Universal Turing Machine and Computability Theory in Isabelle/HOL

Jian Xu[2] Xingyuan Zhang[2] Christian Urban[1]
Sebastiaan J. C. Joosten[3]

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

February 7, 2019

Abstract

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

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

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

1 Turing Machines

theory *Turing*
  imports *Main*
begin

2 Basic definitions of Turing machine

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

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

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

type-synonym *state* = *nat*

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

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

type-synonym *tprog0* = *instr list*

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

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

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

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*)
  by (*cases s*,*force*) (*cases b*;*force*)

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

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

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

abbreviation
  step0 c p ≝ 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 ≝ steps c (p, 0) n

lemma step_red [simp]:
  shows steps c p (Suc n) = step (steps c p n) p
  by (induct n arbitrary: c) (auto)

lemma steps_add [simp]:
  shows steps c p (m + n) = steps (steps c p m) p n
  by (induct m arbitrary: c) (auto)

lemma step_0 [simp]:
  shows step (0, (l, r)) p = (0, (l, r))
  by (cases p, simp)

lemma steps_0 [simp]:
  shows steps (0, (l, r)) p n = (0, (l, r))
  by (induct n) (simp_all)

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

lemma is_final_eq:
  shows is_final (s, tp) = (s = 0)
  by (cases tp) (auto)

lemma is_finalI [intro]:
  shows is_final (0, tp)
  by (simp add: is_final_eq)

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

```
  using assms
  by(induct n;cases c;auto)

lemma is_final:
  assumes a: is_final (steps c p n1)
    and b: n1 ≤ n2
  shows is_final (steps c p n2)
proof −
  obtain n3 where eq: n2 = n1 + n3 using b by (metis le_iff_add)
  from a show is_final (steps c p n2) unfolding eq
    by (simp add: after_is_final)
qed

lemma not_is_final:
  assumes a: ¬ is_final (steps c p n1)
    and b: n2 ≤ n1
  shows ¬ is_final (steps c p n2)
proof (rule notI)
  obtain n3 where eq: n1 = n2 + n3 using b by (metis le_iff_add)
  assume is_final (steps c p n2)
  then have is_final (steps c p n1) unfolding eq
    by (simp add: after_is_final)
  with a show False by simp
qed


lemma before_final:
  assumes steps0 (1, tp) A n = (0, tp′)
  shows ∃ n′. ¬ is_final (steps0 (1, tp) A n′) ∧ steps0 (1, tp) A (Suc n′) = (0, tp′)
  using assms
proof(induct n arbitrary: tp′)
  case (0 tp′)
  have asm: steps0 (1, tp) A 0 = (0, tp′) by fact
  then show ∃n′. ¬ is_final (steps0 (1, tp) A n′) ∧ steps0 (1, tp) A (Suc n′) = (0, tp′)
    by simp
next
  case (Suc n tp′)
  have ih: ⋀tp′. steps0 (1, tp) A n = (0, tp′) ⟹
    ∃n′. ¬ is_final (steps0 (1, tp) A n′) ∧ steps0 (1, tp) A (Suc n′) = (0, tp′) by fact
  have asm: steps0 (1, tp) A (Suc n) = (0, tp′) by fact
  obtain s l r where cases: steps0 (1, tp) A n = (s, l, r)
    by (auto intro: is_final.cases)
  then show ∃n′. ¬ is_final (steps0 (1, tp) A n′) ∧ steps0 (1, tp) A (Suc n′) = (0, tp′)
  proof (cases s = 0)
    case True
    then have steps0 (1, tp) A n = (0, tp′)
      using asm cases by (simp del: steps.simps)
    then show ?thesis using ih by simp
  next
    case False
```

```
    then have ¬ is_final (steps0 (1, tp) A n) ∧ steps0 (1, tp) A (Suc n) = (0, tp′)
      using asm cases by simp
    then show ?thesis by auto
  qed
qed

lemma least_steps:
  assumes steps0 (1, tp) A n = (0, tp′)
  shows ∃ n′. (∀ n′′ < n′. ¬ is_final (steps0 (1, tp) A n′′)) ∧
             (∀ n′′ ≥ n′. is_final (steps0 (1, tp) A n′′))
proof −
  from before_final[OF assms]
  obtain n′ where
    before: ¬ is_final (steps0 (1, tp) A n′) and
    final: steps0 (1, tp) A (Suc n′) = (0, tp′) by auto
  from before
  have ∀ n′′ < Suc n′. ¬ is_final (steps0 (1, tp) A n′′)
    using not_is_final by auto
  moreover
  from final
  have ∀ n′′ ≥ Suc n′. is_final (steps0 (1, tp) A n′′)
    using is_final[of _ _ Suc n′] by (auto simp add: is_final_eq)
  ultimately
  show ∃ n′. (∀ n′′ < n′. ¬ is_final (steps0 (1, tp) A n′′)) ∧ (∀ n′′ ≥ n′. is_final (steps0 (1, tp) A
n′′))
    by blast
qed


abbreviation is_even n ≝ (n::nat) mod 2 = 0

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

abbreviation
  tm_wf0 p ≝ tm_wf (p, 0)

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

lemma hd_repeat_cases:
  P (hd (a ↑ m @ r)) ⟷ (m = 0 ⟶ P (hd r)) ∧ (∀ nat. m = Suc nat ⟶ P a)
  by (cases m,auto)
```

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


instantiation nat::tape begin
definition tape_of_nat where tape_of_nat (n::nat) ≝ Oc ↑ (Suc n)
instance by standard
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 ≝ tape_of_nat_list
instance by standard
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 ≝ tape_of_nat_prod
instance by standard
end

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

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

abbreviation
  adjust0 p ≝ adjust p (Suc (length p div 2))

lemma length_shift [simp]:
  shows length (shift p n) = length p
  by simp

lemma length_adjust [simp]:
  shows length (adjust p n) = length p
  by (induct p) (auto)
```

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

lemma tm_comp_length:
  shows length (A |+| B) = length A + length B
  by auto

lemma tm_comp_wf[intro]:
  ⟦tm_wf (A, 0); tm_wf (B, 0)⟧ ⟹ tm_wf (A |+| B, 0)
  by (fastforce)

lemma tm_comp_step:
  assumes unfinal: ¬ is_final (step0 c A)
  shows step0 c (A |+| B) = step0 c A
proof -
  obtain s l r where eq: c = (s, l, r) by (metis is_final.cases)
  have ¬ is_final (step0 (s, l, r) A) using unfinal eq by simp
  then have case (fetch A s (read r)) of (a, s) ⇒ s ≠ 0
    by (auto simp add: is_final_eq)
  then have fetch (A |+| B) s (read r) = fetch A s (read r)
    apply (cases read r;cases s)
    by (auto simp: tm_comp_length nth_append)
  then show step0 c (A |+| B) = step0 c A by (simp add: eq)
qed

lemma tm_comp_steps:
  assumes ¬ is_final (steps0 c A n)
  shows steps0 c (A |+| B) n = steps0 c A n
  using assms
proof(induct n)
  case 0
  then show steps0 c (A |+| B) 0 = steps0 c A 0 by auto
next
  case (Suc n)
  have ih: ¬ is_final (steps0 c A n) ⟹ steps0 c (A |+| B) n = steps0 c A n by fact
  have fin: ¬ is_final (steps0 c A (Suc n)) by fact
  then have fin1: ¬ is_final (step0 (steps0 c A n) A)
    by (auto simp only: step_red)
  then have fin2: ¬ is_final (steps0 c A n)
    by (metis is_final_eq step_0 surj_pair)

  have steps0 c (A |+| B) (Suc n) = step0 (steps0 c (A |+| B) n) (A |+| B)
    by (simp only: step_red)
  also have ... = step0 (steps0 c A n) (A |+| B) by (simp only: ih[OF fin2])
  also have ... = step0 (steps0 c A n) A by (simp only: tm_comp_step[OF fin1])
```

```
 finally show steps0 c (A |+| B) (Suc n) = steps0 c A (Suc n)
  by (simp only: step_red)
qed

lemma tm_comp_fetch_in_A:
 assumes h1: fetch A s x = (a, 0)
  and h2: s ≤ length A div 2
  and h3: s ≠ 0
 shows fetch (A |+| B) s x = (a, Suc (length A div 2))
 using h1 h2 h3
 apply(cases s;cases x)
 by(auto simp: tm_comp_length nth_append)

lemma tm_comp_exec_after_first:
 assumes h1: ¬ is_final c
  and h2: step0 c A = (0, tp)
  and h3: fst c ≤ length A div 2
 shows step0 c (A |+| B) = (Suc (length A div 2), tp)
 using h1 h2 h3
 apply(case_tac c)
 apply(auto simp del: tm_comp.simps)
  apply(case_tac fetch A a Bk)
  apply(simp del: tm_comp.simps)
  apply(subst tm_comp_fetch_in_A;force)
 apply(case_tac fetch A a (hd ca))
 apply(simp del: tm_comp.simps)
 apply(subst tm_comp_fetch_in_A)
   apply(auto)[4]
 done

lemma step_in_range:
 assumes h1: ¬ is_final (step0 c A)
  and h2: tm_wf (A, 0)
 shows fst (step0 c A) ≤ length A div 2
 using h1 h2
 apply(cases c;cases fst c;cases hd (snd (snd c)))
 by(auto simp add: Let_def case_prod_beta')

lemma steps_in_range:
 assumes h1: ¬ is_final (steps0 (1, tp) A stp)
  and h2: tm_wf (A, 0)
 shows fst (steps0 (1, tp) A stp) ≤ length A div 2
 using h1
proof(induct stp)
 case 0
 then show fst (steps0 (1, tp) A 0) ≤ length A div 2 using h2
  by (auto)
next
 case (Suc stp)
 have ih: ¬ is_final (steps0 (1, tp) A stp) ⟹ fst (steps0 (1, tp) A stp) ≤ length A div 2 by fact
```

```
  have h: ¬ is_final (steps0 (1, tp) A (Suc stp)) by fact
  from ih h h2 show fst (steps0 (1, tp) A (Suc stp)) ≤ length A div 2
    by (metis step_in_range step_red)
qed


lemma tm_comp_next:
  assumes a_ht: steps0 (1, tp) A n = (0, tp')
    and a_wf: tm_wf (A, 0)
  obtains n' where steps0 (1, tp) (A |+| B) n' = (Suc (length A div 2), tp')
proof -
  assume a: ⋀n. steps (1, tp) (A |+| B, 0) n = (Suc (length A div 2), tp') ⟹ thesis
  obtain stp' where fin: ¬ is_final (steps0 (1, tp) A stp') and h: steps0 (1, tp) A (Suc stp') = (0,
tp')
    using before_final[OF a_ht] by blast
  from fin have h1:steps0 (1, tp) (A |+| B) stp' = steps0 (1, tp) A stp'
    by (rule tm_comp_steps)
  from h have h2: step0 (steps0 (1, tp) A stp') A = (0, tp')
    by (simp only: step_red)

  have steps0 (1, tp) (A |+| B) (Suc stp') = step0 (steps0 (1, tp) (A |+| B) stp') (A |+| B)
    by (simp only: step_red)
  also have ... = step0 (steps0 (1, tp) A stp') (A |+| B) using h1 by simp
  also have ... = (Suc (length A div 2), tp')
    by (rule tm_comp_exec_after_first[OF fin h2 steps_in_range[OF fin a_wf]])
  finally show thesis using a by blast
qed

lemma tm_comp_fetch_second_zero:
  assumes h1: fetch B s x = (a, 0)
    and hs: tm_wf (A, 0) s ≠ 0
  shows fetch (A |+| B) (s + (length A div 2)) x = (a, 0)
  using h1 hs
  by(cases x; cases s; fastforce simp: tm_comp_length nth_append)

lemma tm_comp_fetch_second_inst:
  assumes h1: fetch B sa x = (a, s)
    and hs: tm_wf (A, 0) sa ≠ 0 s ≠ 0
  shows fetch (A |+| B) (sa + length A div 2) x = (a, s + length A div 2)
  using h1 hs
  by(cases x; cases sa; fastforce simp: tm_comp_length nth_append)


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

```
  case 0
  then show ?case by simp
next
  case (Suc stp s' l' r')
  obtain s'' l'' r'' where a: steps0 (1, l, r) B stp = (s'', l'', r'')
    by (metis is_final.cases)
  then have ih1: s'' = 0 ⟹ steps0 (Suc (length A div 2), l, r) (A |+| B) stp = (0, l'', r'')
    and ih2: s'' ≠ 0 ⟹ steps0 (Suc (length A div 2), l, r) (A |+| B) stp = (s'' + length A div 2,
l'', r'')
    using Suc by (auto)
  have h: steps0 (1, l, r) B (Suc stp) = (s', l', r') by fact

  { assume s'' = 0
    then have ?case using a h ih1 by (simp del: steps.simps)
  } moreover
  { assume as: s'' ≠ 0 s' = 0
    from as a h
    have step0 (s'', l'', r'') B = (0, l', r') by (simp del: steps.simps)
    with as have ?case
      apply(cases fetch B s'' (read r''))
      by (auto simp add: tm_comp_fetch_second_zero[OF _ a_wf] ih2[OF as(1)]
          simp del: tm_comp.simps steps.simps)
  } moreover
  { assume as: s'' ≠ 0 s' ≠ 0
    from as a h
    have step0 (s'', l'', r'') B = (s', l', r') by (simp del: steps.simps)
    with as have ?case
      apply(simp add: ih2[OF as(1)] del: tm_comp.simps steps.simps)
      apply(case_tac fetch B s'' (read r''))
      apply(auto simp add: tm_comp_fetch_second_inst[OF _ a_wf as] simp del: tm_comp.simps)
      done
  }
  ultimately show ?case by blast
qed


lemma tm_comp_final:
  assumes tm_wf (A, 0)
    and steps0 (1, l, r) B stp = (0, l', r')
  shows steps0 (Suc (length A div 2), l, r)  (A |+| B) stp = (0, l', r')
  using tm_comp_second[OF assms] by (simp)

end
```

3 Hoare Rules for TMs

```
theory Turing_Hoare
  imports Turing
begin
```

type-synonym $assert = tape \Rightarrow bool$

definition
$assert\_imp :: assert \Rightarrow assert \Rightarrow bool$ ($\_ \mapsto \_$ $[0, 0]$ 100)
where
$P \mapsto Q \stackrel{def}{=} \forall\, l\, r.\ P\ (l, r) \longrightarrow Q\ (l, r)$

lemma *refl_assert*[*intro*, *simp*]:
$P \mapsto P$
unfolding *assert_imp_def* **by** *simp*

fun
$holds\_for :: (tape \Rightarrow bool) \Rightarrow config \Rightarrow bool$ ($\_\ holds'\_for\ \_$ $[100, 99]$ 100)
where
$P\ holds\_for\ (s, l, r) = P\ (l, r)$

lemma *is_final_holds*[*simp*]:
assumes $is\_final\ c$
shows $Q\ holds\_for\ (steps\ c\ p\ n) = Q\ holds\_for\ c$
using *assms*
by(*induct n*;*cases c*,*auto*)

definition
$Hoare\_halt :: assert \Rightarrow tprog0 \Rightarrow assert \Rightarrow bool$ $((\{(1\_)\}/\ (\_)/\ \{(1\_)\})\ 50)$
where
$\{P\}\ p\ \{Q\} \stackrel{def}{=} (\forall\, tp.\ P\ tp \longrightarrow (\exists\, n.\ is\_final\ (steps0\ (1, tp)\ p\ n) \land Q\ holds\_for\ (steps0\ (1, tp)\ p\ n)))$

definition
$Hoare\_unhalt :: assert \Rightarrow tprog0 \Rightarrow bool$ $((\{(1\_)\}/\ (\_))\uparrow 50)$
where
$\{P\}\ p \uparrow \stackrel{def}{=} \forall\, tp.\ P\ tp \longrightarrow (\forall\ n\ .\ \neg\ (is\_final\ (steps0\ (1, tp)\ p\ n)))$

lemma *Hoare_haltI*:
assumes $\bigwedge l\ r.\ P\ (l, r) \Longrightarrow \exists\, n.\ is\_final\ (steps0\ (1, (l, r))\ p\ n) \land Q\ holds\_for\ (steps0\ (1, (l, r))\ p\ n)$
shows $\{P\}\ p\ \{Q\}$
unfolding *Hoare_halt_def*
using *assms* **by** *auto*

lemma *Hoare_unhaltI*:
assumes $\bigwedge l\ r\ n.\ P\ (l, r) \Longrightarrow \neg\ is\_final\ (steps0\ (1, (l, r))\ p\ n)$

```
  shows {P} p ↑
  unfolding Hoare_unhalt_def
  using assms by auto
```

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

```
lemma Hoare_plus_halt [case_names A_halt B_halt A_wf]:
  assumes A_halt : {P} A {Q}
    and B_halt : {Q} B {S}
    and A_wf : tm_wf (A, 0)
  shows {P} A |+| B {S}
proof(rule Hoare_haltI)
  fix l r
  assume h: P (l, r)
  then obtain n1 l' r'
    where is_final (steps0 (1, l, r) A n1)
      and a1: Q holds_for (steps0 (1, l, r) A n1)
      and a2: steps0 (1, l, r) A n1 = (0, l', r')
    using A_halt unfolding Hoare_halt_def
    by (metis is_final_eq surj_pair)
  then obtain n2
    where steps0 (1, l, r) (A |+| B) n2 = (Suc (length A div 2), l', r')
    using A_wf by (rule_tac tm_comp_next)
  moreover
  from a1 a2 have Q (l', r') by (simp)
  then obtain n3 l'' r''
    where is_final (steps0 (1, l', r') B n3)
      and b1: S holds_for (steps0 (1, l', r') B n3)
      and b2: steps0 (1, l', r') B n3 = (0, l'', r'')
    using B_halt unfolding Hoare_halt_def
    by (metis is_final_eq surj_pair)
  then have steps0 (Suc (length A div 2), l', r')  (A |+| B) n3 = (0, l'', r'')
    using A_wf by (rule_tac tm_comp_final)
  ultimately show
    ∃ n. is_final (steps0 (1, l, r) (A |+| B) n) ∧ S holds_for (steps0 (1, l, r) (A |+| B) n)
    using b1 b2 by (rule_tac x = n2 + n3 in exI) (simp)
qed
```

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

```
lemma Hoare_plus_unhalt [case_names A_halt B_unhalt A_wf]:
  assumes A_halt: {P} A {Q}
    and B_uhalt: {Q} B ↑
    and A_wf : tm_wf (A, 0)
  shows {P} (A |+| B) ↑
proof(rule_tac Hoare_unhaltI)
  fix n l r
  assume h: P (l, r)
  then obtain n1 l' r'
    where a: is_final (steps0 (1, l, r) A n1)
      and b: Q holds_for (steps0 (1, l, r) A n1)
```

```
    and c: steps0 (1, l, r) A n1 = (0, l′, r′)
   using A_halt unfolding Hoare_halt_def
   by (metis is_final_eq surj_pair)
  then obtain n2 where eq: steps0 (1, l, r) (A |+| B) n2 = (Suc (length A div 2), l′, r′)
   using A_wf by (rule_tac tm_comp_next)
  then show ¬ is_final (steps0 (1, l, r) (A |+| B) n)
  proof(cases n2 ≤ n)
   case True
   from b c have Q (l′, r′) by simp
   then have ∀ n. ¬ is_final (steps0 (1, l′, r′) B n)
    using B_uhalt unfolding Hoare_unhalt_def by simp
   then have ¬ is_final (steps0 (1, l′, r′) B (n − n2)) by auto
   then obtain s′′ l′′ r′′
    where steps0 (1, l′, r′) B (n − n2) = (s′′, l′′, r′′)
     and ¬ is_final (s′′, l′′, r′′) by (metis surj_pair)
   then have steps0 (Suc (length A div 2), l′, r′) (A |+| B) (n − n2) = (s′′+ length A div 2, l′′,
r′′)
    using A_wf by (auto dest: tm_comp_second simp del: tm_wf.simps)
   then have ¬ is_final (steps0 (1, l, r) (A |+| B) (n2 + (n  − n2)))
    using A_wf by (simp only: steps_add eq) simp
   then show ¬ is_final (steps0 (1, l, r) (A |+| B) n)
    using ‹n2 ≤ n› by simp
  next
   case False
   then obtain n3 where n = n2 − n3
    using diff_le_self le_imp_diff_is_add nat_le_linear
     add.commute by metis
   moreover
   with eq show ¬ is_final (steps0 (1, l, r) (A |+| B) n)
    by (simp add: not_is_final[where ?n1.0=n2])
  qed
qed

lemma Hoare_consequence:
  assumes P′ ↦ P {P} p {Q} Q ↦ Q′
  shows {P′} p {Q′}
  using assms
  unfolding Hoare_halt_def assert_imp_def
  by (metis holds_for.simps surj_pair)



end
```

4 Undeciablity of the Halting Problem

```
theory Uncomputable
  imports Turing_Hoare
begin
```

lemma *numeral*:
shows $2 = Suc\ 1$
and $3 = Suc\ 2$
and $4 = Suc\ 3$
and $5 = Suc\ 4$
and $6 = Suc\ 5$
and $7 = Suc\ 6$
and $8 = Suc\ 7$
and $9 = Suc\ 8$
and $10 = Suc\ 9$
and $11 = Suc\ 10$
and $12 = Suc\ 11$
by *simp_all*

lemma *gr1_conv_Suc*:$Suc\ 0 < mr \longleftrightarrow (\exists\ nat.\ mr = Suc\ (Suc\ nat))$ **by** *presburger*

The Copying TM, which duplicates its input.

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

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

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

definition
tcopy :: *instr list*
where
tcopy $\stackrel{def}{=}$ (*tcopy_begin* |+| *tcopy_loop*) |+| *tcopy_end*

fun
inv_begin0 :: $nat \Rightarrow tape \Rightarrow bool$ **and**
inv_begin1 :: $nat \Rightarrow tape \Rightarrow bool$ **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*, *tp*) =
(*if s* = *0 then inv_begin0 n tp else*
if s = *1 then inv_begin1 n tp else*
if s = *2 then inv_begin2 n tp else*
if s = *3 then inv_begin3 n tp else*
if s = *4 then inv_begin4 n tp*
else False)

lemma *split_head_repeat*[*simp*]:
Oc # *list1* = *Bk* ↑ *j* @ *list2* ⟷ *j* = *0* ∧ *Oc* # *list1* = *list2*
Bk # *list1* = *Oc* ↑ *j* @ *list2* ⟷ *j* = *0* ∧ *Bk* # *list1* = *list2*
Bk ↑ *j* @ *list2* = *Oc* # *list1* ⟷ *j* = *0* ∧ *Oc* # *list1* = *list2*
Oc ↑ *j* @ *list2* = *Bk* # *list1* ⟷ *j* = *0* ∧ *Bk* # *list1* = *list2*
by(*cases j*;*force*)+

lemma *inv_begin_step_E*: ⟦*0* < *i*; *0* < *j*⟧ ⟹
∃ *ia*>*0*. *ia* + *j* − *Suc 0* = *i* + *j* ∧ *Oc* # *Oc* ↑ *i* = *Oc* ↑ *ia*
by (*rule_tac x* = *Suc i* **in** *exI*, *simp*)

lemma *inv_begin_step*:
assumes *inv_begin n cf*
and *n* > *0*
shows *inv_begin n* (*step0 cf tcopy_begin*)
using *assms*
unfolding *tcopy_begin_def*
apply(*cases cf*)
apply(*auto simp*: *numeral split*: *if_splits elim*:*inv_begin_step_E*)
apply(*cases hd* (*snd* (*snd cf*));*cases* (*snd* (*snd cf*)),*auto*)
done

lemma *inv_begin_steps*:
assumes *inv_begin n cf*
and *n* > *0*
shows *inv_begin n* (*steps0 cf tcopy_begin stp*)
apply(*induct stp*)
apply(*simp add*: *assms*)

```
 apply(auto simp del: steps.simps)
 apply(rule_tac inv_begin_step)
  apply(simp_all add: assms)
 done

lemma begin_partial_correctness:
 assumes is_final (steps0 (1, [], Oc ↑ n) tcopy_begin stp)
 shows 0 < n ⟹ {inv_begin1 n} tcopy_begin {inv_begin0 n}
proof(rule_tac Hoare_haltI)
 fix l r
 assume h: 0 < n inv_begin1 n (l, r)
 have inv_begin n (steps0 (1, [], Oc ↑ n) tcopy_begin stp)
  using h by (rule_tac inv_begin_steps) (simp_all)
 then show
  ∃ stp. is_final (steps0 (1, l, r) tcopy_begin stp) ∧
  inv_begin0 n holds_for steps (1, l, r) (tcopy_begin, 0) stp
  using h assms
  apply(rule_tac x = stp in exI)
  apply(case_tac (steps0 (1, [], Oc ↑ n) tcopy_begin stp), simp)
  done
qed

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
 unfolding measure_begin_def
 by auto

lemma measure_begin_induct [case_names Step]:
 [[⋀n. ¬ P (f n) ⟹ (f (Suc n), (f n)) ∈ measure_begin]] ⟹ ∃ n. P (f n)
 using wf_measure_begin
 by (metis wf_iff_no_infinite_down_chain)

lemma begin_halts:
 assumes h: x > 0
 shows ∃ stp. is_final (steps0 (1, [], Oc ↑ x) tcopy_begin stp)
```

proof (*induct rule*: *measure_begin_induct*)
  case (*Step n*)
  have $\neg$ *is_final* (*steps0* (*1*, [], *Oc* $\uparrow$ *x*) *tcopy_begin n*) **by** *fact*
  moreover
  have *inv_begin x* (*steps0* (*1*, [], *Oc* $\uparrow$ *x*) *tcopy_begin n*)
    by (*rule_tac inv_begin_steps*) (*simp_all add*: *h*)
  moreover
  obtain *s l r* **where** *eq*: (*steps0* (*1*, [], *Oc* $\uparrow$ *x*) *tcopy_begin n*) = (*s*, *l*, *r*)
    by (*metis measure_begin_state*.*cases*)
  ultimately
  have (*step0* (*s*, *l*, *r*) *tcopy_begin*, *s*, *l*, *r*) $\in$ *measure_begin*
    apply(*auto simp*: *measure_begin_def tcopy_begin_def numeral split*: *if_splits*)
    apply(*subgoal_tac r* = [*Oc*])
     apply(*auto*)
    by (*metis cell*.*exhaust list*.*exhaust list*.*sel*(*3*))
  then
   show (*steps0* (*1*, [], *Oc* $\uparrow$ *x*) *tcopy_begin* (*Suc n*), *steps0* (*1*, [], *Oc* $\uparrow$ *x*) *tcopy_begin n*) $\in$ *measure_begin*
    using *eq* **by** (*simp only*: *step_red*)
qed

lemma *begin_correct*:
  shows *0* < *n* $\Longrightarrow$ {*inv_begin1 n*} *tcopy_begin* {*inv_begin0 n*}
  using *begin_partial_correctness begin_halts* **by** *blast*

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

fun
  inv_loop1_loop :: *nat* $\Rightarrow$ *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
$inv\_loop0 :: nat \Rightarrow tape \Rightarrow bool$ **and**
$inv\_loop1 :: nat \Rightarrow tape \Rightarrow bool$ **and**
$inv\_loop2 :: nat \Rightarrow tape \Rightarrow bool$ **and**
$inv\_loop3 :: nat \Rightarrow tape \Rightarrow bool$ **and**
$inv\_loop4 :: nat \Rightarrow tape \Rightarrow bool$ **and**
$inv\_loop5 :: nat \Rightarrow tape \Rightarrow bool$ **and**
$inv\_loop6 :: nat \Rightarrow tape \Rightarrow bool$
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 $inv\_loop :: nat \Rightarrow config \Rightarrow bool$
where
$inv\_loop\ x\ (s, l, r) =$
  $(if\ s = 0\ then\ inv\_loop0\ x\ (l, r)$
  $else\ if\ s = 1\ then\ inv\_loop1\ x\ (l, r)$
  $else\ if\ s = 2\ then\ inv\_loop2\ x\ (l, r)$
  $else\ if\ s = 3\ then\ inv\_loop3\ x\ (l, r)$
  $else\ if\ s = 4\ then\ inv\_loop4\ x\ (l, r)$
  $else\ if\ s = 5\ then\ inv\_loop5\ x\ (l, r)$
  $else\ if\ s = 6\ then\ inv\_loop6\ x\ (l, r)$
  $else\ False)$

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

lemma *Bk_no_Oc_repeatE*[*elim*]: $Bk\ \#\ list = Oc \uparrow t \Longrightarrow RR$
by (*cases t*, *auto*)

lemma *inv_loop3_Bk_empty_via_2*[*elim*]: $[\![0 < x;\ inv\_loop2\ x\ (b, [])]\!] \Longrightarrow inv\_loop3\ x\ (Bk\ \#\ b, [])$
by (*auto simp*: *inv_loop2.simps inv_loop3.simps*)

```
lemma inv_loop3_Bk_empty[elim]: ⟦0 < x; inv_loop3 x (b, [])⟧ ⟹ inv_loop3 x (Bk # b, [])
  by (auto simp: inv_loop3.simps)

lemma inv_loop5_Oc_empty_via_4[elim]: ⟦0 < x; inv_loop4 x (b, [])⟧ ⟹ inv_loop5 x (b, [Oc])
  by(auto simp: inv_loop4.simps inv_loop5.simps;force)

lemma inv_loop1_Bk[elim]: ⟦0 < x; inv_loop1 x (b, Bk # list)⟧ ⟹ list = Oc # Bk ↑ x @ Oc ↑ x
  by (auto simp: inv_loop1.simps)

lemma inv_loop3_Bk_via_2[elim]: ⟦0 < x; inv_loop2 x (b, Bk # list)⟧ ⟹ inv_loop3 x (Bk # b,
list)
  by(auto simp: inv_loop2.simps inv_loop3.simps;force)

lemma inv_loop3_Bk_move[elim]: ⟦0 < x; inv_loop3 x (b, Bk # list)⟧ ⟹ inv_loop3 x (Bk # b,
list)
  apply(auto simp: inv_loop3.simps)
   apply (rename_tac i j k t)
   apply(rule_tac [!] x = i in exI,
      rule_tac [!] x = j in exI, simp_all)
   apply(case_tac [!] t, auto)
  done

lemma inv_loop5_Oc_via_4_Bk[elim]: ⟦0 < x; inv_loop4 x (b, Bk # list)⟧ ⟹ inv_loop5 x (b, Oc
# list)
  by (auto simp: inv_loop4.simps inv_loop5.simps)

lemma inv_loop6_Bk_via_5[elim]: ⟦0 < x; inv_loop5 x ([], Bk # list)⟧ ⟹ inv_loop6 x ([], Bk #
Bk # list)
  by (auto simp: inv_loop6.simps inv_loop5.simps)

lemma inv_loop5_loop_no_Bk[simp]: inv_loop5_loop x (b, Bk # list) = False
  by (auto simp: inv_loop5.simps)

lemma inv_loop6_exit_no_Bk[simp]: inv_loop6_exit x (b, Bk # list) = False
  by (auto simp: inv_loop6.simps)

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

lemma inv_loop6_loopBk_via_5[elim]:⟦0 < x; inv_loop5_exit x (b, Bk # list); b ≠ []; hd b = Bk⟧
        ⟹ inv_loop6_loop x (tl b, Bk # Bk # list)
  apply(simp only: inv_loop5_exit.simps inv_loop6_loop.simps )
  apply(erule_tac exE)+
  apply(rename_tac i j)
  apply(rule_tac x = i in exI,
      rule_tac x = j in exI,
      rule_tac x = j − Suc (Suc 0) in exI,
      rule_tac x = Suc 0 in exI, auto)
   apply(case_tac [!] j, simp_all)
   apply(case_tac [!] j−1, simp_all)
```

done

lemma *inv_loop6_loop_no_Oc_Bk*[*simp*]: *inv_loop6_loop x* (*b*, *Oc* # *Bk* # *list*) = *False*
by (*auto simp*: *inv_loop6_loop.simps*)

lemma *inv_loop6_exit_Oc_Bk_via_5*[*elim*]: ⟦*x* > *0*; *inv_loop5_exit x* (*b*, *Bk* # *list*); *b* ≠ []; *hd b* = *Oc*⟧ ⟹
inv_loop6_exit x (*tl b*, *Oc* # *Bk* # *list*)
apply(*simp only*: *inv_loop5_exit.simps inv_loop6_exit.simps*)
apply(*erule_tac exE*)+
apply(*rule_tac x* = *x* − *1* **in** *exI*, *rule_tac x* = *1* **in** *exI*, *simp*)
apply(*rename_tac i j*)
apply(*case_tac j*;*case_tac j*−*1*, *auto*)
done

lemma *inv_loop6_Bk_tail_via_5*[*elim*]: ⟦*0* < *x*; *inv_loop5 x* (*b*, *Bk* # *list*); *b* ≠ []⟧ ⟹ *inv_loop6 x* (*tl b*, *hd b* # *Bk* # *list*)
apply(*simp add*: *inv_loop5.simps inv_loop6.simps*)
apply(*cases hd b*, *simp_all*, *auto*)
done

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*)
apply(*simp only*: *inv_loop6_loop.simps*)
apply(*erule_tac exE*)+
apply(*rename_tac i j k t*)
apply(*rule_tac x* = *i* **in** *exI*, *rule_tac x* = *j* **in** *exI*,
rule_tac x = *k* − *1* **in** *exI*, *rule_tac x* = *Suc t* **in** *exI*, *auto*)
apply(*case_tac* [!] *k*, *auto*)
done

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*)
apply(*simp only*: *inv_loop6_loop.simps inv_loop6_exit.simps*)
apply(*erule_tac exE*)+
apply(*rename_tac i j k t*)
apply(*rule_tac x* = *i* − *1* **in** *exI*, *rule_tac x* = *j* **in** *exI*, *auto*)
apply(*case_tac* [!] *k*, *auto*)
done

lemma *inv_loop6_Bk_tail*[*elim*]: ⟦*0* < *x*; *inv_loop6 x* (*b*, *Bk* # *list*); *b* ≠ []⟧ ⟹ *inv_loop6 x* (*tl b*, *hd b* # *Bk* # *list*)
apply(*simp add*: *inv_loop6.simps*)
apply(*case_tac hd b*, *simp_all*, *auto*)
done

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

apply(*auto simp*: *inv_loop1.simps inv_loop2.simps*,*force*)
done

lemma *inv_loop2_Bk_via_Oc*[*elim*]: $[\![0 < x; \mathit{inv\_loop2}\ x\ (b, \mathit{Oc}\ \#\ \mathit{list})]\!] \Longrightarrow \mathit{inv\_loop2}\ x\ (b, \mathit{Bk}\ \#\ \mathit{list})$
by (*auto simp*: *inv_loop2.simps*)

lemma *inv_loop4_Oc_via_3*[*elim*]: $[\![0 < x; \mathit{inv\_loop3}\ x\ (b, \mathit{Oc}\ \#\ \mathit{list})]\!] \Longrightarrow \mathit{inv\_loop4}\ x\ (\mathit{Oc}\ \#\ b, \mathit{list})$
apply(*auto simp*: *inv_loop3.simps inv_loop4.simps*)
apply(*rename_tac i j*)
apply(*rule_tac* [!] $x = i$ **in** *exI*, *auto*)
apply(*rule_tac* [!] $x = \mathit{Suc}\ 0$ **in** *exI*, *rule_tac* [!] $x = j - 1$ **in** *exI*)
apply(*case_tac* [!] *j*, *auto*)
done

lemma *inv_loop4_Oc_move*[*elim*]:
assumes $0 < x$ $\mathit{inv\_loop4}\ x\ (b, \mathit{Oc}\ \#\ \mathit{list})$
shows $\mathit{inv\_loop4}\ x\ (\mathit{Oc}\ \#\ b, \mathit{list})$
proof −
from *assms*[*unfolded inv_loop4.simps*] **obtain** *i j k t* **where**
$i + j = x$
$0 < i$ $0 < j$ $k + t = j$ $(b, \mathit{Oc}\ \#\ \mathit{list}) = (\mathit{Oc} \uparrow k\ @\ \mathit{Bk} \uparrow \mathit{Suc}\ j\ @\ \mathit{Oc} \uparrow i, \mathit{Oc} \uparrow t)$
by *auto*
thus *?thesis* **unfolding** *inv_loop4.simps*
apply(*rule_tac* [!] $x = i$ **in** *exI*,*rule_tac* [!] $x = j$ **in** *exI*)
apply(*rule_tac* [!] $x = \mathit{Suc}\ k$ **in** *exI*,*rule_tac* [!] $x = t - 1$ **in** *exI*)
by(*cases t*,*auto*)
qed

lemma *inv_loop5_exit_no_Oc*[*simp*]: $\mathit{inv\_loop5\_exit}\ x\ (b, \mathit{Oc}\ \#\ \mathit{list}) = \mathit{False}$
by (*auto simp*: *inv_loop5_exit.simps*)

lemma *inv_loop5_exit_Bk_Oc_via_loop*[*elim*]: $[\![\mathit{inv\_loop5\_loop}\ x\ (b, \mathit{Oc}\ \#\ \mathit{list}); b \neq []; \mathit{hd}\ b = \mathit{Bk}]\!]$
$\Longrightarrow \mathit{inv\_loop5\_exit}\ x\ (\mathit{tl}\ b, \mathit{Bk}\ \#\ \mathit{Oc}\ \#\ \mathit{list})$
apply(*simp only*: *inv_loop5_loop.simps inv_loop5_exit.simps*)
apply(*erule_tac exE*)+
apply(*rename_tac i j k t*)
apply(*rule_tac* $x = i$ **in** *exI*)
apply(*case_tac k*, *auto*)
done

lemma *inv_loop5_loop_Oc_Oc_drop*[*elim*]: $[\![\mathit{inv\_loop5\_loop}\ x\ (b, \mathit{Oc}\ \#\ \mathit{list}); b \neq []; \mathit{hd}\ b = \mathit{Oc}]\!]$
$\Longrightarrow \mathit{inv\_loop5\_loop}\ x\ (\mathit{tl}\ b, \mathit{Oc}\ \#\ \mathit{Oc}\ \#\ \mathit{list})$
apply(*simp only*: *inv_loop5_loop.simps*)
apply(*erule_tac exE*)+
apply(*rename_tac i j k t*)
apply(*rule_tac* $x = i$ **in** *exI*, *rule_tac* $x = j$ **in** *exI*)
apply(*rule_tac* $x = k - 1$ **in** *exI*, *rule_tac* $x = \mathit{Suc}\ t$ **in** *exI*)

apply(case_tac k, auto)
done

lemma inv_loop5_Oc_tl[elim]: ⟦inv_loop5 x (b, Oc # list); b ≠ []⟧ ⟹ inv_loop5 x (tl b, hd b # Oc # list)
apply(simp add: inv_loop5.simps)
apply(cases hd b, simp_all, auto)
done

lemma inv_loop1_Bk_Oc_via_6[elim]: ⟦0 < x; inv_loop6 x ([], Oc # list)⟧ ⟹ inv_loop1 x ([], Bk # Oc # list)
by(auto simp: inv_loop6.simps inv_loop1.simps inv_loop6_loop.simps inv_loop6_exit.simps)

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)
by(auto simp: inv_loop6.simps inv_loop1.simps inv_loop6_loop.simps inv_loop6_exit.simps)

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
by (auto simp: inv_loop1.simps inv_loop2.simps inv_loop3.simps inv_loop4.simps
inv_loop5.simps inv_loop6.simps inv_loop5_exit.simps inv_loop5_loop.simps
inv_loop6_loop.simps)

lemma inv_loop_nonemptyE[elim]:
⟦inv_loop5 x (b, [])⟧ ⟹ RR inv_loop6 x (b, []) ⟹ RR
⟦inv_loop1 x (b, Bk # list)⟧ ⟹ b = []
by (auto simp: inv_loop4.simps inv_loop5.simps inv_loop5_exit.simps inv_loop5_loop.simps
inv_loop6.simps inv_loop6_exit.simps inv_loop6_loop.simps inv_loop1.simps)

lemma inv_loop6_Bk_Bk_drop[elim]: ⟦inv_loop6 x ([], Bk # list)⟧ ⟹ inv_loop6 x ([], Bk # Bk # list)
by (simp)

lemma inv_loop_step:
⟦inv_loop x cf; x > 0⟧ ⟹ inv_loop x (step cf (tcopy_loop, 0))
apply(cases cf, cases snd (snd cf); cases hd (snd (snd cf)))
apply(auto simp: inv_loop.simps step.simps tcopy_loop_def numeral split: if_splits)
done

lemma inv_loop_steps:
⟦inv_loop x cf; x > 0⟧ ⟹ inv_loop x (steps cf (tcopy_loop, 0) stp)
apply(induct stp, simp add: steps.simps, simp)
apply(erule_tac inv_loop_step, simp)

done

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*
else if $s = 4$ *then length r*
else if $s = 5$ *then length l*
else if $s = 6$ *then length l*
else $0)$

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

lemma *wf_measure_loop*: *wf measure_loop*
unfolding *measure_loop_def*
by (*auto*)

lemma *measure_loop_induct* [*case_names Step*]:
$[\![\bigwedge n.\ \neg\ P\ (f\ n) \Longrightarrow (f\ (Suc\ n), (f\ n)) \in measure\_loop]\!] \Longrightarrow \exists\ n.\ P\ (f\ n)$
using *wf_measure_loop*
by (*metis wf_iff_no_infinite_down_chain*)

lemma *inv_loop4_not_just_Oc*[*elim*]:
$[\![inv\_loop4\ x\ (l', [\,]);$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ [Oc])) \neq$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'))]\!]$
$\Longrightarrow RR$
$[\![inv\_loop4\ x\ (l', Bk\ \#\ list);$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ Oc\ \#\ list)) \neq$
$length\ (takeWhile\ (\lambda a.\ a = Oc)\ (rev\ l'\ @\ Bk\ \#\ list))]\!]$
$\Longrightarrow RR$
apply(*auto simp*: *inv_loop4.simps*)
apply(*rename_tac i j*)
apply(*case_tac* [!] *j*, *simp_all add*: *List.takeWhile_tail*)
done

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

```
 by (induct x, auto)

lemma takeWhile_replicate:
 P a ⟹ takeWhile P (a↑x) = a↑x
 by (induct x, auto)

lemma inv_loop5_Bk_E[elim]:
 ⟦inv_loop5 x (l′, Bk # list); l′ ≠ [];
  length (takeWhile (λa. a = Oc) (rev (tl l′) @ hd l′ # Bk # list)) ≠
  length (takeWhile (λa. a = Oc) (rev l′ @ Bk # list))⟧
  ⟹ RR
 apply(cases length list;cases length list − 1
     ,auto simp: inv_loop5.simps inv_loop5_exit.simps
     takeWhile_replicate_append takeWhile_replicate)
  apply(cases length list − 2; force simp add: List.takeWhile_tail)+
 done

lemma inv_loop1_hd_Oc[elim]: ⟦inv_loop1 x (l′, Oc # list)⟧ ⟹ hd list = Oc
 by (auto simp: inv_loop1.simps)

lemma inv_loop6_not_just_Bk[dest!]:
 ⟦length (takeWhile P (rev (tl l′) @ hd l′ # list)) ≠
 length (takeWhile P (rev l′ @ list))⟧
 ⟹ l′ = []
 apply(cases l′, simp_all)
 done

lemma inv_loop2_OcE[elim]:
 ⟦inv_loop2 x (l′, Oc # list); l′ ≠ []⟧ ⟹
 length (takeWhile (λa. a = Oc) (rev l′ @ Bk # list)) <
 length (takeWhile (λa. a = Oc) (rev l′ @ Oc # list))
 apply(auto simp: inv_loop2.simps takeWhile_tail takeWhile_replicate_append
    takeWhile_replicate)
 done

lemma loop_halts:
 assumes h: n > 0 inv_loop n (1, l, r)
 shows ∃ stp. is_final (steps0 (1, l, r) tcopy_loop stp)
proof (induct rule: measure_loop_induct)
 case (Step stp)
 have ¬ is_final (steps0 (1, l, r) tcopy_loop stp) by fact
 moreover
 have inv_loop n (steps0 (1, l, r) tcopy_loop stp)
  by (rule_tac inv_loop_steps) (simp_all only: h)
 moreover
 obtain s l′ r′ where eq: (steps0 (1, l, r) tcopy_loop stp) = (s, l′, r′)
  by (metis measure_begin_state.cases)
 ultimately
 have (step0 (s, l′, r′) tcopy_loop, s, l′, r′) ∈ measure_loop
  using h(1)
```

```
  apply(cases r';cases hd r')
    apply(auto simp: inv_loop.simps step.simps tcopy_loop_def numeral measure_loop_def split:
if_splits)
  done
 then
 show (steps0 (1, l, r) tcopy_loop (Suc stp), steps0 (1, l, r) tcopy_loop stp) ∈ measure_loop
  using eq by (simp only: step_red)
qed

lemma loop_correct:
 assumes 0 < n
 shows {inv_loop1 n} tcopy_loop {inv_loop0 n}
 using assms
proof(rule_tac Hoare_haltI)
 fix l r
 assume h: 0 < n inv_loop1 n (l, r)
 then obtain stp where k: is_final (steps0 (1, l, r) tcopy_loop stp)
  using loop_halts
  apply(simp add: inv_loop.simps)
  apply(blast)
  done
 moreover
 have inv_loop n (steps0 (1, l, r) tcopy_loop stp)
  using h
  by (rule_tac inv_loop_steps) (simp_all add: inv_loop.simps)
 ultimately show
  ∃ stp. is_final (steps0 (1, l, r) tcopy_loop stp) ∧
  inv_loop0 n holds_for steps0 (1, l, r) tcopy_loop stp
  using h(1)
  apply(rule_tac x = stp in exI)
  apply(case_tac (steps0 (1, l, r) tcopy_loop stp))
  apply(simp add: inv_loop.simps)
  done
qed
```

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

fun
 inv_end0 :: nat ⇒ tape ⇒  bool and
```

