adjust_halt_eq*:
assumes *exec*: $\mathit{steps0}\ (1, l, r)\ \mathit{ap\ stp} = (0, l', r')$
and *composable_tm*: *composable_tm* $(\mathit{ap}, 0)$
shows $\exists\ \mathit{stp}.\ \mathit{steps0}\ (\mathit{Suc}\ 0, l, r)\ (\mathit{adjust0\ ap})\ \mathit{stp} =$
$(\mathit{Suc}\ (\mathit{length\ ap\ div\ 2}), l', r')$
$\langle \mathit{proof} \rangle$

lemma *composable_tm_twice_compile_tm* [*simp*]: *composable_tm* (*twice_compile_tm*, 0)
$\langle \mathit{proof} \rangle$

lemma *twice_tm_change_term_state*:
$\exists$ *stp ln rn*. *steps0* (*Suc 0*, *Bk* # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *twice_tm stp*
= (*Suc twice_tm_len*, *Bk*↑(*ln*) @ *Bk* # *Bk* # *ires*, *Oc*↑(*Suc* ($2 * \mathit{rs}$)) @ *Bk*↑(*rn*))
$\langle \mathit{proof} \rangle$

lemma *length_wcode_main_first_part_tm_even*[*intro*]: *length wcode_main_first_part_tm mod 2* $= 0$
$\langle \mathit{proof} \rangle$

lemma *twice_tm_append_pre*:
steps0 (*Suc 0*, *Bk* # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *twice_tm stp*
= (*Suc twice_tm_len*, *Bk*↑(*ln*) @ *Bk* # *Bk* # *ires*, *Oc*↑(*Suc* ($2 * \mathit{rs}$)) @ *Bk*↑(*rn*))
$\Longrightarrow$ *steps0* (*Suc 0* + *length wcode_main_first_part_tm div 2*, *Bk* # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))

(wcode_main_first_part_tm @ shift twice_tm (length wcode_main_first_part_tm div 2) @
([(L, 1), (L, 1)] @ shift fourtimes_tm (twice_tm_len + 13) @ [(L, 1), (L, 1)])) stp
= (Suc (twice_tm_len) + length wcode_main_first_part_tm div 2,
Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (2 ∗ rs)) @ Bk↑(rn))
⟨proof⟩

lemma twice_tm_append:
∃ stp ln rn. steps0 (Suc 0 + length wcode_main_first_part_tm div 2, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n))
(wcode_main_first_part_tm @ shift twice_tm (length wcode_main_first_part_tm div 2) @
([(L, 1), (L, 1)] @ shift fourtimes_tm (twice_tm_len + 13) @ [(L, 1), (L, 1)])) stp
= (Suc (twice_tm_len) + length wcode_main_first_part_tm div 2, Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (2 ∗ rs)) @ Bk↑(rn))
⟨proof⟩

lemma mopup_mod2: length (mopup_n_tm k) mod 2 = 0
⟨proof⟩

lemma fetch_wcode_main_tm_Oc[simp]: fetch wcode_main_tm (Suc (twice_tm_len + length wcode_main_first_part_tm div 2)) Oc
= (L, Suc 0)
⟨proof⟩

lemma wcode_jump1:
∃ stp ln rn. steps0 (Suc (twice_tm_len) + length wcode_main_first_part_tm div 2,
Bk↑(m) @ Bk # Bk # ires, Oc↑(Suc (2 ∗ rs)) @ Bk↑(n))
wcode_main_tm stp
= (Suc 0, Bk↑(ln) @ Bk # ires, Bk # Oc↑(Suc (2 ∗ rs)) @ Bk↑(rn))
⟨proof⟩

lemma wcode_main_first_part_len[simp]:
length wcode_main_first_part_tm = 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)) wcode_main_tm 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
 $wcode\_on\_left\_moving\_2\_O\ ires\ rs\ (l, r) =$
 $(\exists\ ln\ rn.\ l = Bk \# Oc \# ires \wedge$
 $r = Oc \# Bk\uparrow(ln) @ Bk \# Bk \# Oc\uparrow(Suc\ rs) @ Bk\uparrow(rn))$

fun *wcode_on_left_moving_2* :: *bin_inv_t*
 where
 $wcode\_on\_left\_moving\_2\ ires\ rs\ (l, r) =$
 $(wcode\_on\_left\_moving\_2\_B\ ires\ rs\ (l, r) \vee$
 $wcode\_on\_left\_moving\_2\_O\ ires\ rs\ (l, r))$

fun *wcode_on_checking_2* :: *bin_inv_t*
 where
 $wcode\_on\_checking\_2\ ires\ rs\ (l, r) =$
 $(\exists\ ln\ rn.\ l = Oc\#ires \wedge$
 $r = Bk \# Oc \# Bk\uparrow(ln) @ Bk \# Bk \# Oc\uparrow(Suc\ rs) @ Bk\uparrow(rn))$

fun *wcode_goon_checking* :: *bin_inv_t*
 where
 $wcode\_goon\_checking\ ires\ rs\ (l, r) =$
 $(\exists\ ln\ rn.\ l = ires \wedge$
 $r = Oc \# Bk \# Oc \# Bk\uparrow(ln) @ Bk \# Bk \# Oc\uparrow(Suc\ rs) @ Bk\uparrow(rn))$

fun *wcode_right_move* :: *bin_inv_t*
 where
 $wcode\_right\_move\ ires\ rs\ (l, r) =$
 $(\exists\ ln\ rn.\ l = Oc \# ires \wedge$
 $r = Bk \# Oc \# Bk\uparrow(ln) @ Bk \# Bk \# Oc\uparrow(Suc\ rs) @ Bk\uparrow(rn))$

fun *wcode_erase2* :: *bin_inv_t*
 where
 $wcode\_erase2\ ires\ rs\ (l, r) =$
 $(\exists\ ln\ rn.\ l = Bk \# Oc \# ires \wedge$
 $tl\ r = Bk\uparrow(ln) @ Bk \# Bk \# Oc\uparrow(Suc\ rs) @ Bk\uparrow(rn))$

fun *wcode_on_right_moving_2* :: *bin_inv_t*
 where
 $wcode\_on\_right\_moving\_2\ ires\ rs\ (l, r) =$
 $(\exists\ ml\ mr\ rn.\ l = Bk\uparrow(ml) @ Oc \# ires \wedge$
 $r = Bk\uparrow(mr) @ Oc\uparrow(Suc\ rs) @ Bk\uparrow(rn) \wedge ml + mr > Suc\ 0)$

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) =
      (∃ ln rn. l = Bk # Bk↑(ln) @ Oc # ires ∧
           r = Bk # Oc↑(Suc (Suc rs)) @ Bk↑(rn))

fun wcode_backto_standard_pos_2_O :: bin_inv_t
 where
  wcode_backto_standard_pos_2_O ires rs (l, r) =
     (∃ ml mr ln rn. l = Oc↑(ml )@ Bk # Bk # Bk↑(ln) @ Oc # ires ∧
              r = Oc↑(mr) @ Bk↑(rn) ∧
              ml + mr = (Suc (Suc rs)) ∧ 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) ∨
      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) =
     (∃ ln rn. l = Bk # Bk # Bk↑(ln) @ Oc # ires ∧
          r = Oc↑(Suc (Suc rs)) @ Bk↑(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 ⇒ 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 = twice_tm_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 *gr1_conv_Suc*:*Suc 0 < mr ⟷ (∃ nat. mr = Suc (Suc nat))* ⟨*proof*⟩

lemma *wcode_on_left_moving_2[simp]*:

wcode_on_left_moving_2 ires rs $(b, Bk \# list) \Longrightarrow$ *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* $(b, Bk \# list)$
$\Longrightarrow$ *wcode_goon_checking ires rs* $(tl\ b, hd\ b \# Bk \# list)$
⟨*proof*⟩

lemma *wcode_erase2_via_move* [*simp*]: *wcode_right_move ires rs* $(b, Bk \# list) \Longrightarrow$ *wcode_erase2 ires rs* $(Bk \# b, list)$
⟨*proof*⟩