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

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

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

lemma inv_end_nonempty[simp]:
  inv_end1 x (b, []) = False
  inv_end1 x ([], list) = False
  inv_end2 x (b, []) = False
  inv_end3 x (b, []) = False
  inv_end4 x (b, []) = False
  inv_end5 x (b, []) = False
  inv_end5 x ([], Oc # list) = False
  by (auto simp: inv_end1.simps inv_end2.simps inv_end3.simps inv_end4.simps inv_end5.simps)

lemma inv_end0_Bk_via_1[elim]: ⟦0 < x; inv_end1 x (b, Bk # list); b ≠ []⟧
  ⟹ inv_end0 x (tl b, hd b # Bk # list)
  by (auto simp: inv_end1.simps inv_end0.simps)

lemma inv_end3_Oc_via_2[elim]: ⟦0 < x; inv_end2 x (b, Bk # list)⟧
  ⟹ inv_end3 x (b, Oc # list)
  apply(auto simp: inv_end2.simps inv_end3.simps)
  by (metis Cons_replicate_eq One_nat_def Suc_inject Suc_pred add_Suc_right cell.distinct(1)
    empty_replicate list.sel(3) neq0_conv self_append_conv2 tl_append2 tl_replicate)
```

lemma *inv_end2_Bk_via_3*[*elim*]: ⟦*0* < *x*; *inv_end3 x* (*b*, *Bk* # *list*)⟧ ⟹ *inv_end2 x* (*Bk* # *b*, *list*)
by (*auto simp*: *inv_end2*.*simps inv_end3*.*simps*)

lemma *inv_end5_Bk_via_4*[*elim*]: ⟦*0* < *x*; *inv_end4 x* ([], *Bk* # *list*)⟧ ⟹
inv_end5 x ([], *Bk* # *Bk* # *list*)
by (*auto simp*: *inv_end4*.*simps inv_end5*.*simps*)

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*)
apply(*auto simp*: *inv_end4*.*simps inv_end5*.*simps*)
apply(*rule_tac x* = *1* **in** *exI*, *simp*)
done

lemma *inv_end0_Bk_via_5*[*elim*]: ⟦*0* < *x*; *inv_end5 x* (*b*, *Bk* # *list*)⟧ ⟹ *inv_end0 x* (*Bk* # *b*, *list*)
by(*auto simp*: *inv_end5*.*simps inv_end0*.*simps gr0_conv_Suc*)

lemma *inv_end2_Oc_via_1*[*elim*]: ⟦*0* < *x*; *inv_end1 x* (*b*, *Oc* # *list*)⟧ ⟹ *inv_end2 x* (*Oc* # *b*, *list*)
by (*auto simp*: *inv_end1*.*simps inv_end2*.*simps*)

lemma *inv_end4_Bk_Oc_via_2*[*elim*]: ⟦*0* < *x*; *inv_end2 x* ([], *Oc* # *list*)⟧ ⟹
inv_end4 x ([], *Bk* # *Oc* # *list*)
by (*auto simp*: *inv_end2*.*simps inv_end4*.*simps*)

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*)
by(*auto simp*: *inv_end2*.*simps inv_end4*.*simps gr0_conv_Suc*)

lemma *inv_end2_Oc_via_3*[*elim*]: ⟦*0* < *x*; *inv_end3 x* (*b*, *Oc* # *list*)⟧ ⟹ *inv_end2 x* (*Oc* # *b*, *list*)
by (*auto simp*: *inv_end2*.*simps inv_end3*.*simps*)

lemma *inv_end4_Bk_via_Oc*[*elim*]: ⟦*0* < *x*; *inv_end4 x* (*b*, *Oc* # *list*)⟧ ⟹ *inv_end4 x* (*b*, *Bk* # *list*)
by (*auto simp*: *inv_end2*.*simps inv_end4*.*simps*)

lemma *inv_end5_Bk_drop_Oc*[*elim*]: ⟦*0* < *x*; *inv_end5 x* ([], *Oc* # *list*)⟧ ⟹ *inv_end5 x* ([], *Bk* # *Oc* # *list*)
by (*auto simp*: *inv_end2*.*simps inv_end5*.*simps*)

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*
by (*auto simp*: *inv_end5_exit*.*simps*)

lemma *inv_end5_loop_no_Bk_Oc*[*simp*]: *inv_end5_loop x* (*tl b*, *Bk* # *Oc* # *list*) = *False*
by (*auto simp*: *inv_end5_loop*.*simps*)

```
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)
 apply(auto simp: inv_end5_loop.simps inv_end5_exit.simps)
 using hd_replicate apply fastforce
 by (metis cell.distinct(1) hd_append2 hd_replicate list.sel(3) self_append_conv2
   split_head_repeat(2))

lemma inv_end5_loop_Oc_Oc_drop[elim]:
 ⟦0 < x; inv_end5_loop x (b, Oc # list); b ≠ []; hd b = Oc⟧ ⟹
 inv_end5_loop x (tl b, Oc # Oc # list)
 apply(simp only: inv_end5_loop.simps inv_end5_exit.simps)
 apply(erule_tac exE)+
 apply(rename_tac i j)
 apply(rule_tac x = i − 1 in exI,
   rule_tac x = Suc j in exI, auto)
  apply(case_tac [!] i, simp_all)
 done

lemma inv_end5_Oc_tail[elim]: ⟦0 < x; inv_end5 x (b, Oc # list); b ≠ []⟧ ⟹
 inv_end5 x (tl b, hd b # Oc # list)
 apply(simp add: inv_end2.simps inv_end5.simps)
 apply(case_tac hd b, simp_all, auto)
 done

lemma inv_end_step:
 ⟦x > 0; inv_end x cf⟧ ⟹ inv_end x (step cf (tcopy_end, 0))
 apply(cases cf, cases snd (snd cf); cases hd (snd (snd cf)))
  apply(auto simp: inv_end.simps step.simps tcopy_end_def numeral split: if_splits)
 done

lemma inv_end_steps:
 ⟦x > 0; inv_end x cf⟧ ⟹ inv_end x (steps cf (tcopy_end, 0) stp)
 apply(induct stp, simp add:steps.simps, simp)
 apply(erule_tac inv_end_step, simp)
 done

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
 unfolding end_LE_def by auto

lemma halt_lemma:
 ⟦wf LE; ∀ n. (¬ P (f n) ⟶ (f (Suc n), (f n)) ∈ LE)⟧ ⟹ ∃ n. P (f n)
 by (metis wf_iff_no_infinite_down_chain)

lemma end_halt:
 ⟦x > 0; inv_end x (Suc 0, l, r)⟧ ⟹
    ∃ stp. is_final (steps (Suc 0, l, r) (tcopy_end, 0) stp)
proof(rule halt_lemma[OF wf_end_le])
 assume great: 0 < x
  and inv_start: inv_end x (Suc 0, l, r)
 show ∀ n. ¬ is_final (steps (Suc 0, l, r) (tcopy_end, 0) n) ⟶
  (steps (Suc 0, l, r) (tcopy_end, 0) (Suc n), steps (Suc 0, l, r) (tcopy_end, 0) n) ∈ end_LE
 proof(rule_tac allI, rule_tac impI)
  fix n
  assume notfinal: ¬ is_final (steps (Suc 0, l, r) (tcopy_end, 0) n)
  obtain s′ l′ r′ where d: steps (Suc 0, l, r) (tcopy_end, 0) n = (s′, l′, r′)
   apply(case_tac steps (Suc 0, l, r) (tcopy_end, 0) n, auto)
   done
  hence inv_end x (s′, l′, r′) ∧ s′ ≠ 0
   using great inv_start notfinal
   apply(drule_tac stp = n in inv_end_steps, auto)
   done
  hence (step (s′, l′, r′) (tcopy_end, 0), s′, l′, r′) ∈ end_LE
   apply(cases r′; cases hd r′)
       apply(auto simp: inv_end.simps step.simps tcopy_end_def numeral end_LE_def split:
if_splits)
   done
  thus (steps (Suc 0, l, r) (tcopy_end, 0) (Suc n),
   steps (Suc 0, l, r) (tcopy_end, 0) n) ∈ end_LE
   using d
   by simp
```

qed
qed

lemma *end_correct*:
$n > 0 \Longrightarrow \{inv\_end1\ n\}\ tcopy\_end\ \{inv\_end0\ n\}$
proof(*rule_tac Hoare_haltI*)
fix *l r*
assume *h*: $0 < n$
inv_end1 n (*l*, *r*)
then have ∃ *stp*. *is_final* (*steps0* (*1*, *l*, *r*) *tcopy_end stp*)
by (*simp add*: *end_halt inv_end.simps*)
then obtain *stp* **where** *is_final* (*steps0* (*1*, *l*, *r*) *tcopy_end stp*) **..**
moreover have *inv_end n* (*steps0* (*1*, *l*, *r*) *tcopy_end stp*)
apply(*rule_tac inv_end_steps*)
using *h* **by**(*simp_all add*: *inv_end.simps*)
ultimately show
∃ *stp*. *is_final* (*steps* (*1*, *l*, *r*) (*tcopy_end*, *0*) *stp*) ∧
inv_end0 n holds_for steps (*1*, *l*, *r*) (*tcopy_end*, *0*) *stp*
using *h*
apply(*rule_tac x* = *stp* **in** *exI*)
apply(*cases* (*steps0* (*1*, *l*, *r*) *tcopy_end stp*))
apply(*simp add*: *inv_end.simps*)
done
qed

lemma *tm_wf_tcopy*[*intro*]:
tm_wf (*tcopy_begin*, *0*)
tm_wf (*tcopy_loop*, *0*)
tm_wf (*tcopy_end*, *0*)
by (*auto simp*: *tm_wf.simps tcopy_end_def tcopy_loop_def tcopy_begin_def*)

lemma *tcopy_correct1*:
assumes $0 < x$
shows {*inv_begin1 x*} *tcopy* {*inv_end0 x*}
proof −
have {*inv_begin1 x*} *tcopy_begin* {*inv_begin0 x*}
by (*metis assms begin_correct*)
moreover
have *inv_begin0 x* ↦ *inv_loop1 x*
unfolding *assert_imp_def*
unfolding *inv_begin0.simps inv_loop1.simps*
unfolding *inv_loop1_loop.simps inv_loop1_exit.simps*
apply(*auto simp add*: *numeral Cons_eq_append_conv*)
by (*rule_tac x* = *Suc 0* **in** *exI*, *auto*)
ultimately have {*inv_begin1 x*} *tcopy_begin* {*inv_loop1 x*}
by (*rule_tac Hoare_consequence*) (*auto*)
moreover
have {*inv_loop1 x*} *tcopy_loop* {*inv_loop0 x*}

by (*metis assms loop_correct*)
ultimately
have {*inv_begin1 x*} (*tcopy_begin* |+| *tcopy_loop*) {*inv_loop0 x*}
by (*rule_tac Hoare_plus_halt*) (*auto*)
moreover
have {*inv_end1 x*} *tcopy_end* {*inv_end0 x*}
by (*metis assms end_correct*)
moreover
have *inv_loop0 x* = *inv_end1 x*
by(*auto simp*: *inv_end1.simps inv_loop1.simps assert_imp_def*)
ultimately
show {*inv_begin1 x*} *tcopy* {*inv_end0 x*}
unfolding *tcopy_def*
by (*rule_tac Hoare_plus_halt*) (*auto*)
qed

abbreviation (*input*)
pre_tcopy n $\stackrel{def}{=}$ λ*tp*. *tp* = ([]::*cell list*, *Oc* $\uparrow$ (*Suc n*))
abbreviation (*input*)
post_tcopy n $\stackrel{def}{=}$ λ*tp*. *tp*= ([*Bk*], <(*n*, *n*::*nat*)>)

lemma *tcopy_correct*:
shows {*pre_tcopy n*} *tcopy* {*post_tcopy n*}
proof −
have {*inv_begin1* (*Suc n*)} *tcopy* {*inv_end0* (*Suc n*)}
by (*rule tcopy_correct1*) (*simp*)
moreover
have *pre_tcopy n* = *inv_begin1* (*Suc n*)
by (*auto*)
moreover
have *inv_end0* (*Suc n*) = *post_tcopy n*
unfolding *fun_eq_iff*
by (*auto simp add*: *inv_end0.simps tape_of_nat_def tape_of_prod_def*)
ultimately
show {*pre_tcopy n*} *tcopy* {*post_tcopy n*}
by *simp*
qed

5 The *Dithering* Turing Machine

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

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

abbreviation (*input*)

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

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

lemma *dither_loops_aux*:
$(steps0\ (1, Bk \uparrow m, [Oc])\ dither\ stp = (1, Bk \uparrow m, [Oc])) \lor$
$(steps0\ (1, Bk \uparrow m, [Oc])\ dither\ stp = (2, Oc \# Bk \uparrow m, []))$
apply(*induct stp*)
apply(*auto simp*: *steps.simps step.simps dither_def numeral*)
done

lemma *dither_loops*:
shows {*dither_unhalt_inv*} *dither* ↑
apply(*rule Hoare_unhaltI*)
using *dither_loops_aux*
apply(*auto simp add*: *numeral tape_of_nat_def*)
by (*metis Suc_neq_Zero is_final_eq*)

lemma *dither_halts_aux*:
shows $steps0\ (1, Bk \uparrow m, [Oc, Oc])\ dither\ 2 = (0, Bk \uparrow m, [Oc, Oc])$
unfolding *dither_def*
by (*simp add*: *steps.simps step.simps numeral*)

lemma *dither_halts*:
shows {*dither_halt_inv*} *dither* {*dither_halt_inv*}
apply(*rule Hoare_haltI*)
using *dither_halts_aux*
apply(*auto simp add*: *tape_of_nat_def*)
by (*metis* (*lifting*, *mono_tags*) *holds_for.simps is_final_eq*)

6 The diagnal argument below shows the undecidability of Halting problem

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

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

lemma *tm_wf0_tcopy*[*intro*, *simp*]: *tm_wf0 tcopy*
by (*auto simp*: *tcopy_def*)

lemma *tm_wf0_dither*[*intro*, *simp*]: *tm_wf0 dither*
by (*auto simp*: *tm_wf.simps dither_def*)

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

Halting Problem is established.

locale *uncomputable* =

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

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

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

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

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

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

lemma *H_halt_inv*:
assumes $\neg$ *halts M ns*
shows {*pre_H_inv M ns*} *H* {*post_H_halt_inv*}
using *assms nh_case* **by** *auto*

lemma *H_unhalt_inv*:
assumes *halts M ns*
shows {*pre_H_inv M ns*} *H* {*post_H_unhalt_inv*}
using *assms h_case* **by** *auto*

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

lemma *tcontra_unhalt*:
assumes $\neg$ *halts tcontra* [*code tcontra*]
shows *False*

```
proof −

 define P1 where P1 ≝ λtp. tp = ([]::cell list, <code_tcontra>)
 define P2 where P2 ≝ λtp. tp = ([Bk], <(code_tcontra, code_tcontra)>)
 define P3 where P3 ≝ λtp. ∃ k. tp = (Bk ↑ k, <1::nat>)


 have H_wf: tm_wf0 (tcopy |+| H) by auto


 have first: {P1} (tcopy |+| H) {P3}
 proof (cases rule: Hoare_plus_halt)
  case A_halt
  show {P1} tcopy {P2} unfolding P1_def P2_def tape_of_nat_def
   by (rule tcopy_correct)
 next
  case B_halt
  show {P2} H {P3}
   unfolding P2_def P3_def
   using H_halt_inv[OF assms]
   by (simp add: tape_of_prod_def tape_of_list_def)
 qed (simp)


 have second: {P3} dither {P3} unfolding P3_def
  by (rule dither_halts)


 have {P1} tcontra {P3}
  unfolding tcontra_def
  by (rule Hoare_plus_halt[OF first second H_wf])

 with assms show False
  unfolding P1_def P3_def
  unfolding halts_def
  unfolding Hoare_halt_def
  apply(auto) apply(rename_tac n)
  apply(drule_tac x = n in spec)
  apply(case_tac steps0 (Suc 0, [], <code tcontra>) tcontra n)
  apply(auto simp add: tape_of_list_def)
  by (metis append_Nil2 replicate_0)
qed


lemma tcontra_halt:
 assumes halts tcontra [code tcontra]
 shows False
proof −
```

```
  define P1 where P1 ≝ λtp. tp = ([]::cell list, <code_tcontra>)
  define P2 where P2 ≝ λtp. tp = ([Bk], <(code_tcontra, code_tcontra)>)
  define Q3 where Q3 ≝ λtp. ∃k. tp = (Bk ↑ k, <0::nat>)


  have H_wf: tm_wf0 (tcopy |+| H) by auto


  have first: {P1} (tcopy |+| H) {Q3}
  proof (cases rule: Hoare_plus_halt)
    case A_halt
    show {P1} tcopy {P2} unfolding P1_def P2_def tape_of_nat_def
      by (rule tcopy_correct)
  next
    case B_halt
    then show {P2} H {Q3}
      unfolding P2_def Q3_def using H_unhalt_inv[OF assms]
      by(simp add: tape_of_prod_def tape_of_list_def)
  qed (simp)


  have second: {Q3} dither ↑ unfolding Q3_def
    by (rule dither_loops)


  have {P1} tcontra ↑
    unfolding tcontra_def
    by (rule Hoare_plus_unhalt[OF first second H_wf])

  with assms show False
    unfolding P1_def
    unfolding halts_def
    unfolding Hoare_halt_def Hoare_unhalt_def
    by (auto simp add: tape_of_list_def)
qed
```

False can finally derived.

```
lemma false: False
  using tcontra_halt tcontra_unhalt
  by auto

end

declare replicate_Suc[simp del]

end
```

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

theory *Abacus_Mopup*
 imports *Uncomputable*
begin

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

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

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

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

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

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

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

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

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

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

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

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

lemma mopup_a_nth:
 ⟦q < n; x < 4⟧ ⟹ mopup_a n ! (4 * q + x) =
                  mopup_a (Suc q) ! ((4 * q) + x)
proof(induct n)
 case (Suc n)
 then show ?case
  by(cases q < n;cases q = n, auto simp add: nth_append)
qed auto

lemma fetch_bef_erase_a_o[simp]:
 ⟦0 < s; s ≤ 2 * n; s mod 2 = Suc 0⟧
 ⟹ (fetch (mopup_a n @ shift mopup_b (2 * n)) s Oc) = (W0, s + 1)
 apply(subgoal_tac ∃ q. s = 2*q + 1, auto)
  apply(subgoal_tac length (mopup_a n) = 4*n)
   apply(auto simp: nth_append)
  apply(subgoal_tac mopup_a n ! (4 * q + 1) =
                mopup_a (Suc q) ! ((4 * q) + 1),
    simp add: nth_append)
  apply(rule mopup_a_nth, auto)
 apply arith
 done

lemma fetch_bef_erase_a_b[simp]:
 ⟦0 < s; s ≤ 2 * n; s mod 2 = Suc 0⟧
  ⟹  (fetch (mopup_a n @ shift mopup_b (2 * n)) s Bk) = (R, s + 2)
 apply(subgoal_tac ∃ q. s = 2*q + 1, auto)
  apply(subgoal_tac length (mopup_a n) = 4*n)
   apply(auto simp: nth_append)
  apply(subgoal_tac mopup_a n ! (4 * q + 0) =
                 mopup_a (Suc q) ! ((4 * q + 0)),
    simp add: nth_append)
  apply(rule mopup_a_nth, auto)
 apply arith
 done

lemma fetch_bef_erase_b_b:
 assumes n < length lm 0 < s s ≤ 2 * n s mod 2 = 0
 shows (fetch (mopup_a n @ shift mopup_b (2 * n)) s Bk) = (R, s − 1)
proof −
```

```
  from assms obtain q where q:s = 2 * q by auto
  then obtain nat where nat:q = Suc nat using assms(2) by (cases q, auto)
  from assms(3) mopup_a_nth[of nat n 2]
  have mopup_a n ! (4 * nat + 2) = mopup_a (Suc nat) ! ((4 * nat) + 2)
    unfolding nat q by auto
  thus ?thesis using assms nat q by (auto simp: nth_append)
qed

lemma fetch_jump_over1_o:
  fetch (mopup_a n @ shift mopup_b (2 * n)) (Suc (2 * n)) Oc
  = (R, Suc (2 * n))
  apply(subgoal_tac length (mopup_a n) = 4 * n)
   apply(auto simp: mopup_b_def nth_append shift.simps)
  done

lemma fetch_jump_over1_b:
  fetch (mopup_a n @ shift mopup_b (2 * n)) (Suc (2 * n)) Bk
  = (R, Suc (Suc (2 * n)))
  apply(subgoal_tac length (mopup_a n) = 4 * n)
   apply(auto simp: mopup_b_def nth_append shift.simps)
  done

lemma fetch_aft_erase_a_o:
  fetch (mopup_a n @ shift mopup_b (2 * n)) (Suc (Suc (2 * n))) Oc
  = (W0, Suc (2 * n + 2))
  apply(subgoal_tac length (mopup_a n) = 4 * n)
   apply(auto simp: mopup_b_def nth_append shift.simps)
  done

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))
  apply(subgoal_tac length (mopup_a n) = 4 * n)
   apply(auto simp: mopup_b_def nth_append shift.simps)
  done

lemma fetch_aft_erase_b_b:
  fetch (mopup_a n @ shift mopup_b (2 * n)) (2*n + 3) Bk
  = (R, Suc (2 * n + 3))
  apply(subgoal_tac length (mopup_a n) = 4 * n)
   apply(subgoal_tac 2*n + 3 = Suc (2*n + 2), simp only: fetch.simps)
    apply(auto simp: mopup_b_def nth_append shift.simps)
  done

lemma fetch_aft_erase_c_o:
  fetch (mopup_a n @ shift mopup_b (2 * n)) (2 * n + 4) Oc
  = (W0, Suc (2 * n + 2))
  apply(subgoal_tac length (mopup_a n) = 4 * n)
   apply(subgoal_tac 2*n + 4 = Suc (2*n + 3), simp only: fetch.simps)
    apply(auto simp: mopup_b_def nth_append shift.simps)
```

done

lemma *fetch_aft_erase_c_b*:
fetch (*mopup_a n* @ *shift mopup_b* (*2* ∗ *n*)) (*2* ∗ *n* + *4*) *Bk*
= (*R*, *Suc* (*2* ∗ *n* + *1*))
apply(*subgoal_tac length* (*mopup_a n*) = *4* ∗ *n*)
apply(*subgoal_tac 2*∗*n* + *4* = *Suc* (*2*∗*n* + *3*), *simp only*: *fetch*.*simps*)
apply(*auto simp*: *mopup_b_def nth_append shift*.*simps*)
done

lemma *fetch_left_moving_o*:
(*fetch* (*mopup_a n* @ *shift mopup_b* (*2* ∗ *n*)) (*2* ∗ *n* + *5*) *Oc*)
= (*L*, *2*∗*n* + *6*)
apply(*subgoal_tac length* (*mopup_a n*) = *4* ∗ *n*)
apply(*subgoal_tac 2*∗*n* + *5* = *Suc* (*2*∗*n* + *4*), *simp only*: *fetch*.*simps*)
apply(*auto simp*: *mopup_b_def nth_append shift*.*simps*)
done

lemma *fetch_left_moving_b*:
(*fetch* (*mopup_a n* @ *shift mopup_b* (*2* ∗ *n*)) (*2* ∗ *n* + *5*) *Bk*)
= (*L*, *2*∗*n* + *5*)
apply(*subgoal_tac length* (*mopup_a n*) = *4* ∗ *n*)
apply(*subgoal_tac 2*∗*n* + *5* = *Suc* (*2*∗*n* + *4*), *simp only*: *fetch*.*simps*)
apply(*auto simp*: *mopup_b_def nth_append shift*.*simps*)
done

lemma *fetch_jump_over2_b*:
(*fetch* (*mopup_a n* @ *shift mopup_b* (*2* ∗ *n*)) (*2* ∗ *n* + *6*) *Bk*)
= (*R*, *0*)
apply(*subgoal_tac length* (*mopup_a n*) = *4* ∗ *n*)
apply(*subgoal_tac 2*∗*n* + *6* = *Suc* (*2*∗*n* + *5*), *simp only*: *fetch*.*simps*)
apply(*auto simp*: *mopup_b_def nth_append shift*.*simps*)
done

lemma *fetch_jump_over2_o*:
(*fetch* (*mopup_a n* @ *shift mopup_b* (*2* ∗ *n*)) (*2* ∗ *n* + *6*) *Oc*)
= (*L*, *2*∗*n* + *6*)
apply(*subgoal_tac length* (*mopup_a n*) = *4* ∗ *n*)
apply(*subgoal_tac 2*∗*n* + *6* = *Suc* (*2*∗*n* + *5*), *simp only*: *fetch*.*simps*)
apply(*auto simp*: *mopup_b_def nth_append shift*.*simps*)
done

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
 apply(auto simp: mopup_bef_erase_a.simps mopup_bef_erase_b.simps)
 by (metis cell.distinct(1) hd_append list.sel(1) list.sel(3) tl_append2 tl_replicate)

lemma mopup_false1:
 ⟦0 < s; s ≤ 2 ∗ n; s mod 2 = Suc 0;  ¬ Suc s ≤ 2 ∗ n⟧
 ⟹ RR
 apply(arith)
 done

lemma mopup_bef_erase_a_implies_two[simp]:
 ⟦n < length lm; 0 < s; s ≤ 2 ∗ n; s mod 2 = Suc 0;
  mopup_bef_erase_a (s, l, Oc # xs) lm n ires; r = Oc # xs⟧
⟹ (Suc s ≤ 2 ∗ n ⟶ mopup_bef_erase_b (Suc s, l, Bk # xs) lm n ires)  ∧
   (¬ Suc s ≤ 2 ∗ n ⟶ mopup_jump_over1 (Suc s, l, Bk # xs) lm n ires)
 apply(auto elim!: mopup_false1)
 done

lemma tape_of_nl_cons: <m # lm> = (if lm = [] then Oc↑(Suc m)
              else Oc↑(Suc m) @ Bk # <lm>)
 by(cases lm, simp_all add: tape_of_list_def  tape_of_nat_def split: if_splits)

lemma drop_tape_of_cons:
 ⟦Suc q < length lm; x = lm ! q⟧ ⟹ <drop q lm> = Oc # Oc ↑ x @ Bk # <drop (Suc q) lm>
 using Suc_lessD append_Cons list.simps(2) Cons_nth_drop_Suc replicate_Suc tape_of_nl_cons
 by metis

lemma erase2jumpover1:
 ⟦q < length list;
        ∀ rn. <drop q list> ≠ Oc # Oc ↑ (list ! q) @ Bk # <drop (Suc q) list> @ Bk ↑ rn⟧
     ⟹ <drop q list> = Oc # Oc ↑ (list ! q)
 apply(erule_tac x = 0 in allE, simp)
 apply(cases Suc q < length list)
  apply(erule_tac notE)
  apply(rule_tac drop_tape_of_cons, simp_all)
 apply(subgoal_tac length list = Suc q, auto)
 apply(subgoal_tac drop q list = [list ! q])
  apply(simp add: tape_of_nat_def tape_of_list_def replicate_Suc)
 by (metis append_Nil2 append_eq_conv_conj Cons_nth_drop_Suc lessI)

lemma erase2jumpover2:
 ⟦q < length list; ∀ rn. <drop q list> @ Bk # Bk ↑ n ≠
```

```
  Oc # Oc ↑ (list ! q) @ Bk # <drop (Suc q) list> @ Bk ↑ rn⟧
  ⟹ RR
  apply(cases Suc q < length list)
   apply(erule_tac x = Suc n in allE, simp)
   apply(erule_tac notE, simp add: replicate_Suc)
   apply(rule_tac drop_tape_of_cons, simp_all)
  apply(subgoal_tac length list = Suc q, auto)
  apply(erule_tac x = n in allE, simp add: tape_of_list_def)
 by (metis append_Nil2 append_eq_conv_conj Cons_nth_drop_Suc lessI replicate_Suc tape_of_list_def
tape_of_nl_cons)

lemma mod_ex1: (a mod 2 = Suc 0) = (∃ q. a = Suc (2 ∗ q))
  by arith

declare replicate_Suc[simp]

lemma mopup_bef_erase_a_2_jump_over[simp]:
  ⟦n < length lm; 0 < s; s mod 2 = Suc 0;  s ≤ 2 ∗ n;
   mopup_bef_erase_a (s, l, Bk # xs) lm n ires; ¬ (Suc (Suc s) ≤ 2 ∗ n)⟧
⟹ mopup_jump_over1 (s′, Bk # l, xs) lm n ires
proof(cases n)
  case (Suc nat)
  assume assms: n < length lm 0 < s s mod 2 = Suc 0 s ≤ 2 ∗ n
    mopup_bef_erase_a (s, l, Bk # xs) lm n ires ¬ (Suc (Suc s) ≤ 2 ∗ n)
  from assms obtain a lm′ where Cons:lm = Cons a lm′ by (cases lm,auto)
  from assms have n:Suc s div 2 = n by auto
  have [simp]:s = Suc (2 ∗ q) ⟷ q = nat for q using assms Suc by presburger
  from assms obtain ln m rn where ln:l = Bk ↑ ln @ Bk # Bk # ires
    and Bk # xs = Oc ↑ m @ Bk # <drop (Suc s div 2) lm> @ Bk ↑ rn
    by (auto simp: mopup_bef_erase_a.simps mopup_jump_over1.simps)
  hence xs:xs = <drop n lm> @ Bk ↑ rn by(cases m;auto simp: n mod_ex1)
  have [intro]:nat < length lm′ ⟹
    ∀ rna. xs ≠ Oc # Oc ↑ (lm′ ! nat) @ Bk # <drop (Suc nat) lm′> @ Bk ↑ rna ⟹
    <drop nat lm′> @ Bk ↑ rn = Oc # Oc ↑ (lm′ ! nat)
    by(cases rn, auto elim: erase2jumpover1 erase2jumpover2 simp:xs Suc Cons)
  have [intro]:<drop nat lm′> ≠ Oc # Oc ↑ (lm′ ! nat) @ Bk # <drop (Suc nat) lm′> @ Bk ↑
0 ⟹ length lm′ ≤ Suc nat
    using drop_tape_of_cons[of nat lm′] by fastforce
  from assms(1,3) have [intro!]:
        0 + Suc (lm′ ! nat) = Suc (lm′ ! nat) ∧
        Bk # Bk ↑ ln = Oc ↑ 0 @ Bk ↑ Suc ln ∧
        ((∃ rna. xs = Oc ↑ Suc (lm′ ! nat) @ Bk # <drop (Suc nat) lm′> @ Bk ↑ rna) ∨
         xs = Oc ↑ Suc (lm′ ! nat) ∧ length lm′ ≤ Suc nat)
    by (auto simp:Cons ln xs Suc)
  from assms(1,3) show ?thesis unfolding Cons ln Suc
   by(auto simp: mopup_bef_erase_a.simps mopup_jump_over1.simps simp del:split_head_repeat)
qed auto


lemma Suc_Suc_div:  ⟦0 < s; s mod 2 = Suc 0; Suc (Suc s) ≤ 2 ∗ n⟧
```

```
      ⟹ (Suc (Suc (s div 2))) ≤ n by(arith)

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) ≤ 2 ∗ n
 shows mopup_bef_erase_a (Suc (Suc s), Bk # l, xs) lm n ires
proof−
 from assms obtain rn m ln where
   rn:l = Bk ↑ ln @ Bk # Bk # ires Bk # xs = Oc ↑ m @ Bk # <drop (Suc s div 2) lm> @ Bk
↑ rn
   by (auto simp: mopup_bef_erase_a.simps)
 hence m:m = 0 using assms by (cases m,auto)
 hence d:drop (Suc (Suc (s div 2))) lm ≠ []
   using assms(1,3,5) by auto arith
 hence Bk # l = Bk ↑ Suc ln @ Bk # Bk # ires ∧
   xs = Oc ↑ Suc (lm ! (Suc s div 2)) @ Bk # <drop ((Suc (Suc s) + 1) div 2) lm> @ Bk ↑ rn
   using rn by(auto intro:drop_tape_of_cons simp:m)
 thus ?thesis unfolding mopup_bef_erase_a.simps by blast
qed

lemma mopup_false2:
 ⟦0 < s; s ≤ 2 ∗ n;
  s mod 2 = Suc 0; Suc s ≠ 2 ∗ n;
  ¬ Suc (Suc s) ≤ 2 ∗ n⟧ ⟹ RR
 by(arith)

lemma ariths[simp]: ⟦0 < s; s ≤ 2 ∗n; s mod 2 ≠ Suc 0⟧ ⟹
                         (s − Suc 0) mod 2 = Suc 0
 ⟦0 < s; s ≤ 2 ∗n; s mod 2 ≠ Suc 0⟧ ⟹
                          s − Suc 0 ≤ 2 ∗ n
 ⟦0 < s; s ≤ 2 ∗n; s mod 2 ≠ Suc 0⟧ ⟹ ¬ s ≤ Suc 0
 by(arith)+

lemma take_suc[intro]:
 ∃ lna. Bk # Bk ↑ ln = Bk ↑ lna
 by(rule_tac x = Suc ln in exI, simp)


lemma mopup_bef_erase[simp]: mopup_bef_erase_a (s, l, []) lm n ires ⟹
              mopup_bef_erase_a (s, l, [Bk]) lm n ires
 ⟦n < length lm; 0 < s; s ≤ 2 ∗ n; s mod 2 = Suc 0; ¬ Suc (Suc s) ≤ 2 ∗n;
   mopup_bef_erase_a (s, l, []) lm n ires⟧
  ⟹  mopup_jump_over1 (s′, Bk # l, []) lm n ires
 mopup_bef_erase_b (s, l, Oc # xs) lm n ires ⟹ l ≠ []
 ⟦n < length lm; 0 < s; s ≤ 2 ∗ n;
          s mod 2 ≠ Suc 0;
          mopup_bef_erase_b (s, l, Bk # xs) lm n ires; r = Bk # xs⟧
        ⟹ mopup_bef_erase_a (s − Suc 0, Bk # l, xs) lm n ires
 ⟦mopup_bef_erase_b (s, l, []) lm n ires⟧ ⟹
```

```
        mopup_bef_erase_a (s − Suc 0, Bk # l, []) lm n ires
  by(auto simp: mopup_bef_erase_b.simps mopup_bef_erase_a.simps)


lemma mopup_jump_over1_in_ctx[simp]:
  assumes mopup_jump_over1 (Suc (2 ∗ n), l, Oc # xs) lm n ires
  shows mopup_jump_over1 (Suc (2 ∗ n), Oc # l, xs) lm n ires
proof −
  from assms obtain ln m1 m2 rn where
     m1 + m2 = Suc (lm ! n)
     l = Oc ↑ m1 @ Bk ↑ ln @ Bk # Bk # ires
     (Oc # xs = Oc ↑ m2 @ Bk # <drop (Suc n) lm> @ Bk ↑ rn ∨
      Oc # xs = Oc ↑ m2 ∧ drop (Suc n) lm = []) unfolding mopup_jump_over1.simps by blast
  thus ?thesis unfolding mopup_jump_over1.simps
   apply(rule_tac x = ln in exI, rule_tac x = Suc m1 in exI
      ,rule_tac x = m2 − 1 in exI)
   by(cases m2, auto)
qed

lemma mopup_jump_over1_2_aft_erase_a[simp]:
  assumes mopup_jump_over1 (Suc (2 ∗ n), l, Bk # xs) lm n ires
  shows mopup_aft_erase_a (Suc (Suc (2 ∗ n)), Bk # l, xs) lm n ires
proof −
  from assms obtain ln m1 m2 rn where
     m1 + m2 = Suc (lm ! n)
     l = Oc ↑ m1 @ Bk ↑ ln @ Bk # Bk # ires
     (Bk # xs = Oc ↑ m2 @ Bk # <drop (Suc n) lm> @ Bk ↑ rn ∨
      Bk # xs = Oc ↑ m2 ∧ drop (Suc n) lm = []) unfolding mopup_jump_over1.simps by blast
  thus ?thesis unfolding mopup_aft_erase_a.simps
   apply( rule_tac x = ln in exI, rule_tac x = Suc 0 in exI,rule_tac x = rn in exI
      , rule_tac x = drop (Suc n) lm in exI)
   by(cases m2, auto)
qed

lemma mopup_aft_erase_a_via_jump_over1[simp]:
  ⟦mopup_jump_over1 (Suc (2 ∗ n), l, []) lm n ires⟧ ⟹
   mopup_aft_erase_a (Suc (Suc (2 ∗ n)), Bk # l, []) lm n ires
proof(rule mopup_jump_over1_2_aft_erase_a)
  assume a:mopup_jump_over1 (Suc (2 ∗ n), l, []) lm n ires
  then obtain ln where ln:length lm ≤ Suc n ⟹ l = Oc # Oc ↑ (lm ! n) @ Bk ↑ ln @ Bk # Bk
# ires
   unfolding mopup_jump_over1.simps by auto
  show mopup_jump_over1 (Suc (2 ∗ n), l, [Bk]) lm n ires
   unfolding mopup_jump_over1.simps
   apply(rule_tac x = ln in exI, rule_tac x = Suc (lm ! n) in exI,
      rule_tac x = 0 in exI)
   using a ln by(auto simp: mopup_jump_over1.simps tape_of_list_def )
qed

lemma tape_of_list_empty[simp]: <[]> = [] by(simp add: tape_of_list_def)
```

```
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 -
  from assms obtain lnl lnr rn ml where
     assms:
     l = Bk ↑ lnr @ Oc ↑ Suc (lm ! n) @ Bk ↑ lnl @ Bk # Bk # ires
     Oc # xs = <ml::nat list> @ Bk ↑ rn
    unfolding mopup_aft_erase_a.simps by auto
  then obtain a list where ml:ml = a # list by (cases ml,cases rn,auto)
  with assms show ?thesis unfolding mopup_aft_erase_b.simps
    apply(auto simp add: tape_of_nl_cons split: if_splits)
     apply(cases a, simp_all)
      apply(rule_tac x = rn in exI, rule_tac x = [] in exI, force)
     apply(rule_tac x = rn in exI, rule_tac x = [a-1] in exI)
     apply(cases a; force simp add: tape_of_list_def tape_of_nat_def)
    apply(cases a)
     apply(rule_tac x = rn in exI, rule_tac x = list in exI, force)
    apply(rule_tac x = rn in exI,rule_tac x = (a-1) # list in exI, simp add: tape_of_nl_cons)
    done
qed

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-
  from assms[unfolded mopup_aft_erase_a.simps] obtain lnl lnr rn ml where
     l = Bk ↑ lnr @ Oc ↑ Suc (lm ! n) @ Bk ↑ lnl @ Bk # Bk # ires
     Bk # xs = <ml::nat list> @ Bk ↑ rn
    by auto
  thus ?thesis unfolding mopup_left_moving.simps
    by(cases lnr;cases ml,auto simp: tape_of_nl_cons)
qed

lemma mopup_aft_erase_a_nonempty[simp]:
  mopup_aft_erase_a (Suc (Suc (2 * n)), l, xs) lm n ires ⟹ l ≠ []
  by(auto simp only: mopup_aft_erase_a.simps)

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 -
  have [intro!]:[Bk] = Bk ↑ 1 by auto
  from assms obtain lnl lnr where l = Bk ↑ lnr @ Oc # Oc ↑ (lm ! n) @ Bk ↑ lnl @ Bk # Bk #
ires
    unfolding mopup_aft_erase_a.simps by auto
  thus ?thesis by(case_tac lnr, auto simp add:mopup_left_moving.simps)
qed
```

```
lemma mopup_aft_erase_b_no_Oc[simp]: mopup_aft_erase_b (2 ∗ n + 3, l, Oc # xs) lm n ires =
False
 by(auto simp: mopup_aft_erase_b.simps)

lemma tape_of_ex1[intro]:
 ∃ rna ml. Oc ↑ a @ Bk ↑ rn = <ml::nat list> @ Bk ↑ rna ∨ Oc ↑ a @ Bk ↑ rn = Bk # <ml>
@ Bk ↑ rna
 by(rule_tac x = rn in exI, rule_tac x = if a = 0 then [] else [a−1] in exI,
    simp add: tape_of_list_def tape_of_nat_def)

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
 apply(cases list = [], simp add: replicate_Suc[THEN sym] del: replicate_Suc split_head_repeat)
  apply(rule_tac rn = Suc rn in tape_of_ex1)
 apply(cases a, simp)
  apply(rule_tac x = rn in exI, rule_tac x = list in exI, simp)
 apply(rule_tac x = rn in exI, rule_tac x = (a−1) # list in exI)
 apply(simp add: tape_of_nl_cons)
 done

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−
 from assms obtain lnl rn lnr ml where assms:
    l = Bk ↑ lnr @ Oc # Oc ↑ (lm ! n) @ Bk ↑ lnl @ Bk # Bk # ires
    Oc # xs = <ml::nat list> @ Bk ↑ rn unfolding mopup_aft_erase_c.simps by auto
 hence Oc # xs = Bk ↑ rn ⟹ False by(cases rn,auto)
 thus ?thesis using assms unfolding mopup_aft_erase_b.simps
   by(cases ml)
     (auto simp add: tape_of_nl_cons split: if_splits intro:mopup_aft_erase_b_via_c_helper
      simp del:split_head_repeat)
qed

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 −
 from assms obtain lnl lnr rn ml where
    l = Bk ↑ lnr @ Oc ↑ Suc (lm ! n) @ Bk ↑ lnl @ Bk # Bk # ires
    (Bk # xs = <ml::nat list> @ Bk ↑ rn ∨ Bk # xs = Bk # <ml> @ Bk ↑ rn)
   unfolding mopup_aft_erase_c.simps by auto
 thus ?thesis unfolding mopup_aft_erase_a.simps
   apply(clarify)
   apply(erule disjE)
    apply(subgoal_tac ml = [], simp, case_tac rn,
      simp, simp, rule conjI)
      apply(rule_tac x = lnl in exI, rule_tac x = Suc lnr in exI, simp)
     apply (insert tape_of_list_empty,blast)
    apply(case_tac ml, simp, simp add: tape_of_nl_cons split: if_splits)
```

```
  apply(rule_tac x = lnl in exI, rule_tac x = Suc lnr in exI)
  apply(rule_tac x = rn in exI, rule_tac x = ml in exI, simp)
  done
qed

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
 by (rule mopup_aft_erase_c_aft_erase_a)
   (auto simp:mopup_aft_erase_c.simps)

lemma mopup_aft_erase_b_2_aft_erase_c[simp]:
 assumes mopup_aft_erase_b (2 ∗ n + 3, l, Bk # xs) lm n ires
 shows mopup_aft_erase_c (4 + 2 ∗ n, Bk # l, xs) lm n ires
proof −
 from assms obtain lnl lnr ml rn where
     l = Bk ↑ lnr @ Oc ↑ Suc (lm ! n) @ Bk ↑ lnl @ Bk # Bk # ires
     Bk # xs = Bk # <ml::nat list> @ Bk ↑ rn ∨ Bk # xs = Bk # Bk # <ml> @ Bk ↑ rn
   unfolding  mopup_aft_erase_b.simps by auto
 thus ?thesis unfolding mopup_aft_erase_c.simps
   by (rule_tac x = lnl in exI) auto
qed

lemma mopup_aft_erase_c_via_b[simp]:
 ⟦mopup_aft_erase_b (2 ∗ n + 3, l, []) lm n ires⟧
 ⟹ mopup_aft_erase_c (4 + 2 ∗ n, Bk # l, []) lm n ires
 by(auto simp add: mopup_aft_erase_b.simps intro:mopup_aft_erase_b_2_aft_erase_c)

lemma mopup_left_moving_nonempty[simp]:
 mopup_left_moving (2 ∗ n + 5, l, Oc # xs) lm n ires ⟹ l ≠ []
 by(auto simp: mopup_left_moving.simps)

lemma exp_ind: a↑(Suc x) = a↑x @ [a]
 by(induct x, auto)

lemma mopup_jump_over2_via_left_moving[simp]:
 ⟦mopup_left_moving (2 ∗ n + 5, l, Oc # xs) lm n ires⟧
 ⟹ mopup_jump_over2 (2 ∗ n + 6, tl l, hd l # Oc # xs) lm n ires
 apply(simp only: mopup_left_moving.simps mopup_jump_over2.simps)
 apply(erule_tac exE)+
 apply(erule conjE, erule disjE, erule conjE)
  apply (simp add: Cons_replicate_eq)
 apply(rename_tac Lnl lnr rn m)
 apply(cases hd l, simp add:  )
  apply(cases lm ! n, simp)
   apply(rule exI, rule_tac x = length xs in exI,
     rule_tac x = Suc 0 in exI, rule_tac x = 0 in exI)
   apply(case_tac Lnl, simp,simp,  simp add: exp_ind[THEN sym])
  apply(cases lm ! n, simp)
  apply(case_tac Lnl, simp, simp)
```

```
 apply(rule_tac x = Lnl in exI, rule_tac x = length xs in exI, auto)
 apply(cases lm ! n, simp)
  apply(case_tac Lnl, simp_all add: numeral_2_eq_2)
 done

lemma mopup_left_moving_nonempty_snd[simp]: mopup_left_moving (2 * n + 5, l, xs) lm n ires
⟹ l ≠ []
 apply(auto simp: mopup_left_moving.simps)
 done

lemma mopup_left_moving_hd_Bk[simp]:
 ⟦mopup_left_moving (2 * n + 5, l, Bk # xs) lm n ires⟧
 ⟹ mopup_left_moving (2 * n + 5, tl l, hd l # Bk # xs) lm n ires
 apply(simp only: mopup_left_moving.simps)
 apply(erule exE)+ apply(rename_tac lnl Lnr rn m)
 apply(case_tac Lnr, auto)
 done

lemma mopup_left_moving_emptylist[simp]:
 ⟦mopup_left_moving (2 * n + 5, l, []) lm n ires⟧
   ⟹ mopup_left_moving (2 * n + 5, tl l, [hd l]) lm n ires
 apply(simp only: mopup_left_moving.simps)
 apply(erule exE)+ apply(rename_tac lnl Lnr rn m)
 apply(case_tac Lnr, auto)
 apply(rule_tac x = 1 in exI, simp)
 done


lemma mopup_jump_over2_Oc_nonempty[simp]:
 mopup_jump_over2 (2 * n + 6, l, Oc # xs) lm n ires ⟹ l ≠ []
 apply(auto simp: mopup_jump_over2.simps )
 done

lemma mopup_jump_over2_context[simp]:
 ⟦mopup_jump_over2 (2 * n + 6, l, Oc # xs) lm n ires⟧
 ⟹  mopup_jump_over2 (2 * n + 6, tl l, hd l # Oc # xs) lm n ires
 apply(simp only: mopup_jump_over2.simps)
 apply(erule_tac exE)+
 apply(simp, erule conjE, erule_tac conjE)
 apply(rename_tac Ln Rn M1 M2)
 apply(case_tac M1, simp)
  apply(rule_tac x = Ln in exI, rule_tac x = Rn in exI,
    rule_tac x = 0 in exI)
  apply(case_tac Ln, simp, simp, simp only: exp_ind[THEN sym], simp)
 apply(rule_tac x = Ln in exI, rule_tac x = Rn in exI,
    rule_tac x = M1−1 in exI, rule_tac x = Suc M2 in exI, simp)
 done

lemma mopup_stop_via_jump_over2[simp]:
 ⟦mopup_jump_over2 (2 * n + 6, l, Bk # xs) lm n ires⟧
```

```
 ⟹ mopup_stop (0, Bk # l, xs) lm n ires
 apply(auto simp: mopup_jump_over2.simps mopup_stop.simps tape_of_nat_def)
 apply(simp add: exp_ind[THEN sym])
 done

lemma mopup_jump_over2_nonempty[simp]: mopup_jump_over2 (2 * n + 6, l, []) lm n ires =
False
 by(auto simp: mopup_jump_over2.simps)

declare fetch.simps[simp del]
lemma mod_ex2: (a mod (2::nat) = 0) = (∃ q. a = 2 * q)
 by arith

lemma mod_2: x mod 2  = 0 ∨  x mod 2 = Suc 0
 by arith


lemma mopup_inv_step:
 ⟦n < length lm; mopup_inv (s, l, r) lm n ires⟧
 ⟹ mopup_inv (step (s, l, r) (mopup_a n @ shift mopup_b (2 * n), 0)) lm n ires
 apply(cases r;cases hd r)
    apply(auto split:if_splits simp add:step.simps mopupfetchs fetch.simps(1))
     apply(drule_tac mopup_false2, simp_all add: mopup_bef_erase_b.simps)
   apply(drule_tac mopup_false2, simp_all)
  apply(drule_tac mopup_false2, simp_all)
 by presburger

declare mopup_inv.simps[simp del]
lemma mopup_inv_steps:
 ⟦n < length lm; mopup_inv (s, l, r) lm n ires⟧ ⟹
   mopup_inv (steps (s, l, r) (mopup_a n @ shift mopup_b (2 * n), 0)  stp) lm n ires
proof(induct stp)
 case (Suc stp)
 then show ?case
  by ( cases steps (s, l, r)
          (mopup_a n @ shift mopup_b (2 * n), 0) stp
     , auto simp add: steps.simps intro:mopup_inv_step)
qed (auto simp add: steps.simps)

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

fun abc_mopup_stage2 :: config ⇒ nat ⇒ nat
```

```
  where
   abc_mopup_stage2 (s, l, r) n =
        (if s > 0 ∧ s ≤ 2∗n then length r
         else if s = 2∗n + 1 then length r
         else if s = 2∗n + 5 then length l
         else if s = 2∗n + 6 then length l
         else if s ≥ 2∗n + 2 ∧ s ≤ 2∗n + 4 then length r
         else 0)

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

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

lemma wf_abc_mopup_measure:
  shows wf abc_mopup_measure
  unfolding abc_mopup_measure_def
  by auto

lemma abc_mopup_measure_induct [case_names Step]:
  ⟦⋀n. ¬ P (f n) ⟹ (f (Suc n), (f n)) ∈ abc_mopup_measure⟧ ⟹ ∃ n. P (f n)
  using wf_abc_mopup_measure
  by (metis wf_iff_no_infinite_down_chain)

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
  by(auto simp: mopup_bef_erase_a.simps mopup_bef_erase_b.simps mopup_aft_erase_b.simps)

declare mopup_inv.simps[simp del]

lemma fetch_mopup_a_shift[simp]:
  assumes 0 < q q ≤ n
  shows fetch (mopup_a n @ shift mopup_b (2 ∗ n)) (2∗q) Bk = (R, 2∗q − 1)
proof(cases q)
  case (Suc nat) with assms
  have mopup_a n ! (4 ∗ nat + 2) = mopup_a (Suc nat) ! ((4 ∗ nat) + 2) using assms
    by (metis Suc_le_lessD add_2_eq_Suc′ less_Suc_eq mopup_a_nth numeral_Bit0)
```

```
  then show ?thesis using assms Suc
    by(auto simp: fetch.simps nth_of.simps nth_append)
qed (insert assms,auto)

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 ∃ stp. P (f stp)
proof (induct rule: abc_mopup_measure_induct)
  case (Step na)
  have h: ¬ P (f na) by fact
  show (f (Suc na), f na) ∈ abc_mopup_measure
  proof(simp add: f)
    obtain a b c where g:steps (Suc 0, l, r) (mopup_a n @ shift mopup_b (2 ∗ n), 0) na = (a, b,
c)
      apply(case_tac steps (Suc 0, l, r) (mopup_a n @ shift mopup_b (2 ∗ n), 0) na, auto)
      done
    then have mopup_inv (a, b, c) lm n ires
      using inv less mopup_inv_steps[of n lm Suc 0 l r ires na]
      apply(simp)
      done
    moreover have a > 0
      using h g
      apply(simp add: f P)
      done
    ultimately
   have ((step (a, b, c) (mopup_a n @ shift mopup_b (2 ∗ n), 0), n), (a, b, c), n) ∈ abc_mopup_measure
      apply(case_tac c;cases hd c)
        apply(auto split:if_splits simp add:step.simps mopup_inv.simps mopup_bef_erase_b.simps)
      by (auto split:if_splits simp: mopupfetchs abc_mopup_measure_def )
    thus ((step (steps (Suc 0, l, r) (mopup_a n @ shift mopup_b (2 ∗ n), 0) na)
      (mopup_a n @ shift mopup_b (2 ∗ n), 0), n),
      steps (Suc 0, l, r) (mopup_a n @ shift mopup_b (2 ∗ n), 0) na, n)
      ∈ abc_mopup_measure
      using g by simp
  qed
qed

lemma mopup_inv_start:
  n < length am ⟹ mopup_inv (Suc 0, Bk # Bk # ires, <am> @ Bk ↑ k) am n ires
  apply(cases am;auto simp: mopup_inv.simps mopup_bef_erase_a.simps mopup_jump_over1.simps)
    apply(auto simp: tape_of_nl_cons)
     apply(rule_tac x = Suc (hd am) in exI, rule_tac x = k in exI, simp)
    apply(cases k;cases n;force)
   apply(cases n; force)
  by(cases n; force split:if_splits)
```

lemma *mopup_correct*:
 assumes *less*: *n* < *length* (*am*::*nat list*)
  and *rs*: *am* ! *n* = *rs*
 shows ∃ *stp i j*. (*steps* (*Suc 0*, *Bk* # *Bk* # *ires*, <*am*> @ *Bk* ↑ *k*) (*mopup_a n* @ *shift mopup_b* (*2* ∗ *n*), *0*) *stp*)
  = (*0*, *Bk*↑*i* @ *Bk* # *Bk* # *ires*, *Oc* # *Oc*↑ *rs* @ *Bk*↑*j*)
 using *less*
proof −
 have *a*: *mopup_inv* (*Suc 0*, *Bk* # *Bk* # *ires*, <*am*> @ *Bk* ↑ *k*) *am n ires*
  using *less*
  apply(*simp add*: *mopup_inv_start*)
  done
 then have ∃ *stp*. *is_final* (*steps* (*Suc 0*, *Bk* # *Bk* # *ires*, <*am*> @ *Bk* ↑ *k*) (*mopup_a n* @ *shift mopup_b* (*2* ∗ *n*), *0*) *stp*)
  using *less mopup_halt*[*of n am Bk* # *Bk* # *ires* <*am*> @ *Bk* ↑ *k ires*
    (λ*stp*. (*steps* (*Suc 0*, *Bk* # *Bk* # *ires*, <*am*> @ *Bk* ↑ *k*) (*mopup_a n* @ *shift mopup_b* (*2* ∗ *n*), *0*) *stp*, *n*))
    (λ(*c*, *n*). *is_final c*)]
  apply(*simp*)
  done
 from *this* **obtain** *stp* **where** *b*:
  is_final (*steps* (*Suc 0*, *Bk* # *Bk* # *ires*, <*am*> @ *Bk* ↑ *k*) (*mopup_a n* @ *shift mopup_b* (*2* ∗ *n*), *0*) *stp*) **..**
 from *a b* **have**
  mopup_inv (*steps* (*Suc 0*, *Bk* # *Bk* # *ires*, <*am*> @ *Bk* ↑ *k*) (*mopup_a n* @ *shift mopup_b* (*2* ∗ *n*), *0*) *stp*)
  am n ires
  apply(*rule_tac mopup_inv_steps*, *simp_all add*: *less*)
  done
 from *b* **and** *this* **show** *?thesis*
  apply(*rule_tac x* = *stp* **in** *exI*, *simp*)
  apply(*case_tac steps* (*Suc 0*, *Bk* # *Bk* # *ires*, <*am*> @ *Bk* ↑ *k*)
   (*mopup_a n* @ *shift mopup_b* (*2* ∗ *n*), *0*) *stp*)
  apply(*simp add*: *mopup_inv.simps mopup_stop.simps rs*)
  using *rs*
  apply(*simp add*: *tape_of_nat_def*)
  done
qed

lemma *wf_mopup*[*intro*]: *tm_wf* (*mopup n*, *0*)
 by(*induct n*, *auto simp add*: *shift.simps mopup_b_def tm_wf.simps*)

end

8 Abacus Machines

theory *Abacus*
 imports *Turing_Hoare Abacus_Mopup*
begin

declare *replicate_Suc*[*simp add*]

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

type-synonym *abc_prog* = *abc_inst list*

type-synonym *abc_state* = *nat*

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

type-synonym *abc_lm* = *nat list*

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

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

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

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

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

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

Fetch instruction out of Abacus program:

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

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

```
fun abc_step_l :: abc_conf ⇒ abc_inst option ⇒ abc_conf
  where
    abc_step_l (s, lm) a = (case a of
          None ⇒ (s, lm) |
          Some (Inc n)  ⇒ (let nv = abc_lm_v lm n in
```

$(s + 1, abc\_lm\_s\ lm\ n\ (nv + 1)))\ |$
$Some\ (Dec\ n\ e) \Rightarrow (let\ nv = abc\_lm\_v\ lm\ n\ in$
$if\ (nv = 0)\ then\ (e, abc\_lm\_s\ lm\ n\ 0)$
$else\ (s + 1,\ abc\_lm\_s\ lm\ n\ (nv - 1)))\ |$
$Some\ (Goto\ n) \Rightarrow (n, lm)$
$)$

Multi-step execution of Abacus machine.

fun *abc_steps_l* :: *abc_conf* ⇒ *abc_prog* ⇒ *nat* ⇒ *abc_conf*
where
abc_steps_l (*s*, *lm*) *p* *0* = (*s*, *lm*) |
abc_steps_l (*s*, *lm*) *p* (*Suc n*) =
abc_steps_l (*abc_step_l* (*s*, *lm*) (*abc_fetch s p*)) *p n*

9 Compiling Abacus machines into Turing machines

9.1 Compiling functions

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

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

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

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

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

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

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

definition *tdec_b* :: *instr list*
where
tdec_b $\stackrel{def}{=}$ [($W1$, 1), (R, 2), (L, 14), (R, 3), (L, 4), (R, 3),

$(R, 5), (W0, 4), (R, 6), (W0, 5), (L, 7), (L, 8),$
$(L, 11), (W0, 7), (W1, 8), (R, 9), (L, 10), (R, 9),$
$(R, 5), (W0, 10), (L, 12), (L, 11), (R, 13), (L, 11),$
$(R, 17), (W0, 13), (L, 15), (L, 14), (R, 16), (L, 14),$
$(R, 0), (W0, 16)]$

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

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

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

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

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

type-synonym *layout* = *nat list*

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

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

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

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

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

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

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

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

| *ci ly ss* (*Goto n*) = *tgoto* (*start_of ly n*)

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

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

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

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

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

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

lemma *length_findnth*:
length (*findnth n*) = *4* ∗ *n*
by (*induct n*, *auto*)

lemma *ci_length* : *length* (*ci ns n ai*) *div 2* = *length_of ai*
apply(*auto simp*: *ci*.*simps tinc_b_def tdec_b_def length_findnth*
split: *abc_inst*.*splits simp del*: *adjust*.*simps*)
done

9.2 Representation of Abacus memory by TM tapes

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

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

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

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

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

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

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

```
declare abc_step_l.simps[simp del] abc_steps_l.simps[simp del]
lemma start_of_nonzero[simp]: start_of ly as > 0 (start_of ly as = 0) = False
  apply(auto simp: start_of.simps)
 done

lemma abc_steps_l_0: abc_steps_l ac ap 0 = ac
 by(cases ac, simp add: abc_steps_l.simps)

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(induct stp arbitrary: as am bs bm)
 case 0
 thus ?case
  by(simp add: abc_steps_l.simps abc_steps_l_0)
next
 case (Suc stp as am bs bm)
 have ind: ⋀as am bs bm. 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)
  by fact
 have h: abc_steps_l (as, am) ap (Suc stp) = (bs, bm) by fact
 obtain as' am' where g: abc_step_l (as, am) (abc_fetch as ap) = (as', am')
  by(cases abc_step_l (as, am) (abc_fetch as ap), auto)
 then have abc_steps_l (as', am') ap (Suc stp) = abc_step_l (bs, bm) (abc_fetch bs ap)
  using h
  by(intro ind, simp add: abc_steps_l.simps)
 thus ?case
  using g
  by(simp add: abc_steps_l.simps)
qed

lemma tm_shift_fetch:
 ⟦fetch A s b = (ac, ns); ns ≠ 0 ⟧
 ⟹ fetch (shift A off) s b = (ac, ns + off)
 apply(cases b;cases s)
   apply(auto simp: fetch.simps shift.simps)
 done

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')
 using assms
 apply(simp add: step.simps)
 apply(cases fetch A s (read r), auto)
  apply(drule_tac [!] off = off in tm_shift_fetch, simp_all)
 done
```

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

lemma tm_shift_eq_steps:
 assumes exec: steps (s, l, r) (A, 0) stp = (s', l', r')
  and notfinal: s' ≠ 0
 shows steps (s + off, l, r) (shift A off, off) stp = (s' + off, l', r')
 using exec notfinal
proof(induct stp arbitrary: s' l' r', simp add: steps.simps)
 fix stp s' l' r'
 assume ind: ⋀s' l' r'. ⟦steps (s, l, r) (A, 0) stp = (s', l', r'); s' ≠ 0⟧
   ⟹ steps (s + off, l, r) (shift A off, off) stp = (s' + off, l', r')
  and h:  steps (s, l, r) (A, 0) (Suc stp) = (s', l', r') s' ≠ 0
 obtain s1 l1 r1 where g: steps (s, l, r) (A, 0) stp = (s1, l1, r1)
  apply(cases steps (s, l, r) (A, 0) stp) by blast
 moreover then have s1 ≠ 0
  using h
  apply(simp add: step_red)
  apply(cases 0 < s1, auto)
  done
 ultimately have steps (s + off, l, r) (shift A off, off) stp =
             (s1 + off, l1, r1)
  apply(intro ind, simp_all)
  done
 thus steps (s + off, l, r) (shift A off, off) (Suc stp) = (s' + off, l', r')
  using h g assms
  apply(simp add: step_red)
  apply(intro tm_shift_eq_step, auto)
  done
qed

lemma startof_ge1[simp]: Suc 0 ≤ start_of ly as
 apply(simp add: start_of.simps)
 done

lemma start_of_Suc1: ⟦ly = layout_of ap;
     abc_fetch as ap = Some (Inc n)⟧
     ⟹ start_of ly (Suc as) = start_of ly as + 2 ∗ n + 9
 apply(auto simp: start_of.simps layout_of.simps
    length_of.simps abc_fetch.simps
    take_Suc_conv_app_nth split: if_splits)
 done

lemma start_of_Suc2:
 ⟦ly = layout_of ap;
 abc_fetch as ap = Some (Dec n e)⟧ ⟹
      start_of ly (Suc as) =
         start_of ly as + 2 ∗ n + 16
 apply(auto simp: start_of.simps layout_of.simps
    length_of.simps abc_fetch.simps
```

```
    take_Suc_conv_app_nth split: if_splits)
  done

lemma start_of_Suc3:
  ⟦ly = layout_of ap;
  abc_fetch as ap = Some (Goto n)⟧ ⟹
  start_of ly (Suc as) = start_of ly as + 1
  apply(auto simp: start_of.simps layout_of.simps
    length_of.simps abc_fetch.simps
    take_Suc_conv_app_nth split: if_splits)
  done

lemma length_ci_inc:
  length (ci ly ss (Inc n)) = 4*n + 18
  apply(auto simp: ci.simps length_findnth tinc_b_def)
  done

lemma length_ci_dec:
  length (ci ly ss (Dec n e)) = 4*n + 32
  apply(auto simp: ci.simps length_findnth tdec_b_def)
  done

lemma length_ci_goto:
  length (ci ly ss (Goto n )) = 2
  apply(auto simp: ci.simps length_findnth tdec_b_def)
  done

lemma take_Suc_last[elim]: Suc as ≤ length xs ⟹
      take (Suc as) xs = take as xs @ [xs ! as]
proof(induct xs arbitrary: as)
  case (Cons a xs)
  then show ?case by ( simp, cases as;simp)
qed simp

lemma concat_suc: Suc as ≤ length xs ⟹
    concat (take (Suc as) xs) = concat (take as xs) @ xs! as
  apply(subgoal_tac take (Suc as) xs = take as xs @ [xs ! as], simp)
  by auto

lemma concat_drop_suc_iff:
  Suc n < length tps ⟹ concat (drop (Suc n) tps) =
      tps ! Suc n @ concat (drop (Suc (Suc n)) tps)
proof(induct tps arbitrary: n)
  case (Cons a tps)
  then show ?case
    apply(cases tps, simp, simp)
    apply(cases n, simp, simp)
    done
qed simp
```

```
declare append_assoc[simp del]

lemma tm_append:
  ⟦n < length tps; tp = tps ! n⟧ ⟹
  ∃ tp1 tp2. concat tps = tp1 @ tp @ tp2 ∧ tp1 =
  concat (take n tps) ∧ tp2 = concat (drop (Suc n) tps)
  apply(rule_tac x = concat (take n tps) in exI)
  apply(rule_tac x = concat (drop (Suc n) tps) in exI)
  apply(auto)
proof(induct n)
  case 0
  then show ?case by(cases tps; simp)
next
  case (Suc n)
  then show ?case
    apply(subgoal_tac concat (take n tps) @ (tps ! n) =
              concat (take (Suc n) tps))
     apply(simp only: append_assoc[THEN sym], simp only: append_assoc)
     apply(subgoal_tac  concat (drop (Suc n) tps) = tps ! Suc n @
                concat (drop (Suc (Suc n)) tps))
      apply (metis append_take_drop_id concat_append)
     apply(rule concat_drop_suc_iff,force)
    by (simp add: concat_suc)
qed

declare append_assoc[simp]

lemma length_tms_of[simp]: length (tms_of aprog) = length aprog
  apply(auto simp: tms_of.simps tpairs_of.simps)
  done

lemma ci_nth:
  ⟦ly = layout_of aprog;
  abc_fetch as aprog = Some ins⟧
  ⟹ ci ly (start_of ly as) ins = tms_of aprog ! as
  apply(simp add: tms_of.simps tpairs_of.simps
     abc_fetch.simps del: map_append split: if_splits)
  done

lemma t_split:⟦
       ly = layout_of aprog;
       abc_fetch as aprog = Some ins⟧
     ⟹ ∃ tp1 tp2. concat (tms_of aprog) =
          tp1 @ (ci ly (start_of ly as) ins) @ tp2
          ∧ tp1 = concat (take as (tms_of aprog)) ∧
           tp2 = concat (drop (Suc as) (tms_of aprog))
  apply(insert tm_append[of as tms_of aprog
       ci ly (start_of ly as) ins], simp)
  apply(subgoal_tac ci ly (start_of ly as) ins = (tms_of aprog) ! as)
   apply(subgoal_tac length (tms_of aprog) = length aprog)
```

```
  apply(simp add: abc_fetch.simps split: if_splits, simp)
 apply(intro ci_nth, auto)
 done

lemma div_apart: ⟦x mod (2::nat) = 0; y mod 2 = 0⟧
     ⟹ (x + y) div 2 = x div 2 + y div 2
 by(auto)

lemma length_layout_of[simp]: length (layout_of aprog) = length aprog
 by(auto simp: layout_of.simps)

lemma length_tms_of_elem_even[intro]:  n < length ap ⟹ length (tms_of ap ! n) mod 2 = 0
 apply(cases ap ! n)
 by (auto simp: tms_of.simps tpairs_of.simps ci.simps length_findnth tinc_b_def tdec_b_def)

lemma compile_mod2: length (concat (take n (tms_of ap))) mod 2 = 0
proof(induct n)
 case 0
 then show ?case by (auto simp add: take_Suc_conv_app_nth)
next
 case (Suc n)
 hence n < length (tms_of ap) ⟹ is_even (length (concat (take (Suc n) (tms_of ap))))
  unfolding take_Suc_conv_app_nth by fastforce
 with Suc show ?case by(cases n < length (tms_of ap), auto)
qed

lemma tpa_states:
 ⟦tp = concat (take as (tms_of ap));
 as ≤ length ap⟧ ⟹
 start_of (layout_of ap) as = Suc (length tp div 2)
proof(induct as arbitrary: tp)
 case 0
 thus ?case
  by(simp add: start_of.simps)
next
 case (Suc as tp)
 have ind: ⋀tp. ⟦tp = concat (take as (tms_of ap)); as ≤ length ap⟧ ⟹
  start_of (layout_of ap) as = Suc (length tp div 2) by fact
 have tp: tp = concat (take (Suc as) (tms_of ap)) by fact
 have le: Suc as ≤ length ap by fact
 have a: start_of (layout_of ap) as = Suc (length (concat (take as (tms_of ap))) div 2)
  using le
  by(intro ind, simp_all)
 from a tp le show ?case
  apply(simp add: start_of.simps take_Suc_conv_app_nth)
  apply(subgoal_tac length (concat (take as (tms_of ap))) mod 2= 0)
   apply(subgoal_tac  length (tms_of ap ! as) mod 2 = 0)
    apply(simp add: Abacus.div_apart)
    apply(simp add: layout_of.simps ci_length  tms_of.simps tpairs_of.simps)
   apply(auto  intro: compile_mod2)
```

```
  done
qed

declare fetch.simps[simp]
lemma append_append_fetch:
 ⟦length tp1 mod 2 = 0; length tp mod 2 = 0;
    length tp1 div 2 < a ∧ a ≤ length tp1 div 2 + length tp div 2⟧
   ⟹fetch (tp1 @ tp @ tp2) a b = fetch tp (a − length tp1 div 2) b
 apply(subgoal_tac ∃ x. a = length tp1 div 2 + x, erule exE)
  apply(rename_tac x)
  apply(case_tac x, simp)
  apply(subgoal_tac length tp1 div 2 + Suc nat =
          Suc (length tp1 div 2 + nat))
   apply(simp only: fetch.simps nth_of.simps, auto)
  apply(cases b, simp)
   apply(subgoal_tac 2 ∗ (length tp1 div 2) = length tp1, simp)
    apply(subgoal_tac 2 ∗ nat < length tp, simp add: nth_append, simp)
   apply(subgoal_tac 2 ∗ (length tp1 div 2) = length tp1, simp)
   apply(subgoal_tac 2 ∗ nat < length tp, simp add: nth_append, auto)
  apply(auto simp: nth_append)
 apply(rule_tac x = a − length tp1 div 2 in exI, simp)
 done

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 ≥ start_of ly as
  and range2: s < start_of ly (Suc as)
 shows fetch tp s b = fetch (ci ly (start_of ly as) ins)
      (Suc s − start_of ly as) b
proof −
 have ∃tp1 tp2. concat (tms_of ap) = tp1 @ ci ly (start_of ly as) ins @ tp2 ∧
  tp1 = concat (take as (tms_of ap)) ∧ tp2 = concat (drop (Suc as) (tms_of ap))
  using assms
  by(intro t_split, simp_all)
 then obtain tp1 tp2 where a: concat (tms_of ap) = tp1 @ ci ly (start_of ly as) ins @ tp2 ∧
  tp1 = concat (take as (tms_of ap)) ∧ tp2 = concat (drop (Suc as) (tms_of ap)) by blast
 then have b: start_of (layout_of ap) as = Suc (length tp1 div 2)
  using fetch
  by(intro tpa_states, simp, simp add: abc_fetch.simps split: if_splits)
 have fetch (tp1 @ (ci ly (start_of ly as) ins) @ tp2)  s b =
      fetch (ci ly (start_of ly as) ins) (s − length tp1 div 2) b
 proof(intro append_append_fetch)
  show length tp1 mod 2 = 0
   using a
   by(auto, rule_tac compile_mod2)
 next
  show length (ci ly (start_of ly as) ins) mod 2 = 0
   by(cases ins, auto simp: ci.simps length_findnth tinc_b_def tdec_b_def)
```

```
  next
    show length tp1 div 2 < s ∧ s ≤
      length tp1 div 2 + length (ci ly (start_of ly as) ins) div 2
    proof -
      have length (ci ly (start_of ly as) ins) div 2 = length_of ins
        using ci_length by simp
      moreover have start_of ly (Suc as) = start_of ly as + length_of ins
        using fetch layout
        apply(simp add: start_of.simps abc_fetch.simps List.take_Suc_conv_app_nth
            split: if_splits)
        apply(simp add: layout_of.simps)
        done
      ultimately show ?thesis
        using b layout range1 range2
        apply(simp)
        done
    qed
  qed
  thus ?thesis
    using b layout a compile
    apply(simp add: tm_of.simps)
    done
qed

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 ≠ 0
  shows fetch tp s b = (ac, ns)
proof -
  have s ≥ start_of ly as
  proof(cases s ≥ start_of ly as)
    case True thus ?thesis by simp
  next
    case False
    have ¬ start_of ly as ≤ s by fact
    then have Suc s − start_of ly as = 0
      by arith
    then have fetch (ci ly (start_of ly as) ins)
       (Suc s − start_of ly as) b = (Nop, 0)
      by(simp add: fetch.simps)
    with notfinal fetch show ?thesis
      by(simp)
  qed
  moreover have s < start_of ly (Suc as)
  proof(cases s < start_of ly (Suc as))
    case True thus ?thesis by simp
```

next
case *False*
have *h*: $\neg$ *s* < *start_of ly* (*Suc as*)
by *fact*
then have *s* > *start_of ly as*
using *abc_fetch layout*
apply(*simp add*: *start_of*.*simps abc_fetch*.*simps split*: *if_splits*)
apply(*simp add*: *List*.*take_Suc_conv_app_nth*, *auto*)
apply(*subgoal_tac layout_of ap* ! *as* > *0*)
apply *arith*
apply(*simp add*: *layout_of*.*simps*)
apply(*cases ap*!*as*, *auto simp*: *length_of*.*simps*)
done
from *this* **and** *h* **have** *fetch* (*ci ly* (*start_of ly as*) *ins*) (*Suc s* − *start_of ly as*) *b* = (*Nop*, *0*)
using *abc_fetch layout*
apply(*cases b*;*cases ins*)
apply(*simp_all add*:*Suc_diff_le start_of_Suc2 start_of_Suc1 start_of_Suc3*)
by (*simp_all only*: *length_ci_inc length_ci_dec length_ci_goto*, *auto*)
from *fetch* **and** *notfinal this* **show** *?thesis***by** *simp*
qed
ultimately show *?thesis*
using *assms*
by(*drule_tac b*= *b* **and** *ins* = *ins* **in** *step_eq_fetch′*, *auto*)
qed

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′* ≠ *0*
shows *step* (*s*, *l*, *r*) (*tp*, *0*) = (*s′*, *l′*, *r′*)
using *assms*
apply(*simp add*: *step*.*simps*)
apply(*cases fetch* (*ci* (*layout_of ap*) (*start_of* (*layout_of ap*) *as*) *ins*)
(*Suc s* − *start_of* (*layout_of ap*) *as*) (*read r*), *simp*)
using *layout*
apply(*drule_tac s* = *s* **and** *b* = *read r* **and** *ac* = *a* **in** *step_eq_fetch*, *auto*)
done

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

```
  shows steps (s, l, r) (tp, 0) stp = (s′, l′, r′)
  using exec notfinal
proof(induct stp arbitrary: s′ l′ r′, simp add: steps.simps)
  fix stp s′ l′ r′
  assume ind:
    ⋀s′ l′ r′. ⟦steps (s, l, r) (ci ly (start_of ly as) ins, start_of ly as − 1) stp = (s′, l′, r′); s′ ≠ 0⟧
          ⟹ steps (s, l, r) (tp, 0) stp = (s′, l′, r′)
    and h: steps (s, l, r) (ci ly (start_of ly as) ins, start_of ly as − 1) (Suc stp) = (s′, l′, r′) s′ ≠ 0
  obtain s1 l1 r1 where g: steps (s, l, r) (ci ly (start_of ly as) ins, start_of ly as − 1) stp =
                    (s1, l1, r1)
    apply(cases steps (s, l, r) (ci ly (start_of ly as) ins, start_of ly as − 1) stp) by blast
  moreover hence s1 ≠ 0
    using h
    apply(simp add: step_red)
    apply(cases 0 < s1, simp_all)
    done
  ultimately have steps (s, l, r) (tp, 0) stp = (s1, l1, r1)
    apply(rule_tac ind, auto)
    done
  thus steps (s, l, r) (tp, 0) (Suc stp) = (s′, l′, r′)
    using h g assms
    apply(simp add: step_red)
    apply(rule_tac step_eq_in, auto)
    done
qed

lemma tm_append_fetch_first:
  ⟦fetch A s b = (ac, ns); ns ≠ 0⟧ ⟹
    fetch (A @ B) s b = (ac, ns)
  by(cases b;cases s;force simp: fetch.simps nth_append split: if_splits)

lemma tm_append_first_step_eq:
  assumes step (s, l, r) (A, off) = (s′, l′, r′)
    and s′ ≠ 0
  shows step (s, l, r) (A @ B, off) = (s′, l′, r′)
  using assms
  apply(simp add: step.simps)
  apply(cases fetch A (s − off) (read r))
  apply(frule_tac  B = B and b = read r in tm_append_fetch_first, auto)
  done

lemma tm_append_first_steps_eq:
  assumes steps (s, l, r) (A, off) stp = (s′, l′, r′)
    and s′ ≠ 0
  shows steps (s, l, r) (A @ B, off) stp = (s′, l′, r′)
  using assms
proof(induct stp arbitrary: s′ l′ r′, simp add: steps.simps)
  fix stp s′ l′ r′
  assume ind: ⋀s′ l′ r′. ⟦steps (s, l, r) (A, off) stp = (s′, l′, r′); s′ ≠ 0⟧
    ⟹ steps (s, l, r) (A @ B, off) stp = (s′, l′, r′)
```

```
  and h: steps (s, l, r) (A, off) (Suc stp) = (s', l', r') s' ≠ 0
 obtain sa la ra where a: steps (s, l, r) (A, off) stp = (sa, la, ra)
  apply(cases steps (s, l, r) (A, off) stp) by blast
 hence steps (s, l, r) (A @ B, off) stp = (sa, la, ra) ∧ sa ≠ 0
  using h ind[of sa la ra]
  apply(cases sa, simp_all)
  done
 thus steps (s, l, r) (A @ B, off) (Suc stp) = (s', l', r')
  using h a
  apply(simp add: step_red)
  apply(intro tm_append_first_step_eq, simp_all)
  done
qed

lemma tm_append_second_fetch_eq:
 assumes
  even: length A mod 2 = 0
  and off: off = length A div 2
  and fetch: fetch B s b = (ac, ns)
  and notfinal: ns ≠ 0
 shows fetch (A @ shift B off) (s + off) b = (ac, ns + off)
 using assms
 by(cases b;cases s,auto simp: nth_append shift.simps split: if_splits)

lemma tm_append_second_step_eq:
 assumes
  exec: step0 (s, l, r) B = (s', l', r')
  and notfinal: s' ≠ 0
  and off: off = length A div 2
  and even: length A mod 2 = 0
 shows step0 (s + off, l, r) (A @ shift B off) = (s' + off, l', r')
 using assms
 apply(simp add: step.simps)
 apply(cases fetch B s (read r))
 apply(frule_tac tm_append_second_fetch_eq, simp_all, auto)
 done


lemma tm_append_second_steps_eq:
 assumes
  exec: steps (s, l, r) (B, 0) stp = (s', l', r')
  and notfinal: s' ≠ 0
  and off: off = length A div 2
  and even: length A mod 2 = 0
 shows steps (s + off, l, r) (A @ shift B off, 0) stp = (s' + off, l', r')
 using exec notfinal
proof(induct stp arbitrary: s' l' r')
 case 0
 thus steps0 (s + off, l, r) (A @ shift B off) 0 = (s' + off, l', r')
  by(simp add: steps.simps)
```

```
next
  case (Suc stp s' l' r')
  have ind: ⋀s' l' r'. ⟦steps0 (s, l, r) B stp = (s', l', r'); s' ≠ 0⟧ ⟹
    steps0 (s + off, l, r) (A @ shift B off) stp = (s' + off, l', r')
    by fact
  have h: steps0 (s, l, r) B (Suc stp) = (s', l', r') by fact
  have k: s' ≠ 0 by fact
  obtain s'' l'' r'' where a: steps0 (s, l, r) B stp = (s'', l'', r'')
    by (metis prod_cases3)
  then have b: s'' ≠ 0
    using h k
    by(intro notI, auto)
  from a b have c: steps0 (s + off, l, r) (A @ shift B off) stp = (s'' + off, l'', r'')
    by(erule_tac ind, simp)
  from c b h a k assms show ?case
    by(auto intro:tm_append_second_step_eq)
qed

lemma tm_append_second_fetch0_eq:
  assumes
    even: length A mod 2 = 0
    and off: off = length A div 2
    and fetch: fetch B s b = (ac, 0)
    and notfinal: s ≠ 0
  shows fetch (A @ shift B off) (s + off) b = (ac, 0)
  using assms
  apply(cases b;cases s)
   apply(auto simp: fetch.simps nth_append shift.simps split: if_splits)
  done

lemma tm_append_second_halt_eq:
  assumes
    exec: steps (Suc 0, l, r) (B, 0) stp = (0, l', r')
    and wf_B: tm_wf (B, 0)
    and off: off = length A div 2
    and even: length A mod 2 = 0
  shows steps (Suc off, l, r) (A @ shift B off, 0) stp = (0, l', r')
proof -
  have ∃n. ¬ is_final (steps0 (1, l, r) B n) ∧ steps0 (1, l, r) B (Suc n) = (0, l', r')
    using exec by(rule_tac before_final, simp)
  then obtain n where a:
    ¬ is_final (steps0 (1, l, r) B n) ∧ steps0 (1, l, r) B (Suc n) = (0, l', r') ..
  obtain s'' l'' r'' where b: steps0 (1, l, r) B n = (s'', l'', r'') ∧ s'' >0
    using a
    by(cases steps0 (1, l, r) B n, auto)
  have c: steps (Suc 0 + off, l, r) (A @ shift B off, 0) n = (s'' + off, l'', r'')
    using a b assms
    by(rule_tac tm_append_second_steps_eq, simp_all)
  obtain ac where d: fetch B s'' (read r'') = (ac, 0)
    using  b a
```

```
    by(cases fetch B s'' (read r''), auto simp: step_red step.simps)
  then have fetch (A @ shift B off) (s'' + off) (read r'') = (ac, 0)
    using assms b
    by(rule_tac tm_append_second_fetch0_eq, simp_all)
  then have e: steps (Suc 0 + off, l, r) (A @ shift B off, 0) (Suc n) = (0, l', r')
    using a b assms c d
    by(simp add: step_red step.simps)
  from a have n < stp
    using exec
  proof(cases n < stp)
    case True thus ?thesis by simp
  next
    case False
    have ¬ n < stp by fact
    then obtain d where n = stp + d
      by (metis add.comm_neutral less_imp_add_positive nat_neq_iff)
    thus ?thesis
      using a e exec
      by(simp)
  qed
  then obtain d where stp = Suc n + d
    by(metis add_Suc less_iff_Suc_add)
  thus ?thesis
    using e
    by(simp only: steps_add, simp)
qed

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 -
  have steps (s, l, r) (A@shift B off, 0) stpa = (Suc (length A div 2), la, ra)
    apply(intro tm_append_first_steps_eq)
     apply(auto simp: assms)
    done
  moreover have steps (1 + off, la, ra) (A @ shift B off, 0) stpb = (sb + off, lb, rb)
    apply(intro tm_append_second_steps_eq)
      apply(auto simp: assms bexec)
    done
  ultimately show steps (s, l, r) (A @ shift B off, 0) (stpa + stpb) = (sb + off, lb, rb)
    apply(simp add: steps_add off)
    done
qed
```

9.3 Crsp of Inc

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

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

fun *at_begin_norm* :: *inc_inv_t*
where
$at\_begin\_norm\ (as, lm)\ (s, l, r)\ ires=$
$(\exists\ lm1\ lm2\ rn.\ lm = lm1\ @\ lm2 \wedge length\ lm1 = s \wedge$
$(if\ lm1 = []\ then\ l = Bk\ \#\ Bk\ \#\ ires$
$else\ l = Bk\ \#\ <rev\ lm1>\ @\ Bk\ \#\ Bk\ \#\ ires\ ) \wedge r = <lm2>@Bk\uparrow rn)$

fun *in_middle* :: *inc_inv_t*
where
$in\_middle\ (as, lm)\ (s, l, r)\ ires =$
$(\exists\ lm1\ lm2\ tn\ m\ ml\ mr\ rn.\ lm\ @\ 0\uparrow tn = lm1\ @\ [m]\ @\ lm2$
$\wedge\ length\ lm1 = s \wedge m + 1 = ml + mr \wedge$
$ml \neq 0 \wedge tn = s + 1 - length\ lm \wedge$
$(if\ lm1 = []\ then\ l = Oc\uparrow ml\ @\ Bk\ \#\ Bk\ \#\ ires$
$else\ l = Oc\uparrow ml@[Bk]@<rev\ lm1>@$
$Bk\ \#\ Bk\ \#\ ires) \wedge (r = Oc\uparrow mr\ @\ [Bk]\ @\ <lm2>@\ Bk\uparrow rn\ \vee$
$(lm2 = [] \wedge r = Oc\uparrow mr))$
$)$

fun *inv_locate_a* :: *inc_inv_t*
where $inv\_locate\_a\ (as, lm)\ (s, l, r)\ ires =$
$(at\_begin\_norm\ (as, lm)\ (s, l, r)\ ires\ \vee$
$at\_begin\_fst\_bwtn\ (as, lm)\ (s, l, r)\ ires\ \vee$
$at\_begin\_fst\_awtn\ (as, lm)\ (s, l, r)\ ires$
$)$

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 $inv\_after\_write\ (as, lm)\ (s, l, r)\ ires =$
$(\exists\ rn\ m\ lm1\ lm2.\ lm = lm1\ @\ m\ \#\ lm2\ \wedge$

```
    (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)
 apply(intro exCI, simp)
 apply(subgoal_tac ∀ n. Q (f n))
  apply(drule_tac f = f in wf_inv_image)
  apply(erule wf_induct)
  apply(auto)
 apply(rename_tac n,induct_tac n; simp)
 done

lemma halt_lemma2′′:
 ⟦P (f n); ¬ P (f (0::nat))⟧ ⟹
     ∃ n. (P (f n) ∧ (∀ i < n. ¬ P (f i)))
 apply(induct n rule: nat_less_induct, auto)
 done

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)
 apply(induct na, simp, simp)
 done

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)
```

```
apply(insert halt_lemma2' [of LE P f Q], simp, erule_tac exE)
apply(subgoal_tac ∃ n. (P (f n) ∧ (∀ i < n. ¬ P (f i))))
 apply(erule_tac exE)+
 apply(rename_tac n na)
 apply(rule_tac x = na in exI, auto)
 apply(rule halt_lemma2''', simp, simp, simp)
apply(erule_tac halt_lemma2'', simp)
done
```

```
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
 by(auto simp: findnth_LE_def lex_pair_def)

declare findnth_inv.simps[simp del]
```

lemma *x_is_2n_arith*[*simp*]:
 $[\![x < Suc\ (Suc\ (2 * n)); Suc\ x\ mod\ 2 = Suc\ 0; \neg\ x < 2 * n]\!]$
 $\Longrightarrow x = 2{*}n$
 by *arith*

lemma *between_sucs*:$x < Suc\ n \Longrightarrow \neg\ x < n \Longrightarrow x = n$ **by** *auto*

lemma *fetch_findnth*[*simp*]:
 $[\![0 < a; a < Suc\ (2 * n); a\ mod\ 2 = Suc\ 0]\!] \Longrightarrow fetch\ (findnth\ n)\ a\ Oc = (R, Suc\ a)$
 $[\![0 < a; a < Suc\ (2 * n); a\ mod\ 2 \neq Suc\ 0]\!] \Longrightarrow fetch\ (findnth\ n)\ a\ Oc = (R, a)$
 $[\![0 < a; a < Suc\ (2 * n); a\ mod\ 2 \neq Suc\ 0]\!] \Longrightarrow fetch\ (findnth\ n)\ a\ Bk = (R, Suc\ a)$
 $[\![0 < a; a < Suc\ (2 * n); a\ mod\ 2 = Suc\ 0]\!] \Longrightarrow fetch\ (findnth\ n)\ a\ Bk = (W1, a)$
 by(*cases a*;*induct n*;*force simp*: *length_findnth nth_append dest*!:*between_sucs*)+

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

lemma *replicate_once*[*intro*]: $\exists\, rn.\ [Bk] = Bk \uparrow rn$
 by (*metis replicate.simps*)

lemma *at_begin_norm_Bk*[*intro*]: *at_begin_norm* (*as*, *am*) (*q*, *aaa*, []) *ires*
 $\Longrightarrow$ *at_begin_norm* (*as*, *am*) (*q*, *aaa*, [*Bk*]) *ires*
 apply(*simp add*: *at_begin_norm.simps*)
 by *fastforce*

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*
 apply(*simp only*: *at_begin_fst_bwtn.simps*)
 using *replicate_once* **by** *blast*

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*
 apply(*auto simp*: *at_begin_fst_awtn.simps*)
 done

```
lemma inv_locate_a_Bk[intro]: inv_locate_a (as, am) (q, aaa, []) ires
        ⟹ inv_locate_a (as, am) (q, aaa, [Bk]) ires
 apply(simp only: inv_locate_a.simps)
 apply(erule disj_forward)
  defer
  apply(erule disj_forward, auto)
 done

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
 apply(simp only: inv_locate_a.simps at_begin_norm.simps
    at_begin_fst_bwtn.simps at_begin_fst_awtn.simps)
 apply(erule_tac disjE, erule exE, erule exE, erule exE,
    rule disjI2, rule disjI2)
  defer
  apply(erule_tac disjE, erule exE, erule exE,
    erule exE, rule disjI2, rule disjI2)
   prefer 2
   apply(simp)
proof−
 fix lm1 tn rn
 assume k: lm1 = am @ 0↑tn ∧ length lm1 = q ∧ (if lm1 = [] then aaa = Bk # Bk #
   ires else aaa = [Bk] @ <rev lm1> @ Bk # Bk # ires) ∧ Bk # xs = Bk↑rn
 thus ∃ lm1 tn rn. lm1 = am @ 0 ↑ tn ∧ length lm1 = q ∧
   (if lm1 = [] then aaa = Bk # Bk # ires else aaa = [Bk] @ <rev lm1> @ Bk # Bk # ires) ∧
Oc # xs = [Oc] @ Bk ↑ rn
   (is ∃ lm1 tn rn. ?P lm1 tn rn)
 proof −
  from k have ?P lm1 tn (rn − 1)
    by (auto simp: Cons_replicate_eq)
  thus ?thesis by blast
 qed
next
 fix lm1 lm2 rn
 assume h1: am = lm1 @ lm2 ∧ length lm1 = q ∧ (if lm1 = []
   then aaa = Bk # Bk # ires else aaa = Bk # <rev lm1> @ Bk # Bk # ires) ∧
   Bk # xs = <lm2> @ Bk↑rn
 from h1 have h2: lm2 = []
   apply(auto split: if_splits;cases lm2;simp add: tape_of_nl_cons split: if_splits)
   done
 from h1 and h2 show ∃ lm1 tn rn. lm1 = am @ 0↑tn ∧ length lm1 = q ∧
   (if lm1 = [] then aaa = Bk # Bk # ires else aaa = [Bk] @ <rev lm1> @ Bk # Bk # ires) ∧
   Oc # xs = [Oc] @ Bk↑rn
   (is ∃ lm1 tn rn. ?P lm1 tn rn)
 proof −
  from h1 and h2  have ?P lm1 0 (rn − 1)
    apply(auto simp:tape_of_nat_def)
    by(cases rn, simp, simp)
  thus ?thesis by blast
 qed
```

```
qed

lemma inv_locate_a[simp]: inv_locate_a (as, am) (q, aaa, []) ires ⟹
          inv_locate_a (as, am) (q, aaa, [Oc]) ires
 apply(insert locate_a_2_locate_a [of as am q aaa []])
 apply(subgoal_tac inv_locate_a (as, am) (q, aaa, [Bk]) ires, auto)
 done


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
 apply(simp only: inv_locate_b.simps in_middle.simps)
 apply(erule exE)+
 apply(rename_tac lm1 lm2 tn m ml mr rn)
 apply(rule_tac x = lm1 in exI, rule_tac x = lm2 in exI,
    rule_tac x = tn in exI, rule_tac x = m in exI)
 apply(rule_tac x = Suc ml in exI, rule_tac x = mr − 1 in exI,
    rule_tac x = rn in exI)
 apply(case_tac mr, simp_all, auto)
 done

lemma tape_nat[simp]:  <[x::nat]> = Oc↑(Suc x)
 apply(simp add: tape_of_nat_def tape_of_list_def)
 done

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
 apply(simp add: inv_locate_b.simps inv_locate_a.simps)
 apply(rule_tac disjI2, rule_tac disjI1)
 apply(simp only: in_middle.simps at_begin_fst_bwtn.simps)
 apply(erule_tac exE)+
 apply(rename_tac lm1 n lm2 tn m ml mr rn)
 apply(rule_tac x = lm1 @ [m] in exI, rule_tac x = tn in exI, simp split: if_splits)
  apply(case_tac mr, simp_all)
  apply(cases length am, simp_all, case_tac tn, simp_all)
  apply(case_tac lm2, simp_all add: tape_of_nl_cons split: if_splits)
    apply(cases am, simp_all)
   apply(case_tac n, simp_all)
  apply(case_tac n, simp_all)
 apply(case_tac mr, simp_all)
 apply(case_tac lm2, simp_all add: tape_of_nl_cons split: if_splits, auto)
  apply(case_tac [!] n, simp_all)
 done

lemma repeat_Bk_no_Oc[simp]: (Oc # r = Bk ↑ rn) = False
 apply(cases rn, simp_all)
 done

lemma repeat_Bk[simp]: (∃ rna. Bk ↑ rn = Bk # Bk ↑ rna) ∨ rn = 0
 apply(cases rn, auto)
```

done

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 −
show *?thesis* **using** *assms* **unfolding** *inv_locate_a.simps inv_locate_b.simps*
at_begin_norm.simps at_begin_fst_bwtn.simps at_begin_fst_awtn.simps
apply(*simp only*:*in_middle.simps*)
apply(*erule disjE*, *erule exE*, *erule exE*, *erule exE*)
apply(*rename_tac Lm1 Lm2 Rn*)
apply(*rule_tac x* = *Lm1* **in** *exI*, *rule_tac x* = *tl Lm2* **in** *exI*)
apply(*rule_tac x* = *0* **in** *exI*, *rule_tac x* = *hd Lm2* **in** *exI*)
apply(*rule_tac x* = *1* **in** *exI*, *rule_tac x* = *hd Lm2* **in** *exI*)
apply(*case_tac Lm2*, *force simp*: *tape_of_nl_cons*)
apply(*case_tac tl Lm2*, *simp_all*)
apply(*case_tac Rn*, *auto simp*: *tape_of_nl_cons*)
apply(*rename_tac tn rn*)
apply(*rule_tac x* = *lm* @ *replicate tn 0* **in** *exI*,
rule_tac x = [] **in** *exI*,
rule_tac x = *Suc tn* **in** *exI*,
rule_tac x = *0* **in** *exI*, *auto simp add*: *replicate_append_same*)
apply(*rule_tac x* = *Suc 0* **in** *exI*, *auto*)
done
qed

lemma *length_equal*: *xs* = *ys* $\Longrightarrow$ *length xs* = *length ys*
by *auto*

lemma *inv_locate_a_Bk_via_b*[*simp*]: $[\![$*inv_locate_b* (*as*, *am*) (*q*, *aaa*, *Bk* # *xs*) *ires*;
$\neg$ ($\exists$ *n*. *xs* = *Bk*$\uparrow$*n*)$]\!]$
$\Longrightarrow$ *inv_locate_a* (*as*, *am*) (*Suc q*, *Bk* # *aaa*, *xs*) *ires*
apply(*simp add*: *inv_locate_b.simps inv_locate_a.simps*)
apply(*rule_tac disjI1*)
apply(*simp only*: *in_middle.simps at_begin_norm.simps*)
apply(*erule_tac exE*)+
apply(*rename_tac lm1 lm2 tn m ml mr rn*)
apply(*rule_tac x* = *lm1* @ [*m*] **in** *exI*, *rule_tac x* = *lm2* **in** *exI*, *simp*)
apply(*subgoal_tac tn* = *0*, *simp* , *auto split*: *if_splits*)
apply(*simp add*: *tape_of_nl_cons*)
apply(*drule_tac length_equal*, *simp*)
apply(*cases length am*, *simp_all*, *erule_tac x* = *rn* **in** *allE*, *simp*)
apply(*drule_tac length_equal*, *simp*)
apply(*case_tac* (*Suc* (*length lm1*) − *length am*), *simp_all*)
apply(*case_tac lm2*, *simp*, *simp*)
done

lemma *locate_b_2_a*[*intro*]:
inv_locate_b (*as*, *am*) (*q*, *aaa*, *Bk* # *xs*) *ires*
$\Longrightarrow$ *inv_locate_a* (*as*, *am*) (*Suc q*, *Bk* # *aaa*, *xs*) *ires*

apply(*cases* ∃ *n*. *xs* = *Bk*↑*n*, *simp*, *simp*)
done

lemma *inv_locate_b_Bk*[*simp*]: *inv_locate_b* (*as*, *am*) (*q*, *l*, []) *ires*
⟹ *inv_locate_b* (*as*, *am*) (*q*, *l*, [*Bk*]) *ires*
by(*force simp add*: *inv_locate_b*.*simps in_middle*.*simps*)

lemma *div_rounding_down*[*simp*]: (2∗*q* − *Suc 0*) *div 2* = (*q* − *1*) (*Suc* (2∗*q*)) *div 2* = *q*
by *arith*+

lemma *even_plus_one_odd*[*simp*]: *x mod 2* = *0* ⟹ *Suc x mod 2* = *Suc 0*
by *arith*

lemma *odd_plus_one_even*[*simp*]: *x mod 2* = *Suc 0* ⟹ *Suc x mod 2* = *0*
by *arith*

lemma *locate_b_2_locate_a*[*simp*]:
⟦*q* > *0*; *inv_locate_b* (*as*, *am*) (*q* − *Suc 0*, *aaa*, *Bk* # *xs*) *ires*⟧
⟹ *inv_locate_a* (*as*, *am*) (*q*, *Bk* # *aaa*, *xs*) *ires*
apply(*insert locate_b_2_a* [*of as am q* − *1 aaa xs ires*], *simp*)
done

lemma *findnth_inv_layout_of_via_crsp*[*simp*]:
crsp (*layout_of ap*) (*as*, *lm*) (*s*, *l*, *r*) *ires*
⟹ *findnth_inv* (*layout_of ap*) *n* (*as*, *lm*) (*Suc 0*, *l*, *r*) *ires*
by(*auto simp*: *crsp*.*simps findnth_inv*.*simps inv_locate_a*.*simps*
at_begin_norm.*simps at_begin_fst_awtn*.*simps at_begin_fst_bwtn*.*simps*)

lemma *findnth_correct_pre*:
assumes *layout*: *ly* = *layout_of ap*
and *crsp*: *crsp ly* (*as*, *lm*) (*s*, *l*, *r*) *ires*
and *not0*: *n* > *0*
and *f*: *f* = (λ *stp*. (*steps* (*Suc 0*, *l*, *r*) (*findnth n*, *0*) *stp*, *n*))
and *P*: *P* = (λ ((*s*, *l*, *r*), *n*). *s* = *Suc* (*2* ∗ *n*))
and *Q*: *Q* = (λ ((*s*, *l*, *r*), *n*). *findnth_inv ly n* (*as*, *lm*) (*s*, *l*, *r*) *ires*)
shows ∃ *stp*. *P* (*f stp*) ∧ *Q* (*f stp*)
proof(*rule_tac LE* = *findnth_LE* **in** *halt_lemma2*)
show *wf findnth_LE* **by**(*intro wf_findnth_LE*)
next
show *Q* (*f 0*)
using *crsp layout*
apply(*simp add*: *f P Q steps*.*simps*)
done
next
show ¬ *P* (*f 0*)

using *not0*
apply(*simp add*: *f P steps*.*simps*)
done
next
have $\neg$ *P* (*f na*) $\wedge$ *Q* (*f na*) $\Longrightarrow$ *Q* (*f* (*Suc na*)) $\wedge$ (*f* (*Suc na*), *f na*) $\in$ *findnth_LE* **for** *na*
proof(*simp add*: *f*,
cases steps (*Suc 0*, *l*, *r*) (*findnth n*, *0*) *na*, *simp add*: *P*)
fix *na a b c*
assume $a \neq$ *Suc* (*2* $*$ *n*) $\wedge$ *Q* ((*a*, *b*, *c*), *n*)
thus *Q* (*step* (*a*, *b*, *c*) (*findnth n*, *0*), *n*) $\wedge$
((*step* (*a*, *b*, *c*) (*findnth n*, *0*), *n*), (*a*, *b*, *c*), *n*) $\in$ *findnth_LE*
apply(*cases c*, *case_tac* [*2*] *hd c*)
apply(*simp_all add*: *step*.*simps findnth_LE_def Q findnth_inv*.*simps mod_2 lex_pair_def split*: *if_splits*)
apply(*auto simp*: *mod_ex1 mod_ex2*)
done
qed
thus $\forall$ *n*. $\neg$ *P* (*f n*) $\wedge$ *Q* (*f n*) $\longrightarrow$
Q (*f* (*Suc n*)) $\wedge$ (*f* (*Suc n*), *f n*) $\in$ *findnth_LE* **by** *blast*
qed