lemma *wcode_on_right_moving_2_via_erase2*[*simp*]:
wcode_erase2 ires rs $(b, Bk \# list) \Longrightarrow$ *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* $(b, Bk \# list)$
$\Longrightarrow$ *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 $(b, Bk \# list) \Longrightarrow$
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* $(b, Oc \# list)$
$\Longrightarrow$
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 $(b, []) \Longrightarrow$
wcode_backto_standard_pos_2 ires rs $(b, [Oc])$
⟨*proof*⟩

lemma *wcode_goon_checking_cases*[*simp*]: *wcode_goon_checking ires rs* $(b, Oc \# list) \Longrightarrow$
$(b = [] \longrightarrow$ *wcode_right_move ires rs* $([Oc], list)) \wedge$
$(b \neq [] \longrightarrow$ *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* $(b, Oc \# list)$
$\Longrightarrow$ *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* = *twice_tm_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*)) *wcode_main_tm stp*) *in*
∃ *n* .*P* (*f n*) ∧ *Q* (*f* (*n*::*nat*))
⟨*proof*⟩

definition *fourtimes_tm_len* :: *nat*
where
fourtimes_tm_len = (*length fourtimes_tm 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 *fourtimes_tm_correct*:
∃*stp ln rn*. *steps0* (*Suc 0*, *Bk* # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))
(*tm_of abc_fourtimes* @ *shift* (*mopup_n_tm 1*) (*length* (*tm_of abc_fourtimes*) *div 2*)) *stp* =
(*0*, *Bk*↑(*ln*) @ *Bk* # *Bk* # *ires*, *Oc*↑(*Suc* (*4* ∗ *rs*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

lemma *composable_fourtimes_tm*[*intro*]: *composable_tm* (*fourtimes_compile_tm*, *0*)
⟨*proof*⟩

lemma *fourtimes_tm_change_term_state*:
∃ *stp ln rn*. *steps0* (*Suc 0*, *Bk* # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *fourtimes_tm stp*
= (*Suc fourtimes_tm_len*, *Bk*↑(*ln*) @ *Bk* # *Bk* # *ires*, *Oc*↑(*Suc* (*4* ∗ *rs*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

lemma *length_twice_tm_even*[*intro*]: *is_even* (*length twice_tm*)
⟨*proof*⟩

lemma *fourtimes_tm_append_pre*:
steps0 (*Suc 0*, *Bk* # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *fourtimes_tm stp*
= (*Suc fourtimes_tm_len*, *Bk*↑(*ln*) @ *Bk* # *Bk* # *ires*, *Oc*↑(*Suc* (*4* ∗ *rs*)) @ *Bk*↑(*rn*))
⟹ *steps0* (*Suc 0* + *length* (*wcode_main_first_part_tm* @
shift twice_tm (*length wcode_main_first_part_tm div 2*) @ [(*L*, *1*), (*L*, *1*)]) *div 2*,
Bk # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))
((*wcode_main_first_part_tm* @
shift twice_tm (*length wcode_main_first_part_tm div 2*) @ [(*L*, *1*), (*L*, *1*)]) @
shift fourtimes_tm (*length* (*wcode_main_first_part_tm* @
shift twice_tm (*length wcode_main_first_part_tm div 2*) @ [(*L*, *1*), (*L*, *1*)]) *div 2*) @ ([(*L*, *1*),
(*L*, *1*)])) *stp*
= ((*Suc fourtimes_tm_len*) + *length* (*wcode_main_first_part_tm* @
shift twice_tm (*length wcode_main_first_part_tm 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 *twice_tm_len_plus_14*[*simp*]: *twice_tm_len* + *14* = *14* + *length* (*shift twice_tm 12*) *div*
2
⟨*proof*⟩

lemma *fourtimes_tm_append*:
∃ *stp ln rn*.
steps0 (*Suc 0* + *length* (*wcode_main_first_part_tm* @ *shift twice_tm*
(*length wcode_main_first_part_tm div 2*) @ [(*L*, *1*), (*L*, *1*)]) *div 2*,
Bk # *Bk* # *ires*, *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))

((*wcode_main_first_part_tm* @ *shift twice_tm* (*length wcode_main_first_part_tm div 2*) @
[(*L*, *1*), (*L*, *1*)]) @ *shift fourtimes_tm* (*twice_tm_len* + *13*) @ [(*L*, *1*), (*L*, *1*)]) *stp*
= (*Suc fourtimes_tm_len* + *length* (*wcode_main_first_part_tm* @ *shift twice_tm*
(*length wcode_main_first_part_tm 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 fourtimes_tm mod 2* = *0*
⟨*proof*⟩

lemma *twice_tm_even*[*simp*]: *2* ∗ (*length twice_tm div 2*) = *length twice_tm*
⟨*proof*⟩

lemma *fourtimes_tm_even*[*simp*]: *2* ∗ (*length fourtimes_tm div 2*) = *length fourtimes_tm*
⟨*proof*⟩

lemma *fetch_wcode_tm_14_Oc*: *fetch wcode_main_tm* (*14* + *length twice_tm div 2* + *fourtimes_tm_len*)
Oc
= (*L*, *Suc 0*)
⟨*proof*⟩

lemma *fetch_wcode_tm_14_Bk*: *fetch wcode_main_tm* (*14* + *length twice_tm div 2* + *fourtimes_tm_len*)
Bk
= (*L*, *Suc 0*)
⟨*proof*⟩

lemma *fetch_wcode_tm_14* [*simp*]: *fetch wcode_main_tm* (*14* + *length twice_tm div 2* + *fourtimes_tm_len*)
b
= (*L*, *Suc 0*)
⟨*proof*⟩

lemma *wcode_jump2*:
∃ *stp ln rn*. *steps0* (*twice_tm_len* + *14* + *fourtimes_tm_len*
, *Bk* # *Bk* # *Bk*↑(*lnb*) @ *Oc* # *ires*, *Oc*↑(*Suc* (*4* ∗ *rs* + *4*)) @ *Bk*↑(*rnb*)) *wcode_main_tm 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*))
wcode_main_tm 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*) $\Longrightarrow$ *b* $\neq$ []
wcode_on_checking_3 ires rs (*b*, *Bk* # *list*) $\Longrightarrow$ *b* $\neq$ []
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*) $\Longrightarrow$
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*)) *wcode_main_tm stp*) *in*
$\exists$ *n* .*P* (*f n*) $\wedge$ *Q* (*f* (*n*::*nat*))
⟨*proof*⟩

declare *wcode_halt_case_inv*.*simps*[*simp del*]
lemma *leading_Oc*[*intro*]: $\exists$ *xs*. (<*rev list* @ [*aa*::*nat*]> :: *cell list*) = *Oc* # *xs*
⟨*proof*⟩

lemma *wcode_halt_case*:
$\exists$ *stp ln rn*. *steps0* (*Suc 0*, *Bk* # *Bk*↑(*m*) @ *Oc* # *Bk* # *Bk* # *ires*, *Bk* # *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *wcode_main_tm 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 *wcode_main_tm_lemma_pre*:
⟦*args* $\neq$ []; *lm* = <*args*::*nat list*>⟧ $\Longrightarrow$
$\exists$ *stp ln rn*. *steps0* (*Suc 0*, *Bk* # *Bk*↑(*m*) @ *rev lm* @ *Bk* # *Bk* # *ires*, *Bk* # *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *wcode_main_tm*
stp
= (*0*, *Bk* # *ires*, *Bk* # *Oc* # *Bk*↑(*ln*) @ *Bk* # *Bk* # *Oc*↑(*bl_bin lm* + *rs* ∗ 2^(*length lm* − *1*)) @ *Bk*↑(*rn*))
⟨*proof*⟩

definition *wcode_prepare_tm* :: *instr list*
where
wcode_prepare_tm $\stackrel{def}{=}$
[(*WO*, *2*), (*L*, *1*), (*L*, *3*), (*R*, *2*), (*R*, *4*), (*WB*, *3*),
(*R*, *4*), (*R*, *5*), (*R*, *6*), (*R*, *5*), (*R*, *7*), (*R*, *5*),

```
    (WO, 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* $\stackrel{def}{=}$ (*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_wcode_prepare_tm*[*simp*]:
fetch wcode_prepare_tm (*Suc 0*) *Bk* = (*WO*, *2*)
fetch wcode_prepare_tm (*Suc 0*) *Oc* = (*L*, *1*)
fetch wcode_prepare_tm (*Suc* (*Suc 0*)) *Bk* = (*L*, *3*)
fetch wcode_prepare_tm (*Suc* (*Suc 0*)) *Oc* = (*R*, *2*)
fetch wcode_prepare_tm (*Suc* (*Suc* (*Suc 0*))) *Bk* = (*R*, *4*)
fetch wcode_prepare_tm (*Suc* (*Suc* (*Suc 0*))) *Oc* = (*WB*, *3*)
fetch wcode_prepare_tm 4 Bk = (*R*, *4*)
fetch wcode_prepare_tm 4 Oc = (*R*, *5*)
fetch wcode_prepare_tm 5 Oc = (*R*, *5*)
fetch wcode_prepare_tm 5 Bk = (*R*, *6*)
fetch wcode_prepare_tm 6 Oc = (*R*, *5*)
fetch wcode_prepare_tm 6 Bk = (*R*, *7*)
fetch wcode_prepare_tm 7 Oc = (*L*, *0*)
fetch wcode_prepare_tm 7 Bk = (*WO*, *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*, [])⟧

$\Longrightarrow$
wprepare_loop_goon m lm $(Bk \# b, [])$
⟨*proof*⟩

lemma *wprepare_loop_goon_nonempty_fst*[*simp*]: ⟦$lm \neq []$; *wprepare_loop_goon m lm* $(b, [])$⟧ $\Longrightarrow b \neq []$
⟨*proof*⟩

lemma *wprepare_add_one2_Bk_empty*[*simp*]:⟦$lm \neq []$; *wprepare_loop_goon m lm* $(b, [])$⟧ $\Longrightarrow$
wprepare_add_one2 m lm $(Bk \# b, [])$
⟨*proof*⟩

lemma *wprepare_add_one2_nonempty_fst*[*simp*]: *wprepare_add_one2 m lm* $(b, []) \Longrightarrow b \neq []$
⟨*proof*⟩

lemma *wprepare_add_one2_Oc*[*simp*]: *wprepare_add_one2 m lm* $(b, []) \Longrightarrow$ *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*]:
⟦$lm \neq []$; *wprepare_add_one m lm* $(b, Bk \# list)$⟧
$\Longrightarrow (b = [] \longrightarrow$ *wprepare_goto_first_end m lm* $([], Oc \# list)) \wedge$
$(b \neq [] \longrightarrow$ *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 $(b, Bk \# list) \Longrightarrow b \neq []$
⟨*proof*⟩

declare *replicate_Suc*[*simp*]

lemma *wprepare_erase_elem_Bk_rest*[*simp*]: *wprepare_goto_first_end m lm* $(b, Bk \# list) \Longrightarrow$
wprepare_erase m lm $(tl\ b, hd\ b \# Bk \# list)$
⟨*proof*⟩

lemma *wprepare_erase_Bk_nonempty_fst*[*simp*]: *wprepare_erase m lm* $(b, Bk \# list) \Longrightarrow b \neq []$
⟨*proof*⟩

lemma *wprepare_goto_start_pos_Bk*[*simp*]: *wprepare_erase m lm* $(b, Bk \# list) \Longrightarrow$
wprepare_goto_start_pos m lm $(Bk \# b, list)$
⟨*proof*⟩

lemma *wprepare_add_one_Bk_nonempty_snd*[*simp*]: ⟦*wprepare_add_one m lm* $(b, Bk \# list)$⟧ $\Longrightarrow list \neq []$
⟨*proof*⟩

lemma *wprepare_goto_first_end_nonempty_snd_tl*[*simp*]:
$[\![lm \neq []; \ wprepare\_goto\_first\_end\ m\ lm\ (b, Bk \# list)]\!] \Longrightarrow list \neq []$
⟨*proof*⟩

lemma *wprepare_erase_Bk_nonempty_list*[*simp*]: $[\![lm \neq []; wprepare\_erase\ m\ lm\ (b, Bk \# list)]\!] \Longrightarrow list \neq []$
⟨*proof*⟩

lemma *wprepare_goto_start_pos_Bk_nonempty*[*simp*]: $[\![lm \neq []; \ wprepare\_goto\_start\_pos\ m\ lm\ (b, Bk \# list)]\!] \Longrightarrow list \neq []$
⟨*proof*⟩

lemma *wprepare_goto_start_pos_Bk_nonempty_fst*[*simp*]: $[\![lm \neq []; \ wprepare\_goto\_start\_pos\ m\ lm\ (b, Bk \# list)]\!] \Longrightarrow b \neq []$
⟨*proof*⟩

lemma *wprepare_loop_goon_Bk_nonempty*[*simp*]: $[\![lm \neq []; wprepare\_loop\_goon\ m\ lm\ (b, Bk \# list)]\!] \Longrightarrow b \neq []$
⟨*proof*⟩

lemma *wprepare_loop_goon_wprepare_add_one2_cases*[*simp*]: $[\![lm \neq []; wprepare\_loop\_goon\ m\ lm\ (b, Bk \# list)]\!] \Longrightarrow$
$(list = [] \longrightarrow wprepare\_add\_one2\ m\ lm\ (Bk \# b, [])) \wedge$
$(list \neq [] \longrightarrow wprepare\_add\_one2\ m\ lm\ (Bk \# b, list))$
⟨*proof*⟩

lemma *wprepare_add_one2_nonempty*[*simp*]: $wprepare\_add\_one2\ m\ lm\ (b, Bk \# list) \Longrightarrow b \neq []$
⟨*proof*⟩

lemma *wprepare_add_one2_cases*[*simp*]: $wprepare\_add\_one2\ m\ lm\ (b, Bk \# list) \Longrightarrow$
$(list = [] \longrightarrow wprepare\_add\_one2\ m\ lm\ (b, [Oc])) \wedge$
$(list \neq [] \longrightarrow wprepare\_add\_one2\ m\ lm\ (b, Oc \# list))$
⟨*proof*⟩

lemma *wprepare_goto_first_end_cases_Oc*[*simp*]: $wprepare\_goto\_first\_end\ m\ lm\ (b, Oc \# list)$
$\Longrightarrow (b = [] \longrightarrow wprepare\_goto\_first\_end\ m\ lm\ ([Oc], list)) \wedge$
$(b \neq [] \longrightarrow wprepare\_goto\_first\_end\ m\ lm\ (Oc \# b, list))$
⟨*proof*⟩

lemma *wprepare_erase_nonempty*[*simp*]: $wprepare\_erase\ m\ lm\ (b, Oc \# list) \Longrightarrow b \neq []$
⟨*proof*⟩

lemma *wprepare_erase_Bk*[*simp*]: $wprepare\_erase\ m\ lm\ (b, Oc \# list)$
$\Longrightarrow wprepare\_erase\ m\ lm\ (b, Bk \# list)$
⟨*proof*⟩

lemma *wprepare_goto_start_pos_Bk_move*[*simp*]: $[\![lm \neq []; wprepare\_goto\_start\_pos\ m\ lm\ (b, Bk \# list)]\!]$

$\Longrightarrow$ *wprepare_goto_start_pos m lm* $(Bk \# b, list)$
⟨*proof*⟩

lemma *wprepare_loop_start_b_nonempty*[*simp*]: *wprepare_loop_start m lm* $(b, aa) \Longrightarrow b \neq []$
⟨*proof*⟩
lemma *exists_exp_of_Bk*[*elim*]: $Bk \# list = Oc\uparrow(mr)$ @ $Bk\uparrow(rn) \Longrightarrow \exists rn.\ list = Bk\uparrow(rn)$
⟨*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*]: $[\![lm \neq []$; *wprepare_loop_start m lm* $(b, [Bk])]\!] \Longrightarrow$
wprepare_loop_goon m lm $(Bk \# b, [])$
⟨*proof*⟩

lemma *wprepare_loop_goon_in_middle_Bk_False2*[*simp*]: *wprepare_loop_start_on_rightmost m lm* $(b, Bk \# a \# lista)$
$\Longrightarrow$ *wprepare_loop_goon_in_middle m lm* $(Bk \# b, a \# lista) = False$
⟨*proof*⟩

lemma *wprepare_loop_goon_on_rightbmost_Bk_False*[*simp*]: $[\![lm \neq []$; *wprepare_loop_start_on_rightmost m lm* $(b, Bk \# a \# lista)]\!]$
$\Longrightarrow$ *wprepare_loop_goon_on_rightmost m lm* $(Bk \# b, a \# lista)$
⟨*proof*⟩

lemma *wprepare_loop_goon_in_middle_Bk_False3*[*simp*]:
assumes $lm \neq []$ *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*]: $[\![lm \neq []$; *wprepare_loop_start m lm* $(b, Bk \# a \# lista)]\!] \Longrightarrow$
wprepare_loop_goon m lm $(Bk \# b, a \# lista)$
⟨*proof*⟩

lemma *start_2_goon*:
$[\![lm \neq []$; *wprepare_loop_start m lm* $(b, Bk \# list)]\!] \Longrightarrow$

$(list = [] \longrightarrow wprepare\_loop\_goon\ m\ lm\ (Bk \# b, [])) \wedge$
$(list \neq [] \longrightarrow wprepare\_loop\_goon\ m\ lm\ (Bk \# b, list))$
⟨*proof*⟩