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*
apply(*auto simp*: *crsp*.*simps inv_locate_a*.*simps at_begin_norm*.*simps*)
done

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*
using *crsp*
apply(*cases n* = *0*)
apply(*rule_tac x* = *0* **in** *exI*, *auto simp*: *steps*.*simps*)
using *assms*
apply(*drule_tac findnth_correct_pre*, *auto*)
using *findnth_inv*.*simps* **by** *auto*

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 ≝ less_than <*lex*> lex_pair

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

declare inc_inv.simps[simp del]

lemma wf_inc_le[intro]: wf inc_LE
  by(auto simp: inc_LE_def lex_triple_def lex_pair_def)

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 −
  from assms show ?thesis
  apply(auto simp: in_middle.simps inv_after_write.simps
      abc_lm_v.simps abc_lm_s.simps inv_locate_b.simps simp del:split_head_repeat)
  apply(rename_tac lm1 lm2 m ml mr rn)
   apply(case_tac [!] mr, auto split: if_splits)
  apply(rename_tac lm1 lm2 m rn)
  apply(rule_tac x = rn in exI, rule_tac x = Suc m in exI,
      rule_tac x = lm1 in exI, simp)
  apply(rule_tac x = lm2 in exI)
  apply(simp only: Suc_diff_le exp_ind)
  by(subgoal_tac lm2 = []; force dest:length_equal)
qed


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
  apply(auto simp:inv_after_move.simps inv_after_write.simps split: if_splits)
  done
```

```
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
   apply(auto simp: inv_after_write.simps )
  done


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
  apply(auto simp: inv_after_move.simps inv_after_clear.simps split: if_splits)
  done


lemma inv_after_move_2_inv_on_left_moving[simp]:
  assumes inv_after_move (as, lm) (s, l, Bk # r) ires
  shows (l = [] ⟶
        inv_on_left_moving (as, lm) (s', [], Bk # Bk # r) ires) ∧
     (l ≠ [] ⟶
        inv_on_left_moving (as, lm) (s', tl l, hd l # Bk # r) ires)
proof (cases l)
  case (Cons a list)
  from assms Cons show ?thesis
    apply(simp only: inv_after_move.simps inv_on_left_moving.simps)
    apply(rule conjI, force, rule impI, rule disjI1, simp only: inv_on_left_moving_norm.simps)
    apply(erule exE)+
    apply(rename_tac rn m lm1 lm2)
    apply(subgoal_tac lm2 = [])
     apply(rule_tac x = lm1 in exI, rule_tac x = lm2 in exI,
       rule_tac x = m in exI, rule_tac x = m in exI,
       rule_tac x = 1 in exI,
       rule_tac x = rn - 1 in exI)
     apply (auto split:if_splits)
      apply(case_tac [1-2] rn, simp_all)
    by(case_tac [!] lm2, simp_all add: tape_of_nl_cons split: if_splits)
next
  case Nil thus ?thesis using assms
    unfolding inv_after_move.simps inv_on_left_moving.simps
    by (auto split:if_splits)
qed


lemma inv_after_move_2_inv_on_left_moving_B[simp]:
  inv_after_move (as, lm) (s, l, []) ires
     ⟹ (l = [] ⟶ inv_on_left_moving (as, lm) (s', [], [Bk]) ires) ∧
       (l ≠ [] ⟶ inv_on_left_moving (as, lm) (s', tl l, [hd l]) ires)
  apply(simp only: inv_after_move.simps inv_on_left_moving.simps)
  apply(subgoal_tac l ≠ [], rule conjI, simp, rule impI, rule disjI1,
     simp only: inv_on_left_moving_norm.simps)
   apply(erule exE)+
```

```
 apply(rename_tac rn m lm1 lm2)
 apply(subgoal_tac lm2 = [])
  apply(rule_tac x = lm1 in exI, rule_tac x = lm2 in exI,
    rule_tac x = m in exI, rule_tac x = m in exI,
    rule_tac x = 1 in exI, rule_tac x = rn - 1 in exI, force)
 apply(metis append_Cons list.distinct(1) list.exhaust replicate_Suc tape_of_nl_cons)
apply(metis append_Cons list.distinct(1) replicate_Suc)
done

lemma inv_after_clear_2_inv_on_right_moving[simp]:
 inv_after_clear (as, lm) (x, l, Bk # r) ires
    ⟹ inv_on_right_moving (as, lm) (y, Bk # l, r) ires
 apply(auto simp: inv_after_clear.simps inv_on_right_moving.simps simp del:split_head_repeat)
 apply(rename_tac rn m lm1 lm2)
 apply(subgoal_tac lm2 ≠ [])
  apply(rule_tac x = lm1 @ [m] in exI, rule_tac x = tl lm2 in exI,
    rule_tac x = hd lm2 in exI, simp del:split_head_repeat)
  apply(rule_tac x = 0 in exI, rule_tac x = hd lm2 in exI)
  apply(simp, rule conjI)
   apply(case_tac [!] lm2::nat list, auto)
   apply(case_tac rn, auto split: if_splits simp: tape_of_nl_cons)
  apply(case_tac [!] rn, simp_all)
 done


lemma inv_on_right_moving_Oc[simp]: inv_on_right_moving (as, lm) (x, l, Oc # r) ires
    ⟹ inv_on_right_moving (as, lm) (y, Oc # l, r) ires
 apply(auto simp: inv_on_right_moving.simps)
  apply(rename_tac lm1 lm2 ml mr rn)
  apply(rule_tac x = lm1 in exI, rule_tac x = lm2 in exI,
    rule_tac x = ml + mr in exI, simp)
  apply(rule_tac x = Suc ml in exI,
    rule_tac x = mr - 1 in exI, simp)
  apply (metis One_nat_def Suc_pred cell.distinct(1) empty_replicate list.inject
    list.sel(3) neq0_conv self_append_conv2 tl_append2 tl_replicate)
 apply(rule_tac x = lm1 in exI, rule_tac x = [] in exI,
    rule_tac x = ml + mr in exI, simp)
 apply(rule_tac x = Suc ml in exI,
    rule_tac x = mr - 1 in exI)
 apply (auto simp add: Cons_replicate_eq)
 done

lemma inv_on_right_moving_2_inv_on_right_moving[simp]:
 inv_on_right_moving (as, lm) (x, l, Bk # r) ires
   ⟹ inv_after_write (as, lm) (y, l, Oc # r) ires
 apply(auto simp: inv_on_right_moving.simps inv_after_write.simps)
 by (metis append.left_neutral append_Cons )

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

```
apply(auto simp: inv_on_right_moving.simps)
by fastforce

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
  by(auto simp: inv_on_left_moving_in_middle_B.simps )

lemma no_inv_on_left_moving_norm_Bk[simp]: inv_on_left_moving_norm (as, lm) (s, l, Bk # r)
ires
      = False
  by(auto simp: inv_on_left_moving_norm.simps)

lemma inv_on_left_moving_in_middle_B_Bk[simp]:
  ⟦inv_on_left_moving_norm (as, lm) (s, l, Oc # r) ires;
   hd l = Bk; l ≠ []⟧ ⟹
    inv_on_left_moving_in_middle_B (as, lm) (s, tl l, Bk # Oc # r) ires
  apply(cases l, simp, simp)
  apply(simp only: inv_on_left_moving_norm.simps
     inv_on_left_moving_in_middle_B.simps)
  apply(erule_tac exE)+ unfolding tape_of_nl_cons
  apply(rename_tac a list lm1 lm2 m ml mr rn)
  apply(rule_tac x = lm1 in exI, rule_tac x = m # lm2 in exI, auto)
   apply(auto simp: tape_of_nl_cons split: if_splits)
  done

lemma inv_on_left_moving_norm_Oc_Oc[simp]: ⟦inv_on_left_moving_norm (as, lm) (s, l, Oc #
r) ires;
          hd l = Oc; l ≠ []⟧
        ⟹ inv_on_left_moving_norm (as, lm)
                     (s, tl l, Oc # Oc # r) ires
  apply(simp only: inv_on_left_moving_norm.simps)
  apply(erule exE)+
  apply(rename_tac lm1 lm2 m ml mr rn)
  apply(rule_tac x = lm1 in exI, rule_tac x = lm2 in exI,
     rule_tac x = m in exI, rule_tac x = ml - 1 in exI,
     rule_tac x = Suc mr in exI, rule_tac x = rn in exI, simp)
  apply(case_tac ml, auto simp: split: if_splits)
  done

lemma inv_on_left_moving_in_middle_B_Bk_Oc[simp]: inv_on_left_moving_norm (as, lm) (s, [],
Oc # r) ires
    ⟹ inv_on_left_moving_in_middle_B (as, lm) (s, [], Bk # Oc # r) ires
  by(auto simp: inv_on_left_moving_norm.simps
     inv_on_left_moving_in_middle_B.simps split: if_splits)

lemma inv_on_left_moving_Oc_cases[simp]:inv_on_left_moving (as, lm) (s, l, Oc # r) ires
   ⟹ (l = [] ⟶ inv_on_left_moving (as, lm) (s, [], Bk # Oc # r) ires)
 ∧  (l ≠ [] ⟶ inv_on_left_moving (as, lm) (s, tl l, hd l # Oc # r) ires)
  apply(simp add: inv_on_left_moving.simps)
```

apply(*cases l* ≠ [], *rule conjI*, *simp*, *simp*)
apply(*cases hd l*, *simp*, *simp*, *simp*)
done

lemma *from_on_left_moving_to_check_left_moving*[*simp*]: *inv_on_left_moving_in_middle_B* (*as*, *lm*)

(*s*, *Bk* # *list*, *Bk* # *r*) *ires*
⟹ *inv_check_left_moving_on_leftmost* (*as*, *lm*)
(*s*′, *list*, *Bk* # *Bk* # *r*) *ires*
apply(*simp only*: *inv_on_left_moving_in_middle_B.simps inv_check_left_moving_on_leftmost.simps*)
apply(*erule_tac exE*)+
apply(*rename_tac lm1 lm2 rn*)
apply(*case_tac rev lm1*, *simp_all*)
apply(*case_tac tl* (*rev lm1*), *simp_all add*: *tape_of_nat_def tape_of_list_def*)
done

lemma *inv_check_left_moving_in_middle_no_Bk*[*simp*]:
inv_check_left_moving_in_middle (*as*, *lm*) (*s*, *l*, *Bk* # *r*) *ires*= *False*
by(*auto simp*: *inv_check_left_moving_in_middle.simps*)

lemma *inv_check_left_moving_on_leftmost_Bk_Bk*[*simp*]:
inv_on_left_moving_in_middle_B (*as*, *lm*) (*s*, [], *Bk* # *r*) *ires*⟹
inv_check_left_moving_on_leftmost (*as*, *lm*) (*s*′, [], *Bk* # *Bk* # *r*) *ires*
apply(*auto simp*: *inv_on_left_moving_in_middle_B.simps*
inv_check_left_moving_on_leftmost.simps split: *if_splits*)
done

lemma *inv_check_left_moving_on_leftmost_no_Oc*[*simp*]: *inv_check_left_moving_on_leftmost* (*as*, *lm*)
(*s*, *list*, *Oc* # *r*) *ires*= *False*
by(*auto simp*: *inv_check_left_moving_on_leftmost.simps split*: *if_splits*)

lemma *inv_check_left_moving_in_middle_Oc_Bk*[*simp*]: *inv_on_left_moving_in_middle_B* (*as*, *lm*)
(*s*, *Oc* # *list*, *Bk* # *r*) *ires*
⟹ *inv_check_left_moving_in_middle* (*as*, *lm*) (*s*′, *list*, *Oc* # *Bk* # *r*) *ires*
apply(*auto simp*: *inv_on_left_moving_in_middle_B.simps*
inv_check_left_moving_in_middle.simps split: *if_splits*)
done

lemma *inv_on_left_moving_2_check_left_moving*[*simp*]:
inv_on_left_moving (*as*, *lm*) (*s*, *l*, *Bk* # *r*) *ires*
⟹ (*l* = [] ⟶ *inv_check_left_moving* (*as*, *lm*) (*s*′, [], *Bk* # *Bk* # *r*) *ires*)
∧ (*l* ≠ [] ⟶
inv_check_left_moving (*as*, *lm*) (*s*′, *tl l*, *hd l* # *Bk* # *r*) *ires*)
by (*cases l*;*cases hd l*, *auto simp*: *inv_on_left_moving.simps inv_check_left_moving.simps*)

lemma *inv_on_left_moving_norm_no_empty*[*simp*]: *inv_on_left_moving_norm* (*as*, *lm*) (*s*, *l*, []) *ires*
= *False*
apply(*auto simp*: *inv_on_left_moving_norm.simps*)
done

```
lemma inv_on_left_moving_no_empty[simp]: inv_on_left_moving (as, lm) (s, l, []) ires = False
 apply(simp add: inv_on_left_moving.simps)
 apply(simp add: inv_on_left_moving_in_middle_B.simps)
 done

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
 using assms
 apply(simp only: inv_check_left_moving_in_middle.simps
    inv_on_left_moving_in_middle_B.simps)
 apply(erule_tac exE)+
 apply(rename_tac lm1 lm2 r' rn)
 apply(rule_tac x = rev (tl (rev lm1)) in exI,
    rule_tac x = [hd (rev lm1)] @ lm2 in exI, auto)
     apply(case_tac [!] rev lm1,case_tac [!] tl (rev lm1))
             apply(simp_all add: tape_of_nat_def tape_of_list_def tape_of_nat_list.simps)
  apply(case_tac [1] lm2, auto simp:tape_of_nat_def)
 apply(case_tac lm2, auto simp:tape_of_nat_def)
 done

lemma inv_check_left_moving_in_middle_Bk_Oc[simp]:
 inv_check_left_moving_in_middle (as, lm) (s, [], Oc # r) ires⟹
   inv_check_left_moving_in_middle (as, lm) (s', [Bk], Oc # r) ires
 apply(auto simp: inv_check_left_moving_in_middle.simps )
 done

lemma inv_on_left_moving_norm_Oc_Oc_via_middle[simp]: inv_check_left_moving_in_middle (as,
lm)
                (s, Oc # list, Oc # r) ires
  ⟹ inv_on_left_moving_norm (as, lm) (s', list, Oc # Oc # r) ires
 apply(auto simp: inv_check_left_moving_in_middle.simps
    inv_on_left_moving_norm.simps)
 apply(rename_tac lm1 lm2 rn)
 apply(rule_tac x = rev (tl (rev lm1)) in exI,
    rule_tac x = lm2 in exI, rule_tac x = hd (rev lm1) in exI)
 apply(rule_tac conjI)
  apply(case_tac rev lm1, simp, simp)
 apply(rule_tac x = hd (rev lm1) − 1 in exI, auto)
  apply(rule_tac [!] x = Suc (Suc 0) in exI, simp)
  apply(case_tac [!] rev lm1, simp_all)
  apply(case_tac [!] last lm1, simp_all add: tape_of_nl_cons split: if_splits)
 done

lemma inv_check_left_moving_Oc_cases[simp]: inv_check_left_moving (as, lm) (s, l, Oc # r) ires
⟹ (l = [] ⟶ inv_on_left_moving (as, lm) (s', [], Bk # Oc # r) ires) ∧
  (l ≠ [] ⟶ inv_on_left_moving (as, lm) (s', tl l, hd l # Oc # r) ires)
 apply(cases l;cases hd l, auto simp: inv_check_left_moving.simps inv_on_left_moving.simps)
```

done

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*
apply(*auto simp*: *inv_check_left_moving*.*simps*
inv_check_left_moving_on_leftmost.*simps* *inv_after_left_moving*.*simps*)
done

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*
by(*simp add*: *inv_check_left_moving*.*simps*
inv_check_left_moving_in_middle.*simps*
inv_check_left_moving_on_leftmost.*simps*)

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*
apply(*auto simp*: *inv_after_left_moving*.*simps* *inv_stop*.*simps*)
done

lemma *inv_stop_Bk_empty*[*simp*]: *inv_after_left_moving* (*as*, *lm*) (*s*, *l*, []) *ires*
$\Longrightarrow$ *inv_stop* (*as*, *lm*) (*s′*, *Bk* # *l*, []) *ires*
by(*auto simp*: *inv_after_left_moving*.*simps*)

lemma *inv_stop_indep_fst*[*simp*]: *inv_stop* (*as*, *lm*) (*x*, *l*, *r*) *ires* $\Longrightarrow$
inv_stop (*as*, *lm*) (*y*, *l*, *r*) *ires*
apply(*simp add*: *inv_stop*.*simps*)
done

lemma *inv_after_clear_no_Oc*[*simp*]: *inv_after_clear* (*as*, *lm*) (*s*, *aaa*, *Oc* # *xs*) *ires*= *False*
apply(*auto simp*: *inv_after_clear*.*simps*)
done

lemma *inv_after_left_moving_no_Oc*[*simp*]:
inv_after_left_moving (*as*, *lm*) (*s*, *aaa*, *Oc* # *xs*) *ires* = *False*
by(*auto simp*: *inv_after_left_moving*.*simps*)

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*
apply(*auto simp*: *inv_after_clear*.*simps*)
done

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* $\Longrightarrow$ $b \neq []$

```
 apply(auto simp: inv_on_left_moving.simps inv_on_left_moving_norm.simps split: if_splits)
 done

lemma inv_check_left_moving_Suc_nonempty[simp]:
 inv_check_left_moving (as, abc_lm_s lm n (Suc (abc_lm_v lm n))) (s, b, Oc # list) ires ⟹ b ≠
[]
 apply(auto simp: inv_check_left_moving.simps inv_check_left_moving_in_middle.simps split: if_splits)
 done

lemma tinc_correct_pre:
 assumes layout: ly = layout_of ap
  and inv_start: inv_locate_a (as, lm) (n, l, r) ires
  and lm': lm' = abc_lm_s lm n (Suc (abc_lm_v lm n))
  and f: f = steps (Suc 0, l, r) (tinc_b, 0)
  and P: P = (λ (s, l, r). s = 10)
  and Q: Q = (λ (s, l, r). inc_inv n (as, lm) (s, l, r) ires)
 shows ∃ stp. P (f stp) ∧ Q (f stp)
proof(rule_tac LE = inc_LE in halt_lemma2)
 show wf inc_LE by(auto)
next
 show Q (f 0)
  using inv_start
  apply(simp add: f P Q steps.simps inc_inv.simps)
  done
next
 show ¬ P (f 0)
  apply(simp add: f P steps.simps)
  done
next
 have ¬ P (f n) ∧ Q (f n) ⟹ Q (f (Suc n)) ∧ (f (Suc n), f n)
     ∈ inc_LE for n
 proof(simp add: f,
    cases steps (Suc 0, l, r) (tinc_b, 0) n, simp add: P)
  fix n a b c
  assume a ≠ 10 ∧ Q (a, b, c)
  thus  Q (step (a, b, c) (tinc_b, 0)) ∧ (step (a, b, c) (tinc_b, 0), a, b, c) ∈ inc_LE
   apply(simp add:Q)
   apply(simp add: inc_inv.simps)
   apply(cases c; cases hd c)
     apply(auto simp: Let_def step.simps tinc_b_def split: if_splits)
              apply(simp_all add: inc_inv.simps inc_LE_def lex_triple_def lex_pair_def
     inc_measure_def numeral)
   done
 qed
 thus ∀ n. ¬ P (f n) ∧ Q (f n) ⟶ Q (f (Suc n)) ∧ (f (Suc n), f n) ∈ inc_LE by blast
qed

lemma tinc_correct:
 assumes layout: ly = layout_of ap
  and inv_start: inv_locate_a (as, lm) (n, l, r) ires
```

```
  and lm′: lm′ = abc_lm_s lm n (Suc (abc_lm_v lm n))
 shows ∃ stp l′ r′. steps (Suc 0, l, r) (tinc_b, 0) stp = (10, l′, r′)
        ∧ inv_stop (as, lm′) (10, l′, r′) ires
 using assms
 apply(drule_tac tinc_correct_pre, auto)
 apply(rule_tac x = stp in exI, simp)
 apply(simp add: inc_inv.simps)
 done

lemma is_even_4[simp]: (4::nat) ∗ n mod 2 = 0
 apply(arith)
 done

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 −
 have ∃ stp l′ r′. steps (Suc 0, l, r) (findnth n, 0) stp = (Suc (2 ∗ n), l′, r′)
  ∧ inv_locate_a (as, lm) (n, l′, r′) ires
  using assms
  apply(rule_tac findnth_correct, simp_all add: crsp layout)
  done
 from this obtain stp l′ r′ where a:
  steps (Suc 0, l, r) (findnth n, 0) stp = (Suc (2 ∗ n), l′, r′)
  ∧ inv_locate_a (as, lm) (n, l′, r′) ires by blast
 moreover have
  ∃ stp la ra. steps (Suc 0, l′, r′) (tinc_b, 0) stp = (10, la, ra)
              ∧ inv_stop (as, lma) (10, la, ra) ires
  using assms a
 proof(rule_tac lm′ = lma and n = n and lm = lm and ly = ly and ap = ap in tinc_correct,
    simp, simp)
  show lma = abc_lm_s lm n (Suc (abc_lm_v lm n))
   using aexec
   apply(simp add: abc_step_l.simps)
   done
 qed
 from this obtain stpa la ra where b:
  steps (Suc 0, l′, r′) (tinc_b, 0) stpa = (10, la, ra)
  ∧ inv_stop (as, lma) (10, la, ra) ires by blast
 from a b show ∃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
  apply(rule_tac x = stp + stpa in exI)
  using tm_append_steps[of Suc 0 l r findnth n stp l′ r′ tinc_b stpa 10 la ra length (findnth n) div
2]
  apply(simp add: length_findnth inv_stop.simps)
  apply(cases stpa, simp_all add: steps.simps)
  done
```

```
qed

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(cases (abc_step_l (as, lm) (Some (Inc n))))
 fix a b
 assume aexec: abc_step_l (as, lm) (Some (Inc n)) = (a, b)
 then have ∃ stp k. steps (Suc 0, l, r) (findnth n @ shift tinc_b (2 ∗ n), 0) stp
      = (2∗n + 10, Bk # Bk # ires, <b> @ Bk↑k) ∧ stp > 0
  using assms
  apply(rule_tac crsp_step_inc_pre, simp_all)
  done
 thus ?thesis
  using assms aexec
  apply(erule_tac exE)
  apply(erule_tac exE)
  apply(erule_tac conjE)
  apply(rename_tac stp k)
  apply(rule_tac x = stp in exI, simp add: ci.simps tm_shift_eq_steps)
  apply(drule_tac off = (start_of (layout_of ap) as − Suc 0) in tm_shift_eq_steps)
   apply(auto simp: crsp.simps abc_step_l.simps fetch start_of_Suc1)
  done
qed
```

9.4 Crsp of Dec n e

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

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

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

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

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

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

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

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

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

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

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

declare fetch.simps[simp del]


lemma x_plus_helpers:
  x + 4 = Suc (x + 3)
  x + 5 = Suc (x + 4)
  x + 6 = Suc (x + 5)
  x + 7 = Suc (x + 6)
  x + 8 = Suc (x + 7)
  x + 9 = Suc (x + 8)
  x + 10 = Suc (x + 9)
  x + 11 = Suc (x + 10)
  x + 12 = Suc (x + 11)
  x + 13 = Suc (x + 12)
  14 + x = Suc (x + 13)
  15 + x = Suc (x + 14)
  16 + x = Suc (x + 15)
  by auto

lemma fetch_Dec[simp]:
  fetch (ci ly (start_of ly as) (Dec n e)) (Suc (2 ∗ n)) Bk = (W1, start_of ly as + 2 ∗n)
```

fetch (ci ly (start_of ly as) (Dec n e)) (Suc (2 ∗ n)) Oc = (R, Suc (start_of ly as) + 2 ∗n)
fetch (ci (ly) (start_of ly as) (Dec n e)) (Suc (Suc (2 ∗ n))) Oc
= (R, start_of ly as + 2∗n + 2)
fetch (ci (ly) (start_of ly as) (Dec n e)) (Suc (Suc (2 ∗ n))) Bk
= (L, start_of ly as + 2∗n + 13)
fetch (ci (ly) (start_of ly as) (Dec n e)) (Suc (Suc (Suc (2 ∗ n)))) Oc
= (R, start_of ly as + 2∗n + 2)
fetch (ci (ly) (start_of ly as) (Dec n e)) (Suc (Suc (Suc (2 ∗ n)))) Bk
= (L, start_of ly as + 2∗n + 3)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 4) Oc = (W0, start_of ly as + 2∗n + 3)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 4) Bk = (R, start_of ly as + 2∗n + 4)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 5) Bk = (R, start_of ly as + 2∗n + 5)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 6) Bk = (L, start_of ly as + 2∗n + 6)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 6) Oc = (L, start_of ly as + 2∗n + 7)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 7) Bk = (L, start_of ly as + 2∗n + 10)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 8) Bk = (W1, start_of ly as + 2∗n + 7)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 8) Oc = (R, start_of ly as + 2∗n + 8)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 9) Bk = (L, start_of ly as + 2∗n + 9)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 9) Oc = (R, start_of ly as + 2∗n + 8)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 10) Bk = (R, start_of ly as + 2∗n + 4)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 10) Oc = (W0, start_of ly as + 2∗n + 9)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 11) Oc = (L, start_of ly as + 2∗n + 10)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 11) Bk = (L, start_of ly as + 2∗n + 11)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 12) Oc = (L, start_of ly as + 2∗n + 10)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 12) Bk = (R, start_of ly as + 2∗n + 12)
fetch (ci (ly) (start_of ly as) (Dec n e)) (2 ∗ n + 13) Bk = (R, start_of ly as + 2∗n + 16)
fetch (ci (ly) (start_of ly as) (Dec n e)) (14 + 2 ∗ n) Oc = (L, start_of ly as + 2∗n + 13)
fetch (ci (ly) (start_of ly as) (Dec n e)) (14 + 2 ∗ n) Bk = (L, start_of ly as + 2∗n + 14)
fetch (ci (ly) (start_of ly as) (Dec n e)) (15 + 2 ∗ n) Oc = (L, start_of ly as + 2∗n + 13)
fetch (ci (ly) (start_of ly as) (Dec n e)) (15 + 2 ∗ n) Bk = (R, start_of ly as + 2∗n + 15)
fetch (ci (ly) (start_of (ly) as) (Dec n e)) (16 + 2 ∗ n) Bk = (R, start_of (ly) e)
unfolding x_plus_helpers fetch.simps
by(auto simp: ci.simps shift.simps nth_append tdec_b_def length_findnth adjust.simps)

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
apply(rule_tac x = Suc (Suc 0) **in** exI)
apply(auto simp: steps.simps step.simps length_ci_dec)
apply(cases r, simp_all)
done

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
  apply(rule_tac x = (Suc 0) in exI, cases hd r, simp_all)
  apply(auto simp: steps.simps step.simps length_ci_dec)
  apply(cases r, simp_all)
  done

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

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

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

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

definition abc_dec_1_LE ::
  ((config × nat ×
  nat) × (config × nat × nat)) set
  where abc_dec_1_LE ≝ (inv_image lex_triple abc_dec_1_measure)

lemma wf_dec_le: wf abc_dec_1_LE
  by(auto intro:wf_inv_image simp:abc_dec_1_LE_def lex_triple_def lex_pair_def)
```

```
lemma startof_Suc2:
  abc_fetch as ap = Some (Dec n e) ⟹
      start_of (layout_of ap) (Suc as) =
          start_of (layout_of ap) as + 2 * n + 16
  apply(auto simp: start_of.simps layout_of.simps
     length_of.simps abc_fetch.simps
     take_Suc_conv_app_nth split: if_splits)
  done

lemma start_of_less_2:
  start_of ly e ≤ start_of ly (Suc e)
  apply(cases e < length ly)
   apply(auto simp: start_of.simps take_Suc take_Suc_conv_app_nth)
  done

lemma start_of_less_1: start_of ly e ≤ start_of ly (e + d)
proof(induct d)
  case 0 thus ?case by simp
next
  case (Suc d)
  have start_of ly e ≤ start_of ly (e + d)  by fact
  moreover have start_of ly (e + d) ≤ start_of ly (Suc (e + d))
    by(rule_tac start_of_less_2)
  ultimately show?case
    by(simp)
qed

lemma start_of_less:
  assumes e < as
  shows start_of ly e ≤ start_of ly as
proof −
  obtain d where  as = e + d
    using assms by (metis less_imp_add_positive)
  thus ?thesis
    by(simp add: start_of_less_1)
qed

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(cases e = Suc as)
  case True
  have e = Suc as by fact
  moreover hence start_of ly (Suc as) = start_of ly as + 2*n + 16
    using layout fetch
    by(simp add: startof_Suc2)
  ultimately show ?thesis by (simp)
next
```

```
 case False
 have e ≠ Suc as by fact
 then have e > Suc as using great by arith
 then have start_of ly (Suc as) ≤ start_of ly e
  by(simp add: start_of_less)
 moreover have start_of ly (Suc as) = start_of ly as + 2∗n + 16
  using layout fetch
  by(simp add: startof_Suc2)
 ultimately show ?thesis
  by arith
qed

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)
 using start_of_ge[of as aprog n e ly] start_of_less[of e as ly]
 apply(cases e < as, simp)
 apply(cases e = as, simp, simp)
 done

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)
 using start_of_ge[of as aprog n e ly] start_of_less[of e as ly]
 apply(cases e < as, simp, simp)
 apply(cases e = as, simp, simp)
 done

lemma inv_locate_b_nonempty[simp]: inv_locate_b (as, lm) (n, [], []) ires = False
 apply(auto simp: inv_locate_b.simps in_middle.simps split: if_splits)
 done

lemma inv_locate_b_no_Bk[simp]: inv_locate_b (as, lm) (n, [], Bk # list) ires = False
 apply(auto simp: inv_locate_b.simps in_middle.simps split: if_splits)
 done

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
 apply(simp only: dec_first_on_right_moving.simps)
 apply(erule exE)+
 apply(rename_tac lm1 lm2 m ml mr rn)
 apply(rule_tac x = lm1 in exI, rule_tac x = lm2 in exI,
   rule_tac x = m in exI, rule_tac x = Suc ml in exI,
   rule_tac x = mr − 1 in exI)
   apply(case_tac [!] mr, auto)
 done

lemma dec_first_on_right_moving_Bk_nonempty[simp]:
 dec_first_on_right_moving n (as, am) (s, l, Bk # xs) ires ⟹ l ≠ []
 apply(auto simp: dec_first_on_right_moving.simps split: if_splits)
 done

lemma replicateE:
 ⟦¬ length lm1 < length am;
  am @ replicate (length lm1 − length am) 0 @ [0::nat] =
                                  lm1 @ m # lm2;
  0 < m⟧
  ⟹ RR
 apply(subgoal_tac lm2 = [], simp)
 apply(drule_tac length_equal, simp)
 done

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
 apply(simp only: dec_first_on_right_moving.simps
   dec_after_clear.simps abc_lm_s.simps abc_lm_v.simps)
 apply(erule_tac exE)+
```

```
 apply(rename_tac lm1 lm2 m ml mr rn)
 apply(cases n < length am)
 by(rule_tac x = lm1 in exI, rule_tac x = lm2 in exI,
    rule_tac x = m − 1 in exI, auto elim:replicateE)

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)
 apply(subgoal_tac l ≠ [],
    simp only: dec_first_on_right_moving.simps
    dec_after_clear.simps abc_lm_s.simps abc_lm_v.simps)
  apply(erule_tac exE)+
 apply(rename_tac lm1 lm2 m ml mr rn)
  apply(cases n < length am, simp)
   apply(rule_tac x = lm1 in exI, rule_tac x = m − 1 in exI, auto)
   apply(case_tac [1−2] m, auto)
 apply(auto simp: dec_first_on_right_moving.simps split: if_splits)
 done

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
 apply(auto simp: dec_after_clear.simps)
 done

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
 apply(auto simp: dec_after_clear.simps dec_right_move.simps split: if_splits)
 done

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
 apply(auto simp: dec_after_clear.simps dec_right_move.simps split: if_splits)
 done

lemma dec_right_move_no_Oc[simp]:dec_right_move (as, am) (s, l, Oc # r) ires = False
 apply(auto simp: dec_right_move.simps)
 done

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
 apply(auto simp: dec_right_move.simps dec_check_right_move.simps split: if_splits)
 done

lemma lm_iff_empty[simp]: (<lm::nat list> = []) = (lm = [])
 apply(cases lm, simp_all add: tape_of_nl_cons)
```

done

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*
apply(*simp add*: *dec_right_move*.*simps*)
done

lemma *dec_check_right_move_nonempty*[*simp*]: *dec_check_right_move* (*as*, *am*) (*s*, *l*, *r*) *ires*$\Longrightarrow$ $l \neq []$
apply(*auto simp*: *dec_check_right_move*.*simps split*: *if_splits*)
done

lemma *dec_check_right_move_Oc_tail*[*simp*]: ⟦*dec_check_right_move* (*as*, *am*) (*s*, *l*, *Oc* # *r*) *ires*⟧
$\Longrightarrow$ *dec_after_write* (*as*, *am*) (*s'*, *tl l*, *hd l* # *Oc* # *r*) *ires*
apply(*auto simp*: *dec_check_right_move*.*simps dec_after_write*.*simps*)
apply(*rename_tac lm1 lm2 m rn*)
apply(*rule_tac x* = *lm1* **in** *exI*, *rule_tac x* = *lm2* **in** *exI*, *rule_tac x* = *m* **in** *exI*, *auto*)
done

lemma *dec_left_move_Bk_tail*[*simp*]: ⟦*dec_check_right_move* (*as*, *am*) (*s*, *l*, *Bk* # *r*) *ires*⟧
$\Longrightarrow$ *dec_left_move* (*as*, *am*) (*s'*, *tl l*, *hd l* # *Bk* # *r*) *ires*
apply(*auto simp*: *dec_check_right_move*.*simps dec_left_move*.*simps inv_after_move*.*simps*)
apply(*rename_tac lm1 lm2 m rn*)
apply(*rule_tac x* = *lm1* **in** *exI*, *rule_tac x* = *m* **in** *exI*, *auto split*: *if_splits*)
apply(*case_tac* [!] *lm2*, *simp_all add*: *tape_of_nl_cons split*: *if_splits*)
apply(*rule_tac* [!] *x* = (*Suc rn*) **in** *exI*, *simp_all*)
done

lemma *dec_left_move_tail*[*simp*]: ⟦*dec_check_right_move* (*as*, *am*) (*s*, *l*, []) *ires*⟧
$\Longrightarrow$ *dec_left_move* (*as*, *am*) (*s'*, *tl l*, [*hd l*]) *ires*
apply(*auto simp*: *dec_check_right_move*.*simps dec_left_move*.*simps inv_after_move*.*simps*)
apply(*rename_tac lm1 m*)
apply(*rule_tac x* = *lm1* **in** *exI*, *rule_tac x* = *m* **in** *exI*, *auto*)
done

lemma *dec_left_move_no_Oc*[*simp*]: *dec_left_move* (*as*, *am*) (*s*, *aaa*, *Oc* # *xs*) *ires* = *False*
apply(*auto simp*: *dec_left_move*.*simps inv_after_move*.*simps*)
done

lemma *dec_left_move_nonempty*[*simp*]: *dec_left_move* (*as*, *am*) (*s*, *l*, *r*) *ires*
$\Longrightarrow l \neq []$
apply(*auto simp*: *dec_left_move*.*simps split*: *if_splits*)
done

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*
apply(*simp add*: *inv_on_left_moving_in_middle_B*.*simps*)
apply(*rule_tac x* = [*m*] **in** *exI*, *auto*)

```
  done


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
  apply(simp only: inv_on_left_moving_in_middle_B.simps)
  apply(rule_tac x = lm1 @ [m ] in exI, rule_tac x = [] in exI, simp)
  apply(simp add: tape_of_nl_cons split: if_splits)
  done

lemma inv_on_left_moving_Bk_tail[simp]: dec_left_move (as, am) (s, l, Bk # r) ires
     ⟹ inv_on_left_moving (as, am) (s', tl l, hd l # Bk # r) ires
  apply(auto simp: dec_left_move.simps inv_on_left_moving.simps split: if_splits)
  done

lemma inv_on_left_moving_tail[simp]: dec_left_move (as, am) (s, l, []) ires
        ⟹ inv_on_left_moving (as, am) (s', tl l, [hd l]) ires
  apply(auto simp: dec_left_move.simps inv_on_left_moving.simps split: if_splits)
  done

lemma dec_on_right_moving_Oc_mv[simp]: dec_after_write (as, am) (s, l, Oc # r) ires
     ⟹ dec_on_right_moving (as, am) (s', Oc # l, r) ires
  apply(auto simp: dec_after_write.simps dec_on_right_moving.simps)
  apply(rename_tac lm1 lm2 m rn)
  apply(rule_tac x = lm1 @ [m] in exI, rule_tac x = tl lm2 in exI,
    rule_tac x = hd lm2 in exI, simp)
  apply(rule_tac x = Suc 0 in exI,rule_tac x =  Suc (hd lm2) in exI)
  apply(case_tac lm2, auto split: if_splits simp: tape_of_nl_cons)
  done

lemma dec_after_write_Oc_via_Bk[simp]: dec_after_write (as, am) (s, l, Bk # r) ires
     ⟹ dec_after_write (as, am) (s', l, Oc # r) ires
  apply(auto simp: dec_after_write.simps)
  done

lemma dec_after_write_Oc_empty[simp]: dec_after_write (as, am) (s, aaa, []) ires
        ⟹ dec_after_write (as, am) (s', aaa, [Oc]) ires
  apply(auto simp: dec_after_write.simps)
  done

lemma dec_on_right_moving_Oc_move[simp]: dec_on_right_moving (as, am) (s, l, Oc # r) ires
     ⟹ dec_on_right_moving (as, am) (s', Oc # l, r) ires
  apply(simp only: dec_on_right_moving.simps)
  apply(erule_tac exE)+
  apply(rename_tac lm1 lm2 m ml mr rn)
  apply(erule conjE)+
  apply(rule_tac x = lm1 in exI, rule_tac x = lm2 in exI,
    rule_tac x = m in exI, rule_tac x = Suc ml in exI,
    rule_tac x = mr - 1 in exI, simp)
```

```
 apply(case_tac mr, auto)
 done

lemma dec_on_right_moving_nonempty[simp]: dec_on_right_moving (as, am) (s, l, r) ires⟹ l
≠ []
 apply(auto simp: dec_on_right_moving.simps split: if_splits)
 done

lemma dec_after_clear_Bk_tail[simp]: dec_on_right_moving (as, am) (s, l, Bk # r) ires
    ⟹ dec_after_clear (as, am) (s', tl l, hd l # Bk # r) ires
 apply(auto simp: dec_on_right_moving.simps dec_after_clear.simps simp del:split_head_repeat)
 apply(rename_tac lm1 lm2 m ml mr rn)
 apply(case_tac mr, auto split: if_splits)
 done

lemma dec_after_clear_tail[simp]: dec_on_right_moving (as, am) (s, l, []) ires
        ⟹ dec_after_clear (as, am) (s', tl l, [hd l]) ires
 apply(auto simp: dec_on_right_moving.simps dec_after_clear.simps)
 apply(simp_all split: if_splits)
 apply(rule_tac x = lm1 in exI, simp)
 done

lemma dec_false_1[simp]:
 ⟦abc_lm_v am n = 0; inv_locate_b (as, am) (n, aaa, Oc # xs) ires⟧
 ⟹ False
 apply(auto simp: inv_locate_b.simps in_middle.simps)
 apply(rename_tac lm1 lm2 m ml Mr rn)
  apply(case_tac length lm1 ≥ length am, auto)
   apply(subgoal_tac lm2 = [], simp, subgoal_tac m = 0, simp)
    apply(case_tac Mr, auto simp: )
   apply(subgoal_tac Suc (length lm1) − length am =
            Suc (length lm1 − length am),
    simp add: exp_ind del: replicate.simps, simp)
   apply(drule_tac xs = am @ replicate (Suc (length lm1) − length am) 0
    and ys = lm1 @ m # lm2 in length_equal, simp)
  apply(case_tac Mr, auto simp: abc_lm_v.simps)
 apply(rename_tac lm1 m ml Mr)
 apply(case_tac Mr = 0, simp_all split: if_splits)
 apply(subgoal_tac Suc (length lm1) − length am =
            Suc (length lm1 − length am),
    simp add: exp_ind del: replicate.simps, simp)
 done

lemma inv_on_left_moving_Bk_tl[simp]:
 ⟦inv_locate_b (as, am) (n, aaa, Bk # xs) ires;
  abc_lm_v am n = 0⟧
  ⟹ inv_on_left_moving (as, abc_lm_s am n 0)
            (s, tl aaa, hd aaa # Bk # xs) ires
 apply(simp add: inv_on_left_moving.simps)
 apply(simp only: inv_locate_b.simps in_middle.simps)
```

```
 apply(erule_tac exE)+
 apply(rename_tac Lm1 Lm2 tn M ml Mr rn)
 apply(subgoal_tac ¬ inv_on_left_moving_in_middle_B
       (as, abc_lm_s am n 0) (s, tl aaa, hd aaa # Bk # xs) ires, simp)
  apply(simp only: inv_on_left_moving_norm.simps)
  apply(erule_tac conjE)+
  apply(rule_tac x = Lm1 in exI, rule_tac x = Lm2 in exI,
    rule_tac x =  M in exI, rule_tac x = M in exI,
    rule_tac x = Suc 0 in exI, simp add: abc_lm_s.simps)
  apply(case_tac Mr, auto simp: abc_lm_v.simps)
  apply(simp only: exp_ind[THEN sym] replicate_Suc Nat.Suc_diff_le)
 apply(auto simp: inv_on_left_moving_in_middle_B.simps split: if_splits)
 done


lemma inv_on_left_moving_tl[simp]:
 ⟦abc_lm_v am n = 0; inv_locate_b (as, am) (n, aaa, []) ires⟧
  ⟹ inv_on_left_moving (as, abc_lm_s am n 0) (s, tl aaa, [hd aaa]) ires
 apply(simp add: inv_on_left_moving.simps)
 apply(simp only: inv_locate_b.simps in_middle.simps)
 apply(erule_tac exE)+
 apply(rename_tac Lm1 Lm2 tn M ml Mr rn)
 apply(simp add: inv_on_left_moving.simps)
 apply(subgoal_tac ¬ inv_on_left_moving_in_middle_B
       (as, abc_lm_s am n 0) (s, tl aaa, [hd aaa]) ires, simp)
  apply(simp only: inv_on_left_moving_norm.simps)
  apply(erule_tac conjE)+
  apply(rule_tac x = Lm1 in exI, rule_tac x = Lm2 in exI,
    rule_tac x =  M in exI, rule_tac x = M in exI,
    rule_tac x = Suc 0 in exI, simp add: abc_lm_s.simps)
  apply(case_tac Mr, simp_all, auto simp: abc_lm_v.simps)
   apply(simp_all only: exp_ind Nat.Suc_diff_le del: replicate_Suc, simp_all)
 apply(auto simp: inv_on_left_moving_in_middle_B.simps split: if_splits)
  apply(case_tac [!] M, simp_all)
 done

declare dec_inv_1.simps[simp del]

declare inv_locate_n_b.simps [simp del]

lemma dec_first_on_right_moving_Oc_via_inv_locate_n_b[simp]:
 ⟦inv_locate_n_b (as, am) (n, aaa, Oc # xs) ires⟧
 ⟹ dec_first_on_right_moving n (as, abc_lm_s am n (abc_lm_v am n))
                               (s, Oc # aaa, xs) ires
 apply(auto simp: inv_locate_n_b.simps dec_first_on_right_moving.simps
    abc_lm_s.simps abc_lm_v.simps)
 apply(rename_tac Lm1 Lm2 m rn)
   apply(rule_tac x = Lm1 in exI, rule_tac x = Lm2 in exI,
    rule_tac x = m in exI, simp)
   apply(rule_tac x = Suc (Suc 0) in exI,
```

rule_tac $x = m - 1$ **in** exI, simp)
apply (metis One_nat_def Suc_pred cell.distinct(1) empty_replicate list.inject list.sel(3)
neq0_conv self_append_conv2 tl_append2 tl_replicate)
apply(rename_tac Lm1 Lm2 m rn)
apply(rule_tac $x = Lm1$ **in** exI, rule_tac $x = Lm2$ **in** exI,
rule_tac $x = m$ **in** exI,
simp add: Suc_diff_le exp_ind del: replicate.simps)
apply(rule_tac $x = Suc\ (Suc\ 0)$ **in** exI,
rule_tac $x = m - 1$ **in** exI, simp)
apply (metis cell.distinct(1) empty_replicate gr_zeroI list.inject replicateE self_append_conv2)
apply(rename_tac Lm1 m)
apply(rule_tac $x = Lm1$ **in** exI, rule_tac $x = []$ **in** exI,
rule_tac $x = m$ **in** exI, simp)
apply(rule_tac $x = Suc\ (Suc\ 0)$ **in** exI,
rule_tac $x = m - 1$ **in** exI, simp)
apply(case_tac m, auto)
apply(rename_tac Lm1 m)
apply(rule_tac $x = Lm1$ **in** exI, rule_tac $x = []$ **in** exI, rule_tac $x = m$ **in** exI,
simp add: Suc_diff_le exp_ind del: replicate.simps, simp)
done

lemma inv_on_left_moving_nonempty[simp]: inv_on_left_moving (as, am) (s, [], r) ires
$=$ False
apply(simp add: inv_on_left_moving.simps inv_on_left_moving_norm.simps
inv_on_left_moving_in_middle_B.simps)
done

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
apply(simp add: inv_check_left_moving.simps inv_check_left_moving_in_middle.simps)
done

lemma start_of_lessE[elim]: $\llbracket$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 $\leq$ Suc (start_of (layout_of ap) as $+ 2 * n$)$\rrbracket$
$\Longrightarrow$ RR
using start_of_less[of e as layout_of ap] start_of_ge[of as ap n e layout_of ap]
apply(cases as $<$ e, simp)
apply(cases as $=$ e, simp, simp)
done

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 $= (\lambda$ 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(rule_tac LE = abc_dec_1_LE in halt_lemma2)
  show wf abc_dec_1_LE by(intro wf_dec_le)
next
  show Q (f 0)
    using layout fetch
    apply(simp add: f steps.simps Q dec_inv_1.simps)
    apply(subgoal_tac e > as ∨ e = as ∨ e < as)
     apply(auto simp: inv_start)
    done
next
  show ¬ P (f 0)
    using layout fetch
    apply(simp add: f steps.simps P)
    done
next
  show ∀ n. ¬ P (f n) ∧ Q (f n) ⟶ Q (f (Suc n)) ∧ (f (Suc n), f n) ∈ abc_dec_1_LE
    using fetch
  proof(rule_tac allI, rule_tac impI)
    fix na
    assume ¬ P (f na) ∧ Q (f na)
    thus Q (f (Suc na)) ∧ (f (Suc na), f na) ∈ abc_dec_1_LE
      apply(simp add: f)
      apply(cases 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) na, simp)
    proof −
      fix a b c
      assume ¬ P ((a, b, c), start_of ly as, n) ∧ Q ((a, b, c), start_of ly as, n)
      thus Q (step (a, b, c) (ci ly (start_of ly as) (Dec n e), start_of ly as − Suc 0), start_of ly as,
n) ∧
             ((step (a, b, c) (ci ly (start_of ly as) (Dec n e), start_of ly as − Suc 0), start_of ly as,
n),
               (a, b, c), start_of ly as, n) ∈ abc_dec_1_LE
        apply(simp add: Q)
        apply(cases c;cases hd c)
          apply(simp_all add: dec_inv_1.simps Let_def split: if_splits)
        using fetch layout dec_0
                     apply(auto simp: step.simps P dec_inv_1.simps Let_def abc_dec_1_LE_def
          lex_triple_def lex_pair_def)
        using dec_0
        apply(drule_tac dec_false_1, simp_all)
        done
    qed
  qed
qed

lemma crsp_step_dec_b_e_pre:
  assumes ly = layout_of ap
```

```
  and inv_start: inv_locate_b (as, lm) (n, la, ra) ires
  and dec_0: abc_lm_v lm n = 0
  and fetch: abc_fetch as ap = Some (Dec n e)
 shows ∃ stp lb rb.
    steps (Suc (start_of ly as) + 2 ∗ n, la, ra) (ci ly (start_of ly as) (Dec n e),
    start_of ly as − Suc 0) stp = (start_of ly e, lb, rb) ∧
    dec_inv_1 ly n e (as, lm) (start_of ly e, lb, rb) ires
 using assms
 apply(drule_tac crsp_step_dec_b_e_pre′, auto)
 apply(rename_tac stp a b)
 apply(rule_tac x = stp in exI, simp)
 done

lemma crsp_abc_step_via_stop[simp]:
 ⟦abc_lm_v lm n = 0;
 inv_stop (as, abc_lm_s lm n (abc_lm_v lm n)) (start_of ly e, lb, rb) ires⟧
 ⟹ crsp ly (abc_step_l (as, lm) (Some (Dec n e))) (start_of ly e, lb, rb) ires
 apply(auto simp: crsp.simps abc_step_l.simps inv_stop.simps)
 done

lemma crsp_step_dec_b_e:
 assumes layout: ly = layout_of ap
  and inv_start: inv_locate_a (as, lm) (n, l, r) ires
  and dec_0: abc_lm_v lm n = 0
  and fetch: abc_fetch as ap = Some (Dec n e)
 shows ∃ stp > 0. crsp ly (abc_step_l (as, lm) (Some (Dec n e)))
 (steps (start_of ly as + 2 ∗ n, l, r) (ci ly (start_of ly as) (Dec n e), start_of ly as − Suc 0) stp)
ires
proof −
 let ?P = ci ly (start_of ly as) (Dec n e)
 let ?off = start_of ly as − Suc 0
 have ∃ stp la ra. steps (start_of ly as + 2 ∗ n, l, r) (?P, ?off) stp = (Suc (start_of ly as) + 2∗n,
la, ra)
        ∧ inv_locate_b (as, lm) (n, la, ra) ires
  using inv_start
  apply(cases r = [] ∨ hd r = Bk, simp_all)
  done
 from this obtain stpa la ra where a:
  steps (start_of ly as + 2 ∗ n, l, r) (?P, ?off) stpa = (Suc (start_of ly as) + 2∗n, la, ra)
        ∧ inv_locate_b (as, lm) (n, la, ra) ires by blast
 have ∃ stp lb rb. steps (Suc (start_of ly as) + 2 ∗ n, la, ra) (?P, ?off) stp = (start_of ly e, lb,
rb)
        ∧ dec_inv_1 ly n e (as, lm) (start_of ly e, lb, rb) ires
  using assms a
  apply(rule_tac crsp_step_dec_b_e_pre, auto)
  done
 from this obtain stpb lb rb where b:
  steps (Suc (start_of ly as) + 2 ∗ n, la, ra) (?P, ?off) stpb = (start_of ly e, lb, rb)
        ∧ dec_inv_1 ly n e (as, lm) (start_of ly e, lb, rb) ires  by blast
 from a b show ∃ stp > 0. crsp ly (abc_step_l (as, lm) (Some (Dec n e)))
```

```
  (steps (start_of ly as + 2 * n, l, r) (?P, ?off) stp) ires
  apply(rule_tac x = stpa + stpb in exI)
  using dec_0
  apply(simp add: dec_inv_1.simps )
  apply(cases stpa, simp_all add: steps.simps)
  done
qed

fun dec_inv_2 :: layout ⇒ nat ⇒ nat ⇒ dec_inv_t
 where
  dec_inv_2 ly n e (as, am) (s, l, r) ires =
      (let ss = start_of ly as in
       let am' = abc_lm_s am n (abc_lm_v am n − Suc 0) in
       let am'' = abc_lm_s am n (abc_lm_v am n) in
        if s = 0 then False
        else if s = ss + 2 * n then
            inv_locate_a (as, am) (n, l, r) ires
        else if s = ss + 2 * n + 1 then
            inv_locate_n_b (as, am) (n, l, r) ires
        else if s = ss + 2 * n + 2 then
            dec_first_on_right_moving n (as, am'') (s, l, r) ires
        else if s = ss + 2 * n + 3 then
            dec_after_clear (as, am') (s, l, r) ires
        else if s = ss + 2 * n + 4 then
            dec_right_move (as, am') (s, l, r) ires
        else if s = ss + 2 * n + 5 then
            dec_check_right_move (as, am') (s, l, r) ires
        else if s = ss + 2 * n + 6 then
            dec_left_move (as, am') (s, l, r) ires
        else if s = ss + 2 * n + 7 then
            dec_after_write (as, am') (s, l, r) ires
        else if s = ss + 2 * n + 8 then
            dec_on_right_moving (as, am') (s, l, r) ires
        else if s = ss + 2 * n + 9 then
            dec_after_clear (as, am') (s, l, r) ires
        else if s = ss + 2 * n + 10 then
            inv_on_left_moving (as, am') (s, l, r) ires
        else if s = ss + 2 * n + 11 then
            inv_check_left_moving (as, am') (s, l, r) ires
        else if s = ss + 2 * n + 12 then
            inv_after_left_moving (as, am') (s, l, r) ires
        else if s = ss + 2 * n + 16 then
            inv_stop (as, am') (s, l, r) ires
        else False)

declare dec_inv_2.simps[simp del]
fun abc_dec_2_stage1 :: config ⇒ nat ⇒ nat ⇒ nat
 where
  abc_dec_2_stage1 (s, l, r) ss n =
        (if s ≤ ss + 2*n + 1 then 7
```

```
      else if s = ss + 2∗n + 2 then 6
      else if s = ss + 2∗n + 3 then 5
      else if s ≥ ss + 2∗n + 4 ∧ s ≤ ss + 2∗n + 9 then 4
      else if s = ss + 2∗n + 6 then 3
      else if s = ss + 2∗n + 10 ∨ s = ss + 2∗n + 11 then 2
      else if s = ss + 2∗n + 12 then 1
      else 0)

fun abc_dec_2_stage2 :: config ⇒ nat ⇒ nat ⇒ nat
  where
   abc_dec_2_stage2 (s, l, r) ss n =
     (if s ≤ ss + 2 ∗ n + 1 then (ss + 2 ∗ n + 16 − s)
      else if s = ss + 2∗n + 10 then length l
      else if s = ss + 2∗n + 11 then length l
      else if s = ss + 2∗n + 4 then length r − 1
      else if s = ss + 2∗n + 5 then length r
      else if s = ss + 2∗n + 7 then length r − 1
      else if s = ss + 2∗n + 8 then
           length r + length (takeWhile (λ a. a = Oc) l) − 1
      else if s = ss + 2∗n + 9 then
           length r + length (takeWhile (λ a. a = Oc) l) − 1
      else 0)

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

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

if $r \neq []$ $\wedge$ *hd r* $=$ *Oc then 1*
else 0
else 0)

fun *abc_dec_2_measure* :: (*config* $\times$ *nat* $\times$ *nat*) $\Rightarrow$ (*nat* $\times$ *nat* $\times$ *nat* $\times$ *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* $\times$ *nat* $\times$ *nat* $\times$ *nat*) $\times$ (*nat* $\times$ *nat* $\times$ *nat* $\times$ *nat*)) *set*
where *lex_square* $\stackrel{def}{=}$ *less_than* <∗*lex*∗> *lex_triple*

definition *abc_dec_2_LE* ::
((*config* $\times$ *nat* $\times$
nat) $\times$ (*config* $\times$ *nat* $\times$ *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*
by(*auto simp*:*abc_dec_2_LE_def lex_square_def lex_triple_def lex_pair_def*)

lemma *fix_add*: *fetch ap* ((*x*::*nat*) + 2∗*n*) *b* = *fetch ap* (2∗*n* + *x*) *b*
using *Suc_1 add.commute* **by** *metis*

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*⟧
$\Longrightarrow$ *RR*
by(*auto simp*:*gr0_conv_Suc inv_locate_n_b.simps abc_lm_v.simps split*: *if_splits*)

lemma *inv_locate_n_b_nonemptyE*[*elim*]:
⟦*0* < *abc_lm_v am n*; *inv_locate_n_b* (*as*, *am*)
(*n*, *aaa*, []) *ires*⟧ $\Longrightarrow$ *RR*
apply(*auto simp*: *inv_locate_n_b.simps abc_lm_v.simps split*: *if_splits*)
done

lemma *no_Ocs_dec_after_write*[*simp*]: *dec_after_write* (*as*, *am*) (*s*, *aa*, *r*) *ires*
$\Longrightarrow$ *takeWhile* (λ*a*. *a* = *Oc*) *aa* = []
apply(*simp only* : *dec_after_write.simps*)
apply(*erule exE*)+
apply(*erule_tac conjE*)+
apply(*cases aa*, *simp*)
apply(*cases hd aa*, *simp only*: *takeWhile.simps* , *simp_all split*: *if_splits*)
done

lemma *fewer_Ocs_dec_on_right_moving*[*simp*]:
⟦*dec_on_right_moving* (*as*, *lm*) (*s*, *aa*, []) *ires*;
length (*takeWhile* (λ*a*. *a* = *Oc*) (*tl aa*))
$\neq$ *length* (*takeWhile* (λ*a*. *a* = *Oc*) *aa*) − *Suc 0*⟧

$\Longrightarrow$ *length* (*takeWhile* ($\lambda a.\ a = Oc$) (*tl aa*)) <
length (*takeWhile* ($\lambda a.\ a = Oc$) *aa*) $-$ *Suc 0*
apply(*simp only*: *dec_on_right_moving.simps*)
apply(*erule_tac exE*)+
apply(*erule_tac conjE*)+
apply(*rename_tac lm1 lm2 m ml Mr rn*)
apply(*case_tac Mr*, *auto split*: *if_splits*)
done

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*)
apply(*simp only*: *dec_after_clear.simps*)
apply(*erule_tac exE*)+
apply(*erule conjE*)+
apply(*simp split*: *if_splits*)
done

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*)
apply(*simp add*: *dec_after_clear.simps split*: *if_splits*)
done

lemma *inv_check_left_moving_nonemptyE*[*elim*]:
inv_check_left_moving (*as*, *lm*) (*s*, [], *Oc* # *xs*) *ires*
$\Longrightarrow$ *RR*
apply(*simp add*: *inv_check_left_moving.simps inv_check_left_moving_in_middle.simps*)
done

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*
apply(*simp only*: *at_begin_norm.simps inv_locate_n_b.simps*)
apply(*erule_tac exE*)+
apply(*rename_tac lm1 lm2 rn*)
apply(*rule_tac x* = *lm1* **in** *exI*, *simp*)
apply(*case_tac length lm2*, *simp*)
apply(*case_tac lm2*, *simp*, *simp*)
apply(*case_tac lm2*, *auto simp*: *tape_of_nl_cons split*: *if_splits*)
done

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*

```
 apply(simp only: at_begin_fst_awtn.simps inv_locate_n_b.simps )
 apply(erule exE)+
 apply(rename_tac lm1 tn rn)
 apply(erule conjE)+
 apply(rule_tac x = lm1 in exI, rule_tac x = [] in exI,
    rule_tac x = Suc tn in exI, rule_tac x = 0 in exI)
 apply(simp add: exp_ind del: replicate.simps)
 apply(rule conjI)+
  apply(auto)
 done

lemma inv_locate_n_b_Oc_via_inv_locate_n_a[simp]:
 ⟦0 < abc_lm_v am n; inv_locate_a (as, am) (n, aaa, Oc # xs) ires⟧
 ⟹ inv_locate_n_b (as, am) (n, Oc#aaa, xs) ires
 apply(auto simp: inv_locate_a.simps at_begin_fst_bwtn.simps)
 done

lemma more_Oc_dec_on_right_moving[simp]:
 ⟦dec_on_right_moving (as, am) (s, aa, Bk # xs) ires;
  Suc (length (takeWhile (λa. a = Oc) (tl aa)))
  ≠ length (takeWhile (λa. a = Oc) aa)⟧
 ⟹ Suc (length (takeWhile (λa. a = Oc) (tl aa)))
  < length (takeWhile (λa. a = Oc) aa)
 apply(simp only: dec_on_right_moving.simps)
 apply(erule exE)+
 apply(rename_tac ml mr rn)
 apply(case_tac ml, auto split: if_splits )
 done

lemma crsp_step_dec_b_suc_pre:
 assumes layout: ly = layout_of ap
  and crsp: crsp ly (as, lm) (s, l, r) ires
  and inv_start: inv_locate_a (as, lm) (n, la, ra) ires
  and fetch: abc_fetch as ap = Some (Dec n e)
  and dec_suc: 0 < abc_lm_v lm n
  and f: f = (λ stp. (steps (start_of ly as + 2 * n, la, ra) (ci ly (start_of ly as) (Dec n e),
       start_of ly as − Suc 0) stp, start_of ly as, n))
  and P: P = (λ ((s, l, r), ss, x). s = start_of ly as + 2*n + 16)
  and Q: Q = (λ ((s, l, r), ss, x). dec_inv_2 ly x e (as, lm) (s, l, r) ires)
 shows ∃ stp. P (f stp) ∧ Q(f stp)
proof(rule_tac LE = abc_dec_2_LE in halt_lemma2)
 show wf abc_dec_2_LE by(intro wf_dec2_le)
next
 show Q (f 0)
  using layout fetch inv_start
  apply(simp add: f steps.simps Q)
  apply(simp only: dec_inv_2.simps)
  apply(auto simp: Let_def start_of_ge start_of_less inv_start dec_inv_2.simps)
  done
next
```

```
  show ¬ P (f 0)
    using layout fetch
    apply(simp add: f steps.simps P)
    done
next
  show ∀ n. ¬ P (f n) ∧ Q (f n) ⟶ Q (f (Suc n)) ∧ (f (Suc n), f n) ∈ abc_dec_2_LE
    using fetch
  proof(rule_tac allI, rule_tac impI)
    fix na
    assume ¬ P (f na) ∧ Q (f na)
    thus Q (f (Suc na)) ∧ (f (Suc na), f na) ∈ abc_dec_2_LE
      apply(simp add: f)
      apply(cases steps ((start_of ly as + 2 * n), la, ra)
        (ci ly (start_of ly as) (Dec n e), start_of ly as − Suc 0) na, simp)
    proof −
      fix a b c
      assume ¬ P ((a, b, c), start_of ly as, n) ∧ Q ((a, b, c), start_of ly as, n)
      thus Q (step (a, b, c) (ci ly (start_of ly as) (Dec n e), start_of ly as − Suc 0), start_of ly as,
n) ∧
           ((step (a, b, c) (ci ly (start_of ly as) (Dec n e), start_of ly as − Suc 0), start_of ly as,
n),
              (a, b, c), start_of ly as, n) ∈ abc_dec_2_LE
        apply(simp add: Q)
        apply(erule_tac conjE)
        apply(cases c; cases hd c)
          apply(simp_all add: dec_inv_2.simps Let_def)
          apply(simp_all split: if_splits)
        using fetch layout dec_suc
                          apply(auto simp: step.simps P dec_inv_2.simps Let_def abc_dec_2_LE_def
lex_triple_def lex_pair_def lex_square_def
          fix_add numeral_3_eq_3)
        done
    qed
  qed
qed

lemma crsp_abc_step_l_start_of[simp]:
  ⟦inv_stop (as, abc_lm_s lm n (abc_lm_v lm n − Suc 0))
  (start_of (layout_of ap) as + 2 * n + 16, a, b) ires;
   abc_lm_v lm n > 0;
   abc_fetch as ap = Some (Dec n e)⟧
  ⟹ crsp (layout_of ap) (abc_step_l (as, lm) (Some (Dec n e)))
  (start_of (layout_of ap) as + 2 * n + 16, a, b) ires
  by(auto simp: inv_stop.simps crsp.simps  abc_step_l.simps startof_Suc2)

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 ∃ 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
 using assms
 apply(drule_tac crsp_step_dec_b_suc_pre, auto)
 apply(rename_tac stp a b)
 apply(rule_tac x = stp in exI)
 apply(case_tac stp, simp_all add: steps.simps dec_inv_2.simps)
 done

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 ∃ 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
 using assms
 apply(cases abc_lm_v lm n = 0)
  apply(rule_tac crsp_step_dec_b_e, simp_all)
 apply(rule_tac crsp_step_dec_b_suc, simp_all)
 done

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(simp add: ci.simps)
 let ?off = start_of ly as − Suc 0
 let ?A = findnth n
 let ?B = adjust (shift (shift tdec_b (2 ∗ n)) ?off) (start_of ly e)
 have ∃ stp la ra. steps (s, l, r) (shift ?A ?off @ ?B, ?off) stp = (start_of ly as + 2∗n, la, ra)
          ∧ inv_locate_a (as, lm) (n, la, ra) ires
 proof −
  have ∃ stp l′ r′. steps (Suc 0, l, r) (?A, 0) stp = (Suc (2 ∗ n), l′, r′) ∧
           inv_locate_a (as, lm) (n, l′, r′) ires
   using assms
   apply(rule_tac findnth_correct, simp_all)
   done
  then obtain stp l′ r′ where a:
   steps (Suc 0, l, r) (?A, 0) stp = (Suc (2 ∗ n), l′, r′) ∧
   inv_locate_a (as, lm) (n, l′, r′) ires by blast
  then have steps (Suc 0 + ?off, l, r) (shift ?A ?off, ?off) stp = (Suc (2 ∗ n) + ?off, l′, r′)
   apply(rule_tac tm_shift_eq_steps, simp_all)
   done
  moreover have s = start_of ly as
```

```
    using crsp
    apply(auto simp: crsp.simps)
    done
  ultimately show ∃ stp la ra. steps (s, l, r) (shift ?A ?off @ ?B, ?off) stp = (start_of ly as +
2∗n, la, ra)
               ∧ inv_locate_a (as, lm) (n, la, ra) ires
    using a
    apply(drule_tac B = ?B in tm_append_first_steps_eq, auto)
    apply(rule_tac x = stp in exI, simp)
    done
 qed
 from this obtain stpa la ra where a:
  steps (s, l, r) (shift ?A ?off @ ?B, ?off) stpa = (start_of ly as + 2∗n, la, ra)
               ∧ inv_locate_a (as, lm) (n, la, ra) ires by blast
 have ∃ stp. crsp ly (abc_step_l (as, lm) (Some (Dec n e)))
        (steps (start_of ly as + 2∗n, la, ra) (shift ?A ?off @ ?B, ?off) stp) ires ∧ stp > 0
  using assms a
  apply(drule_tac crsp_step_dec_b, auto)
  apply(rename_tac stp)
  apply(rule_tac x = stp in exI, simp add: ci.simps)
  done
 then obtain stpb where b:
  crsp ly (abc_step_l (as, lm) (Some (Dec n e)))
  (steps (start_of ly as + 2∗n, la, ra) (shift ?A ?off @ ?B, ?off) stpb) ires ∧ stpb > 0 ..
 from a b show ∃ stp>0. crsp ly (abc_step_l (as, lm) (Some (Dec n e)))
  (steps (s, l, r) (shift ?A ?off @ ?B, ?off) stp) ires
  apply(rule_tac x = stpa + stpb in exI)
  apply(simp)
  done
qed
```

9.5 Crsp of Goto

```
lemma crsp_step_goto:
 assumes layout: ly = layout_of ap
  and crsp: crsp ly (as, lm) (s, l, r) ires
 shows ∃ 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
 using crsp
 apply(rule_tac x = Suc 0 in exI)
 apply(cases r;cases hd r)
  apply(simp_all add: ci.simps steps.simps step.simps crsp.simps fetch.simps abc_step_l.simps)
 done

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
 using assms
 apply(cases ins, simp_all)
  apply(rule crsp_step_inc, simp_all)
  apply(rule crsp_step_dec, simp_all)
 apply(rule_tac crsp_step_goto, simp_all)
 done

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 −
 have ∃ 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
  using assms
  apply(rule_tac crsp_step_in, simp_all)
  done
 from this obtain stp where d: 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 ..
 obtain s′ l′ r′ where e:
  (steps (s, l, r) (ci ly (start_of ly as) ins, start_of ly as − 1) stp) = (s′, l′, r′)
  apply(cases (steps (s, l, r) (ci ly (start_of ly as) ins, start_of ly as − 1) stp))
  by blast
 then have steps (s, l, r) (tp, 0) stp = (s′, l′, r′)
  using assms d
  apply(rule_tac steps_eq_in)
    apply(simp_all)
  apply(cases (abc_step_l (as, lm) (Some ins)), simp add: crsp.simps)
  done
 thus ∃stp>0. crsp ly (abc_step_l (as, lm) (Some ins)) (steps (s, l, r) (tp, 0) stp) ires
  using d e
  apply(rule_tac x = stp in exI, simp)
  done
qed

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
 using crsp
proof(induct n)
 case 0
 then show ?case apply(rule_tac x = 0 in exI)
```

```
  by(simp add: steps.simps abc_steps_l.simps)
next
 case (Suc n)
 then obtain stp where crsp ly (abc_steps_l (as, lm) ap n) (steps0 (s, l, r) tp stp) ires
  by blast
 thus ?case
  apply(cases (abc_steps_l (as, lm) ap n), auto)
  apply(frule_tac abc_step_red, simp)
  apply(cases abc_fetch (fst (abc_steps_l (as, lm) ap n)) ap, simp add: abc_step_l.simps, auto)
  apply(cases steps (s, l, r) (tp, 0) stp, simp)
  using assms
  apply(drule_tac s = fst (steps0 (s, l, r) (tm_of ap) stp)
    and l = fst (snd (steps0 (s, l, r) (tm_of ap) stp))
    and r = snd (snd (steps0 (s, l, r) (tm_of ap) stp)) in crsp_step, auto)
  by (metis steps_add)
qed


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)
 using assms
 apply(drule_tac n = stp in crsp_steps, auto)
 apply(rename_tac stpA)
 apply(rule_tac x = stpA in exI)
 apply(case_tac steps (Suc 0, l, r) (tm_of ap, 0) stpA, simp add: crsp.simps)
 done
```

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]
 apply(simp add: layout_of.simps)
 done

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])
 apply(auto)
 apply(simp add: layout_of.simps start_of.simps)
 done

lemma tpairs_id_cons:
 tpairs_of (xs @ [x]) = tpairs_of xs @ [(start_of (layout_of (xs @ [x])) (length xs), x)]
 apply(auto simp: tpairs_of.simps layout_id_cons )
 done
```

```
lemma map_length_ci:
  (map (length ∘ (λ(xa, y). ci (layout_of xs @ [length_of x]) xa y)) (tpairs_of xs)) =
  (map (length ∘ (λ(x, y). ci (layout_of xs) x y)) (tpairs_of xs))
  apply(auto simp: ci.simps adjust.simps) apply(rename_tac A B)
  apply(case_tac B, auto simp: ci.simps adjust.simps)
  done

lemma length_tp'[simp]:
  ⟦ly = layout_of ap; tp = tm_of ap⟧ ⟹
     length tp = 2 * sum_list (take (length ap) (layout_of ap))
proof(induct ap arbitrary: ly tp rule: rev_induct)
  case Nil
  thus ?case
    by(simp add: tms_of.simps tm_of.simps tpairs_of.simps)
next
  fix x xs ly tp
  assume ind: ⋀ly tp. ⟦ly = layout_of xs; tp = tm_of xs⟧ ⟹
    length tp = 2 * sum_list (take (length xs) (layout_of xs))
    and layout: ly = layout_of (xs @ [x])
    and tp: tp = tm_of (xs @ [x])
  obtain ly' where a: ly' = layout_of xs
    by metis
  obtain tp' where b: tp' = tm_of xs
    by metis
  have c: length tp' = 2 * sum_list (take (length xs) (layout_of xs))
    using a b
    by(erule_tac ind, simp)
  thus length tp = 2 *
    sum_list (take (length (xs @ [x])) (layout_of (xs @ [x])))
    using tp b
   apply(auto simp: layout_id_cons tm_of.simps tms_of.simps length_concat tpairs_id_cons map_length_ci)
    apply(cases x)
      apply(auto simp: ci.simps tinc_b_def tdec_b_def length_findnth adjust.simps length_of.simps
        split: abc_inst.splits)
    done
qed

lemma length_tp:
  ⟦ly = layout_of ap; tp = tm_of ap⟧ ⟹
  start_of ly (length ap) = Suc (length tp div 2)
  apply(frule_tac length_tp', simp_all)
  apply(simp add: start_of.simps)
  done

lemma compile_correct_halt:
  assumes layout: ly = layout_of ap
    and compile: tp = tm_of ap
    and crsp: crsp ly (0, lm) (Suc 0, l, r) ires
    and abc_halt: abc_steps_l (0, lm) ap stp = (length ap, am)
    and rs_loc: n < length am
```

```
    and rs: abc_lm_v am n = rs
    and off: off = length tp div 2
  shows ∃ stp i j. steps (Suc 0, l, r) (tp @ shift (mopup n) off, 0) stp = (0, Bk↑i @ Bk # Bk #
ires, Oc↑Suc rs @ Bk↑j)
proof −
  have ∃ stp k. steps (Suc 0, l, r) (tp, 0) stp = (Suc off, Bk # Bk # ires, <am> @ Bk↑k)
    using assms tp_correct′[of ly ap tp lm l r ires stp am]
    by(simp add: length_tp)
  then obtain stp k where steps (Suc 0, l, r) (tp, 0) stp = (Suc off, Bk # Bk # ires, <am> @
Bk↑k)
    by blast
  then have a: steps (Suc 0, l, r) (tp@shift (mopup n) off , 0) stp = (Suc off, Bk # Bk # ires,
<am> @ Bk↑k)
    using assms
    by(auto intro: tm_append_first_steps_eq)
  have ∃ stp i j. (steps (Suc 0, Bk # Bk # ires, <am> @ Bk ↑ k) (mopup_a n @ shift mopup_b
(2 ∗ n), 0) stp)
    = (0, Bk↑i @ Bk # Bk # ires, Oc # Oc↑ rs @ Bk↑j)
    using assms
    by(rule_tac mopup_correct, auto simp: abc_lm_v.simps)
  then obtain stpb i j where
    steps (Suc 0, Bk # Bk # ires, <am> @ Bk ↑ k) (mopup_a n @ shift mopup_b (2 ∗ n), 0) stpb
    = (0, Bk↑i @ Bk # Bk # ires, Oc # Oc↑ rs @ Bk↑j) by blast
  then have b: steps (Suc 0 + off, Bk # Bk # ires, <am> @ Bk ↑ k) (tp @ shift (mopup n) off,
0) stpb
    = (0, Bk↑i @ Bk # Bk # ires, Oc # Oc↑ rs @ Bk↑j)
    using assms wf_mopup
    apply(drule_tac tm_append_second_halt_eq, auto)
    done
  from a b show ?thesis
    by(rule_tac x = stp + stpb in exI, simp add: steps_add)
qed

declare mopup.simps[simp del]
lemma abc_step_red2:
  abc_steps_l (s, lm) p (Suc n) = (let (as′, am′) = abc_steps_l (s, lm) p n in
                                  abc_step_l (as′, am′) (abc_fetch as′ p))
  apply(cases abc_steps_l (s, lm) p n, simp)
  apply(drule_tac abc_step_red, simp)
  done

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

```
proof(induct stp arbitrary: as am)
 case 0
 thus ?case
  using crsp
  by(rule_tac x = 0 in exI, auto simp: abc_steps_l.simps steps.simps crsp)
next
 case (Suc stp as am)
 have ind:
  ⋀ as am. ⟦as < length ap; abc_steps_l (0, lm) ap stp = (as, am)⟧
  ⟹ ∃ stpa≥stp. crsp ly (as, am) (steps (Suc 0, l, r) (tp, 0) stpa) ires by fact
 have a: as < length ap by fact
 have b: abc_steps_l (0, lm) ap (Suc stp) = (as, am) by fact
 obtain as' am' where c: abc_steps_l (0, lm) ap stp = (as', am')
  by(cases abc_steps_l (0, lm) ap stp, auto)
 then have d: as' < length ap
  using a b
  by(simp add: abc_step_red2, cases as' < length ap, simp,
    simp add: abc_fetch.simps abc_steps_l.simps abc_step_l.simps)
 have ∃ stpa≥stp. crsp ly (as', am') (steps (Suc 0, l, r) (tp, 0) stpa) ires
  using d c ind by simp
 from this obtain stpa where e:
  stpa ≥ stp ∧  crsp ly (as', am') (steps (Suc 0, l, r) (tp, 0) stpa) ires
  by blast
 obtain s' l' r' where f: steps (Suc 0, l, r) (tp, 0) stpa = (s', l', r')
  by(cases steps (Suc 0, l, r) (tp, 0) stpa, auto)
 obtain ins where g: abc_fetch as' ap = Some ins using d
  by(cases abc_fetch as' ap,auto simp: abc_fetch.simps)
 then have ∃ stp> (0::nat). crsp ly (abc_step_l (as', am') (Some ins))
  (steps (s', l', r') (tp, 0) stp) ires
  using layout compile e f
  by(rule_tac crsp_step, simp_all)
 then obtain stpb where stpb > 0 ∧ crsp ly (abc_step_l (as', am') (Some ins))
  (steps (s', l', r') (tp, 0) stpb) ires ..
 from this show ?case using b e g f c
  by(rule_tac x = stpa + stpb in exI, simp add: steps_add abc_step_red2)
qed

lemma compile_correct_unhalt:
 assumes layout: ly = layout_of ap
  and compile: tp = tm_of ap
  and crsp: crsp ly (0, lm) (1, l, r) ires
  and off: off = length tp div 2
  and abc_unhalt: ∀ stp. (λ (as, am). as < length ap) (abc_steps_l (0, lm) ap stp)
 shows ∀ stp.¬ is_final (steps (1, l, r) (tp @ shift (mopup n) off, 0) stp)
 using assms
proof(rule_tac allI, rule_tac notI)
 fix stp
 assume h: is_final (steps (1, l, r) (tp @ shift (mopup n) off, 0) stp)
 obtain as am where a: abc_steps_l (0, lm) ap stp = (as, am)
  by(cases abc_steps_l (0, lm) ap stp, auto)
```

```
  then have b: as < length ap
   using abc_unhalt
   by(erule_tac x = stp in allE, simp)
  have ∃ stpa≥stp. crsp ly (as, am) (steps (1, l, r) (tp, 0) stpa) ires
   using assms b a
   apply(simp add: numeral)
   apply(rule_tac crsp_steps2)
     apply(simp_all)
   done
  then obtain stpa where
   stpa≥stp ∧ crsp ly (as, am) (steps (1, l, r) (tp, 0) stpa) ires ..
  then obtain s' l' r' where b: (steps (1, l, r) (tp, 0) stpa) = (s', l', r') ∧
     stpa≥stp ∧ crsp ly (as, am) (s', l', r') ires
   by(cases steps (1, l, r) (tp, 0) stpa, auto)
  hence c:
   (steps (1, l, r) (tp @ shift (mopup n) off, 0) stpa) = (s', l', r')
   by(rule_tac tm_append_first_steps_eq, simp_all add: crsp.simps)
  from b have d: s' > 0 ∧ stpa ≥ stp
   by(simp add: crsp.simps)
  then obtain diff where e: stpa = stp + diff by (metis le_iff_add)
  obtain s'' l'' r'' where f:
   steps (1, l, r) (tp @ shift (mopup n) off, 0) stp = (s'', l'', r'') ∧ is_final (s'', l'', r'')
   using h
   by(cases steps (1, l, r) (tp @ shift (mopup n) off, 0) stp, auto)

  then have is_final (steps (s'', l'', r'') (tp @ shift (mopup n) off, 0) diff)
   by(auto intro: after_is_final)
  then have is_final (steps (1, l, r) (tp @ shift (mopup n) off, 0) stpa)
   using e f by simp
  from this and c d show False by simp
qed

end
```

10 Alternative Definitions for Translating Abacus Machines to TMs

```
theory Abacus_Defs
  imports Abacus
begin

abbreviation
  layout ≝ layout_of

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

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

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

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

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

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

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

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

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

lemma
compile ap s i = *ci* (*layout ap*) *s i*
apply(*cases i*)
apply(*simp add*: *ci*.*simps shift*.*simps start_of*.*simps tinc_b_def*)
apply(*simp add*: *ci*.*simps shift*.*simps start_of*.*simps tdec_b_def*)
apply(*simp add*: *ci*.*simps shift*.*simps start_of*.*simps*)
done

end

```
theory Rec_Def
  imports Main
begin

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

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

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

termination
  apply(relation measures [λ (r, xs). size r, (λ (r, xs). last xs)])
        apply(auto simp add: less_Suc_eq_le intro: trans_le_add2 size_list_estimation'[THEN
trans_le_add1])
  done

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

end

theory *Abacus_Hoare*
  imports *Abacus*
begin

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

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

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

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

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

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

lemma *abc_Hoare_haltI*:
  assumes ⋀*lm*. *P lm* ⟹ ∃ *n*. *abc_final* (*abc_steps_l* (0, *lm*) *p n*) *p* ∧ *Q abc_holds_for* (*abc_steps_l* (0, *lm*) *p n*)
  shows {*P*} (*p*::*abc_prog*) {*Q*}
  unfolding *abc_Hoare_halt_def*
  using *assms* **by** *auto*

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

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

where
$abc\_Hoare\_unhalt\ P\ p \stackrel{def}{=} \forall\, args.\ P\ args \longrightarrow (\forall\ n\ .abc\_notfinal\ (abc\_steps\_l\ (0, args)\ p\ n)\ p)$

lemma *abc_Hoare_unhaltI*:
assumes $\bigwedge args\ n.\ P\ args \Longrightarrow abc\_notfinal\ (abc\_steps\_l\ (0, args)\ p\ n)\ p$
shows $\{P\}\ (p{::}abc\_prog) \uparrow$
unfolding *abc_Hoare_unhalt_def*
using *assms* **by** *auto*

fun $abc\_inst\_shift :: abc\_inst \Rightarrow nat \Rightarrow abc\_inst$
where
$abc\_inst\_shift\ (Inc\ m)\ n = Inc\ m\ |$
$abc\_inst\_shift\ (Dec\ m\ e)\ n = Dec\ m\ (e + n)\ |$
$abc\_inst\_shift\ (Goto\ m)\ n = Goto\ (m + n)$

fun $abc\_shift :: abc\_inst\ list \Rightarrow nat \Rightarrow abc\_inst\ list$
where
$abc\_shift\ xs\ n = map\ (\lambda\, x.\ abc\_inst\_shift\ x\ n)\ xs$

fun $abc\_comp :: abc\_inst\ list \Rightarrow abc\_inst\ list \Rightarrow$
$abc\_inst\ list$ (**infixl** $[+]$ *99*)
where
$abc\_comp\ al\ bl = (let\ al\_len = length\ al\ in$
$al\ @\ abc\_shift\ bl\ al\_len)$

lemma *abc_comp_first_step_eq_pre*:
$s < length\ A$
$\Longrightarrow abc\_step\_l\ (s, lm)\ (abc\_fetch\ s\ (A\ [+]\ B)) =$
$abc\_step\_l\ (s, lm)\ (abc\_fetch\ s\ A)$
by(*simp add*: *abc_step_l.simps abc_fetch.simps nth_append*)