lemma *add_one_2_add_one*: $wprepare\_add\_one\ m\ lm\ (b, Oc \# list)$
$\Longrightarrow (hd\ b = Oc \longrightarrow (b = [] \longrightarrow wprepare\_add\_one\ m\ lm\ ([], Bk \# Oc \# list)) \wedge$
$(b \neq [] \longrightarrow wprepare\_add\_one\ m\ lm\ (tl\ b, Oc \# Oc \# list))) \wedge$
$(hd\ b \neq Oc \longrightarrow (b = [] \longrightarrow wprepare\_add\_one\ m\ lm\ ([], Bk \# Oc \# list)) \wedge$
$(b \neq [] \longrightarrow wprepare\_add\_one\ m\ lm\ (tl\ b, hd\ b \# Oc \# list)))$
⟨*proof*⟩

lemma *wprepare_loop_start_on_rightmost_Oc*[*simp*]: $wprepare\_loop\_start\_on\_rightmost\ m\ lm$
$(b, Oc \# list) \Longrightarrow$
$wprepare\_loop\_start\_on\_rightmost\ m\ lm\ (Oc \# b, list)$
⟨*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*: $wprepare\_loop\_start\ m\ lm\ (b, Oc \# list) \Longrightarrow$
$wprepare\_loop\_start\ m\ lm\ (Oc \# b, list)$
⟨*proof*⟩

lemma *wprepare_loop_goon_Oc_nonempty*[*simp*]: $wprepare\_loop\_goon\ m\ lm\ (b, Oc \# list)$
$\Longrightarrow b \neq []$
⟨*proof*⟩

lemma *wprepare_goto_start_pos_Oc_nonempty*[*simp*]: $wprepare\_goto\_start\_pos\ m\ lm\ (b, Oc$
$\# list) \Longrightarrow b \neq []$
⟨*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*: $[\![rev\ b\ @\ Oc\uparrow(mr) = Oc\uparrow(Suc\ m)\ @\ Bk \# Bk \# <lm>;$
$b \neq []; 0 < mr; Oc \# list = Oc\uparrow(mr)\ @\ Bk\uparrow(rn)]\!]$
$\Longrightarrow wprepare\_loop\_start\_on\_rightmost\ m\ lm\ (Oc \# b, list)$
⟨*proof*⟩

lemma *wprepare_loop2*: $[\![rev\ b\ @\ Oc\uparrow(mr)\ @\ Bk \# <a \# lista> = Oc\uparrow(Suc\ m)\ @\ Bk \# Bk \#$
$<lm>;$
$b \neq []; Oc \# list = Oc\uparrow(mr)\ @\ Bk \# <(a::nat) \# lista>\ @\ Bk\uparrow(rn)]\!]$
$\Longrightarrow wprepare\_loop\_start\_in\_middle\ m\ lm\ (Oc \# b, list)$
⟨*proof*⟩

lemma *wprepare_loop_goon_in_middle_cases*[*simp*]: $wprepare\_loop\_goon\_in\_middle\ m\ lm\ (b,$
$Oc \# list) \Longrightarrow$

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*>)) *wcode_prepare_tm stp*) *in*
∃ *n* .*P* (*f n*) ∧ *Q* (*f n*)
⟨*proof*⟩

lemma *composable_tm_wcode_prepare_tm*[*intro*]: *composable_tm* (*wcode_prepare_tm*, *0*)
⟨*proof*⟩

lemma *is_28_even*[*intro*]: (*28* + (*length twice_compile_tm* + *length fourtimes_compile_tm*)) *mod 2* = *0*
⟨*proof*⟩

lemma *b_le_28*[*elim*]: $(a, b) \in set\ wcode\_main\_first\_part\_tm \Longrightarrow$
$b \leq (28 + (length\ twice\_compile\_tm + length\ fourtimes\_compile\_tm))\ div\ 2$
$\langle proof \rangle$

lemma *composable_tm_change_termi*:
assumes $composable\_tm\ (tp, 0)$
shows $list\_all\ (\lambda(acn, st).\ (st \leq Suc\ (length\ tp\ div\ 2)))\ (adjust0\ tp)$
$\langle proof \rangle$

lemma *composable_tm_shift*:
assumes $list\_all\ (\lambda(acn, st).\ (st \leq y))\ tp$
shows $list\_all\ (\lambda(acn, st).\ (st \leq y + off))\ (shift\ tp\ off)$
$\langle proof \rangle$

declare *length_tp'*[*simp del*]

lemma *length_mopup_1*[*simp*]: $length\ (mopup\_n\_tm\ (Suc\ 0)) = 16$
$\langle proof \rangle$

lemma *twice_plus_28_elim*[*elim*]: $(a, b) \in set\ (shift\ (adjust0\ twice\_compile\_tm)\ 12) \Longrightarrow$
$b \leq (28 + (length\ twice\_compile\_tm + length\ fourtimes\_compile\_tm))\ div\ 2$
$\langle proof \rangle$

lemma *length_plus_28_elim2*[*elim*]: $(a, b) \in set\ (shift\ (adjust0\ fourtimes\_compile\_tm)\ (twice\_tm\_len$
$+\ 13))$
$\Longrightarrow b \leq (28 + (length\ twice\_compile\_tm + length\ fourtimes\_compile\_tm))\ div\ 2$
$\langle proof \rangle$

lemma *composable_tm_wcode_main_tm*[*intro*]: $composable\_tm\ (wcode\_main\_tm, 0)$
$\langle proof \rangle$

lemma *prepare_mainpart_lemma*:
$args \neq [] \Longrightarrow$
$\exists\ stp\ ln\ rn.\ steps0\ (Suc\ 0, [], <m \# args>)\ (wcode\_prepare\_tm\ |+|\ wcode\_main\_tm)\ stp$
$= (0,\ Bk \# Oc\uparrow(Suc\ m), Bk \# Oc \# Bk\uparrow(ln)\ @\ Bk \# Bk \# Oc\uparrow(bl\_bin\ (<args>))$
$@\ Bk\uparrow(rn))$
$\langle proof \rangle$

definition $tinres :: cell\ list \Rightarrow cell\ list \Rightarrow bool$
where
$tinres\ xs\ ys = (\exists n.\ xs = ys\ @\ Bk \uparrow n \vee ys = xs\ @\ Bk \uparrow n)$

lemma *tinres_fetch_congr*[*simp*]: $tinres\ r\ r' \Longrightarrow$
$fetch\ t\ ss\ (read\ r) =$
$fetch\ t\ ss\ (read\ r')$
$\langle proof \rangle$

lemma *nonempty_hd_tinres*[*simp*]: $[\![$*tinres r r'*; $r \neq []$; $r' \neq []]\!] \Longrightarrow$ *hd r* = *hd r'*
⟨*proof*⟩

lemma *tinres_nonempty*[*simp*]:
$[\![$*tinres r* []; $r \neq []]\!] \Longrightarrow$ *hd r* = *Bk*
$[\![$*tinres* [] *r'*; $r' \neq []]\!] \Longrightarrow$ *hd r'* = *Bk*
$[\![$*tinres r* []; $r \neq []]\!] \Longrightarrow$ *tinres* (*tl r*) []
tinres r r' $\Longrightarrow$ *tinres* (*b* # *r*) (*b* # *r'*)
⟨*proof*⟩

lemma *ex_move_tl*[*intro*]: $\exists$ *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'* $\Longrightarrow$ *tinres* (*tl r*) (*tl r'*)
⟨*proof*⟩

lemma *tinres_empty*[*simp*]:
$[\![$*tinres* [] *r'*$]\!] \Longrightarrow$ *tinres* [] (*tl r'*)
tinres r [] $\Longrightarrow$ *tinres* (*Bk* # *tl r*) [*Bk*]
tinres r [] $\Longrightarrow$ *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*)$]\!]$
$\Longrightarrow$ *la* = *lb* ∧ *tinres ra rb* ∧ *sa* = *sb*
⟨*proof*⟩

definition *wcode_adjust_tm* :: *instr list*
where
wcode_adjust_tm = [(*WO*, *1*), (*R*, *2*), (*Nop*, *2*), (*R*, *3*), (*R*, *3*), (*R*, *4*),
(*L*, *8*), (*L*, *5*), (*L*, *6*), (*WB*, *5*), (*L*, *6*), (*R*, *7*),
(*WO*, *2*), (*Nop*, *7*), (*L*, *9*), (*WB*, *8*), (*L*, *9*), (*L*, *10*),
(*L*, *11*), (*L*, *10*), (*R*, *0*), (*L*, *11*)]

lemma *fetch_wcode_adjust_tm*[*simp*]:
fetch wcode_adjust_tm (*Suc 0*) *Bk* = (*WO*, *1*)
fetch wcode_adjust_tm (*Suc 0*) *Oc* = (*R*, *2*)
fetch wcode_adjust_tm (*Suc* (*Suc 0*)) *Oc* = (*R*, *3*)
fetch wcode_adjust_tm (*Suc* (*Suc* (*Suc 0*))) *Oc* = (*R*, *4*)
fetch wcode_adjust_tm (*Suc* (*Suc* (*Suc 0*))) *Bk* = (*R*, *3*)
fetch wcode_adjust_tm 4 Bk = (*L*, *8*)
fetch wcode_adjust_tm 4 Oc = (*L*, *5*)
fetch wcode_adjust_tm 5 Oc = (*WB*, *5*)

fetch wcode_adjust_tm 5 Bk = (*L*, *6*)
fetch wcode_adjust_tm 6 Oc = (*R*, *7*)
fetch wcode_adjust_tm 6 Bk = (*L*, *6*)
fetch wcode_adjust_tm 7 Bk = (*WO*, *2*)
fetch wcode_adjust_tm 8 Bk = (*L*, *9*)
fetch wcode_adjust_tm 8 Oc = (*WB*, *8*)
fetch wcode_adjust_tm 9 Oc = (*L*, *10*)
fetch wcode_adjust_tm 9 Bk = (*L*, *9*)
fetch wcode_adjust_tm 10 Bk = (*L*, *11*)
fetch wcode_adjust_tm 10 Oc = (*L*, *10*)
fetch wcode_adjust_tm 11 Oc = (*L*, *11*)
fetch wcode_adjust_tm 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