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(*induct n*)
case *0*
thus *?thesis*
by(*simp add*: *abc_steps_l.simps*)
next
case (*Suc n*)
have *ind*: $[\![abc\_final\ (abc\_steps\_l\ (0, lm)\ p\ n)\ p;\ p \neq []]\!] \Longrightarrow$
$\exists n'.\ abc\_notfinal\ (abc\_steps\_l\ (0, lm)\ p\ n')\ p \wedge abc\_final\ (abc\_steps\_l\ (0, lm)\ p\ (Suc\ n'))\ p$
by *fact*
have *final*: $abc\_final\ (abc\_steps\_l\ (0, lm)\ p\ (Suc\ n))\ p$ **by** *fact*
have *notnull*: $p \neq []$ **by** *fact*
show *?thesis*
proof(*cases* $abc\_final\ (abc\_steps\_l\ (0, lm)\ p\ n)\ p$)
case *True*

```
    have abc_final (abc_steps_l (0, lm) p n) p by fact
    then have ∃n'. abc_notfinal (abc_steps_l (0, lm) p n') p ∧ abc_final (abc_steps_l (0, lm) p
(Suc n')) p
      using ind notnull
      by simp
    thus ?thesis
      by simp
  next
    case False
    have ¬ abc_final (abc_steps_l (0, lm) p n) p by fact
    from final this have abc_notfinal (abc_steps_l (0, lm) p n) p
      by(case_tac abc_steps_l (0, lm) p n, simp add: abc_step_red2
          abc_step_l.simps abc_fetch.simps split: if_splits)
    thus ?thesis
      using final
      by(rule_tac x = n in exI, simp)
  qed
qed

lemma notfinal_Suc:
  abc_notfinal (abc_steps_l (0, lm) A (Suc n)) A ⟹
  abc_notfinal (abc_steps_l (0, lm) A n) A
  apply(case_tac abc_steps_l (0, lm) A n)
  apply(simp add: abc_step_red2 abc_fetch.simps abc_step_l.simps split: if_splits)
  done

lemma abc_comp_frist_steps_eq_pre:
  assumes notfinal: abc_notfinal (abc_steps_l (0, lm)  A n) A
    and notnull: A ≠ []
  shows abc_steps_l (0, lm) (A [+] B) n = abc_steps_l (0, lm) A n
  using notfinal
proof(induct n)
  case 0
  thus ?case
    by(simp add: abc_steps_l.simps)
next
  case (Suc n)
   have ind: abc_notfinal (abc_steps_l (0, lm) A n) A ⟹ abc_steps_l (0, lm) (A [+] B) n =
abc_steps_l (0, lm) A n
    by fact
  have h: abc_notfinal (abc_steps_l (0, lm) A (Suc n)) A by fact
  then have a: abc_notfinal (abc_steps_l (0, lm) A n) A
    by(simp add: notfinal_Suc)
  then have b: abc_steps_l (0, lm) (A [+] B) n = abc_steps_l (0, lm) A n
    using ind by simp
  obtain s lm' where c: abc_steps_l (0, lm) A n = (s, lm')
    by (metis prod.exhaust)
  then have d: s < length A ∧ abc_steps_l (0, lm) (A [+] B) n = (s, lm')
    using a b by simp
  thus ?case
```

```
  using c
  by(simp add: abc_step_red2 abc_fetch.simps abc_step_l.simps nth_append)
qed

declare abc_shift.simps[simp del] abc_comp.simps[simp del]
lemma halt_steps2: st ≥ length A ⟹ abc_steps_l (st, lm) A stp = (st, lm)
 apply(induct stp)
 by(simp_all add: abc_step_red2 abc_steps_l.simps abc_step_l.simps abc_fetch.simps)

lemma halt_steps: abc_steps_l (length A, lm) A n = (length A, lm)
 apply(induct n, simp add: abc_steps_l.simps)
 apply(simp add: abc_step_red2 abc_step_l.simps nth_append abc_fetch.simps)
 done

lemma abc_steps_add:
 abc_steps_l (as, lm) ap (m + n) =
      abc_steps_l (abc_steps_l (as, lm) ap m) ap n
 apply(induct m arbitrary: n as lm, simp add: abc_steps_l.simps)
proof −
 fix m n as lm
 assume ind:
  ⋀n as lm. abc_steps_l (as, lm) ap (m + n) =
            abc_steps_l (abc_steps_l (as, lm) ap m) ap n
 show abc_steps_l (as, lm) ap (Suc m + n) =
        abc_steps_l (abc_steps_l (as, lm) ap (Suc m)) ap n
  apply(insert ind[of as lm Suc n], simp)
  apply(insert ind[of as lm Suc 0], simp add: abc_steps_l.simps)
  apply(case_tac (abc_steps_l (as, lm) ap m), simp)
  apply(simp add: abc_steps_l.simps)
  apply(case_tac abc_step_l (a, b) (abc_fetch a ap),
     simp add: abc_steps_l.simps)
  done
qed

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(cases na > nb)
 case True
 then obtain d where na = nb + d
  by (metis add_Suc_right less_iff_Suc_add)
 thus ?thesis using assms halt_steps
  by(simp add: abc_steps_add)
next
 case False
 then obtain d where nb = na + d
  by (metis add.comm_neutral less_imp_add_positive nat_neq_iff)
 thus ?thesis using assms halt_steps
  by(simp add: abc_steps_add)
```

```
qed

lemma abc_comp_frist_steps_halt_eq':
  assumes final: abc_steps_l (0, lm) A n = (length A, lm')
    and notnull: A ≠ []
  shows ∃ n'. abc_steps_l (0, lm) (A [+] B) n' = (length A, lm')
proof -
  have ∃ n'. abc_notfinal (abc_steps_l (0, lm) A n') A ∧
    abc_final (abc_steps_l (0, lm) A (Suc n')) A
    using assms
    by(rule_tac n = n in abc_before_final, simp_all)
  then obtain na where a:
    abc_notfinal (abc_steps_l (0, lm) A na) A ∧
        abc_final (abc_steps_l (0, lm) A (Suc na)) A ..
  obtain sa lma where b: abc_steps_l (0, lm) A na = (sa, lma)
    by (metis prod.exhaust)
  then have c: abc_steps_l (0, lm) (A [+] B) na = (sa, lma)
    using a abc_comp_frist_steps_eq_pre[of lm A na B] assms
    by simp
  have d: sa < length A using b a by simp
  then have e: abc_step_l (sa, lma) (abc_fetch sa (A [+] B)) =
    abc_step_l (sa, lma) (abc_fetch sa A)
    by(rule_tac abc_comp_first_step_eq_pre)
  from a have abc_steps_l (0, lm) A (Suc na) = (length A, lm')
    using final equal_when_halt
    by(case_tac abc_steps_l (0, lm) A (Suc na) , simp)
  then have abc_steps_l (0, lm) (A [+] B) (Suc na) = (length A, lm')
    using a b c e
    by(simp add: abc_step_red2)
  thus ?thesis
    by blast
qed

lemma abc_exec_null: abc_steps_l sam [] n = sam
  apply(cases sam)
  apply(induct n)
   apply(auto simp: abc_step_red2)
   apply(auto simp: abc_step_l.simps abc_steps_l.simps abc_fetch.simps)
  done

lemma abc_comp_frist_steps_halt_eq:
  assumes final: abc_steps_l (0, lm) A n = (length A, lm')
  shows ∃ n'. abc_steps_l (0, lm) (A [+] B) n' = (length A, lm')
  using final
  apply(case_tac A = [])
   apply(rule_tac x = 0 in exI, simp add: abc_steps_l.simps abc_exec_null)
  apply(rule_tac abc_comp_frist_steps_halt_eq', simp_all)
  done
```

```
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)
  using assms
  apply(auto simp: abc_step_l.simps abc_fetch.simps nth_append abc_comp.simps abc_shift.simps
split : if_splits )
  apply(case_tac [!] B ! s, auto simp: Let_def)
  done

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')
  using assms
proof(induct n arbitrary: sa lm')
  case 0
  thus ?case
    by(simp add: abc_steps_l.simps)
next
  case (Suc n)
  have ind: ⋀sa lm'. abc_steps_l (0, lm) B n = (sa, lm') ⟹
    abc_steps_l (length A, lm) (A [+] B) n = (sa + length A, lm') by fact
  have exec: abc_steps_l (0, lm) B (Suc n) = (sa, lm') by fact
  obtain sb lmb where a:  abc_steps_l (0, lm) B n = (sb, lmb)
    by (metis prod.exhaust)
  then have abc_steps_l (length A, lm) (A [+] B) n = (sb + length A, lmb)
    using ind by simp
  moreover have abc_step_l (sb + length A, lmb) (abc_fetch (sb + length A) (A [+] B)) = (sa
+ length A, lm')
    using a exec abc_comp_second_step_eq
    by(simp add: abc_step_red2)
  ultimately show ?case
    by(simp add: abc_step_red2)
qed

lemma length_abc_comp[simp, intro]:
  length (A [+] B) = length A + length B
  by(auto simp: abc_comp.simps abc_shift.simps)

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(rule_tac abc_Hoare_haltI)
  fix lm
  assume a: P lm
  then obtain na lma where
    abc_final (abc_steps_l (0, lm) A na) A
    and b: abc_steps_l (0, lm) A na = (length A, lma)
    and c: Q abc_holds_for (length A, lma)
```

using *A_halt* **unfolding** *abc_Hoare_halt_def*
by (*metis* (*full_types*) *abc_final.simps abc_holds_for.simps prod.exhaust*)
have $\exists$ *n*. *abc_steps_l* (*0*, *lm*) (*A* [+] *B*) *n* = (*length A*, *lma*)
using *abc_comp_frist_steps_halt_eq b*
by(*simp*)
then obtain *nx* **where** *h1*: *abc_steps_l* (*0*, *lm*) (*A* [+] *B*) *nx* = (*length A*, *lma*) **..**
from *c* **have** *Q lma*
using *c* **unfolding** *abc_holds_for.simps*
by *simp*
then obtain *nb lmb* **where**
abc_final (*abc_steps_l* (*0*, *lma*) *B nb*) *B*
and *d*: *abc_steps_l* (*0*, *lma*) *B nb* = (*length B*, *lmb*)
and *e*: *S abc_holds_for* (*length B*, *lmb*)
using *B_halt* **unfolding** *abc_Hoare_halt_def*
by (*metis* (*full_types*) *abc_final.simps abc_holds_for.simps prod.exhaust*)
have *h2*: *abc_steps_l* (*length A*, *lma*) (*A* [+] *B*) *nb* = (*length B* + *length A*, *lmb*)
using *d abc_comp_second_steps_eq*
by *simp*
thus $\exists$ *n*. *abc_final* (*abc_steps_l* (*0*, *lm*) (*A* [+] *B*) *n*) (*A* [+] *B*) $\wedge$
S abc_holds_for abc_steps_l (*0*, *lm*) (*A* [+] *B*) *n*
using *h1 e*
by(*rule_tac x* = *nx* + *nb* **in** *exI*, *simp add*: *abc_steps_add*)
qed

lemma *abc_unhalt_append_eq*:
assumes *unhalt*: {*P*} (*A*::*abc_prog*) $\uparrow$
and *P*: *P args*
shows *abc_steps_l* (*0*, *args*) (*A* [+] *B*) *stp* = *abc_steps_l* (*0*, *args*) *A stp*
proof(*induct stp*)
case *0*
thus *?case*
by(*simp add*: *abc_steps_l.simps*)
next
case (*Suc stp*)
have *ind*: *abc_steps_l* (*0*, *args*) (*A* [+] *B*) *stp* = *abc_steps_l* (*0*, *args*) *A stp*
by *fact*
obtain *s nl* **where** *a*: *abc_steps_l* (*0*, *args*) *A stp* = (*s*, *nl*)
by (*metis prod.exhaust*)
then have *b*: *s* < *length A*
using *unhalt P*
apply(*auto simp*: *abc_Hoare_unhalt_def*)
by (*metis abc_notfinal.simps*)
thus *?case*
using *a ind*
by(*simp add*: *abc_step_red2 abc_step_l.simps abc_fetch.simps nth_append abc_comp.simps*)
qed

lemma *abc_Hoare_plus_unhalt1*:
{*P*} (*A*::*abc_prog*) $\uparrow \Longrightarrow$ {*P*} (*A* [+] *B*) $\uparrow$
apply(*rule abc_Hoare_unhaltI*)

```
  apply(subst abc_unhalt_append_eq,force,force)
  by (metis (mono_tags, lifting) abc_notfinal.elims(3) abc_notfinal.simps add_diff_inverse_nat
    abc_Hoare_unhalt_def le_imp_less_Suc length_abc_comp not_less_eq order_refl trans_le_add1)

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
  apply(subgoal_tac ∃ d. x = y + d, auto)
   apply(cases abc_steps_l (0, args) A y,simp)
   apply(rule classical, simp add: abc_steps_add leI halt_steps2)
  by arith

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)
proof −
  obtain st nl stp where a: abc_final (abc_steps_l (0, args) A stp) A
    and b: Q abc_holds_for (length A, nl)
    and c: abc_steps_l (0, args) A stp = (st, nl)
    using halt P unfolding abc_Hoare_halt_def
    by (metis abc_holds_for.simps prod.exhaust)
  obtain stpa where d:
    abc_notfinal (abc_steps_l (0, args) A stpa) A ∧ abc_final (abc_steps_l (0, args) A (Suc stpa))
A
    using abc_before_final[of args A stp,OF a notnull] by metis
  thus ?thesis
  proof(cases n < Suc stpa)
    case True
    have h: n < Suc stpa by fact
    then have abc_notfinal (abc_steps_l (0, args) A n) A
      using d
      by(rule_tac notfinal_all_before, auto)
    moreover then have abc_steps_l (0, args) (A [+] B) n = abc_steps_l (0, args) A n
      using notnull
      by(rule_tac abc_comp_frist_steps_eq_pre, simp_all)
    ultimately show ?thesis
      by(case_tac abc_steps_l (0, args) A n, simp)
  next
    case False
    have ¬ n < Suc stpa by fact
    then obtain d where i1: n = Suc stpa + d
      by (metis add_Suc less_iff_Suc_add not_less_eq)
    have abc_steps_l (0, args) A (Suc stpa) = (length A, nl)
      using d a c
      apply(case_tac abc_steps_l (0, args) A stp, simp add: equal_when_halt)
      by(case_tac abc_steps_l (0, args) A (Suc stpa), simp add: equal_when_halt)
    moreover have  abc_steps_l (0, args) (A [+] B) stpa = abc_steps_l (0, args) A stpa
```

```
      using notnull d
      by(rule_tac abc_comp_frist_steps_eq_pre, simp_all)
    ultimately have i2: abc_steps_l (0, args) (A [+] B) (Suc stpa) = (length A, nl)
      using d
      apply(case_tac abc_steps_l (0, args) A stpa, simp)
     by(simp add: abc_step_red2 abc_steps_l.simps abc_fetch.simps abc_comp.simps nth_append)
    obtain s' nl' where i3:abc_steps_l (0, nl) B d = (s', nl')
      by (metis prod.exhaust)
    then have i4: abc_steps_l (0, args) (A [+] B) (Suc stpa + d) = (length A + s', nl')
      using i2  apply(simp only: abc_steps_add)
      using abc_comp_second_steps_eq[of nl B d s' nl']
      by simp
    moreover have s' < length B
      using unhalt b i3
      apply(simp add: abc_Hoare_unhalt_def)
      apply(erule_tac x = nl in allE, simp)
      by(erule_tac x = d in allE, simp)
    ultimately show ?thesis
      using i1
      by(simp)
  qed
qed

lemma abc_comp_null_left[simp]: [] [+] A = A
proof(induct A)
  case (Cons a A)
  then show ?case
   apply(cases a)
    by(auto simp: abc_comp.simps abc_shift.simps)
qed (auto simp: abc_comp.simps abc_shift.simps)

lemma abc_comp_null_right[simp]: A [+] [] = A
proof(induct A)
  case (Cons a A)
  then show ?case
   apply(cases a)
    by(auto simp: abc_comp.simps abc_shift.simps)
qed (auto simp: abc_comp.simps abc_shift.simps)

lemma abc_Hoare_plus_unhalt2:
  ⟦{Q} (B::abc_prog)↑; {P} (A::abc_prog) {Q}⟧⟹ {P} (A [+] B) ↑
  apply(case_tac A = [])
   apply(simp add: abc_Hoare_halt_def abc_Hoare_unhalt_def abc_exec_null)
  apply(rule_tac abc_Hoare_unhaltI)
  apply(erule_tac abc_Hoare_plus_unhalt2', simp)
   apply(simp, simp)
  done

end
```

```
theory Recursive
  imports Abacus Rec_Def Abacus_Hoare
begin

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

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

The compilation of *z*-operator.

```
definition rec_ci_z :: abc_inst list
  where
    rec_ci_z ≝ [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))
```

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

```
fun rec_ci :: recf ⇒ abc_inst list × nat × nat
 where
  rec_ci z = (rec_ci_z, 1, 2) |
  rec_ci s = (rec_ci_s, 1, 3) |
  rec_ci (id m n) = (rec_ci_id m n, m, m + 2) |
  rec_ci (Cn n f gs) =
    (let cied_gs = map (λ g. rec_ci g) gs in
     let (fprog, fpara, fn) = rec_ci f in
     let pstr = Max (set (Suc n # fn # (map (λ (aprog, p, n). n) cied_gs))) in
     let qstr = pstr + Suc (length gs) in
     (cn_merge_gs cied_gs pstr [+] mv_boxes 0 qstr n [+]
       mv_boxes pstr 0 (length gs) [+] fprog [+]
         mv_box fpara pstr [+] empty_boxes (length gs) [+]
          mv_box pstr n [+] mv_boxes qstr 0 n, n,  qstr + n)) |
  rec_ci (Pr n f g) =
      (let (fprog, fpara, fn) = rec_ci f in
       let (gprog, gpara, gn) = rec_ci g in
       let p = Max (set ([n + 3, fn, gn])) in
       let e = length gprog + 7 in
        (mv_box n p [+] fprog [+] mv_box n (Suc n) [+]
           (([Dec p e] [+] gprog [+]
             [Inc n, Dec (Suc n) 3, Goto 1]) @
                [Dec (Suc (Suc n)) 0, Inc (Suc n), Goto (length gprog + 4)]),
          Suc n, p + 1)) |
  rec_ci (Mn n f) =
      (let (fprog, fpara, fn) = rec_ci f in
       let len = length (fprog) in
         (fprog @ [Dec (Suc n) (len + 5), Dec (Suc n) (len + 3),
          Goto (len + 1), Inc n, Goto 0], n, max (Suc n) fn))

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

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

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

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

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

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

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

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

```
fun addition_inv :: nat × nat list ⇒ nat ⇒ nat ⇒ nat ⇒
              nat list ⇒ bool
  where
    addition_inv (as, lm′) m n p lm =
        (let sn = lm ! n in
         let sm = lm ! m in
         lm ! p = 0 ∧
            (if as = 0 then ∃ x. x ≤ lm ! m ∧ lm′ = lm[m := x,
                            n := (sn + sm − x), p := (sm − x)]
            else if as = 1 then ∃ x. x < lm ! m ∧ lm′ = lm[m := x,
                       n := (sn + sm − x − 1), p := (sm − x − 1)]
            else if as = 2 then ∃ x. x < lm ! m ∧ lm′ = lm[m := x,
                         n := (sn + sm − x), p := (sm − x − 1)]
            else if as = 3 then ∃ x. x < lm ! m ∧ lm′ = lm[m := x,
                            n := (sn + sm − x), p := (sm − x)]
            else if as = 4 then ∃ x. x ≤ lm ! m ∧ lm′ = lm[m := x,
                              n := (sn + sm), p := (sm − x)]
            else if as = 5 then ∃ x. x < lm ! m ∧ lm′ = lm[m := x,
                           n := (sn + sm), p := (sm − x − 1)]
            else if as = 6 then ∃ x. x < lm ! m ∧ lm′ =
                  lm[m := Suc x, n := (sn + sm), p := (sm − x − 1)]
            else if as = 7 then lm′ = lm[m := sm, n := (sn + sm)]
            else False))

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

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

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

fun *addition_measure* :: $((nat \times nat\ list) \times nat \times nat) \Rightarrow$
$(nat \times nat \times nat)$
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*
by(*auto simp*: *addition_LE_def lex_triple_def lex_pair_def*)

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

lemma *update_zero_to_zero*[*simp*]: $[\![am\ !\ n = (0{::}nat); n < length\ am]\!] \Longrightarrow am[n := 0] = am$
apply(*simp add*: *list_update_same_conv*)
done

lemma *addition_inv_init*:
$[\![m \neq n; max\ m\ n < p; length\ lm > p; lm\ !\ p = 0]\!] \Longrightarrow$
$addition\_inv\ (0, lm)\ m\ n\ p\ lm$
apply(*simp add*: *addition_inv.simps Let_def*)
apply(*rule_tac* $x = lm\ !\ m$ **in** *exI*, *simp*)
done

lemma *abs_fetch*[*simp*]:
$abc\_fetch\ 0\ (addition\ m\ n\ p) = Some\ (Dec\ m\ 4)$
$abc\_fetch\ (Suc\ 0)\ (addition\ m\ n\ p) = Some\ (Inc\ n)$
$abc\_fetch\ 2\ (addition\ m\ n\ p) = Some\ (Inc\ p)$
$abc\_fetch\ 3\ (addition\ m\ n\ p) = Some\ (Goto\ 0)$
$abc\_fetch\ 4\ (addition\ m\ n\ p) = Some\ (Dec\ p\ 7)$
$abc\_fetch\ 5\ (addition\ m\ n\ p) = Some\ (Inc\ m)$
$abc\_fetch\ 6\ (addition\ m\ n\ p) = Some\ (Goto\ 4)$
by(*simp_all add*: *abc_fetch.simps addition.simps*)

lemma *exists_small_list_elem1*[*simp*]:
$[\![m \neq n; p < length\ lm; lm\ !\ p = 0; m < p; n < p; x \leq lm\ !\ m; 0 < x]\!]$
$\Longrightarrow \exists xa{<}lm\ !\ m.\ lm[m := x, n := lm\ !\ n + lm\ !\ m - x,$
$p := lm\ !\ m - x, m := x - Suc\ 0] =$
$lm[m := xa, n := lm\ !\ n + lm\ !\ m - Suc\ xa,$
$p := lm\ !\ m - Suc\ xa]$
apply(*cases x*, *simp*, *simp*)
apply(*rule_tac* $x = x - 1$ **in** *exI*, *simp add*: *list_update_swap*
list_update_overwrite)
done

lemma *exists_small_list_elem2*[*simp*]:
$[\![m \neq n; p < \mathit{length}\ lm; lm\ !\ p = 0; m < p; n < p; x < lm\ !\ m]\!]$
$\Longrightarrow \exists xa{<}lm\ !\ m.\ lm[m := x, n := lm\ !\ n + lm\ !\ m - \mathit{Suc}\ x,$
$p := lm\ !\ m - \mathit{Suc}\ x, n := lm\ !\ n + lm\ !\ m - x]$
$= lm[m := xa, n := lm\ !\ n + lm\ !\ m - xa,$
$p := lm\ !\ m - \mathit{Suc}\ xa]$
apply(*rule_tac* $x = x$ **in** *exI*,
simp add: *list_update_swap list_update_overwrite*)
done

lemma *exists_small_list_elem3*[*simp*]:
$[\![m \neq n; p < \mathit{length}\ lm; lm\ !\ p = 0; m < p; n < p; x < lm\ !\ m]\!]$
$\Longrightarrow \exists xa{<}lm\ !\ m.\ lm[m := x, n := lm\ !\ n + lm\ !\ m - x,$
$p := lm\ !\ m - \mathit{Suc}\ x, p := lm\ !\ m - x]$
$= lm[m := xa, n := lm\ !\ n + lm\ !\ m - xa,$
$p := lm\ !\ m - xa]$
apply(*rule_tac* $x = x$ **in** *exI*, *simp add*: *list_update_overwrite*)
done

lemma *exists_small_list_elem4*[*simp*]:
$[\![m \neq n; p < \mathit{length}\ lm; lm\ !\ p = (0{::}\mathit{nat}); m < p; n < p; x < lm\ !\ m]\!]$
$\Longrightarrow \exists xa{\leq}lm\ !\ m.\ lm[m := x, n := lm\ !\ n + lm\ !\ m - x,$
$p := lm\ !\ m - x] =$
$lm[m := xa, n := lm\ !\ n + lm\ !\ m - xa,$
$p := lm\ !\ m - xa]$
apply(*rule_tac* $x = x$ **in** *exI*, *simp*)
done

lemma *exists_small_list_elem5*[*simp*]:
$[\![m \neq n; p < \mathit{length}\ lm; lm\ !\ p = 0; m < p; n < p;$
$x \leq lm\ !\ m; lm\ !\ m \neq x]\!]$
$\Longrightarrow \exists xa{<}lm\ !\ m.\ lm[m := x, n := lm\ !\ n + lm\ !\ m,$
$p := lm\ !\ m - x, p := lm\ !\ m - \mathit{Suc}\ x]$
$= lm[m := xa, n := lm\ !\ n + lm\ !\ m,$
$p := lm\ !\ m - \mathit{Suc}\ xa]$
apply(*rule_tac* $x = x$ **in** *exI*, *simp add*: *list_update_overwrite*)
done

lemma *exists_small_list_elem6*[*simp*]:
$[\![m \neq n; p < \mathit{length}\ lm; lm\ !\ p = 0; m < p; n < p; x < lm\ !\ m]\!]$
$\Longrightarrow \exists xa{<}lm\ !\ m.\ lm[m := x, n := lm\ !\ n + lm\ !\ m,$
$p := lm\ !\ m - \mathit{Suc}\ x, m := \mathit{Suc}\ x]$
$= lm[m := \mathit{Suc}\ xa, n := lm\ !\ n + lm\ !\ m,$
$p := lm\ !\ m - \mathit{Suc}\ xa]$
apply(*rule_tac* $x = x$ **in** *exI*,
simp add: *list_update_swap list_update_overwrite*)
done

lemma *exists_small_list_elem7*[*simp*]:
$[\![m \neq n; p < \mathit{length}\ lm; lm\ !\ p = 0; m < p; n < p; x < lm\ !\ m]\!]$

```
 ⟹ ∃xa≤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]
 apply(rule_tac x = Suc x in exI, simp)
 done

lemma abc_steps_zero: abc_steps_l asm ap 0 = asm
 apply(cases asm, simp add: abc_steps_l.simps)
 done

lemma list_double_update_2:
 lm[a := x, b := y, a := z] = lm[b := y, a:=z]
 by (metis list_update_overwrite list_update_swap)

declare Let_def[simp]
lemma addition_halt_lemma:
 ⟦m ≠ n; max m n < p; length lm > p⟧ ⟹
 ∀na. ¬ (λ(as, lm′) (m, p). as = 7)
     (abc_steps_l (0, lm) (addition m n p) na) (m, p) ∧
 addition_inv (abc_steps_l (0, lm) (addition m n p) na) m n p lm
⟶ addition_inv (abc_steps_l (0, lm) (addition m n p)
                 (Suc na)) m n p lm
 ∧ ((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) ∈ addition_LE
proof −
 assume assms:m≠n max m n < p length lm > p
 { fix na
  obtain a b where ab:abc_steps_l (0, lm) (addition m n p) na = (a, b) by force
  assume assms2: ¬ (λ(as, lm′) (m, p). as = 7)
     (abc_steps_l (0, lm) (addition m n p) na) (m, p)
    addition_inv (abc_steps_l (0, lm) (addition m n p) na) m n p lm
  have r1:addition_inv (abc_steps_l (0, lm) (addition m n p)
                 (Suc na)) m n p lm using assms(1−3) assms2
    unfolding abc_step_red2 ab abc_step_l.simps abc_lm_v.simps abc_lm_s.simps
     addition_inv.simps
    by (auto split:if_splits simp add: addition_inv.simps Suc_diff_Suc)
  have r2:((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) ∈ addition_LE using assms(1−3) assms2
    unfolding abc_step_red2 ab
    apply(auto split:if_splits simp add: addition_inv.simps abc_steps_zero)
    by(auto simp add: addition_LE_def lex_triple_def lex_pair_def
      abc_step_l.simps abc_lm_v.simps abc_lm_s.simps split: if_splits)
  note r1 r2
 }
 thus ?thesis by auto
qed

lemma addition_correct′:
 ⟦m ≠ n; max m n < p; length lm > p; lm ! p = 0⟧ ⟹
 ∃ stp. (λ (as, lm′). as = 7 ∧ addition_inv (as, lm′) m n p lm)
```

```
                  (abc_steps_l (0, lm) (addition m n p) stp)
  apply(insert halt_lemma2[of addition_LE
       λ ((as, lm'), m, p). addition_inv (as, lm') m n p lm
       λ stp. (abc_steps_l (0, lm) (addition m n p) stp, m, p)
       λ ((as, lm'), m, p). as = 7],
     simp add: abc_steps_zero addition_inv_init)
  apply(drule_tac addition_halt_lemma,force,force)
  apply (simp,erule_tac exE)
  apply(rename_tac na)
  apply(rule_tac x = na in exI)
  apply(auto)
  done

lemma length_addition[simp]: length (addition a b c) = 7
  by(auto simp: addition.simps)

lemma addition_correct:
  assumes m ≠ n max m n < p length lm > p lm ! p = 0
  shows {λ a. a = lm} (addition m n p) {λ nl. addition_inv (7, nl) m n p lm}
  using assms
proof(rule_tac abc_Hoare_haltI, simp)
  fix lma
  assume m ≠ n m < p ∧ n < p p < length lm lm ! p = 0
  then have ∃ stp. (λ (as, lm'). as = 7 ∧ addition_inv (as, lm') m n p lm)
                  (abc_steps_l (0, lm) (addition m n p) stp)
    by(rule_tac addition_correct', auto simp: addition_inv.simps)
  then obtain stp where (λ (as, lm'). as = 7 ∧ addition_inv (as, lm') m n p lm)
                  (abc_steps_l (0, lm) (addition m n p) stp)
    using exE by presburger
  thus ∃ na. abc_final (abc_steps_l (0, lm) (addition m n p) na) (addition m n p) ∧
               (λnl. addition_inv (7, nl) m n p lm) abc_holds_for abc_steps_l (0, lm) (addition m n
p) na
    by(auto intro:exI[of _ stp])
qed

lemma compile_s_correct':
  {λnl. nl = n # 0 ↑ 2 @ anything} addition 0 (Suc 0) 2 [+] [Inc (Suc 0)] {λnl. nl = n # Suc n
# 0 # anything}
proof(rule_tac abc_Hoare_plus_halt)
  show {λnl. nl = n # 0 ↑ 2 @ anything} addition 0 (Suc 0) 2 {λ nl. addition_inv (7, nl) 0 (Suc
0) 2 (n # 0 ↑ 2 @ anything)}
    by(rule_tac addition_correct, auto simp: numeral_2_eq_2)
next
  show {λnl. addition_inv (7, nl) 0 (Suc 0) 2 (n # 0 ↑ 2 @ anything)} [Inc (Suc 0)] {λnl. nl =
n # Suc n # 0 # anything}
    by(rule_tac abc_Hoare_haltI, rule_tac x = 1 in exI, auto simp: addition_inv.simps
      abc_steps_l.simps abc_step_l.simps abc_fetch.simps numeral_2_eq_2 abc_lm_s.simps abc_lm_v.simps)
qed

declare rec_exec.simps[simp del]
```

lemma *abc_comp_commute*: $(A\ [+]\ B)\ [+]\ C = A\ [+]\ (B\ [+]\ C)$
apply(*auto simp*: *abc_comp.simps abc_shift.simps*)
apply(*rename_tac x*)
apply(*case_tac x*, *auto*)
done

lemma *compile_z_correct*:
$[\![rec\_ci\ z = (ap, arity, fp);\ rec\_exec\ z\ [n] = r]\!] \Longrightarrow$
$\{\lambda nl.\ nl = n\ \#\ 0 \uparrow (fp - arity)\ @\ anything\}\ ap\ \{\lambda nl.\ nl = n\ \#\ r\ \#\ 0 \uparrow (fp - Suc\ arity)\ @\ anything\}$
apply(*rule_tac abc_Hoare_haltI*)
apply(*rule_tac* $x = 1$ **in** *exI*)
apply(*auto simp*: *abc_steps_l.simps rec_ci.simps rec_ci_z_def numeral_2_eq_2 abc_fetch.simps abc_step_l.simps rec_exec.simps*)
done

lemma *compile_s_correct*:
$[\![rec\_ci\ s = (ap, arity, fp);\ rec\_exec\ s\ [n] = r]\!] \Longrightarrow$
$\{\lambda nl.\ nl = n\ \#\ 0 \uparrow (fp - arity)\ @\ anything\}\ ap\ \{\lambda nl.\ nl = n\ \#\ r\ \#\ 0 \uparrow (fp - Suc\ arity)\ @\ anything\}$
apply(*auto simp*: *rec_ci.simps rec_ci_s_def compile_s_correct′ rec_exec.simps*)
done

lemma *compile_id_correct′*:
assumes $n < length\ args$
shows $\{\lambda nl.\ nl = args\ @\ 0 \uparrow 2\ @\ anything\}\ addition\ n\ (length\ args)\ (Suc\ (length\ args))$
$\{\lambda nl.\ nl = args\ @\ rec\_exec\ (recf.id\ (length\ args)\ n)\ args\ \#\ 0\ \#\ anything\}$
proof −
have $\{\lambda nl.\ nl = args\ @\ 0 \uparrow 2\ @\ anything\}\ addition\ n\ (length\ args)\ (Suc\ (length\ args))$
$\{\lambda nl.\ addition\_inv\ (7, nl)\ n\ (length\ args)\ (Suc\ (length\ args))\ (args\ @\ 0 \uparrow 2\ @\ anything)\}$
using *assms*
by(*rule_tac addition_correct*, *auto simp*: *numeral_2_eq_2 nth_append*)
thus *?thesis*
using *assms*
by(*simp add*: *addition_inv.simps rec_exec.simps nth_append numeral_2_eq_2 list_update_append*)
qed

lemma *compile_id_correct*:
$[\![n < m;\ length\ xs = m;\ rec\_ci\ (recf.id\ m\ n) = (ap, arity, fp);\ rec\_exec\ (recf.id\ m\ n)\ xs = r]\!]$
$\Longrightarrow \{\lambda nl.\ nl = xs\ @\ 0 \uparrow (fp - arity)\ @\ anything\}\ ap\ \{\lambda nl.\ nl = xs\ @\ r\ \#\ 0 \uparrow (fp - Suc\ arity)\ @\ anything\}$
apply(*auto simp*: *rec_ci.simps rec_ci_id.simps compile_id_correct′*)
done

lemma *cn_merge_gs_tl_app*:
$cn\_merge\_gs\ (gs\ @\ [g])\ pstr =$

```
    cn_merge_gs gs pstr [+] cn_merge_gs [g] (pstr + length gs)
  apply(induct gs arbitrary: pstr, simp add: cn_merge_gs.simps, auto)
  apply(simp add: abc_comp_commute)
  done

lemma footprint_ge:
  rec_ci a = (p, arity, fp) ⟹ arity < fp
proof(induct a)
  case (Cn x1 a x3)
  then show ?case by(cases rec_ci a, auto simp:rec_ci.simps)
next
  case (Pr x1 a1 a2)
  then show ?case by(cases rec_ci a1;cases rec_ci a2, auto simp:rec_ci.simps)
next
case (Mn x1 a)
  then show ?case by(cases rec_ci a, auto simp:rec_ci.simps)
qed (auto simp: rec_ci.simps)

lemma param_pattern:
  ⟦terminate f xs; rec_ci f = (p, arity, fp)⟧ ⟹ length xs = arity
proof(induct arbitrary: p arity fp rule: terminate.induct)
  case (termi_cn f xs gs n) thus ?case
    by(cases rec_ci f, (auto simp: rec_ci.simps))
next
  case (termi_pr x g xs n f) thus ?case
    by (cases rec_ci f, cases rec_ci g, auto simp: rec_ci.simps)
next
  case (termi_mn xs n f r) thus ?case
    by (cases rec_ci f, auto simp: rec_ci.simps)
qed (auto simp: rec_ci.simps)

lemma replicate_merge_anywhere:
  x↑a @ x↑b @ ys = x↑(a+b) @ ys
  by(simp add:replicate_add)

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

fun mv_box_stage1 :: nat × nat list ⇒ nat ⇒ nat
```

```
 where
  mv_box_stage1 (as, lm) m  =
        (if as = 3 then 0
         else 1)

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

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

fun mv_box_measure :: ((nat × nat list) × nat) ⇒ (nat × nat × nat)
 where
  mv_box_measure ((as, lm), m) =
   (mv_box_stage1 (as, lm) m, mv_box_stage2 (as, lm) m,
    mv_box_stage3 (as, lm) m)

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

definition lex_triple ::
 ((nat × (nat × nat)) × (nat × (nat × nat))) set
 where
  lex_triple ≝ less_than <*lex*> lex_pair

definition mv_box_LE ::
 (((nat × nat list) × nat) × ((nat × nat list) × nat)) set
 where
  mv_box_LE ≝ (inv_image lex_triple mv_box_measure)

lemma wf_lex_triple: wf lex_triple
 by (auto simp:lex_triple_def lex_pair_def)

lemma wf_mv_box_le[intro]: wf mv_box_LE
 by(auto intro:wf_lex_triple simp: mv_box_LE_def)

declare mv_box_inv.simps[simp del]

lemma mv_box_inv_init:
 ⟦m < length initlm; n < length initlm; m ≠ n⟧ ⟹
 mv_box_inv (0, initlm) m n initlm
 apply(simp add: abc_steps_l.simps mv_box_inv.simps)
 apply(rule_tac x = initlm ! m in exI,
```

$\quad$ *rule_tac x = initlm ! n* **in** *exI*, *simp*)
done

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*
apply(*simp_all add*: *mv_box.simps abc_fetch.simps*)
done

lemma *replicate_Suc_iff_anywhere*: $x \# x{\uparrow}b\ @\ ys = x{\uparrow}(Suc\ b)\ @\ ys$
by *simp*

lemma *exists_smaller_in_list0*[*simp*]:
$[\![m \neq n;\ m < length\ initlm;\ n < length\ initlm;$
$k + l = initlm\ !\ m + initlm\ !\ n;\ k \leq initlm\ !\ m;\ 0 < k]\!]$
$\Longrightarrow \exists\, ka\ la.\ initlm[m := k, n := l, m := k - Suc\ 0] =$
$initlm[m := ka, n := la] \wedge$
$Suc\ (ka + la) = initlm\ !\ m + initlm\ !\ n \wedge$
$ka < initlm\ !\ m$
apply(*rule_tac x = k − Suc 0* **in** *exI*, *rule_tac x = l* **in** *exI*, *auto*)
apply(*subgoal_tac*
$initlm[m := k, n := l, m := k - Suc\ 0] =$
$initlm[n := l, m := k, m := k - Suc\ 0]$,*force intro*:*list_update_swap*)
by(*simp add*: *list_update_swap*)

lemma *exists_smaller_in_list1*[*simp*]:
$[\![m \neq n;\ m < length\ initlm;\ n < length\ initlm;$
$Suc\ (k + l) = initlm\ !\ m + initlm\ !\ n;$
$k < initlm\ !\ m]\!]$
$\Longrightarrow \exists\, ka\ la.\ initlm[m := k, n := l, n := Suc\ l] =$
$initlm[m := ka, n := la] \wedge$
$ka + la = initlm\ !\ m + initlm\ !\ n \wedge$
$ka \leq initlm\ !\ m$
apply(*rule_tac x = k* **in** *exI*, *rule_tac x = Suc l* **in** *exI*, *auto*)
done

lemma *abc_steps_prop*[*simp*]:
$[\![length\ initlm > max\ m\ n;\ m \neq n]\!] \Longrightarrow$
$\neg\ (\lambda(as, lm)\ m.\ as = 3)$
$(abc\_steps\_l\ (0, initlm)\ (mv\_box\ m\ n)\ na)\ m \wedge$
$mv\_box\_inv\ (abc\_steps\_l\ (0, initlm)$
$(mv\_box\ m\ n)\ na)\ m\ n\ initlm \longrightarrow$
$mv\_box\_inv\ (abc\_steps\_l\ (0, initlm)$
$(mv\_box\ m\ n)\ (Suc\ na))\ m\ n\ initlm \wedge$
$((abc\_steps\_l\ (0, initlm)\ (mv\_box\ m\ n)\ (Suc\ na), m),$
$abc\_steps\_l\ (0, initlm)\ (mv\_box\ m\ n)\ na, m) \in mv\_box\_LE$
apply(*rule impI*, *simp add*: *abc_step_red2*)
apply(*cases* (*abc_steps_l* (*0*, *initlm*) (*mv_box m n*) *na*),

```
    simp)
  apply(auto split:if_splits simp add:abc_steps_l.simps mv_box_inv.simps)
     apply(auto simp add: mv_box_LE_def lex_triple_def lex_pair_def
    abc_step_l.simps abc_steps_l.simps
    mv_box_inv.simps abc_lm_v.simps abc_lm_s.simps
    split: if_splits )
  apply(rule_tac x = k in exI, rule_tac x = Suc l in exI, simp)
  done

lemma mv_box_inv_halt:
  ⟦length initlm > max m n; m ≠ n⟧ ⟹
  ∃ stp. (λ (as, lm). as = 3 ∧
  mv_box_inv (as, lm) m n initlm)
          (abc_steps_l (0::nat, initlm) (mv_box m n) stp)
  apply(insert halt_lemma2[of mv_box_LE
      λ ((as, lm), m). mv_box_inv (as, lm) m n initlm
      λ stp. (abc_steps_l (0, initlm) (mv_box m n) stp, m)
      λ ((as, lm), m). as = (3::nat)
      ])
  apply(insert wf_mv_box_le)
  apply(simp add: mv_box_inv_init abc_steps_zero)
  apply(erule_tac exE)
  by (metis (no_types, lifting) case_prodE' case_prodI)

lemma mv_box_halt_cond:
  ⟦m ≠ n; mv_box_inv (a, b) m n lm; a = 3⟧ ⟹
  b = lm[n := lm ! m + lm ! n, m := 0]
  apply(simp add: mv_box_inv.simps, auto)
  apply(simp add: list_update_swap)
  done

lemma mv_box_correct':
  ⟦length lm > max m n; m ≠ n⟧ ⟹
  ∃ stp. abc_steps_l (0::nat, lm) (mv_box m n) stp
  = (3, (lm[n := (lm ! m + lm ! n)])[m := 0::nat])
  by(drule mv_box_inv_halt, auto dest:mv_box_halt_cond)

lemma length_mvbox[simp]: length (mv_box m n) = 3
  by(simp add: mv_box.simps)

lemma mv_box_correct:
  ⟦length lm > max m n; m≠n⟧
  ⟹ {λ nl. nl = lm} mv_box m n {λ nl. nl = lm[n := (lm ! m + lm ! n), m:=0]}
  apply(drule_tac mv_box_correct', simp)
  apply(auto simp: abc_Hoare_halt_def)
  by (metis abc_final.simps abc_holds_for.simps length_mvbox)

declare list_update.simps(2)[simp del]

lemma zero_case_rec_exec[simp]:
```

⟦length xs < gf; gf ≤ ft; n < length gs⟧
⟹ (rec_exec (gs ! n) xs # 0 ↑ (ft − Suc (length xs)) @ map (λi. 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 (λi. rec_exec i xs) (take n gs) @ rec_exec (gs ! n) xs # 0 ↑ (length
gs − Suc n) @ 0 # 0 ↑ length xs @ anything
using list_update_append[of rec_exec (gs ! n) xs # 0 ↑ (ft − Suc (length xs)) @ map (λi.
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]
apply(auto)
apply(cases length gs − n, simp, simp add: list_update.simps replicate_Suc_iff_anywhere Suc_diff_Suc
del: replicate_Suc)
apply(simp add: list_update.simps)
done

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. {λnl. nl = xs @ 0 ↑ (xb − xa) @ xc} x {λnl. 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
{λnl. nl = xs @ 0 # 0 ↑ (ft + length gs) @ anything}
cn_merge_gs (map rec_ci (take n gs)) ft
{λnl. nl = xs @ 0 ↑ (ft − length xs) @
map (λi. rec_exec i xs) (take n gs) @ 0↑(length gs − n) @ 0 ↑ Suc (length xs) @
anything}
using g_cond
proof(induct n)
case 0
have ft > length xs
using ft
by simp
thus ?case
apply(rule_tac abc_Hoare_haltI)
apply(rule_tac x = 0 **in** exI, simp add: abc_steps_l.simps replicate_add[THEN sym]
replicate_Suc[THEN sym] del: replicate_Suc)
done
next
case (Suc n)
have ind′: ∀ g∈set (take n 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})) ⟹
{λnl. nl = xs @ 0 # 0 ↑ (ft + length gs) @ anything} cn_merge_gs (map rec_ci (take n gs)) ft
{λnl. nl = xs @ 0 ↑ (ft − length xs) @ map (λi. rec_exec i xs) (take n gs) @ 0 ↑ (length gs −
n) @ 0 ↑ Suc (length xs) @ anything}
by fact
have g_newcond: ∀ g∈set (take (Suc n) gs).
terminate g xs ∧ (∀ x xa xb. rec_ci g = (x, xa, xb) ⟶ (∀ xc. {λnl. nl = xs @ 0 ↑ (xb − xa)

@ *xc*} *x* {λ*nl*. *nl* = *xs* @ *rec_exec g xs* # *0* ↑ (*xb* − *Suc xa*) @ *xc*}))
by *fact*
from *g_newcond* **have** *ind*:
{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*}
apply(*rule_tac ind*′, *rule_tac ballI*, *erule_tac x* = *g* **in** *ballE*, *simp_all add*: *take_Suc*)
by(*cases n* < *length gs*, *simp add*:*take_Suc_conv_app_nth*, *simp*)
show *?case*
proof(*cases n* < *length gs*)
case *True*
have *h*: *n* < *length gs* **by** *fact*
thus *?thesis*
proof(*simp add*: *take_Suc_conv_app_nth cn_merge_gs_tl_app*)
obtain *gp ga gf* **where** *a*: *rec_ci* (*gs*!*n*) = (*gp*, *ga*, *gf*)
by (*metis prod_cases3*)
moreover have *min* (*length gs*) *n* = *n*
using *h* **by** *simp*
moreover have
{λ*nl*. *nl* = *xs* @ *0* # *0* ↑ (*ft* + *length gs*) @ *anything*}
cn_merge_gs (*map rec_ci* (*take n gs*)) *ft* [+] (*gp* [+] *mv_box ga* (*ft* + *n*))
{λ*nl*. *nl* = *xs* @ *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(*rule_tac abc_Hoare_plus_halt*)
show {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*}
using *ind* **by** *simp*
next
have *x*: *gs*!*n* ∈ *set* (*take* (*Suc n*) *gs*)
using *h*
by(*simp add*: *take_Suc_conv_app_nth*)
have *b*: *terminate* (*gs*!*n*) *xs*
using *a g_newcond h x*
by(*erule_tac x* = *gs*!*n* **in** *ballE*, *simp_all*)
hence *c*: *length xs* = *ga*
using *a param_pattern* **by** *metis*
have *d*: *gf* > *ga* **using** *footprint_ge a* **by** *simp*
have *e*: *ft* ≥ *gf*
using *ft a h Max_ge image_eqI*
by(*simp*, *rule_tac max.coboundedI2*, *rule_tac Max_ge*, *simp*,
rule_tac insertI2,
rule_tac f = (λ(*aprog*, *p*, *n*). *n*) **and** *x* = *rec_ci* (*gs*!*n*) **in** *image_eqI*, *simp*,
rule_tac x = *gs*!*n* **in** *image_eqI*, *simp*, *simp*)
show {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*}
gp [+] *mv_box ga* (*ft* + *n*)
{λ*nl*. *nl* = *xs* @ *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(rule_tac abc_Hoare_plus_halt)
show {λnl. nl = xs @ 0 ↑ (ft − length xs) @ map (λi. rec_exec i xs) (take n gs) @ 0 ↑
(length gs − n) @ 0 ↑ Suc (length xs) @ anything} gp
{λnl. nl = xs @ (rec_exec (gs!n) xs) # 0 ↑ (ft − Suc (length xs)) @ map (λi. rec_exec
i xs)
(take n gs) @ 0 ↑ (length gs − n) @ 0 # 0 ↑ length xs @ anything}
proof −
have
({λnl. nl = xs @ 0 ↑ (gf − ga) @ 0↑(ft − gf)@map (λi. rec_exec i xs) (take n gs) @ 0
↑ (length gs − n) @ 0 ↑ Suc (length xs) @ anything}
gp {λnl. nl = xs @ (rec_exec (gs!n) xs) # 0 ↑ (gf − Suc ga) @
0↑(ft − gf)@map (λi. rec_exec i xs) (take n gs) @ 0 ↑ (length gs − n) @ 0 ↑ Suc (length
xs) @ anything})
using a g_newcond h x
apply(erule_tac x = gs!n **in** ballE)
apply(simp, simp)
done
thus ?thesis
using a b c d e
by(simp add: replicate_merge_anywhere)
qed
next
show
{λnl. nl = xs @ rec_exec (gs ! n) xs #
0 ↑ (ft − Suc (length xs)) @ map (λi. rec_exec i xs) (take n gs) @ 0 ↑ (length gs − n) @
0 # 0 ↑ length xs @ anything}
mv_box ga (ft + n)
{λnl. nl = xs @ 0 ↑ (ft − length xs) @ map (λi. rec_exec i xs) (take n gs) @
rec_exec (gs ! n) xs # 0 ↑ (length gs − Suc n) @ 0 # 0 ↑ length xs @ anything}
proof −
have {λnl. nl = xs @ rec_exec (gs ! n) xs # 0 ↑ (ft − Suc (length xs)) @
map (λi. rec_exec i xs) (take n gs) @ 0 ↑ (length gs − n) @ 0 # 0 ↑ length xs @
anything}
mv_box ga (ft + n) {λnl. nl = (xs @ rec_exec (gs ! n) xs # 0 ↑ (ft − Suc (length xs)) @
map (λi. rec_exec i xs) (take n gs) @ 0 ↑ (length gs − n) @ 0 # 0 ↑ length xs @
anything)
[ft + n := (xs @ rec_exec (gs ! n) xs # 0 ↑ (ft − Suc (length xs)) @ map (λi. rec_exec i
xs) (take n gs) @
0 ↑ (length gs − n) @ 0 # 0 ↑ length xs @ anything) ! ga +
(xs @ rec_exec (gs ! n) xs # 0 ↑ (ft − Suc (length xs)) @
map (λi. rec_exec i xs) (take n gs) @ 0 ↑ (length gs − n) @ 0 # 0 ↑ length xs @
anything) !
(ft + n), ga := 0]}
using a c d e h
apply(rule_tac mv_box_correct)
apply(simp, arith, arith)
done
moreover have (xs @ rec_exec (gs ! n) xs # 0 ↑ (ft − Suc (length xs)) @
map (λi. rec_exec i xs) (take n gs) @ 0 ↑ (length gs − n) @ 0 # 0 ↑ length xs @

anything)
[ft + n := (xs @ rec_exec (gs ! n) xs # 0 ↑ (ft − Suc (length xs)) @ map (λi. rec_exec i xs) (take n gs) @
0 ↑ (length gs − n) @ 0 # 0 ↑ length xs @ anything) ! ga +
(xs @ rec_exec (gs ! n) xs # 0 ↑ (ft − Suc (length xs)) @
map (λi. rec_exec i xs) (take n gs) @ 0 ↑ (length gs − n) @ 0 # 0 ↑ length xs @ anything) !
(ft + n), ga := 0]=
xs @ 0 ↑ (ft − length xs) @ map (λi. rec_exec i xs) (take n gs) @ rec_exec (gs ! n) xs # 0 ↑ (length gs − Suc n) @ 0 # 0 ↑ length xs @ anything
using a c d e h
by(simp add: list_update_append nth_append length_replicate split: if_splits del: list_update.simps(2), auto)
ultimately show ?thesis
by(simp)
qed
qed
qed
ultimately show
{λnl. nl = xs @ 0 # 0 ↑ (ft + length gs) @ anything}
cn_merge_gs (map rec_ci (take n gs)) ft [+] (case rec_ci (gs ! n) of (gprog, gpara, gn) ⇒
gprog [+] mv_box gpara (ft + min (length gs) n))
{λnl. nl = xs @ 0 ↑ (ft − length xs) @ map (λi. rec_exec i xs) (take n gs) @ rec_exec (gs ! n) xs # 0 ↑ (length gs − Suc n) @ 0 # 0 ↑ length xs @ anything}
by simp
qed
next
case False
have h: ¬ n < length gs **by** fact
hence ind′:
{λnl. nl = xs @ 0 # 0 ↑ (ft + length gs) @ anything} cn_merge_gs (map rec_ci gs) ft
{λnl. nl = xs @ 0 ↑ (ft − length xs) @ map (λi. rec_exec i xs) gs @ 0 ↑ Suc (length xs) @ anything}
using ind
by simp
thus ?thesis
using h
by(simp)
qed
qed

lemma compile_cn_gs_correct:
assumes
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}))
and ft: ft = max (Suc (length xs)) (Max (insert ffp ((λ(aprog, p, n). n) ‘ rec_ci ‘ set gs)))
shows
{λnl. nl = xs @ 0 # 0 ↑ (ft + length gs) @ anything}
cn_merge_gs (map rec_ci gs) ft