$wadjust\_loop\_on\_left\_moving\_O\ m\ rs\ (l, r) =$
$(\exists\ ml\ mr\ ln\ rn.\ l = Oc\uparrow(ml)\ @\ Bk\ \#\ Oc\uparrow(Suc\ m\ ) \land$
$r = Oc\ \#\ Bk\uparrow(ln)\ @\ Bk\ \#\ Bk\ \#\ Oc\uparrow(mr)\ @\ Bk\uparrow(rn) \land$
$ml + mr = Suc\ rs \land mr > 0)$

fun $wadjust\_loop\_on\_left\_moving\_B :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
where
$wadjust\_loop\_on\_left\_moving\_B\ m\ rs\ (l, r) =$
$(\exists\ ml\ mr\ nl\ nr\ rn.\ l = Bk\uparrow(nl)\ @\ Oc\ \#\ Oc\uparrow(ml)\ @\ Bk\ \#\ Oc\uparrow(Suc\ m) \land$
$r = Bk\uparrow(nr)\ @\ Bk\ \#\ Bk\ \#\ Oc\uparrow(mr)\ @\ Bk\uparrow(rn) \land$
$ml + mr = Suc\ rs \land mr > 0)$

fun $wadjust\_loop\_on\_left\_moving :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
where
$wadjust\_loop\_on\_left\_moving\ m\ rs\ (l, r) =$
$(wadjust\_loop\_on\_left\_moving\_O\ m\ rs\ (l, r) \lor$
$wadjust\_loop\_on\_left\_moving\_B\ m\ rs\ (l, r))$

fun $wadjust\_loop\_right\_move2 :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
where
$wadjust\_loop\_right\_move2\ m\ rs\ (l, r) =$
$(\exists\ ml\ mr\ ln\ rn.\ l = Oc\ \#\ Oc\uparrow(ml)\ @\ Bk\ \#\ Oc\uparrow(Suc\ m) \land$
$r = Bk\uparrow(ln)\ @\ Bk\ \#\ Bk\ \#\ Oc\uparrow(mr)\ @\ Bk\uparrow(rn) \land$
$ml + mr = Suc\ rs \land mr > 0)$

fun $wadjust\_erase2 :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
where
$wadjust\_erase2\ m\ rs\ (l, r) =$
$(\exists\ ln\ rn.\ l = Bk\uparrow(ln)\ @\ Bk\ \#\ Oc\ \#\ Oc\uparrow(Suc\ rs)\ @\ Bk\ \#\ Oc\uparrow(Suc\ m) \land$
$tl\ r = Bk\uparrow(rn))$

fun $wadjust\_on\_left\_moving\_O :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
where
$wadjust\_on\_left\_moving\_O\ m\ rs\ (l, r) =$
$(\exists\ rn.\ l = Oc\uparrow(Suc\ rs)\ @\ Bk\ \#\ Oc\uparrow(Suc\ m) \land$
$r = Oc\ \#\ Bk\uparrow(rn))$

fun $wadjust\_on\_left\_moving\_B :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
where
$wadjust\_on\_left\_moving\_B\ m\ rs\ (l, r) =$
$(\exists\ ln\ rn.\ l = Bk\uparrow(ln)\ @\ Oc\ \#\ Oc\uparrow(Suc\ rs)\ @\ Bk\ \#\ Oc\uparrow(Suc\ m) \land$
$r = Bk\uparrow(rn))$

fun $wadjust\_on\_left\_moving :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
where
$wadjust\_on\_left\_moving\ m\ rs\ (l, r) =$
$(wadjust\_on\_left\_moving\_O\ m\ rs\ (l, r) \lor$
$wadjust\_on\_left\_moving\_B\ m\ rs\ (l, r))$

fun $wadjust\_goon\_left\_moving\_B :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$

where
  wadjust_goon_left_moving_B m rs $(l, r) =$
    $(\exists\ rn.\ l = Oc\uparrow(Suc\ m) \wedge$
      $r = Bk \# Oc\uparrow(Suc\ (Suc\ rs))\ @\ Bk\uparrow(rn))$

fun *wadjust_goon_left_moving_O* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
  where
  wadjust_goon_left_moving_O m rs $(l, r) =$
    $(\exists\ ml\ mr\ rn.\ l = Oc\uparrow(ml)\ @\ Bk \# Oc\uparrow(Suc\ m) \wedge$
      $r = Oc\uparrow(mr)\ @\ Bk\uparrow(rn) \wedge$
      $ml + mr = Suc\ (Suc\ rs) \wedge mr > 0)$

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* $(l, r) \vee$
     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 $(l, r) =$
    $(\exists\ rn.\ l = [] \wedge$
      $r = Bk \# Oc\uparrow(Suc\ m\ )@\ Bk \# Oc\uparrow(Suc\ (Suc\ rs))\ @\ Bk\uparrow(rn))$

fun *wadjust_backto_standard_pos_O* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
  where
  wadjust_backto_standard_pos_O m rs $(l, r) =$
    $(\exists\ ml\ mr\ rn.\ l = Oc\uparrow(ml) \wedge$
      $r = Oc\uparrow(mr)\ @\ Bk \# Oc\uparrow(Suc\ (Suc\ rs))\ @\ Bk\uparrow(rn) \wedge$
      $ml + mr = Suc\ m \wedge mr > 0)$

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* $(l, r) \vee$
    wadjust_backto_standard_pos_O m rs $(l, r))$

fun *wadjust_stop* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
  where
  wadjust_stop m rs $(l, r) =$
    $(\exists\ rn.\ l = [Bk] \wedge$
      $r = Oc\uparrow(Suc\ m\ )@\ Bk \# Oc\uparrow(Suc\ (Suc\ rs))\ @\ Bk\uparrow(rn))$

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* × *config*) ⇒ *nat* × *nat* × *nat* × *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* × *config*) × *nat* × *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*, []) ⟹ *c* ≠ []
⟨*proof*⟩

lemma *wadjust_loop_check_fst_nonempty*[*simp*]: *wadjust_loop_check m rs* (*c*, []) ⟹ *c* ≠ []
⟨*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*, []) ⟹ *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, []); $c \neq$ []⟧ ⟹
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*])) ∧
($c \neq$ [] ⟶ *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, []); $c \neq$ []; *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, []); $c \neq$ []; *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, []); $c \neq$ []⟧ ⟹
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) ⟹ $c \neq$ []
⟨*proof*⟩

lemma *wadjust_erase2_via_loop_check_Bk*[*simp*]: *wadjust_loop_check m rs* $(c, Bk \# list)$
$\Longrightarrow$ *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*); $hd\ c = Bk$⟧
$\Longrightarrow$ *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* $(c, Bk \# list)$; $c \neq []$; $hd\ c = Oc$⟧ $\Longrightarrow$
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*); $c \neq []$⟧ $\Longrightarrow$
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* $(c, Bk \# list)$; $hd\ c = Bk$⟧
$\Longrightarrow$ *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* $(c, Bk \# list)$; $hd\ c = Oc$⟧
$\Longrightarrow$ *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* $(c, b) \Longrightarrow c \neq []$
wadjust_loop_on_left_moving m rs $(c, b) \Longrightarrow c \neq []$
wadjust_loop_right_move2 m rs $(c, b) \Longrightarrow c \neq []$
wadjust_erase2 m rs $(c, Bk \# list) \Longrightarrow c \neq []$
wadjust_on_left_moving m rs $(c,b) \Longrightarrow c \neq []$
wadjust_on_left_moving_O m rs $(c, Bk \# list) = False$
wadjust_goon_left_moving m rs $(c, b) \Longrightarrow c \neq []$
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 $(c, Bk \# list)$
$\Longrightarrow$ *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 $(c, Bk \# list) \Longrightarrow$ *wadjust_loop_start m rs* $(c, Oc \# list)$
⟨*proof*⟩

lemma *wadjust_on_left_moving_Bk_via_erase*[*simp*]: *wadjust_erase2 m rs* $(c, Bk \# list) \Longrightarrow$
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* $(c, Bk \# list); hd\ c = Bk]\!]$
$\Longrightarrow$ *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* $(c, Bk \# list); hd\ c = Oc]\!]$
$\Longrightarrow$ *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* $(c, Bk \# list)$
$\Longrightarrow$