```
{λnl. nl = xs @ 0 ↑ (ft − length xs) @
        map (λi. rec_exec i xs) gs @ 0 ↑ Suc (length xs) @ anything}
 using assms
 using compile_cn_gs_correct′[of length gs gs xs ft ffp anything ]
 apply(auto)
 done

lemma length_mvboxes[simp]: length (mv_boxes aa ba n) = 3∗n
 by(induct n, auto simp: mv_boxes.simps)

lemma exp_suc: a↑Suc b = a↑b @ [a]
 by(simp add: exp_ind del: replicate.simps)

lemma last_0[simp]:
 ⟦Suc n ≤ ba − aa;  length lm2 = Suc n;
  length lm3 = ba − Suc (aa + n)⟧
 ⟹ (last lm2 # lm3 @ butlast lm2 @ 0 # lm4) ! (ba − aa) = (0::nat)
proof −
 assume h: Suc n ≤ ba − aa
  and g: length lm2 = Suc n length lm3 = ba − Suc (aa + n)
 from h and g have k: ba − aa = Suc (length lm3 + n)
  by arith
 from  k show
  (last lm2 # lm3 @ butlast lm2 @ 0 # lm4) ! (ba − aa) = 0
  apply(simp, insert g)
  apply(simp add: nth_append)
  done
qed

lemma butlast_last[simp]: length lm1 = aa ⟹
    (lm1 @ 0↑n @ last lm2 # lm3 @ butlast lm2 @ 0 # lm4) ! (aa + n) = last lm2
 apply(simp add: nth_append)
 done

lemma arith_as_simp[simp]: ⟦Suc n ≤ ba − aa; aa < ba⟧ ⟹
          (ba < Suc (aa + (ba − Suc (aa + n) + n))) = False
 apply arith
 done

lemma butlast_elem[simp]: ⟦Suc n ≤ ba − aa; aa < ba; length lm1 = aa;
    length lm2 = Suc n; length lm3 = ba − Suc (aa + n)⟧
   ⟹ (lm1 @ 0↑n @ last lm2 # lm3 @ butlast lm2 @ 0 # lm4) ! (ba + n) = 0
 using nth_append[of lm1 @ (0::′a)↑n @ last lm2 # lm3 @ butlast lm2
   (0::′a) # lm4 ba + n]
 apply(simp)
 done

lemma update_butlast_eq0[simp]:
 ⟦Suc n ≤ ba − aa; aa < ba; length lm1 = aa; length lm2 = Suc n;
         length lm3 = ba − Suc (aa + n)⟧
```

$\Longrightarrow$ (*lm1* @ *0*↑*n* @ *last lm2* # *lm3* @ *butlast lm2* @ (*0*::*nat*) # *lm4*)
[*ba* + *n* := *last lm2*, *aa* + *n* := *0*] =
lm1 @ *0* # *0*↑*n* @ *lm3* @ *lm2* @ *lm4*
using *list_update_append*[*of lm1* @ *0*↑*n* @ *last lm2* # *lm3* @ *butlast lm2 0* # *lm4*
ba + *n last lm2*]
apply(*simp add*: *list_update_append list_update.simps*(2−) *replicate_Suc_iff_anywhere exp_suc*
del: *replicate_Suc*)
apply(*cases lm2*, *simp*, *simp*)
done

lemma *update_butlast_eq1*[*simp*]:
⟦*Suc* (*length lm1* + *n*) ≤ *ba*; *length lm2* = *Suc n*; *length lm3* = *ba* − *Suc* (*length lm1* + *n*);
¬ *ba* − *Suc* (*length lm1*) < *ba* − *Suc* (*length lm1* + *n*); ¬ *ba* + *n* − *length lm1* < *n*⟧
$\Longrightarrow$ (*0*::*nat*) ↑ *n* @ (*last lm2* # *lm3* @ *butlast lm2* @ *0* # *lm4*)[*ba* − *length lm1* := *last lm2*,
0 := *0*] =
0 # *0* ↑ *n* @ *lm3* @ *lm2* @ *lm4*
apply(*subgoal_tac ba* − *length lm1* = *Suc n* + *length lm3*, *simp add*: *list_update.simps*(2−)
list_update_append)
apply(*simp add*: *replicate_Suc_iff_anywhere exp_suc del*: *replicate_Suc*)
apply(*cases lm2*, *simp*, *simp*)
apply(*auto*)
done

lemma *mv_boxes_correct*:
⟦*aa* + *n* ≤ *ba*; *ba* > *aa*; *length lm1* = *aa*; *length lm2* = *n*; *length lm3* = *ba* − *aa* − *n*⟧
$\Longrightarrow$ {λ *nl*. *nl* = *lm1* @ *lm2* @ *lm3* @ *0*↑*n* @ *lm4*} (*mv_boxes aa ba n*)
{λ *nl*. *nl* = *lm1* @ *0*↑*n* @ *lm3* @ *lm2* @ *lm4*}
proof(*induct n arbitrary*: *lm2 lm3 lm4*)
case *0*
thus *?case*
by(*simp add*: *mv_boxes.simps abc_Hoare_halt_def*, *rule_tac* *x* = *0* **in** *exI*, *simp add*: *abc_steps_l.simps*)
next
case (*Suc n*)
have *ind*:
⋀*lm2 lm3 lm4*.
⟦*aa* + *n* ≤ *ba*; *aa* < *ba*; *length lm1* = *aa*; *length lm2* = *n*; *length lm3* = *ba* − *aa* − *n*⟧
$\Longrightarrow$ {λ*nl*. *nl* = *lm1* @ *lm2* @ *lm3* @ *0* ↑ *n* @ *lm4*} *mv_boxes aa ba n* {λ*nl*. *nl* = *lm1* @ *0* ↑ *n*
@ *lm3* @ *lm2* @ *lm4*}
by *fact*
have *h1*: *aa* + *Suc n* ≤ *ba* **by** *fact*
have *h2*: *aa* < *ba* **by** *fact*
have *h3*: *length lm1* = *aa* **by** *fact*
have *h4*: *length lm2* = *Suc n* **by** *fact*
have *h5*: *length lm3* = *ba* − *aa* − *Suc n* **by** *fact*
have {λ*nl*. *nl* = *lm1* @ *lm2* @ *lm3* @ *0* ↑ *Suc n* @ *lm4*} *mv_boxes aa ba n* [+] *mv_box* (*aa* + *n*)
(*ba* + *n*)
{λ*nl*. *nl* = *lm1* @ *0* ↑ *Suc n* @ *lm3* @ *lm2* @ *lm4*}
proof(*rule_tac abc_Hoare_plus_halt*)
have {λ*nl*. *nl* = *lm1* @ *butlast lm2* @ (*last lm2* # *lm3*) @ *0* ↑ *n* @ (*0* # *lm4*)} *mv_boxes aa*
ba n

```
      {λ nl. nl = lm1 @ 0↑n @ (last lm2 # lm3) @ butlast lm2 @ (0 # lm4)}
    using h1 h2 h3 h4 h5
    by(rule_tac ind, simp_all)
  moreover have  lm1 @ butlast lm2 @ (last lm2 # lm3) @ 0 ↑ n @ (0 # lm4)
            = lm1 @ lm2 @ lm3 @ 0 ↑ Suc n @ lm4
    using h4
    by(simp add: replicate_Suc[THEN sym] exp_suc del: replicate_Suc,
       cases lm2, simp_all)
  ultimately show {λnl. nl = lm1 @ lm2 @ lm3 @ 0 ↑ Suc n @ lm4} mv_boxes aa ba n
      {λ nl. nl = lm1 @ 0↑n @ last lm2 # lm3 @ butlast lm2 @ 0 # lm4}
    by (metis append_Cons)
 next
  let ?lm = lm1 @ 0 ↑ n @ last lm2 # lm3 @ butlast lm2 @ 0 # lm4
  have {λnl. nl = ?lm} mv_box (aa + n) (ba + n)
      {λ nl. nl = ?lm[(ba + n) := ?lm!(aa+n) + ?lm!(ba+n), (aa+n):=0]}
    using h1 h2 h3 h4  h5
    by(rule_tac mv_box_correct, simp_all)
  moreover have ?lm[(ba + n) := ?lm!(aa+n) + ?lm!(ba+n), (aa+n):=0]
            =  lm1 @ 0 ↑ Suc n @ lm3 @ lm2 @ lm4
    using h1 h2 h3 h4 h5
    by(auto simp: nth_append list_update_append split: if_splits)
  ultimately show {λnl. nl = lm1 @ 0 ↑ n @ last lm2 # lm3 @ butlast lm2 @ 0 # lm4} mv_box
(aa + n) (ba + n)
      {λnl. nl = lm1 @ 0 ↑ Suc n @ lm3 @ lm2 @ lm4}
    by simp
 qed
 thus ?case
  by(simp add: mv_boxes.simps)
qed

lemma update_butlast_eq2[simp]:
 ⟦Suc n ≤ aa − length lm1; length lm1 < aa;
 length lm2 = aa − Suc (length lm1 + n);
 length lm3 = Suc n;
 ¬ aa − Suc (length lm1) < aa − Suc (length lm1 + n);
 ¬ aa + n − length lm1 < n⟧
 ⟹ butlast lm3 @ ((0::nat) # lm2 @ 0 ↑ n @ last lm3 # lm4)[0 := last lm3, aa − length lm1
:= 0] = lm3 @ lm2 @ 0 # 0 ↑ n @ lm4
 apply(subgoal_tac aa − length lm1 = length lm2 + Suc n)
  apply(simp add: list_update.simps list_update_append)
  apply(simp add: replicate_Suc[THEN sym] exp_suc del: replicate_Suc)
  apply(cases lm3, simp, simp)
 apply(auto)
 done

lemma mv_boxes_correct2:
 ⟦n ≤ aa − ba;
  ba < aa;
  length (lm1::nat list) = ba;
  length (lm2::nat list) = aa − ba − n;
```

```
  length (lm3::nat list) = n⟧
 ⟹{λ nl. nl = lm1 @ 0↑n @ lm2 @ lm3 @ lm4}
           (mv_boxes aa ba n)
   {λ nl. nl = lm1 @ lm3 @ lm2 @ 0↑n @ lm4}
proof(induct n arbitrary: lm2 lm3 lm4)
 case 0
 thus ?case
  by(simp add: mv_boxes.simps abc_Hoare_halt_def, rule_tac x = 0 in exI, simp add: abc_steps_l.simps)
next
 case (Suc n)
 have ind:
  ⋀lm2 lm3 lm4.
  ⟦n ≤ aa − ba; ba < aa; length lm1 = ba; length lm2 = aa − ba − n; length lm3 = n⟧
  ⟹ {λnl. nl = lm1 @ 0 ↑ n @ lm2 @ lm3 @ lm4} mv_boxes aa ba n {λnl. nl = lm1 @ lm3 @
lm2 @ 0 ↑ n @ lm4}
   by fact
 have h1: Suc n ≤ aa − ba by fact
 have h2: ba < aa by fact
 have h3: length lm1 = ba by fact
 have h4: length lm2 = aa − ba − Suc n by fact
 have h5: length lm3 = Suc n by fact
 have {λnl. nl = lm1 @ 0 ↑ Suc n @ lm2 @ lm3 @ lm4}  mv_boxes aa ba n [+] mv_box (aa +
n) (ba + n)
   {λnl. nl = lm1 @ lm3 @ lm2 @ 0 ↑ Suc n @ lm4}
 proof(rule_tac abc_Hoare_plus_halt)
   have {λ nl. nl = lm1 @ 0 ↑ n @ (0 # lm2) @ (butlast lm3) @ (last lm3 # lm4)} mv_boxes
aa ba n
        {λ nl. nl = lm1 @ butlast lm3 @ (0 # lm2) @ 0↑n @ (last lm3 # lm4)}
    using h1 h2 h3 h4 h5
    by(rule_tac ind, simp_all)
   moreover have lm1 @ 0 ↑ n @ (0 # lm2) @ (butlast lm3) @ (last lm3 # lm4)
             = lm1 @ 0 ↑ Suc n @ lm2 @ lm3 @ lm4
    using h5
    by(simp add: replicate_Suc_iff_anywhere exp_suc
       del: replicate_Suc, cases lm3, simp_all)
   ultimately show {λnl. nl = lm1 @ 0 ↑ Suc n @ lm2 @ lm3 @ lm4} mv_boxes aa ba n
    {λ nl. nl = lm1 @ butlast lm3 @ (0 # lm2) @ 0↑n @ (last lm3 # lm4)}
    by metis
 next
   thm mv_box_correct
   let ?lm = lm1 @ butlast lm3 @ (0 # lm2) @ 0 ↑ n @ last lm3 # lm4
   have {λnl. nl = ?lm} mv_box (aa + n) (ba + n)
      {λnl. nl = ?lm[ba+n := ?lm!(aa+n)+?lm!(ba+n), (aa+n):=0]}
    using h1 h2 h3 h4 h5
    by(rule_tac mv_box_correct, simp_all)
   moreover have ?lm[ba+n := ?lm!(aa+n)+?lm!(ba+n), (aa+n):=0]
          = lm1 @ lm3 @ lm2 @ 0 ↑ Suc n @ lm4
    using h1 h2 h3 h4 h5
    by(auto simp: nth_append list_update_append split: if_splits)
   ultimately show {λnl. nl = lm1 @ butlast lm3 @ (0 # lm2) @ 0 ↑ n @ last lm3 # lm4}
```

mv_box $(aa + n)$ $(ba + n)$
  $\{\lambda nl.\ nl = lm1\ @\ lm3\ @\ lm2\ @\ 0 \uparrow Suc\ n\ @\ lm4\}$
   by *simp*
 qed
 thus *?case*
  by(*simp add*: *mv_boxes.simps*)
qed

lemma *save_paras*:
 $\{\lambda nl.\ nl = xs\ @\ 0 \uparrow (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) - length\ xs)\ @$
 $map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ 0 \uparrow Suc\ (length\ xs)\ @\ anything\}$
 $mv\_boxes\ 0\ (Suc\ (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) + length\ gs))\ (length\ xs)$
 $\{\lambda nl.\ nl = 0 \uparrow max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))\ @\ map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ 0\ \#\ xs\ @\ anything\}$
proof $-$
 let $?ft = max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))$
 have $\{\lambda nl.\ nl = []\ @\ xs\ @\ (0 \uparrow (?ft - length\ xs)\ @\ map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ [0])\ @$
   $0 \uparrow (length\ xs)\ @\ anything\}\ mv\_boxes\ 0\ (Suc\ ?ft + length\ gs)\ (length\ xs)$
  $\{\lambda nl.\ nl = []\ @\ 0 \uparrow (length\ xs)\ @\ (0 \uparrow (?ft - length\ xs)\ @\ map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ [0])\ @\ xs\ @\ anything\}$
  by(*rule_tac mv_boxes_correct*, *auto*)
 thus *?thesis*
  by(*simp add*: *replicate_merge_anywhere*)
qed

lemma *length_le_max_insert_rec_ci*[*intro*]:
 $length\ gs \le ffp \Longrightarrow length\ gs \le max\ x1\ (Max\ (insert\ ffp\ (x2\ `\ x3\ `\ set\ gs)))$
 apply(*rule_tac max.coboundedI2*)
 apply(*simp add*: *Max_ge_iff*)
 done

lemma *restore_new_paras*:
 $ffp \ge length\ gs$
 $\Longrightarrow \{\lambda nl.\ nl = 0 \uparrow max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))\ @\ map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ 0\ \#\ xs\ @\ anything\}$
  $mv\_boxes\ (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))))\ 0\ (length\ gs)$
 $\{\lambda nl.\ nl = map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ 0 \uparrow max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))\ @\ 0\ \#\ xs\ @\ anything\}$
proof $-$
 let $?ft = max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))$
 assume j: $ffp \ge length\ gs$
 hence $\{\lambda\ nl.\ nl = []\ @\ 0 \uparrow length\ gs\ @\ 0 \uparrow (?ft - length\ gs)\ @\ map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ ((0\ \#\ xs)\ @\ anything)\}$
   $mv\_boxes\ ?ft\ 0\ (length\ gs)$
   $\{\lambda\ nl.\ nl = []\ @\ map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ 0 \uparrow (?ft - length\ gs)\ @\ 0 \uparrow length\ gs\ @\ ((0\ \#\ xs)\ @\ anything)\}$
  by(*rule_tac mv_boxes_correct2*, *auto*)

moreover have *?ft* $\geq$ *length gs*
  using *j*
  by(*auto*)
ultimately show *?thesis*
  using *j*
  by(*simp add*: *replicate_merge_anywhere le_add_diff_inverse*)
qed

lemma *le_max_insert*[*intro*]: *ffp* $\leq$ *max x0* (*Max* (*insert ffp* (*x1* ' *x2* ' *set gs*)))
 by (*rule max.coboundedI2*) *auto*

declare *max_less_iff_conj*[*simp del*]

lemma *save_rs*:
 ⟦*far* = *length gs*;
 ffp $\leq$ *max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ' *rec_ci* ' *set gs*)));
 far < *ffp*⟧
$\Longrightarrow$ {λ*nl*. *nl* = *map* (λ*i*. *rec_exec i xs*) *gs* @
 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*))) @ *xs* @ *anything*}
  mv_box far (*max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ' *rec_ci* ' *set 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*}
proof −
 let *?ft* = *max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((λ(*aprog*, *p*, *n*). *n*) ' *rec_ci* ' *set gs*)))
 thm *mv_box_correct*
 let *?lm*= *map* (λ*i*. *rec_exec i xs*) *gs* @ *rec_exec* (*Cn* (*length xs*) *f gs*) *xs* # *0* ↑ *?ft* @ *xs* @
anything
 assume *h*: *far* = *length gs ffp* $\leq$ *?ft far* < *ffp*
 hence {λ *nl*. *nl* = *?lm*} *mv_box far ?ft* {λ *nl*. *nl* = *?lm*[*?ft* := *?lm*!*far* + *?lm*!*?ft*, *far* := *0*]}
  apply(*rule_tac mv_box_correct*)
  by(*auto*)
 moreover have *?lm*[*?ft* := *?lm*!*far* + *?lm*!*?ft*, *far* := *0*]
  = *map* (λ*i*. *rec_exec i xs*) *gs* @
  0 ↑ (*?ft* − *length gs*) @
  rec_exec (*Cn* (*length xs*) *f gs*) *xs* # *0* ↑ *length gs* @ *xs* @ *anything*
  using *h*
  apply(*simp add*: *nth_append*)
  using *list_update_length*[*of map* (λ*i*. *rec_exec i xs*) *gs* @ *rec_exec* (*Cn* (*length xs*) *f gs*) *xs* #
   0 ↑ (*?ft* − *Suc* (*length gs*)) *0 0* ↑ *length gs* @ *xs* @ *anything rec_exec* (*Cn* (*length xs*) *f gs*)
xs]
  apply(*simp add*: *replicate_merge_anywhere replicate_Suc_iff_anywhere del*: *replicate_Suc*)
 by(*simp add*: *list_update_append list_update.simps replicate_Suc_iff_anywhere del*: *replicate_Suc*)
 ultimately show *?thesis*
  by(*simp*)
qed

lemma *length_empty_boxes*[*simp*]: *length* (*empty_boxes n*) = *2*∗*n*

```
 apply(induct n, simp, simp)
 done

lemma empty_one_box_correct:
 {λnl. nl = 0 ↑ n @ x # lm} [Dec n 2, Goto 0] {λnl. nl = 0 # 0 ↑ n @ lm}
proof(induct x)
 case 0
 thus ?case
  by(simp add: abc_Hoare_halt_def,
     rule_tac x = 1 in exI, simp add: abc_steps_l.simps
     abc_step_l.simps abc_fetch.simps abc_lm_v.simps nth_append abc_lm_s.simps
     replicate_Suc[THEN sym] exp_suc del: replicate_Suc)
next
 case (Suc x)
 have {λnl. nl = 0 ↑ n @ x # lm} [Dec n 2, Goto 0] {λnl. nl = 0 # 0 ↑ n @ lm}
  by fact
 then obtain stp where abc_steps_l (0, 0 ↑ n @ x # lm) [Dec n 2, Goto 0] stp
               = (Suc (Suc 0), 0 # 0 ↑ n @ lm)
  apply(auto simp: abc_Hoare_halt_def)
  by (smt abc_final.simps abc_holds_for.elims(2) length_Cons list.size(3))
 moreover have abc_steps_l (0, 0↑n @ Suc x # lm) [Dec n 2, Goto 0] (Suc (Suc 0))
     = (0,  0 ↑ n @ x # lm)
  by(auto simp: abc_steps_l.simps abc_step_l.simps abc_fetch.simps abc_lm_v.simps
     nth_append abc_lm_s.simps list_update.simps list_update_append)
 ultimately have abc_steps_l (0, 0↑n @ Suc x # lm) [Dec n 2, Goto 0] (Suc (Suc 0) + stp)
           = (Suc (Suc 0), 0 # 0↑n @ lm)
  by(simp only: abc_steps_add)
 thus ?case
  apply(simp add: abc_Hoare_halt_def)
  apply(rule_tac x = Suc (Suc stp) in exI, simp)
  done
qed

lemma empty_boxes_correct:
 length lm ≥ n ⟹
 {λ nl. nl = lm} empty_boxes n {λ nl. nl = 0↑n @ drop n lm}
proof(induct n)
 case 0
 thus ?case
  by(simp add: empty_boxes.simps abc_Hoare_halt_def,
     rule_tac x = 0 in exI, simp add: abc_steps_l.simps)
next
 case (Suc n)
 have ind: n ≤ length lm ⟹ {λnl. nl = lm} empty_boxes n {λnl. nl = 0 ↑ n @ drop n lm} by
fact
 have h: Suc n ≤ length lm by fact
 have {λnl. nl = lm} empty_boxes n [+] [Dec n 2, Goto 0] {λnl. nl = 0 # 0 ↑ n @ drop (Suc n)
lm}
 proof(rule_tac abc_Hoare_plus_halt)
  show {λnl. nl = lm} empty_boxes n {λnl. nl = 0 ↑ n @ drop n lm}
```

using *h*
by(*rule_tac ind*, *simp*)
next
show $\{\lambda nl.\ nl = 0 \uparrow n\ @\ drop\ n\ lm\}$ [*Dec n 2*, *Goto 0*] $\{\lambda nl.\ nl = 0\ \#\ 0 \uparrow n\ @\ drop\ (Suc\ n)\ lm\}$
using *empty_one_box_correct*[*of n lm* ! *n drop* (*Suc n*) *lm*]
using *h*
by(*simp add*: *Cons_nth_drop_Suc*)
qed
thus *?case*
by(*simp add*: *empty_boxes.simps*)
qed

lemma *insert_dominated*[*simp*]: $length\ gs \le ffp \Longrightarrow$
$length\ gs + (max\ xs\ (Max\ (insert\ ffp\ (x1\ `\ x2\ `\ set\ gs))) - length\ gs) =$
$max\ xs\ (Max\ (insert\ ffp\ (x1\ `\ x2\ `\ set\ gs)))$
apply(*rule_tac le_add_diff_inverse*)
apply(*rule_tac max.coboundedI2*)
apply(*simp add*: *Max_ge_iff*)
done

lemma *clean_paras*:
$ffp \ge length\ gs \Longrightarrow$
$\{\lambda nl.\ nl = map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @$
$0 \uparrow (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) - length\ gs)\ @$
$rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow length\ gs\ @\ xs\ @\ anything\}$
empty_boxes (*length gs*)
$\{\lambda nl.\ nl = 0 \uparrow max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))\ @$
$rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow length\ gs\ @\ xs\ @\ anything\}$
proof−
let $?ft = max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))$
assume *h*: $length\ gs \le ffp$
let $?lm = map\ (\lambda i.\ rec\_exec\ i\ xs)\ gs\ @\ 0 \uparrow (?ft - length\ gs)\ @$
$rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow length\ gs\ @\ xs\ @\ anything$
have $\{\lambda\ nl.\ nl = ?lm\}$ *empty_boxes* (*length gs*) $\{\lambda\ nl.\ nl = 0 \uparrow length\ gs\ @\ drop\ (length\ gs)\ ?lm\}$
by(*rule_tac empty_boxes_correct*, *simp*)
moreover have $0 \uparrow length\ gs\ @\ drop\ (length\ gs)\ ?lm$
$= 0 \uparrow ?ft\ @\ rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow length\ gs\ @\ xs\ @\ anything$
using *h*
by(*simp add*: *replicate_merge_anywhere*)
ultimately show *?thesis*
by *metis*
qed

lemma *restore_rs*:

$\{\lambda nl.\ nl = 0 \uparrow max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))$ @
$rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow length\ gs$ @ xs @ $anything\}$
$mv\_box\ (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))))\ (length\ xs)$
$\{\lambda nl.\ nl = 0 \uparrow length\ xs$ @
$rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#$
$0 \uparrow (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) - (length\ xs))$ @
$0 \uparrow length\ gs$ @ xs @ $anything\}$
proof $-$
let $?ft = max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))$
let $?lm = 0\uparrow(length\ xs)$ @ $0\uparrow(?ft - (length\ xs))$ @ $rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow length\ gs$ @ xs @ $anything$
thm *mv_box_correct*
have $\{\lambda\ nl.\ nl = ?lm\}\ mv\_box\ ?ft\ (length\ xs)\ \{\lambda\ nl.\ nl = ?lm[length\ xs := ?lm!?ft + ?lm!(length\ xs),\ ?ft := 0]\}$
by(*rule_tac mv_box_correct*, *simp*, *simp*)
moreover have $?lm[length\ xs := ?lm!?ft + ?lm!(length\ xs),\ ?ft := 0]$
$= 0 \uparrow length\ xs$ @ $rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow (?ft - (length\ xs))$ @ $0 \uparrow length\ gs$ @ xs @ $anything$
apply(*auto simp*: *list_update_append nth_append*)
apply(*cases ?ft*, *simp_all add*: *Suc_diff_le list_update.simps*)
apply(*simp add*: *exp_suc replicate_Suc*[*THEN sym*] *del*: *replicate_Suc*)
done
ultimately show *?thesis*
by(*simp add*: *replicate_merge_anywhere*)
qed

lemma *restore_orgin_paras*:
$\{\lambda nl.\ nl = 0 \uparrow length\ xs$ @
$rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#$
$0 \uparrow (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) - length\ xs)$ @ $0 \uparrow length\ gs$ @ xs @ $anything\}$
$mv\_boxes\ (Suc\ (max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) + length\ gs))\ 0\ (length\ xs)$
$\{\lambda nl.\ nl = xs$ @ $rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow$
$(max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs))) + length\ gs)$ @ $anything\}$
proof $-$
let $?ft = max\ (Suc\ (length\ xs))\ (Max\ (insert\ ffp\ ((\lambda(aprog, p, n).\ n)\ `\ rec\_ci\ `\ set\ gs)))$
thm *mv_boxes_correct2*
have $\{\lambda\ nl.\ nl = []$ @ $0\uparrow(length\ xs)$ @ $(rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow (?ft - length\ xs)$ @ $0 \uparrow length\ gs)$ @ xs @ $anything\}$
$mv\_boxes\ (Suc\ ?ft + length\ gs)\ 0\ (length\ xs)$
$\{\lambda\ nl.\ nl = []$ @ xs @ $(rec\_exec\ (Cn\ (length\ xs)\ f\ gs)\ xs\ \#\ 0 \uparrow (?ft - length\ xs)$ @ $0 \uparrow length\ gs)$ @ $0\uparrow length\ xs$ @ $anything\}$
by(*rule_tac mv_boxes_correct2*, *auto*)
thus *?thesis*
by(*simp add*: *replicate_merge_anywhere*)

qed

lemma *compile_cn_correct′*:
assumes *f_ind*:
$\bigwedge$ *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 −
let *?ft* = *max* (*Suc* (*length xs*)) (*Max* (*insert ffp* ((*λ*(*aprog*, *p*, *n*). *n*) ‘ *rec_ci* ‘ *set gs*)))
let *?A* = *cn_merge_gs* (*map rec_ci gs*) *?ft*
let *?B* = *mv_boxes 0* (*Suc* (*?ft*+*length gs*)) (*length xs*)
let *?C* = *mv_boxes ?ft 0* (*length gs*)
let *?D* = *fap*
let *?E* = *mv_box far ?ft*
let *?F* = *empty_boxes* (*length gs*)
let *?G* = *mv_box ?ft* (*length xs*)
let *?H* = *mv_boxes* (*Suc* (*?ft* + *length gs*)) *0* (*length xs*)
let *?P1* = *λnl. nl* = *xs* @ *0* # *0* ↑ (*?ft* + *length gs*) @ *anything*
let *?S* = *λnl. nl* = *xs* @ *rec_exec* (*Cn* (*length xs*) *f gs*) *xs* # *0* ↑ (*?ft* + *length gs*) @ *anything*
let *?Q1* = *λ nl. nl* = *xs* @ *0*↑(*?ft* − *length xs*) @ *map* (*λ i. rec_exec i xs*) *gs* @ *0*↑(*Suc* (*length xs*)) @ *anything*
show {*?P1*} (*?A* [+] (*?B* [+] (*?C* [+] (*?D* [+] (*?E* [+] (*?F* [+] (*?G* [+] *?H*))))))) {*?S*}
proof(*rule_tac abc_Hoare_plus_halt*)
show {*?P1*} *?A* {*?Q1*}

```
    using g_cond
    by(rule_tac compile_cn_gs_correct, auto)
  next
   let ?Q2 = λnl. nl = 0 ↑ ?ft @
              map (λi. rec_exec i xs) gs @ 0 # xs @ anything
   show {?Q1} (?B [+] (?C [+] (?D [+] (?E [+] (?F [+] (?G [+] ?H)))))) {?S}
   proof(rule_tac abc_Hoare_plus_halt)
    show {?Q1} ?B {?Q2}
     by(rule_tac save_paras)
   next
    let ?Q3 = λ nl. nl = map (λi. rec_exec i xs) gs @ 0↑?ft @ 0 # xs @ anything
    show {?Q2} (?C [+] (?D [+] (?E [+] (?F [+] (?G [+] ?H))))) {?S}
    proof(rule_tac abc_Hoare_plus_halt)
     have ffp ≥ length gs
      using compile term_f
      apply(subgoal_tac length gs = far)
       apply(drule_tac footprint_ge, simp)
      by(drule_tac param_pattern, auto)
     thus {?Q2} ?C {?Q3}
      by(erule_tac restore_new_paras)
    next
     let ?Q4 = λ nl. nl = map (λi. rec_exec i xs) gs @ rec_exec (Cn (length xs) f gs) xs # 0↑?ft
@ xs @ anything
     have a: far = length gs
      using compile term_f
      by(drule_tac param_pattern, auto)
     have b:?ft ≥ ffp
      by auto
     have c: ffp > far
      using compile
      by(erule_tac footprint_ge)
     show {?Q3} (?D [+] (?E [+] (?F [+] (?G [+] ?H)))) {?S}
     proof(rule_tac abc_Hoare_plus_halt)
       have {λnl. nl = map (λg. rec_exec g xs) gs @ 0 ↑ (ffp − far) @ 0↑(?ft − ffp + far) @ 0
# xs @ anything} fap
        {λnl. nl = map (λg. rec_exec g xs) gs @ rec_exec (Cn (length xs) f gs) xs #
        0 ↑ (ffp − Suc far) @ 0↑(?ft − ffp + far) @ 0 # xs @ anything}
        by(rule_tac f_ind, simp add: rec_exec.simps)
      thus {?Q3} fap {?Q4}
       using a b c
       by(simp add: replicate_merge_anywhere,
          cases ?ft, simp_all add: exp_suc del: replicate_Suc)
     next
      let ?Q5 = λnl. nl = map (λi. rec_exec i xs) gs @
        0↑(?ft − length gs) @ rec_exec (Cn (length xs) f gs) xs # 0↑(length gs)@ xs @ anything
      show {?Q4} (?E [+] (?F [+] (?G [+] ?H))) {?S}
      proof(rule_tac abc_Hoare_plus_halt)
       from a b c show {?Q4} ?E {?Q5}
        by(erule_tac save_rs, simp_all)
      next
```

```
          let ?Q6 = λnl. nl = 0↑?ft @ rec_exec (Cn (length xs) f gs) xs # 0↑(length gs)@ xs @
anything
          show {?Q5} (?F [+] (?G [+] ?H)) {?S}
          proof(rule_tac abc_Hoare_plus_halt)
            have length gs ≤ ffp using a b c
              by simp
            thus {?Q5} ?F {?Q6}
              by(erule_tac clean_paras)
          next
          let ?Q7 = λnl. nl = 0↑length xs @ rec_exec (Cn (length xs) f gs) xs # 0↑(?ft − (length
xs)) @ 0↑(length gs)@ xs @ anything
            show {?Q6} (?G [+] ?H) {?S}
            proof(rule_tac abc_Hoare_plus_halt)
              show {?Q6} ?G {?Q7}
                by(rule_tac restore_rs)
            next
              show {?Q7} ?H {?S}
                by(rule_tac restore_orgin_paras)
            qed
          qed
        qed
      qed
    qed
  qed
 qed
qed

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 ↑ (xb − Suc xa) @ xc}))
   and compile: rec_ci (Cn n f gs) = (ap, arity, fp)
   and len: length xs = n
 shows {λnl. nl = xs @ 0 ↑ (fp − arity) @ anything} ap {λnl. nl = xs @ rec_exec (Cn n f gs)
xs # 0 ↑ (fp − Suc arity) @ anything}
proof(cases rec_ci f)
 fix fap far ffp
 assume h: rec_ci f = (fap, far, ffp)
 then have f_newind: ⋀ anything .{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 f (map (λg. rec_exec g xs) gs) # 0 ↑ (ffp −
Suc far) @ anything}
   by(rule_tac f_ind, simp_all)
```

thus $\{\lambda nl.\ nl = xs\ @\ 0 \uparrow (fp - arity)\ @\ anything\}$ *ap* $\{\lambda nl.\ nl = xs\ @\ rec\_exec\ (Cn\ n\ f\ gs)\ xs\ \#\ 0 \uparrow (fp - Suc\ arity)\ @\ anything\}$
using *compile len h termi_f g_cond*
apply(*auto simp*: *rec_ci.simps abc_comp_commute*)
apply(*rule_tac compile_cn_correct′*, *simp_all*)
done
qed

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*
$\{\lambda nl.\ nl = xs\ @\ 0\ \#\ 0 \uparrow (ft - n)\ @\ anything\}$
using *mv_box_correct*[*of n ft xs* $@\ 0\ \#\ 0 \uparrow (ft - n)\ @\ anything$]
by(*auto*)

lemma *length_under_max*[*simp*]: $length\ xs < max\ (length\ xs + 3)\ fft$
by *auto*

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*)
$\{\lambda nl.\ nl = xs\ @\ 0\ \#\ rec\_exec\ f\ xs\ \#\ 0 \uparrow (ft - Suc\ n)\ @\ anything\}$
using *mv_box_correct*[*of n Suc n xs* $@\ rec\_exec\ f\ xs\ \#\ 0 \uparrow (ft - n)\ @\ anything$]
apply(*auto simp*: *list_update_append list_update.simps nth_append split*: *if_splits*)
apply(*cases* (*max* (*length xs* + *3*) (*max fft gft*)), *simp_all add*: *list_update.simps Suc_diff_le*)
done

lemma *less_then_max_plus2*[*simp*]: $n + (2{::}nat) < max\ (n + 3)\ x$
by *auto*

lemma *less_then_max_plus3*[*simp*]: $n < max\ (n + (3{::}nat))\ x$
by *auto*

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*))
$\{\lambda nl.\ nl = xs\ @\ 0 \uparrow (max\ (n + 3)\ (max\ fft\ gft) - n)\ @\ x\ \#\ anything\}$
proof −
assume *h*: *length xs* = *n*
let *?ft* = *max* (*n*+*3*) (*max fft gft*)
let $?lm = xs\ @\ x\ \#\ 0{\uparrow}(?ft - Suc\ n)\ @\ 0\ \#\ anything$
have *g*: $?ft > n + 2$
by *simp*
thm *mv_box_correct*
have *a*: $\{\lambda\ nl.\ nl = ?lm\}$ *mv_box n ?ft* $\{\lambda\ nl.\ nl = ?lm[?ft := ?lm!n + ?lm!?ft, n := 0]\}$
using *h*
by(*rule_tac mv_box_correct*, *auto*)
have *b*: $?lm = xs\ @\ x\ \#\ 0 \uparrow (max\ (n + 3)\ (max\ fft\ gft) - n)\ @\ anything$
by(*cases ?ft*, *simp_all add*: *Suc_diff_le exp_suc del*: *replicate_Suc*)

```
 have c: ?lm[?ft := ?lm!n + ?lm!?ft, n := 0] = xs @ 0 ↑ (max (n + 3) (max fft gft) − n) @ x # anything
  using h g
  apply(auto simp: nth_append list_update_append split: if_splits)
  using list_update_append[of x # 0 ↑ (max (length xs + 3) (max fft gft) − Suc (length xs)) 0 # anything
     max (length xs + 3) (max fft gft) − length xs x]
  apply(auto simp: if_splits)
  apply(simp add: list_update.simps replicate_Suc[THEN sym] del: replicate_Suc)
  done
 from a c show ?thesis
  using h
  apply(simp)
  using b
  by simp
qed

lemma max_less_suc_suc[simp]: max n (Suc n) < Suc (Suc (max (n + 3) x + anything − Suc 0))
 by arith

lemma suc_less_plus_3[simp]: Suc n < max (n + 3) x
 by arith

lemma mv_box_ok_suc_simp[simp]:
 length xs = n
 ⟹ {λnl. nl = xs @ rec_exec f xs # 0 ↑ (max (n + 3) (max fft gft) − Suc n) @ x # anything} mv_box n (Suc n)
   {λnl. nl = xs @ 0 # rec_exec f xs # 0 ↑ (max (n + 3) (max fft gft) − Suc (Suc n)) @ x # anything}
 using mv_box_correct[of n Suc n xs @ rec_exec f xs # 0 ↑ (max (n + 3) (max fft gft) − Suc n) @ x # anything]
 apply(simp add: nth_append list_update_append list_update.simps)
 apply(cases max (n + 3) (max fft gft), simp_all)
 apply(cases max (n + 3) (max fft gft) − 1, simp_all add: Suc_diff_le list_update.simps(2))
 done

lemma abc_append_frist_steps_eq_pre:
 assumes notfinal: abc_notfinal (abc_steps_l (0, lm)  A n) A
  and notnull: A ≠ []
 shows abc_steps_l (0, lm) (A @ B) n = abc_steps_l (0, lm) A n
 using notfinal
proof(induct n)
 case 0
 thus ?case
  by(simp add: abc_steps_l.simps)
next
 case (Suc n)
 have ind: abc_notfinal (abc_steps_l (0, lm) A n) A ⟹ abc_steps_l (0, lm) (A @ B) n = abc_steps_l (0, lm) A n
```

```
    by fact
  have h: abc_notfinal (abc_steps_l (0, lm) A (Suc n)) A by fact
  then have a: abc_notfinal (abc_steps_l (0, lm) A n) A
    by(simp add: notfinal_Suc)
  then have b: abc_steps_l (0, lm) (A @ B) n = abc_steps_l (0, lm) A n
    using ind by simp
  obtain s lm' where c: abc_steps_l (0, lm) A n = (s, lm')
    by (metis prod.exhaust)
  then have d: s < length A ∧ abc_steps_l (0, lm) (A @ B) n = (s, lm')
    using a b by simp
  thus ?case
    using c
    by(simp add: abc_step_red2 abc_fetch.simps abc_step_l.simps nth_append)
qed

lemma abc_append_first_step_eq_pre:
  st < length A
 ⟹ abc_step_l (st, lm) (abc_fetch st (A @ B)) =
    abc_step_l (st, lm) (abc_fetch st A)
  by(simp add: abc_step_l.simps abc_fetch.simps nth_append)

lemma abc_append_frist_steps_halt_eq':
  assumes final: abc_steps_l (0, lm) A n = (length A, lm')
    and notnull: A ≠ []
  shows ∃ n'. abc_steps_l (0, lm) (A @ B) n' = (length A, lm')
proof -
  have ∃ n'. abc_notfinal (abc_steps_l (0, lm) A n') A ∧
    abc_final (abc_steps_l (0, lm) A (Suc n')) A
    using assms
    by(rule_tac n = n in abc_before_final, simp_all)
  then obtain na where a:
    abc_notfinal (abc_steps_l (0, lm) A na) A ∧
            abc_final (abc_steps_l (0, lm) A (Suc na)) A ..
  obtain sa lma where b: abc_steps_l (0, lm) A na = (sa, lma)
    by (metis prod.exhaust)
  then have c: abc_steps_l (0, lm) (A @ B) na = (sa, lma)
    using a abc_append_frist_steps_eq_pre[of lm A na B] assms
    by simp
  have d: sa < length A using b a by simp
  then have e: abc_step_l (sa, lma) (abc_fetch sa (A @ B)) =
    abc_step_l (sa, lma) (abc_fetch sa A)
    by(rule_tac abc_append_first_step_eq_pre)
  from a have abc_steps_l (0, lm) A (Suc na) = (length A, lm')
    using final equal_when_halt
    by(cases abc_steps_l (0, lm) A (Suc na) , simp)
  then have abc_steps_l (0, lm) (A @ B) (Suc na) = (length A, lm')
    using a b c e
    by(simp add: abc_step_red2)
  thus ?thesis
    by blast
```

qed

lemma *abc_append_frist_steps_halt_eq*:
  assumes *final*: *abc_steps_l* $(0, lm)$ A $n = (length\ A, lm')$
  shows $\exists\ n'$. *abc_steps_l* $(0, lm)$ $(A\ @\ B)$ $n' = (length\ A, lm')$
  using *final*
  apply(*cases* $A = []$)
  apply(*rule_tac* $x = 0$ **in** *exI*, *simp add*: *abc_steps_l.simps abc_exec_null*)
  apply(*rule_tac abc_append_frist_steps_halt_eq'*, *simp_all*)
  done

lemma *suc_suc_max_simp*[*simp*]: $Suc\ (Suc\ (max\ (xs + 3)\ fft - Suc\ (Suc\ (\ xs))))$
        $= max\ (\ xs + 3)\ fft - (\ xs)$
  by *arith*

lemma *contract_dec_ft_length_plus_7*[*simp*]: $[\![ft = max\ (n + 3)\ (max\ fft\ gft); length\ xs = n]\!] \Longrightarrow$
    $\{\lambda nl.\ nl = xs\ @\ (x - Suc\ y)\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [x - Suc\ y])\ \#\ 0 \uparrow (ft - Suc\ (Suc\ n))\ @\ Suc\ y\ \#\ anything\}$
    $[Dec\ ft\ (length\ gap + 7)]$
    $\{\lambda nl.\ nl = xs\ @\ (x - Suc\ y)\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [x - Suc\ y])\ \#\ 0 \uparrow (ft - Suc\ (Suc\ n))\ @\ y\ \#\ anything\}$
  apply(*simp add*: *abc_Hoare_halt_def*)
  apply(*rule_tac* $x = 1$ **in** *exI*)
  apply(*auto simp*: *abc_steps_l.simps abc_step_l.simps abc_fetch.simps nth_append*
    abc_lm_v.simps abc_lm_s.simps list_update_append)
  using *list_update_length*
   $[of\ (x - Suc\ y)\ \#\ rec\_exec\ (Pr\ (length\ xs)\ f\ g)\ (xs\ @\ [x - Suc\ y])\ \#$
      $0 \uparrow (max\ (length\ xs + 3)\ (max\ fft\ gft) - Suc\ (Suc\ (length\ xs)))\ Suc\ y\ anything\ y]$
  apply(*simp*)
  done

lemma *adjust_paras'*:
  $length\ xs = n \Longrightarrow \{\lambda nl.\ nl = xs\ @\ x\ \#\ y\ \#\ anything\}\ \ [Inc\ n]\ [+]\ [Dec\ (Suc\ n)\ 2, Goto\ 0]$
    $\{\lambda nl.\ nl = xs\ @\ Suc\ x\ \#\ 0\ \#\ anything\}$
proof(*rule_tac abc_Hoare_plus_halt*)
  assume $length\ xs = n$
  thus $\{\lambda nl.\ nl = xs\ @\ x\ \#\ y\ \#\ anything\}\ [Inc\ n]\ \{\lambda\ nl.\ nl = xs\ @\ Suc\ x\ \#\ y\ \#\ anything\}$
   apply(*simp add*: *abc_Hoare_halt_def*)
   apply(*rule_tac* $x = 1$ **in** *exI*, *force simp add*: *abc_steps_l.simps abc_step_l.simps*
     abc_fetch.simps abc_comp.simps
     abc_lm_v.simps abc_lm_s.simps nth_append list_update_append list_update.simps(2))
   done
next
  assume *h*: $length\ xs = n$
  thus $\{\lambda nl.\ nl = xs\ @\ Suc\ x\ \#\ y\ \#\ anything\}\ [Dec\ (Suc\ n)\ 2, Goto\ 0]\ \{\lambda nl.\ nl = xs\ @\ Suc\ x\ \#\ 0\ \#\ anything\}$
  proof(*induct y*)
   case *0*
   thus *?case*
    apply(*simp add*: *abc_Hoare_halt_def*)

apply(*rule_tac x = 1* **in** *exI*, *simp add*: *abc_steps_l.simps abc_step_l.simps abc_fetch.simps abc_comp.simps*
abc_lm_v.simps abc_lm_s.simps nth_append list_update_append list_update.simps(*2*))
done
next
case (*Suc y*)
have *length xs* = *n* ⟹
{λ*nl*. *nl* = *xs* @ *Suc x* # *y* # *anything*} [*Dec* (*Suc n*) *2*, *Goto 0*] {λ*nl*. *nl* = *xs* @ *Suc x* # *0* # *anything*} **by** *fact*
then obtain *stp* **where**
abc_steps_l (*0*, *xs* @ *Suc x* # *y* # *anything*) [*Dec* (*Suc n*) *2*, *Goto 0*] *stp* = (*2*, *xs* @ *Suc x* # *0* # *anything*)
using *h*
apply(*auto simp*: *abc_Hoare_halt_def numeral_2_eq_2*)
by (*metis* (*mono_tags*, *lifting*) *abc_final.simps abc_holds_for.elims*(*2*) *length_Cons list.size*(*3*))
moreover have *abc_steps_l* (*0*, *xs* @ *Suc x* # *Suc y* # *anything*) [*Dec* (*Suc n*) *2*, *Goto 0*] *2* =
(*0*, *xs* @ *Suc x* # *y* # *anything*)
using *h*
by(*simp add*: *abc_steps_l.simps numeral_2_eq_2 abc_step_l.simps abc_fetch.simps*
abc_lm_v.simps abc_lm_s.simps nth_append list_update_append list_update.simps(*2*))
ultimately show *?case*
apply(*simp add*: *abc_Hoare_halt_def*)
by(*rule exI*[*of _ 2* + *stp*], *simp only*: *abc_steps_add*, *simp*)
qed
qed

lemma *adjust_paras*:
length xs = *n* ⟹ {λ*nl*. *nl* = *xs* @ *x* # *y* # *anything*} [*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)]
{λ*nl*. *nl* = *xs* @ *Suc x* # *0* # *anything*}
using *adjust_paras*′[*of xs n x y anything*]
by(*simp add*: *abc_comp.simps abc_shift.simps numeral_2_eq_2 numeral_3_eq_3*)

lemma *rec_ci_SucSuc_n*[*simp*]: ⟦*rec_ci g* = (*gap*, *gar*, *gft*); ∀ *y*<*x*. *terminate g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])]);
length xs = *n*; *Suc y*≤*x*⟧ ⟹ *gar* = *Suc* (*Suc n*)
by(*auto dest*:*param_pattern elim*!:*allE*[*of _ y*])

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*)
using *le*
proof(*induct x*)
case *0*
thus *?case*
using *h*
by(*rule_tac x* = *0* **in** *exI*,
auto simp: *abc_steps_l.simps abc_step_l.simps abc_fetch.simps nth_append abc_lm_s.simps*

```
abc_lm_v.simps)
next
  case (Suc x)
  have x ≤ y ⟹ ∃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) by fact
  moreover have Suc x ≤ y by fact
  moreover then have ∃ stp. abc_steps_l (length A, xs @ m # (y − Suc x) # Suc x # anything)
(A @ [Dec (Suc (Suc n)) 0, Inc (Suc n), Goto (length gap + 4)]) stp
              = (length A, xs @ m # (y − x) # x # anything)
    using h
    apply(rule_tac x = 3 in exI)
    by(simp add: abc_steps_l.simps numeral_3_eq_3 abc_step_l.simps abc_fetch.simps nth_append
       abc_lm_v.simps abc_lm_s.simps list_update_append list_update.simps(2))
  ultimately show ?case
    apply(auto simp add: abc_steps_add)
    by (metis abc_steps_add)
qed