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 $(c, Bk \# list) \Longrightarrow$
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 $(c, Oc \# list) \Longrightarrow$ *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* $(c, Oc \# list) \Longrightarrow$
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 $(c, Oc \# list)$
$\Longrightarrow$ *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*) $\Longrightarrow$
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*⟧ $\Longrightarrow$
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*⟧
$\Longrightarrow$ *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*) $\Longrightarrow$
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*⟧
$\Longrightarrow$ *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*⟧ $\Longrightarrow$
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*) $\Longrightarrow$
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*) $\Longrightarrow$
wadjust_backto_standard_pos_B m rs ([], *Bk* # *Oc* # *list*)

$\langle proof \rangle$

lemma *wadjust_backto_standard_pos_B_Bk_Oc_via_O*[*simp*]:
$[\![$*wadjust_backto_standard_pos_O m rs* (*c*, *Oc* # *list*); $c \neq []$; *hd c* = *Bk*$]\!]$
$\Longrightarrow$ *wadjust_backto_standard_pos_B m rs* (*tl c*, *Bk* # *Oc* # *list*)
$\langle proof \rangle$

lemma *wadjust_backto_standard_pos_B_Oc_Oc_via_O*[*simp*]: $[\![$*wadjust_backto_standard_pos_O m rs* (*c*, *Oc* # *list*); $c \neq []$; *hd c* = *Oc*$]\!]$
$\Longrightarrow$ *wadjust_backto_standard_pos_O m rs* (*tl c*, *Oc* # *Oc* # *list*)
$\langle proof \rangle$

lemma *wadjust_backto_standard_pos_cases*[*simp*]: *wadjust_backto_standard_pos m rs* (*c*, *Oc* # *list*)
$\Longrightarrow$ ($c = []$ $\longrightarrow$ *wadjust_backto_standard_pos m rs* ([], *Bk* # *Oc* # *list*)) $\wedge$
($c \neq []$ $\longrightarrow$ *wadjust_backto_standard_pos m rs* (*tl c*, *hd c* # *Oc* # *list*))
$\langle proof \rangle$

lemma *wadjust_loop_right_move_nonempty_snd*[*simp*]: *wadjust_loop_right_move m rs* (*c*, []) = *False*
$\langle proof \rangle$

lemma *wadjust_loop_erase_nonempty_snd*[*simp*]: *wadjust_loop_erase m rs* (*c*, []) = *False*
$\langle proof \rangle$

lemma *wadjust_loop_erase_cases2*[*simp*]: $[\![$*Suc* (*Suc rs*) = *a*; *wadjust_loop_erase m rs* (*c*, *Bk* # *list*)$]\!]$
$\Longrightarrow$ *a* − *length* (*takeWhile* ($\lambda a.\ a = Oc$) (*tl* (*dropWhile* ($\lambda a.\ a = Oc$) (*rev* (*tl c*) @ *hd c* # *Bk* # *list*))))
< *a* − *length* (*takeWhile* ($\lambda a.\ a = Oc$) (*tl* (*dropWhile* ($\lambda a.\ a = Oc$) (*rev c* @ *Bk* # *list*)))) $\vee$
a − *length* (*takeWhile* ($\lambda a.\ a = Oc$) (*tl* (*dropWhile* ($\lambda a.\ a = Oc$) (*rev* (*tl c*) @ *hd c* # *Bk* # *list*)))) =
a − *length* (*takeWhile* ($\lambda a.\ a = Oc$) (*tl* (*dropWhile* ($\lambda a.\ a = Oc$) (*rev c* @ *Bk* # *list*))))
$\langle proof \rangle$

lemma *dropWhile_exp1*: *dropWhile* ($\lambda a.\ a = Oc$) ($Oc\uparrow(n)$ @ *xs*) = *dropWhile* ($\lambda a.\ a = Oc$) *xs*
$\langle proof \rangle$
lemma *takeWhile_exp1*: *takeWhile* ($\lambda a.\ a = Oc$) ($Oc\uparrow(n)$ @ *xs*) = $Oc\uparrow(n)$ @ *takeWhile* ($\lambda a.\ a = Oc$) *xs*
$\langle proof \rangle$

lemma *wadjust_correctness_helper_1*:
assumes *Suc* (*Suc rs*) = *a* *wadjust_loop_right_move2 m rs* (*c*, *Bk* # *list*)
shows *a* − *length* (*takeWhile* ($\lambda a.\ a = Oc$) (*tl* (*dropWhile* ($\lambda a.\ a = Oc$) (*rev c* @ *Oc* # *list*))))
< *a* − *length* (*takeWhile* ($\lambda a.\ a = Oc$) (*tl* (*dropWhile* ($\lambda a.\ a = Oc$) (*rev c* @ *Bk* # *list*))))
$\langle proof \rangle$

lemma *wadjust_correctness_helper_2*:
$[\![$*Suc* (*Suc rs*) = *a*; *wadjust_loop_on_left_moving m rs* (*c*, *Bk* # *list*)$]\!]$

$\Longrightarrow a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ (tl\ c)\ @\ hd\ c\ \#\ Bk\ \#\ list))))$
$< a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Bk\ \#\ list)))) \vee$
$a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ (tl\ c)\ @\ hd\ c\ \#\ Bk\ \#\ list)))) =$
$a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Bk\ \#\ list))))$
$\langle proof \rangle$

lemma *wadjust_loop_check_empty_false*[*simp*]: *wadjust_loop_check m rs* ([], *b*) = *False*
$\langle proof \rangle$

lemma *wadjust_loop_check_cases*: $[\![Suc\ (Suc\ rs) = a;\ wadjust\_loop\_check\ m\ rs\ (c, Oc\ \#\ list)]\!]$
$\Longrightarrow a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ (tl\ c)\ @\ hd\ c\ \#\ Oc\ \#\ list))))$
$< a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Oc\ \#\ list)))) \vee$
$a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ (tl\ c)\ @\ hd\ c\ \#\ Oc\ \#\ list)))) =$
$a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Oc\ \#\ list))))$
$\langle proof \rangle$

lemma *wadjust_loop_erase_cases_or*:
$[\![Suc\ (Suc\ rs) = a;\ wadjust\_loop\_erase\ m\ rs\ (c, Oc\ \#\ list)]\!]$
$\Longrightarrow a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Bk\ \#\ list))))$
$< a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Oc\ \#\ list)))) \vee$
$a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Bk\ \#\ list)))) =$
$a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Oc\ \#\ list))))$
$\langle proof \rangle$

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*)
$\Longrightarrow$ *wadjust_start m rs* (*c*, *Oc* # *list*)
$\langle proof \rangle$

lemma *wadjust_stop_Bk*[*simp*]: *wadjust_backto_standard_pos m rs* (*c*, *Bk* # *list*)
$\Longrightarrow$ *wadjust_stop m rs* (*Bk* # *c*, *list*)
$\langle proof \rangle$

lemma *wadjust_loop_start_Oc*[*simp*]:
assumes *wadjust_start m rs* (*c*, *Oc* # *list*)
shows *wadjust_loop_start m rs* (*Oc* # *c*, *list*)
$\langle proof \rangle$

lemma *erase2_Bk_if_Oc*[*simp*]: *wadjust_erase2 m rs* (*c*, *Oc* # *list*)
$\Longrightarrow$ *wadjust_erase2 m rs* (*c*, *Bk* # *list*)
$\langle proof \rangle$

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* = (λ (*len*, *st*, *l*, *r*). *st* = *0*) *in*
let Q = (λ (*len*, *st*, *l*, *r*). *wadjust_inv st m rs* (*l*, *r*)) *in*
let f = (λ *stp*. (*Suc* (*Suc rs*), *steps0* (*Suc 0*, *Bk* # *Oc*↑(*Suc m*),
Bk # *Oc* # *Bk*↑(*ln*) @ *Bk* # *Oc*↑(*Suc rs*) @ *Bk*↑(*rn*)) *wcode_adjust_tm stp*)) *in*
∃ *n* .*P* (*f n*) ∧ *Q* (*f n*)
⟨*proof*⟩

lemma *composable_tm_wcode_adjust_tm*[*intro*]: *composable_tm* (*wcode_adjust_tm*, *0*)
⟨*proof*⟩

lemma *bl_bin_nonzero*[*simp*]: *args* ≠ [] $\Longrightarrow$ *bl_bin* (<*args*::*nat list*>) > *0*
⟨*proof*⟩

lemma *wcode_lemma_pre*′:
args ≠ [] $\Longrightarrow$
∃ *stp rn*. *steps0* (*Suc 0*, [], <*m* # *args*>)
((*wcode_prepare_tm* |+| *wcode_main_tm*) |+| *wcode_adjust_tm*) *stp*
= (*0*, [*Bk*], *Oc*↑(*Suc m*) @ *Bk* # *Oc*↑(*Suc* (*bl_bin* (<*args*>))) @ *Bk*↑(*rn*))
⟨*proof*⟩

The initialization TM *wcode_tm*.

definition *wcode_tm* :: *instr list*
where
wcode_tm = (*wcode_prepare_tm* |+| *wcode_main_tm*) |+| *wcode_adjust_tm*

The correctness of *wcode_tm*.

lemma *wcode_lemma_1*:
args ≠ [] $\Longrightarrow$
∃ *stp ln rn*. *steps0* (*Suc 0*, [], <*m* # *args*>) (*wcode_tm*) *stp* =
(*0*, [*Bk*], *Oc*↑(*Suc m*) @ *Bk* # *Oc*↑(*Suc* (*bl_bin* (<*args*>))) @ *Bk*↑(*rn*))
⟨*proof*⟩

lemma *wcode_lemma*:
args ≠ [] $\Longrightarrow$
∃ *stp ln rn*. *steps0* (*Suc 0*, [], <*m* # *args*>) (*wcode_tm*) *stp* =
(*0*, [*Bk*], <[*m* ,*bl_bin* (<*args*>)]> @ *Bk*↑(*rn*))
⟨*proof*⟩

6.2 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.

6.2.1 Definition of the machine utm

definition *utm* :: *instr list*
where
$utm = (let\ (aprog, rs\_pos, a\_md) = rec\_ci\ rec\_F\ in$
$let\ abc\_F = aprog\ [+]\ dummy\_abc\ (Suc\ (Suc\ 0))\ in$
$(wcode\_tm\ |+|\ (tm\_of\ abc\_F\ @\ shift\ (mopup\_n\_tm\ (Suc\ (Suc\ 0)))\ (length\ (tm\_of\ abc\_F)\ div\ 2))))$

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_tm* :: *instr list*
where
$f\_tprog\_tm = tm\_of\ (f\_aprog)$

definition *utm_with_two_args* :: *instr list*
where
$utm\_with\_two\_args \stackrel{def}{=}$
$f\_tprog\_tm\ @\ shift\ (mopup\_n\_tm\ (Suc\ (Suc\ 0)))\ (length\ f\_tprog\_tm\ div\ 2)$

definition *utm_pre_tm* :: *instr list*
where
$utm\_pre\_tm = wcode\_tm\ |+|\ utm\_with\_two\_args$

lemma *fabr_spike_1*:
$utm\_with\_two\_args = tm\_of\ (fst\ (rec\_ci\ rec\_F)\ [+]\ dummy\_abc\ (Suc\ (Suc\ 0)))\ @\ shift\ (mopup\_n\_tm\ (Suc\ (Suc\ 0)))$
$(length\ (tm\_of\ (fst\ (rec\_ci\ rec\_F)\ [+]\ dummy\_abc\ (Suc\ (Suc\ 0))))\ div\ 2)$
⟨*proof*⟩

lemma *fabr_spike_2*:
$utm = wcode\_tm\ |+|$
$tm\_of\ (fst\ (rec\_ci\ rec\_F)\ [+]\ dummy\_abc\ (Suc\ (Suc\ 0)))\ @\ shift\ (mopup\_n\_tm\ (Suc\ (Suc\ 0)))$
$(length\ (tm\_of\ (fst\ (rec\_ci\ rec\_F)\ [+]\ dummy\_abc\ (Suc\ (Suc\ 0))))\ div\ 2)$
⟨*proof*⟩

theorem *fabr_spike_3*: $utm = wcode\_tm\ |+|\ utm\_with\_two\_args$
⟨*proof*⟩

corollary *fabr_spike_4*: $utm = utm\_pre\_tm$
⟨*proof*⟩

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 *utm_with_two_args_halt_eq*:
assumes *composable_tm*: *composable_tm (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 *∃ stp m n. steps0 (Suc 0, [Bk], <[code tp, bl2wc (<lm>)]> @ Bk↑(i)) utm_with_two_args stp =*
(0, Bk↑(m), Oc↑(rs) @ Bk↑(n))
⟨*proof*⟩

lemma *composable_tm_wcode_tm*[*intro*]: *composable_tm (wcode_tm, 0)*
⟨*proof*⟩

lemma *utm_halt_lemma_pre*:
assumes *composable_tm (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 *∃ stp m n. steps0 (Suc 0, [], <code tp # args>) utm_pre_tm stp =*
(0, Bk↑(m), Oc↑(rs) @ Bk↑(n))
⟨*proof*⟩

6.2.2 The correctness of utm, the halt case

lemma *utm_halt_lemma′*:
assumes *composable_tm*: *composable_tm (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 *∃ 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

$st = 0 \wedge (\exists\ m.\ l = Bk\uparrow(m)) \wedge (\exists\ rs\ n.\ r = Oc\uparrow(Suc\ rs)\ @\ Bk\uparrow(n)))$

lemma *nstd_case1*: $0 < a \Longrightarrow NSTD\ (trpl\_code\ (a, b, c))$
⟨*proof*⟩

lemma *nonzero_bl2wc*[*simp*]: $\forall m.\ b \neq Bk\uparrow(m) \Longrightarrow 0 < bl2wc\ b$
⟨*proof*⟩

lemma *nstd_case2*: $\forall m.\ b \neq Bk\uparrow(m) \Longrightarrow NSTD\ (trpl\_code\ (a, b, c))$
⟨*proof*⟩

lemma *even_not_odd*[*elim*]: $Suc\ (2 * x) = 2 * y \Longrightarrow RR$
⟨*proof*⟩

declare *replicate_Suc*[*simp del*]

lemma *bl2nat_zero_eq*[*simp*]: $(bl2nat\ c\ 0 = 0) = (\exists n.\ c = Bk\uparrow(n))$
⟨*proof*⟩

lemma *bl2wc_exp_ex*:
$[\![Suc\ (bl2wc\ c) = 2\ \hat{}\ \ m]\!] \Longrightarrow \exists\ rs\ n.\ c = Oc\uparrow(rs)\ @\ Bk\uparrow(n)$
⟨*proof*⟩

lemma *lg_bin*:
assumes $\forall rs\ n.\ c \neq Oc\uparrow(Suc\ rs)\ @\ Bk\uparrow(n)$
$bl2wc\ c = 2\ \hat{}\ lg\ (Suc\ (bl2wc\ c))\ 2 - Suc\ 0$
shows $bl2wc\ c = 0$
⟨*proof*⟩

lemma *nstd_case3*:
$\forall rs\ n.\ c \neq Oc\uparrow(Suc\ rs)\ @\ Bk\uparrow(n) \Longrightarrow\ NSTD\ (trpl\_code\ (a, b, c))$
⟨*proof*⟩

lemma *NSTD_1*: $\neg\ TSTD\ (a, b, c)$
$\Longrightarrow rec\_exec\ rec\_NSTD\ [trpl\_code\ (a, b, c)] = Suc\ 0$
⟨*proof*⟩

lemma *nonstop_t_uhalt_eq*:
$[\![composable\_tm\ (tp, 0);$
$steps0\ (Suc\ 0, Bk\uparrow(l), <lm>)\ tp\ stp = (a, b, c);$
$\neg\ TSTD\ (a, b, c)]\!]$
$\Longrightarrow rec\_exec\ rec\_nonstop\ [code\ tp, bl2wc\ (<lm>), stp] = Suc\ 0$
⟨*proof*⟩

lemma *nonstop_true*:
$[\![composable\_tm\ (tp, 0);$
$\forall\ stp.\ (\neg\ TSTD\ (steps0\ (Suc\ 0, Bk\uparrow(l), <lm>)\ tp\ stp))]\!]$
$\Longrightarrow \forall y.\ rec\_exec\ rec\_nonstop\ ([code\ tp, bl2wc\ (<lm>), y]) = (Suc\ 0)$
⟨*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 *composable_tm (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′*:
⟦*composable_tm (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*:
⟦*composable_tm (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 *composable_tm*: *composable_tm (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>)]>) utm_with_two_args 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>)]>) utm_with_two_args stp)*
shows ∀ *stp*. ¬ *is_final (steps0 (Suc 0, Bk↑(m), <[code tp, bl2wc (<lm>)]> @ Bk↑(n)) utm_with_two_args stp)*
(**is** ∀ *stp*. *?P stp*)
⟨*proof*⟩

lemma *tutm_uhalt*:
⟦*composable_tm* (*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*)) *utm_with_two_args stp*)
⟨*proof*⟩

lemma *utm_uhalt_lemma_pre*:
assumes *composable_tm*: *composable_tm* (*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_tm stp*)
⟨*proof*⟩

6.2.3 The correctness of utm, the unhalt case.

lemma *utm_uhalt_lemma′*:
assumes *composable_tm*: *composable_tm* (*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 *composable_tm*: *composable_tm* (*p*, *0*)
and *result*: *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 *composable_tm*: *composable_tm* (*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 *composable_tm*: *composable_tm* (*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 *composable_tm* (*p*, *0*)
and ⦃(λ*tp*. *tp* = ([], <*args*>))⦄ *p* ↑

and $args \neq []$
shows $\{\!\{(\lambda tp.\ tp = ([], {<}code\ p \# args{>}))\}\!\}\ utm \uparrow$
$\langle proof \rangle$

end

Chapter 7

Code extraction for interpreters and compilers

```
theory GeneratedCode
  imports HaltingProblems_K_H
        Abacus_Hoare

        HOL−Library.Code_Binary_Nat

begin



fun
  dummy_cellId :: cell ⇒ cell
  where
dummy_cellId Oc = Oc |
dummy_cellId Bk = Bk



fun
  dummy_abc_inst_Id ::  abc_inst ⇒ bool
  where
dummy_abc_inst_Id (Inc n) = True |
dummy_abc_inst_Id (Dec n s) = True |
dummy_abc_inst_Id (Goto n) = True



fun tape_of_nat_imp :: nat ⇒ cell list
  where
  tape_of_nat_imp n = <n>
```

```
fun tape_of_nat_list_imp :: nat list ⇒ cell list
  where
  tape_of_nat_list_imp ns = <ns>



export-code  dummy_cellId
       step steps
       is_final
       mk_composable0 shift adjust seq_tm


       tape_of_nat_list_imp tape_of_nat_imp


       tm_semi_id_eq0 tm_semi_id_gt0
       tm_onestroke

       tm_copy_begin_orig tm_copy_loop_orig tm_copy_end_new
       tm_weak_copy

       tm_skip_first_arg tm_erase_right_then_dblBk_left
       tm_check_for_one_arg

       tm_strong_copy


       dummy_abc_inst_Id
       abc_step_l abc_steps_l
       abc_lm_v abc_lm_s abc_fetch
       abc_final abc_notfinal abc_out_of_prog


       layout_of start_of
       tinc tdec tgoto ci tpairs_of
       tm_of tms_of
       mopup_n_tm app_mopup


       tm_to_nat_list tm_to_nat
       nat_list_to_tm nat_to_tm

       num_of_nat num_of_integer

       list_encode list_decode prod_encode prod_decode
       triangle


  in Haskell file HaskellCode/
```

end

Bibliography

[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.