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)
  using loop_back'[of A gap xs n x x m anything] assms
  apply(auto) apply(rename_tac stp)
  apply(rule_tac x = stp + 1 in exI)
  apply(simp only: abc_steps_add, simp)
  apply(simp add: abc_steps_l.simps abc_step_l.simps abc_fetch.simps nth_append abc_lm_v.simps
     abc_lm_s.simps)
  done

lemma rec_exec_pr_0_simps: rec_exec (Pr n f g) (xs @ [0]) = rec_exec f xs
  by(simp add: rec_exec.simps)

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])])
  apply(induct y)
   apply(simp add: rec_exec.simps)
  apply(simp add: rec_exec.simps)
  done

lemma suc_max_simp[simp]: Suc (max (n + 3) fft − Suc (Suc (Suc n))) = max (n + 3) fft −
Suc (Suc n)
  by arith

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 −
let *?A* = [*Dec ft* (*length gap* + *7*)]
let *?B* = *gap*
let *?C* = [*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)]
let *?D* = [*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length gap* + *4*)]
have ∃ *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*)
((*?A* [+] (*?B* [+] *?C*)) @ *?D*) *stp* = (*length* (*?A* [+] (*?B* [+] *?C*)),
xs @ (*x* − *y*) # *0* # *rec_exec g* (*xs* @ [*x* − *Suc y*, *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*])])
0 ↑ (*ft* − *Suc* (*Suc* (*Suc n*))) @ *y* # *anything*)
proof −
have ∃ *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*)
((*?A* [+] (*?B* [+] *?C*))) *stp* = (*length* (*?A* [+] (*?B* [+] *?C*)), *xs* @ (*x* − *y*) # *0* #
rec_exec g (*xs* @ [*x* − *Suc y*, *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*])]) # *0* ↑ (*ft* − *Suc* (*Suc* (*Suc n*))) @ *y* # *anything*)
proof −
have {λ *nl*. *nl* = *xs* @ (*x* − *Suc y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*]) # *0* ↑ (*ft* − *Suc* (*Suc n*)) @ *Suc y* # *anything*}
(*?A* [+] (*?B* [+] *?C*))
{λ *nl*. *nl* = *xs* @ (*x* − *y*) # *0* #
rec_exec g (*xs* @ [*x* − *Suc y*, *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*])]) # *0* ↑ (*ft* − *Suc* (*Suc* (*Suc n*))) @ *y* # *anything*}
proof(*rule_tac abc_Hoare_plus_halt*)
show {λ*nl*. *nl* = *xs* @ (*x* − *Suc y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*]) # *0* ↑ (*ft* − *Suc* (*Suc n*)) @ *Suc y* # *anything*}
[*Dec ft* (*length gap* + *7*)]
{λ*nl*. *nl* = *xs* @ (*x* − *Suc y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*]) # *0* ↑ (*ft* − *Suc* (*Suc n*)) @ *y* # *anything*}
using *ft len*
by(*simp*)
next
show
{λ*nl*. *nl* = *xs* @ (*x* − *Suc y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*]) # *0* ↑ (*ft* − *Suc* (*Suc n*)) @ *y* # *anything*}

```
      ?B [+] ?C
      {λnl. nl = xs @ (x − y) # 0 # rec_exec g (xs @ [x − Suc y, rec_exec (Pr n f g) (xs @ [x
− Suc y])]) # 0 ↑ (ft − Suc (Suc (Suc n))) @ y # anything}
    proof(rule_tac abc_Hoare_plus_halt)
      have a: gar = Suc (Suc n)
        using compile_g termi_g len less
        by simp
      have b: gft > gar
        using compile_g
        by(erule_tac footprint_ge)
      show {λnl. nl = xs @ (x − Suc y) # rec_exec (Pr n f g) (xs @ [x − Suc y]) # 0 ↑ (ft −
Suc (Suc n)) @ y # anything} gap
            {λnl. nl = xs @ (x − Suc y) # rec_exec (Pr n f g) (xs @ [x − Suc y]) #
                rec_exec g (xs @ [x − Suc y, rec_exec (Pr n f g) (xs @ [x − Suc y])]) # 0 ↑ (ft −
Suc (Suc (Suc n))) @ y # anything}
      proof −
        have
          {λnl. nl = xs @ (x − Suc y) # rec_exec (Pr n f g) (xs @ [x − Suc y]) # 0 ↑ (gft − gar)
@ 0↑(ft − gft) @ y # anything} gap
          {λnl. nl = xs @ (x − Suc y) # rec_exec (Pr n f g) (xs @ [x − Suc y]) #
          rec_exec g (xs @ [(x − Suc y), rec_exec (Pr n f g) (xs @ [x − Suc y])]) # 0 ↑ (gft − Suc
gar) @ 0↑(ft − gft) @ y # anything}
          using g_ind less by simp
        thus ?thesis
          using a b ft
          by(simp add: replicate_merge_anywhere numeral_3_eq_3)
      qed
    next
      show {λnl. nl = xs @ (x − Suc y) #
              rec_exec (Pr n f g) (xs @ [x − Suc y]) #
          rec_exec g (xs @ [x − Suc y, rec_exec (Pr n f g) (xs @ [x − Suc y])]) # 0 ↑ (ft − Suc
(Suc (Suc n))) @ y # anything}
        [Inc n, Dec (Suc n) 3, Goto (Suc 0)]
        {λnl. nl = xs @ (x − y) # 0 # rec_exec g (xs @ [x − Suc y, rec_exec (Pr n f g)
              (xs @ [x − Suc y])]) # 0 ↑ (ft − Suc (Suc (Suc n))) @ y # anything}
        using len less
        using adjust_paras[of xs n x − Suc y  rec_exec (Pr n f g) (xs @ [x − Suc y])
            rec_exec g (xs @ [x − Suc y, rec_exec (Pr n f g) (xs @ [x − Suc y])]) #
          0 ↑ (ft − Suc (Suc (Suc n))) @ y # anything]
        by(simp add: Suc_diff_Suc)
    qed
  qed
  thus ?thesis
    apply(simp add: abc_Hoare_halt_def, auto)
    apply(rename_tac na)
    apply(rule_tac x = na in exI, case_tac 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)
        ([Dec ft (length gap + 7)] [+] (gap [+] [Inc n, Dec (Suc n) 3, Goto (Suc 0)])) na, simp)
    done
```

qed
then obtain *stpa* **where** *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*)
((*?A* [+] (*?B* [+] *?C*))) *stpa* = (*length* (*?A* [+] (*?B* [+] *?C*)),
xs @ (*x* − *y*) # *0* # *rec_exec g* (*xs* @ [*x* − *Suc y*, *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*])])
0 ↑ (*ft* − *Suc* (*Suc* (*Suc n*))) @ *y* # *anything*) **..**
thus *?thesis*
by(*erule_tac abc_append_frist_steps_halt_eq*)
qed
moreover have
∃ *stp*. *abc_steps_l* (*length* (*?A* [+] (*?B* [+] *?C*)),
xs @ (*x* − *y*) # *0* # *rec_exec g* (*xs* @ [*x* − *Suc y*, *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*])]) # *0* ↑ (*ft* − *Suc* (*Suc* (*Suc n*))) @ *y* # *anything*)
((*?A* [+] (*?B* [+] *?C*)) @ *?D*) *stp* = (*0*, *xs* @ (*x* − *y*) # *rec_exec g* (*xs* @ [*x* − *Suc y*, *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*])]) #
0 # *0* ↑ (*ft* − *Suc* (*Suc* (*Suc n*))) @ *y* # *anything*)
using *len*
by(*rule_tac loop_back*, *simp_all*)
moreover have *rec_exec g* (*xs* @ [*x* − *Suc y*, *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *Suc y*])]) = *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *y*])
using *less*
apply(*cases x* − *y*, *simp_all add*: *rec_exec_pr_Suc_simps*)
apply(*rename_tac nat*)
by(*subgoal_tac nat* = *x* − *Suc y*, *simp*, *arith*)
ultimately show *?thesis*
using *code ft*
apply (*auto simp add*: *abc_steps_add replicate_Suc_iff_anywhere*)
apply(*rename_tac stp stpa*)
apply(*rule_tac x* = *stp* + *stpa* **in** *exI*)
by (*simp add*: *abc_steps_add replicate_Suc_iff_anywhere del*: *replicate_Suc*)
qed

lemma *abc_lm_s_simp0*[*simp*]:
length xs = *n* ⟹ *abc_lm_s* (*xs* @ *x* # *rec_exec* (*Pr n f g*) (*xs* @ [*x*]) # *0* ↑ (*max* (*n* + *3*) (*max fft gft*) − *Suc* (*Suc n*)) @ *0* # *anything*) (*max* (*n* + *3*) (*max fft gft*)) *0* =
xs @ *x* # *rec_exec* (*Pr n f g*) (*xs* @ [*x*]) # *0* ↑ (*max* (*n* + *3*) (*max fft gft*) − *Suc n*) @ *anything*
apply(*simp add*: *abc_lm_s.simps*)
using *list_update_length*[*of xs* @ *x* # *rec_exec* (*Pr n f g*) (*xs* @ [*x*]) # *0* ↑ (*max* (*n* + *3*) (*max fft gft*) − *Suc* (*Suc n*))
0 anything 0]
apply(*auto simp*: *Suc_diff_Suc*)
apply(*simp add*: *exp_suc*[*THEN sym*] *Suc_diff_Suc del*: *replicate_Suc*)
done

lemma *index_at_zero_elem*[*simp*]:
(*xs* @ *x* # *re* # *0* ↑ (*max* (*length xs* + *3*)
(*max fft gft*) − *Suc* (*Suc* (*length xs*))) @ *0* # *anything*) !
max (*length xs* + *3*) (*max fft gft*) = *0*
using *nth_append_length*[*of xs* @ *x* # *re* #
0 ↑ (*max* (*length xs* + *3*) (*max fft gft*) − *Suc* (*Suc* (*length xs*))) *0 anything*]

by(*simp*)

lemma *pr_loop_correct*:
assumes *less*: $y \leq x$
and *len*: *length xs* = *n*
and *compile_g*: *rec_ci g* = (*gap*, *gar*, *gft*)
and *termi_g*: ∀ *y<x*. *terminate g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])])
and *g_ind*: ∀ *y<x*. (∀ *anything*. {λ*nl*. *nl* = *xs* @ *y* # *rec_exec* (*Pr n f g*) (*xs* @ [*y*]) # *0* ↑ (*gft* − *gar*) @ *anything*} *gap*
{λ*nl*. *nl* = *xs* @ *y* # *rec_exec* (*Pr n f g*) (*xs* @ [*y*]) # *rec_exec g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])]) # *0* ↑ (*gft* − *Suc gar*) @ *anything*})
shows {λ*nl*. *nl* = *xs* @ (*x* − *y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *y*]) # *0* ↑ (*max* (*n* + *3*) (*max fft gft*) − *Suc* (*Suc n*)) @ *y* # *anything*}
([*Dec* (*max* (*n* + *3*) (*max fft gft*)) (*length gap* + *7*)] [+] (*gap* [+] [*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)])) @ [*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length gap* + *4*)]
{λ*nl*. *nl* = *xs* @ *x* # *rec_exec* (*Pr n f g*) (*xs* @ [*x*]) # *0* ↑ (*max* (*n* + *3*) (*max fft gft*) − *Suc n*) @ *anything*}
using *less*
proof(*induct y*)
case *0*
thus *?case*
using *len*
apply(*simp add*: *abc_Hoare_halt_def*)
apply(*rule_tac x* = *1* **in** *exI*)
by(*auto simp*: *abc_steps_l.simps abc_step_l.simps abc_fetch.simps*
nth_append abc_comp.simps abc_shift.simps, *simp add*: *abc_lm_v.simps*)
next
case (*Suc y*)
let *?ft* = *max* (*n* + *3*) (*max fft gft*)
let *?C* = [*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*)]
have *ind*: $y \leq x \Longrightarrow$
{λ*nl*. *nl* = *xs* @ (*x* − *y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *y*]) # *0* ↑ (*?ft* − *Suc* (*Suc n*)) @ *y* # *anything*}
?C {λ*nl*. *nl* = *xs* @ *x* # *rec_exec* (*Pr n f g*) (*xs* @ [*x*]) # *0* ↑ (*?ft* − *Suc n*) @ *anything*} **by** *fact*
have *less*: *Suc y* ≤ *x* **by** *fact*
have *stp1*:
∃ *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*)
?C stp = (*0*, *xs* @ (*x* − *y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *y*]) # *0* ↑ (*?ft* − *Suc* (*Suc n*)) @ *y* # *anything*)
using *assms less*
by(*rule_tac pr_loop*, *auto*)
then obtain *stp1* **where** *a*:
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*)
?C stp1 = (*0*, *xs* @ (*x* − *y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *y*]) # *0* ↑ (*?ft* − *Suc* (*Suc n*)) @ *y* # *anything*) **..**

moreover have
∃ *stp*. *abc_steps_l* (*0*, *xs* @ (*x* − *y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *y*]) # *0* ↑ (*?ft* − *Suc* (*Suc* *n*)) @ *y* # *anything*)
?C stp = (*length ?C*, *xs* @ *x* # *rec_exec* (*Pr n f g*) (*xs* @ [*x*]) # *0* ↑ (*?ft* − *Suc n*) @ *anything*)
using *ind less*
apply(*auto simp*: *abc_Hoare_halt_def*)
apply(*rename_tac na*,*case_tac abc_steps_l* (*0*, *xs* @ (*x* − *y*) # *rec_exec* (*Pr n f g*)
(*xs* @ [*x* − *y*]) # *0* ↑ (*?ft* − *Suc* (*Suc n*)) @ *y* # *anything*) *?C na*, *rule_tac x* = *na* **in** *exI*)
by *simp*
then obtain *stp2* **where** *b*:
abc_steps_l (*0*, *xs* @ (*x* − *y*) # *rec_exec* (*Pr n f g*) (*xs* @ [*x* − *y*]) # *0* ↑ (*?ft* − *Suc* (*Suc n*)) @ *y* # *anything*)
?C stp2 = (*length ?C*, *xs* @ *x* # *rec_exec* (*Pr n f g*) (*xs* @ [*x*]) # *0* ↑ (*?ft* − *Suc n*) @ *anything*)
..
from *a b* **show** *?case*
apply(*simp add*: *abc_Hoare_halt_def*)
apply(*rule_tac x* = *stp1* + *stp2* **in** *exI*, *simp add*: *abc_steps_add*)**.**
qed

lemma *compile_pr_correct′*:
assumes *termi_g*: ∀ *y*<*x*. *terminate g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])])
and *g_ind*:
∀ *y*<*x*. (∀ *anything*. {λ*nl*. *nl* = *xs* @ *y* # *rec_exec* (*Pr n f g*) (*xs* @ [*y*]) # *0* ↑ (*gft* − *gar*) @ *anything*} *gap*
{λ*nl*. *nl* = *xs* @ *y* # *rec_exec* (*Pr n f g*) (*xs* @ [*y*]) # *rec_exec g* (*xs* @ [*y*, *rec_exec* (*Pr n f g*) (*xs* @ [*y*])]) # *0* ↑ (*gft* − *Suc gar*) @ *anything*})
and *termi_f*: *terminate f xs*
and *f_ind*: ⋀ *anything*. {λ*nl*. *nl* = *xs* @ *0* ↑ (*fft* − *far*) @ *anything*} *fap* {λ*nl*. *nl* = *xs* @ *rec_exec f xs* # *0* ↑ (*fft* − *Suc far*) @ *anything*}
and *len*: *length xs* = *n*
and *compile1*: *rec_ci f* = (*fap*, *far*, *fft*)
and *compile2*: *rec_ci g* = (*gap*, *gar*, *gft*)
shows
{λ*nl*. *nl* = *xs* @ *x* # *0* ↑ (*max* (*n* + *3*) (*max fft gft*) − *n*) @ *anything*}
mv_box n (*max* (*n* + *3*) (*max fft gft*)) [+]
(*fap* [+] (*mv_box n* (*Suc n*) [+]
([*Dec* (*max* (*n* + *3*) (*max fft gft*)) (*length gap* + *7*)] [+] (*gap* [+] [*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)]) @
[*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length gap* + *4*)])))
{λ*nl*. *nl* = *xs* @ *x* # *rec_exec* (*Pr n f g*) (*xs* @ [*x*]) # *0* ↑ (*max* (*n* + *3*) (*max fft gft*) − *Suc n*) @ *anything*}
proof −
let *?ft* = *max* (*n*+*3*) (*max fft gft*)
let *?A* = *mv_box n ?ft*
let *?B* = *fap*
let *?C* = *mv_box n* (*Suc n*)
let *?D* = [*Dec ?ft* (*length gap* + *7*)]
let *?E* = *gap* [+] [*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)]
let *?F* = [*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length gap* + *4*)]
let *?P* = λ*nl*. *nl* = *xs* @ *x* # *0* ↑ (*?ft* − *n*) @ *anything*

```
 let ?S = λnl. nl = xs @ x # rec_exec (Pr n f g) (xs @ [x]) # 0 ↑ (?ft − Suc n) @ anything
 let ?Q1 = λnl. nl = xs @ 0 ↑ (?ft − n) @  x # anything
 show {?P} (?A [+] (?B [+] (?C [+] (?D [+] ?E @ ?F)))) {?S}
 proof(rule_tac abc_Hoare_plus_halt)
  show {?P} ?A {?Q1}
   using len by simp
 next
  let ?Q2 = λnl. nl = xs @ rec_exec f xs # 0 ↑ (?ft − Suc n) @  x # anything
  have a: ?ft ≥ fft
   by arith
  have b: far = n
   using compile1 termi_f len
   by(drule_tac param_pattern, auto)
  have c: fft > far
   using compile1
   by(simp add: footprint_ge)
  show {?Q1} (?B [+] (?C [+] (?D [+] ?E @ ?F))) {?S}
  proof(rule_tac abc_Hoare_plus_halt)
   have {λnl. nl = xs @ 0 ↑ (fft − far) @ 0↑(?ft − fft) @ x # anything} fap
        {λnl. nl = xs @ rec_exec f xs # 0 ↑ (fft − Suc far) @ 0↑(?ft − fft) @ x # anything}
    by(rule_tac f_ind)
   moreover have fft − far + ?ft − fft = ?ft − far
    using a b c by arith
   moreover have fft − Suc n + ?ft − fft = ?ft − Suc n
    using a b c by arith
   ultimately show {?Q1} ?B {?Q2}
    using b
    by(simp add: replicate_merge_anywhere)
  next
   let ?Q3 = λ nl. nl = xs @ 0 # rec_exec f xs # 0↑(?ft − Suc (Suc n)) @ x # anything
   show {?Q2} (?C [+] (?D [+] ?E @ ?F)) {?S}
   proof(rule_tac abc_Hoare_plus_halt)
    show {?Q2} (?C) {?Q3}
      using mv_box_correct[of n Suc n xs @ rec_exec f xs # 0 ↑ (max (n + 3) (max fft gft) − Suc n) @ x # anything]
      using len
      by(auto)
   next
    show {?Q3} (?D [+] ?E @ ?F) {?S}
     using pr_loop_correct[of x x xs n g  gap gar gft f fft anything] assms
     by(simp add: rec_exec_pr_0_simps)
   qed
  qed
 qed
qed

lemma compile_pr_correct:
 assumes g_ind: ∀ y<x. terminate g (xs @ [y, rec_exec (Pr n f g) (xs @ [y])]) ∧
 (∀ x xa xb. rec_ci g = (x, xa, xb) ⟶
 (∀ xc. {λnl. nl = xs @ y # rec_exec (Pr n f g) (xs @ [y]) # 0 ↑ (xb − xa) @ xc} x
```

$\{\lambda nl.\ nl = xs\ @\ y\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])\ \#\ rec\_exec\ g\ (xs\ @\ [y,\ rec\_exec\ (Pr\ n\ f\ g)$
$(xs\ @\ [y])])\ \#\ 0\uparrow(xb - Suc\ xa)\ @\ xc\}))$
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 $\{\lambda nl.\ nl = xs\ @\ x\ \#\ 0\uparrow(fp - arity)\ @\ anything\}\ ap\ \{\lambda nl.\ nl = xs\ @\ x\ \#\ rec\_exec\ (Pr$
$n\ f\ g)\ (xs\ @\ [x])\ \#\ 0\uparrow(fp - Suc\ arity)\ @\ anything\}$
proof($cases\ rec\_ci\ f$, $cases\ rec\_ci\ g$)
fix $fap\ far\ fft\ gap\ gar\ gft$
assume h: $rec\_ci\ f = (fap, far, fft)\ rec\_ci\ g = (gap, gar, gft)$
have g:
$\forall y{<}x.\ (terminate\ g\ (xs\ @\ [y,\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])]) \wedge$
$(\forall anything.\ \{\lambda nl.\ nl = xs\ @\ y\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])\ \#\ 0\uparrow(gft - gar)\ @\ anything\}$
gap
$\{\lambda nl.\ nl = xs\ @\ y\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])\ \#\ rec\_exec\ g\ (xs\ @\ [y,\ rec\_exec\ (Pr\ n\ f\ g)$
$(xs\ @\ [y])])\ \#\ 0\uparrow(gft - Suc\ gar)\ @\ anything\}))$
using $g\_ind\ h$
by($auto$)
hence $termi\_g$: $\forall\ y{<}x.\ terminate\ g\ (xs\ @\ [y,\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])])$
by $simp$
from g **have** g_newind:
$\forall\ y{<}x.\ (\forall anything.\ \{\lambda nl.\ nl = xs\ @\ y\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])\ \#\ 0\uparrow(gft - gar)\ @$
$anything\}\ gap$
$\{\lambda nl.\ nl = xs\ @\ y\ \#\ rec\_exec\ (Pr\ n\ f\ g)\ (xs\ @\ [y])\ \#\ rec\_exec\ g\ (xs\ @\ [y,\ rec\_exec\ (Pr\ n\ f\ g)$
$(xs\ @\ [y])])\ \#\ 0\uparrow(gft - Suc\ gar)\ @\ anything\})$
by $auto$
have f_newind: $\bigwedge\ anything.\ \{\lambda nl.\ nl = xs\ @\ 0\uparrow(fft - far)\ @\ anything\}\ fap\ \{\lambda nl.\ nl = xs\ @$
$rec\_exec\ f\ xs\ \#\ 0\uparrow(fft - Suc\ far)\ @\ anything\}$
using h
by($rule\_tac\ f\_ind$, $simp$)
show $?thesis$
using $termi\_f\ termi\_g\ h\ compile$
apply($simp\ add$: $rec\_ci.simps\ abc\_comp\_commute$, $auto$)
using $g\_newind\ f\_newind\ len$
by($rule\_tac\ compile\_pr\_correct'$, $simp\_all$)
qed

fun mn_ind_inv ::
$nat \times nat\ list \Rightarrow nat \Rightarrow nat \Rightarrow nat \Rightarrow nat\ list \Rightarrow nat\ list \Rightarrow bool$
where
$mn\_ind\_inv\ (as, lm')\ ss\ x\ rsx\ suf\_lm\ lm =$
$(if\ as = ss\ then\ lm' = lm\ @\ x\ \#\ rsx\ \#\ suf\_lm$
$else\ if\ as = ss + 1\ then$
$\exists y.\ (lm' = lm\ @\ x\ \#\ y\ \#\ suf\_lm) \wedge y \le rsx$
$else\ if\ as = ss + 2\ then$
$\exists y.\ (lm' = lm\ @\ x\ \#\ y\ \#\ suf\_lm) \wedge y \le rsx$

$\quad$ *else if* $as = ss + 3$ *then* $lm' = lm$ @ x # 0 # *suf_lm*
$\quad$ *else if* $as = ss + 4$ *then* $lm' = lm$ @ *Suc* x # 0 # *suf_lm*
$\quad$ *else if* $as = 0$ *then* $lm' = lm$ @ *Suc* x # 0 # *suf_lm*
$\quad$ *else False*
)

fun *mn_stage1* :: *nat* × *nat list* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
mn_stage1 (*as*, *lm*) *ss n* =
$\quad$ (*if as* = *0 then 0*
$\quad$ *else if as* = *ss* + *4 then 1*
$\quad$ *else if as* = *ss* + *3 then 2*
$\quad$ *else if as* = *ss* + *2* ∨ *as* = *ss* + *1 then 3*
$\quad$ *else if as* = *ss then 4*
$\quad$ *else 0*
)

fun *mn_stage2* :: *nat* × *nat list* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
mn_stage2 (*as*, *lm*) *ss n* =
$\quad$ (*if as* = *ss* + *1* ∨ *as* = *ss* + *2 then* (*lm* ! (*Suc n*))
$\quad$ *else 0*)

fun *mn_stage3* :: *nat* × *nat list* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
mn_stage3 (*as*, *lm*) *ss n* = (*if as* = *ss* + *2 then 1 else 0*)

fun *mn_measure* :: ((*nat* × *nat list*) × *nat* × *nat*) ⇒
$\quad$ (*nat* × *nat* × *nat*)
where
mn_measure ((*as*, *lm*), *ss*, *n*) =
(*mn_stage1* (*as*, *lm*) *ss n*, *mn_stage2* (*as*, *lm*) *ss n*,
$\quad$ *mn_stage3* (*as*, *lm*) *ss n*)

definition *mn_LE* :: (((*nat* × *nat list*) × *nat* × *nat*) ×
$\quad$ ((*nat* × *nat list*) × *nat* × *nat*)) *set*
where *mn_LE* $\stackrel{def}{=}$ (*inv_image lex_triple mn_measure*)

lemma *wf_mn_le*[*intro*]: *wf mn_LE*
by(*auto intro*:*wf_inv_image wf_lex_triple simp*: *mn_LE_def*)

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

lemma *put_in_tape_small_enough0*[*simp*]:
0 < *rsx* ⟹
∃ *y*. (*xs* @ *x* # *rsx* # *anything*)[*Suc* (*length xs*) := *rsx* − *Suc 0*] = *xs* @ *x* # *y* # *anything* ∧ *y* ≤ *rsx*
apply(*rule_tac x* = *rsx* − *1* **in** *exI*)

```
 apply(simp add: list_update_append list_update.simps)
 done

lemma put_in_tape_small_enough1[simp]:
 ⟦y ≤ rsx; 0 < y⟧
          ⟹ ∃ ya. (xs @ x # y # anything)[Suc (length xs) := y − Suc 0] = xs @ x # ya #
anything ∧ ya ≤ rsx
 apply(rule_tac x = y − 1 in exI)
 apply(simp add: list_update_append list_update.simps)
 done

lemma abc_comp_null[simp]: (A [+] B = []) = (A = [] ∧ B = [])
 by(auto simp: abc_comp.simps abc_shift.simps)

lemma rec_ci_not_null[simp]: (rec_ci f ≠ ([], a, b))
proof(cases f)
 case (Cn x41 x42 x43)
 then show ?thesis
   by(cases rec_ci x42, auto simp: mv_box.simps rec_ci.simps rec_ci_id.simps)
next
 case (Pr x51 x52 x53)
 then show ?thesis
   apply(cases rec_ci x52, cases rec_ci x53)
   by (auto simp: mv_box.simps rec_ci.simps rec_ci_id.simps)
next
 case (Mn x61 x62)
 then show ?thesis
   by(cases rec_ci x62) (auto simp: rec_ci.simps rec_ci_id.simps)
qed (auto simp: rec_ci_z_def rec_ci_s_def rec_ci.simps addition.simps rec_ci_id.simps)


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 = (λ stp. (abc_steps_l (length fap, xs @ x # rsx # anything) (fap @ B) stp, (length
fap), arity))
   and P: P =(λ ((as, lm), ss, arity). as = 0)
   and Q: Q = (λ ((as, lm), ss, arity). mn_ind_inv (as, lm) (length fap) x rsx anything xs)
 shows ∃ stp. P (f stp) ∧ Q (f stp)
proof(rule_tac halt_lemma2)
 show wf mn_LE
   using wf_mn_le by simp
next
 show Q (f 0)
   by(auto simp: Q f abc_steps_l.simps mn_ind_inv.simps)
next
 have fap ≠ []
```

```
    using compile by auto
  thus ¬ P (f 0)
    by(auto simp: f P abc_steps_l.simps)
next
  have fap ≠ []
    using compile by auto
  then have ⟦¬ P (f stp); Q (f stp)⟧ ⟹ Q (f (Suc stp)) ∧ (f (Suc stp), f stp) ∈ mn_LE for stp
    using ge len
    apply(cases (abc_steps_l (length fap, xs @ x # rsx # anything) (fap @ B) stp))
    apply(simp add: abc_step_red2 B f P Q)
      apply(auto split:if_splits simp add:abc_steps_l.simps  mn_ind_inv.simps abc_steps_zero B
abc_fetch.simps nth_append)
    by(auto simp: mn_LE_def lex_triple_def lex_pair_def
        abc_step_l.simps abc_steps_l.simps mn_ind_inv.simps
        abc_lm_v.simps abc_lm_s.simps nth_append abc_fetch.simps
        split: if_splits)
  thus ∀ stp. ¬ P (f stp) ∧ Q (f stp) ⟶ Q (f (Suc stp)) ∧ (f (Suc stp), f stp) ∈ mn_LE
    by(auto)
qed

lemma abc_Hoare_haltE:
  {λ nl. nl = lm1} p {λ nl. nl = lm2}
    ⟹ ∃ stp. abc_steps_l (0, lm1) p stp = (length p, lm2)
  by(auto simp:abc_Hoare_halt_def elim!: abc_holds_for.elims)

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:  (∀ xc. ({λnl. nl = xs @ x # 0 ↑ (fft − far) @ xc} fap
    {λnl. nl = xs @ x # rec_exec f (xs @ [x]) # 0 ↑ (fft − Suc far) @ xc}))
    and exec_less: rec_exec f (xs @ [x]) > 0
    and compile: rec_ci f = (fap, far, fft)
  shows ∃ stp > 0. abc_steps_l (0, xs @ x # 0 ↑ (ft − Suc arity) @ anything) (fap @ B) stp =
    (0, xs @ Suc x # 0 ↑ (ft − Suc arity) @ anything)
proof −
  have ∃ stp. abc_steps_l (0, xs @ x # 0 ↑ (ft − Suc arity) @ anything) (fap @ B) stp =
    (length fap, xs @ x # rec_exec f (xs @ [x]) # 0 ↑ (ft − Suc (Suc arity)) @ anything)
  proof −
    have ∃ stp. abc_steps_l (0, xs @ x # 0 ↑ (ft − Suc arity) @ anything) fap stp =
      (length fap, xs @ x # rec_exec f (xs @ [x]) # 0 ↑ (ft − Suc (Suc arity)) @ anything)
    proof −
      have {λnl. nl = xs @ x # 0 ↑ (fft − far) @ 0↑(ft − fft) @ anything} fap
          {λnl. nl = xs @ x # rec_exec f (xs @ [x]) # 0 ↑ (fft − Suc far) @ 0↑(ft − fft) @ anything}
        using ind by simp
      moreover have fft > far
        using compile
        by(erule_tac footprint_ge)
```

```
    ultimately show ?thesis
      using ft far
      apply(drule_tac abc_Hoare_haltE)
      by(simp add: replicate_merge_anywhere max_absorb2)
  qed
  then obtain stp where abc_steps_l (0, xs @ x # 0 ↑ (ft − Suc arity) @ anything) fap stp =
    (length fap, xs @ x # rec_exec f (xs @ [x]) # 0 ↑ (ft − Suc (Suc arity)) @ anything) ..
  thus ?thesis
    by(erule_tac abc_append_frist_steps_halt_eq)
 qed
 moreover have
  ∃ stp > 0. abc_steps_l (length fap, xs @ x # rec_exec f (xs @ [x]) # 0 ↑ (ft − Suc (Suc arity))
@ anything) (fap @ B) stp =
  (0, xs @ Suc x # 0 # 0 ↑ (ft − Suc (Suc arity)) @ anything)
  using mn_correct[of f fap far fft rec_exec f (xs @ [x]) xs arity B
    (λstp. (abc_steps_l (length fap, xs @ x # rec_exec f (xs @ [x]) # 0 ↑ (ft − Suc (Suc arity))
@ anything) (fap @ B) stp, length fap, arity))
    x 0 ↑ (ft − Suc (Suc arity)) @ anything (λ((as, lm), ss, arity). as = 0)
     (λ((as, lm), ss, aritya). mn_ind_inv (as, lm) (length fap) x (rec_exec f (xs @ [x])) (0 ↑ (ft
− Suc (Suc arity)) @ anything) xs) ]
   B compile  exec_less len
  apply(subgoal_tac fap ≠ [], auto)
  apply(rename_tac stp y)
  apply(rule_tac x = stp in exI, auto simp: mn_ind_inv.simps)
  by(case_tac stp, simp_all add: abc_steps_l.simps)
 moreover have fft > far
  using compile
  by(erule_tac footprint_ge)
 ultimately show ?thesis
  using ft far
  apply(auto) apply(rename_tac stp1 stp2)
  by(rule_tac x = stp1 + stp2 in exI,
    simp add: abc_steps_add replicate_Suc[THEN sym] diff_Suc_Suc del: replicate_Suc)
qed

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: ∀ i≤x. (∀ xc. ({λnl. nl = xs @ i # 0 ↑ (fft − far) @ xc} fap
  {λnl. nl = xs @ i # rec_exec f (xs @ [i]) # 0 ↑ (fft − Suc far) @ xc}))
  and exec_ge: ∀ i≤x. rec_exec f (xs @ [i]) > 0
  and compile: rec_ci f = (fap, far, fft)
  and far: far = Suc arity
 shows ∃ stp > x. abc_steps_l (0, xs @ 0 # 0 ↑ (ft − Suc arity) @ anything) (fap @ B) stp =
       (0, xs @ Suc x # 0 ↑ (ft − Suc arity) @ anything)
 using ind_all exec_ge
proof(induct x)
 case 0
```

thus *?case*
using *assms*
by(*rule_tac mn_loop*, *simp_all*)
next
case (*Suc x*)
have *ind'*: $[\![\forall i{\le}x.\ \forall xc.\ \{\lambda nl.\ nl = xs\ @\ i\ \#\ 0 \uparrow (fft - far)\ @\ xc\}\ fap\ \{\lambda nl.\ nl = xs\ @\ i\ \#\ rec\_exec\ f\ (xs\ @\ [i])\ \#\ 0 \uparrow (fft - Suc\ far)\ @\ xc\};$
$\forall i{\le}x.\ 0 < rec\_exec\ f\ (xs\ @\ [i])]\!] \Longrightarrow$
$\exists stp > x.\ abc\_steps\_l\ (0, xs\ @\ 0\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)\ (fap\ @\ B)\ stp = (0, xs\ @\ Suc\ x\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)$ **by** *fact*
have *exec_ge*: $\forall i{\le}Suc\ x.\ 0 < rec\_exec\ f\ (xs\ @\ [i])$ **by** *fact*
have *ind_all*: $\forall i{\le}Suc\ x.\ \forall xc.\ \{\lambda nl.\ nl = xs\ @\ i\ \#\ 0 \uparrow (fft - far)\ @\ xc\}\ fap$
$\{\lambda nl.\ nl = xs\ @\ i\ \#\ rec\_exec\ f\ (xs\ @\ [i])\ \#\ 0 \uparrow (fft - Suc\ far)\ @\ xc\}$ **by** *fact*
have *ind*: $\exists stp > x.\ abc\_steps\_l\ (0, xs\ @\ 0\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)\ (fap\ @\ B)\ stp =$
$(0, xs\ @\ Suc\ x\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)$ **using** *ind' exec_ge ind_all* **by** *simp*
have *stp*: $\exists\ stp > 0.\ abc\_steps\_l\ (0, xs\ @\ Suc\ x\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)\ (fap\ @\ B)\ stp =$
$(0, xs\ @\ Suc\ (Suc\ x)\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)$
using *ind_all exec_ge B ft len far compile*
by(*rule_tac mn_loop*, *simp_all*)
from *ind stp* **show** *?case*
apply(*auto simp add*: *abc_steps_add*)
apply(*rename_tac stp1 stp2*)
by(*rule_tac* $x = stp1 + stp2$ **in** *exI*, *simp add*: *abc_steps_add*)
qed

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 −
have $\exists stp{>}x.\ abc\_steps\_l\ (0, xs\ @\ 0\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)\ (fap\ @\ B)\ stp = (0, xs\ @\ Suc\ x\ \#\ 0 \uparrow (ft - Suc\ arity)\ @\ anything)$
using *assms*
by(*rule_tac mn_loop_correct'*, *simp_all*)
thus *?thesis*
by(*auto*)
qed

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: ⋀anything. {λnl. nl = xs @ r # 0 ↑ (fft − far) @ anything} fap
     {λnl. nl = xs @ r # 0 # 0 ↑ (fft − Suc far) @ anything}
  and ind_all: ∀ i < r. (∀ xc. ({λnl. nl = xs @ i # 0 ↑ (fft − far) @ xc} fap
  {λnl. nl = xs @ i # rec_exec f (xs @ [i]) # 0 ↑ (fft − Suc far) @ xc}))
  and exec_less: ∀ 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 {λnl. nl = xs @ 0 ↑ (max (Suc arity) fft − arity) @ anything}
  fap @ B
  {λnl. nl = xs @ rec_exec (Mn arity f) xs # 0 ↑ (max (Suc arity) fft − Suc arity) @ anything}
proof −
 have a: far = Suc arity
  using len compile termi_f
  by(drule_tac param_pattern, auto)
 have b: fft > far
  using compile
  by(erule_tac footprint_ge)
 have ∃ stp. abc_steps_l (0, xs @ 0 # 0 ↑ (ft − Suc arity) @ anything) (fap @ B) stp =
  (0, xs @ r # 0 ↑ (ft − Suc arity) @ anything)
  using assms a
  apply(cases r, rule_tac x = 0 in exI, simp add: abc_steps_l.simps, simp)
  by(rule_tac mn_loop_correct, auto)
 moreover have
  ∃ stp. abc_steps_l (0, xs @ r # 0 ↑ (ft − Suc arity) @ anything) (fap @ B) stp =
  (length fap, xs @ r # rec_exec f (xs @ [r]) # 0 ↑ (ft − Suc (Suc arity)) @ anything)
 proof −
  have ∃ stp. abc_steps_l (0, xs @ r # 0 ↑ (ft − Suc arity) @ anything) fap stp =
   (length fap, xs @ r # rec_exec f (xs @ [r]) # 0 ↑ (ft − Suc (Suc arity)) @ anything)
  proof −
   have {λnl. nl = xs @ r # 0 ↑ (fft − far) @ 0↑(ft − fft) @ anything} fap
      {λnl. nl = xs @ r # rec_exec f (xs @ [r]) # 0 ↑ (fft − Suc far) @ 0↑(ft − fft) @ anything}
    using f_ind exec by simp
   thus ?thesis
    using ft a b
    apply(drule_tac abc_Hoare_haltE)
    by(simp add: replicate_merge_anywhere max_absorb2)
  qed
  then obtain stp where abc_steps_l (0, xs @ r # 0 ↑ (ft − Suc arity) @ anything) fap stp =
   (length fap, xs @ r # rec_exec f (xs @ [r]) # 0 ↑ (ft − Suc (Suc arity)) @ anything) ..
  thus ?thesis
   by(erule_tac abc_append_frist_steps_halt_eq)
 qed
 moreover have
  ∃ stp. abc_steps_l (length fap, xs @ r # rec_exec f (xs @ [r]) # 0 ↑ (ft − Suc (Suc arity)) @
anything) (fap @ B) stp =
      (length fap + 5, xs @ r # rec_exec f (xs @ [r]) # 0 ↑ (ft − Suc (Suc arity)) @ anything)
  using ft a b len B exec
```

```
  apply(rule_tac x = 1 in exI, auto)
  by(auto simp: abc_steps_l.simps B abc_step_l.simps
     abc_fetch.simps nth_append max_absorb2 abc_lm_v.simps abc_lm_s.simps)
 moreover have rec_exec (Mn (length xs) f) xs = r
  using exec exec_less
  apply(auto simp: rec_exec.simps Least_def)
  thm the_equality
  apply(rule_tac the_equality, auto)
   apply(metis exec_less less_not_refl3 linorder_not_less)
  by (metis le_neq_implies_less less_not_refl3)
 ultimately show ?thesis
  using ft a b len B exec
  apply(auto simp: abc_Hoare_halt_def)
  apply(rename_tac stp1 stp2 stp3)
  apply(rule_tac x = stp1 + stp2 + stp3  in exI)
  by(simp add: abc_steps_add replicate_Suc_iff_anywhere max_absorb2 Suc_diff_Suc del: replicate_Suc)
qed

lemma compile_mn_correct:
 assumes len: length xs = n
  and termi_f: terminate f (xs @ [r])
  and f_ind: ⋀ap arity fp anything. rec_ci f = (ap, arity, fp) ⟹
 {λnl. nl = xs @ r # 0 ↑ (fp − arity) @ anything} ap {λnl. nl = xs @ r # 0 # 0 ↑ (fp − Suc
arity) @ anything}
  and exec: rec_exec f (xs @ [r]) = 0
  and ind_all:
  ∀ i<r. terminate f (xs @ [i]) ∧
 (∀ x xa xb. rec_ci f = (x, xa, xb) ⟶
 (∀ xc. {λnl. nl = xs @ i # 0 ↑ (xb − xa) @ xc} x {λnl. nl = xs @ i # rec_exec f (xs @ [i]) #
0 ↑ (xb − Suc xa) @ xc})) ∧
 0 < rec_exec f (xs @ [i])
  and compile: rec_ci (Mn n f) = (ap, arity, fp)
 shows {λnl. nl = xs @ 0 ↑ (fp − arity) @ anything} ap
 {λnl. nl = xs @ rec_exec (Mn n f) xs # 0 ↑ (fp − Suc arity) @ anything}
proof(cases rec_ci f)
 fix fap far fft
 assume h: rec_ci f = (fap, far, fft)
 hence f_newind:
  ⋀anything. {λnl. nl = xs @ r # 0 ↑ (fft − far) @ anything} fap
     {λnl. nl = xs @ r # 0 # 0 ↑ (fft − Suc far) @ anything}
  by(rule_tac f_ind, simp)
 have newind_all:
  ∀ i < r. (∀ xc. ({λnl. nl = xs @ i # 0 ↑ (fft − far) @ xc} fap
  {λnl. nl = xs @ i # rec_exec f (xs @ [i]) # 0 ↑ (fft − Suc far) @ xc}))
  using ind_all h
  by(auto)
 have all_less: ∀  i<r. rec_exec f (xs @ [i]) > 0
  using ind_all by auto
 show ?thesis
  using h compile f_newind newind_all all_less len termi_f exec
```

apply(*auto simp*: *rec_ci.simps*)
by(*rule_tac compile_mn_correct'*, *auto*)
qed

lemma *recursive_compile_correct*:
⟦*terminate recf args*; *rec_ci recf* = (*ap*, *arity*, *fp*)⟧
⟹ {λ *nl*. *nl* = *args* @ *0*↑(*fp* − *arity*) @ *anything*} *ap*
{λ *nl*. *nl* = *args*@ *rec_exec recf args* # *0*↑(*fp* − *Suc arity*) @ *anything*}
apply(*induct arbitrary*: *ap arity fp anything rule*: *terminate.induct*)
apply(*simp_all add*: *compile_s_correct compile_z_correct compile_id_correct*
compile_cn_correct compile_pr_correct compile_mn_correct)
done

definition *dummy_abc* :: *nat* ⇒ *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*
by(*auto simp*: *abc_list_crsp_def*)

lemma *abc_list_crsp_lm_v*:
abc_list_crsp lma lmb ⟹ *abc_lm_v lma n* = *abc_lm_v lmb n*
by(*auto simp*: *abc_list_crsp_def abc_lm_v.simps*
nth_append)

lemma *abc_list_crsp_elim*:
⟦*abc_list_crsp lma lmb*; ∃ *n*. *lma* = *lmb* @ *0*↑*n* ∨ *lmb* = *lma* @ *0*↑*n* ⟹ *P* ⟧ ⟹ *P*
by(*auto simp*: *abc_list_crsp_def*)

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*])
by(*auto simp*: *abc_list_crsp_def list_update_append*)

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*])
apply(*auto simp*: *abc_list_crsp_def list_update_append*)
apply(*rename_tac N*)
apply(*rule_tac x* = *N* + *length lmb* − *Suc m* **in** *exI*)
apply(*rule_tac disjI1*)
apply(*simp add*: *upd_conv_take_nth_drop min_absorb1*)
done

```
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])
  apply(auto simp: abc_list_crsp_def list_update_append)
  apply(rename_tac N)
  apply(rule_tac x = N + length lma − Suc m in exI)
  apply(rule_tac disjI2)
  apply(simp add: upd_conv_take_nth_drop min_absorb1)
  done

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])
  by(auto simp: abc_list_crsp_def list_update_append replicate_merge_anywhere)

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)
  by(auto simp: abc_lm_s.simps)

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'
  apply(cases i, auto simp: abc_step_l.simps
    abc_list_crsp_lm_s abc_list_crsp_lm_v
    split: abc_inst.splits if_splits)
  done

lemma abc_list_crsp_steps:
  ⟦abc_steps_l (0, lm @ 0↑m) aprog stp = (a, lm'); aprog ≠ []⟧
    ⟹ ∃ lma. abc_steps_l (0, lm) aprog stp = (a, lma) ∧
                              abc_list_crsp lm' lma
proof(induct stp arbitrary: a lm')
  case (Suc stp)
  then show ?case apply(cases abc_steps_l (0, lm @ 0↑m) aprog stp, simp add: abc_step_red)
  proof −
   fix stp a lm' aa b
   assume ind:
    ⋀a lm'. aa = a ∧ b = lm' ⟹
    ∃ lma. abc_steps_l (0, lm) aprog stp = (a, lma) ∧
                              abc_list_crsp lm' lma
    and h: abc_step_l (aa, b) (abc_fetch aa aprog) = (a, lm')
    abc_steps_l (0, lm @ 0↑m) aprog stp = (aa, b)
    aprog ≠ []
   have ∃ lma. abc_steps_l (0, lm) aprog stp = (aa, lma) ∧
        abc_list_crsp b lma
    apply(rule_tac ind, simp)
    done
   from this obtain lma where g2:
```

```
    abc_steps_l (0, lm) aprog stp = (aa, lma) ∧
   abc_list_crsp b lma  ..
  hence g3: abc_steps_l (0, lm) aprog (Suc stp)
       = abc_step_l (aa, lma) (abc_fetch aa aprog)
    apply(rule_tac abc_step_red, simp)
    done
  show ∃lma. abc_steps_l (0, lm) aprog (Suc stp) = (a, lma) ∧ abc_list_crsp lm′ lma
    using g2 g3 h
    apply(auto)
    apply(cases abc_step_l (aa, b) (abc_fetch aa aprog),
       cases abc_step_l (aa, lma) (abc_fetch aa aprog), simp)
    apply(rule_tac abc_list_crsp_step, auto)
    done
 qed
qed (force simp add: abc_steps_l.simps)

lemma list_crsp_simp2: abc_list_crsp (lm1 @ 0↑n) lm2 ⟹ abc_list_crsp lm1 lm2
proof(induct n)
 case 0
 thus ?case
  by(auto simp: abc_list_crsp_def)
next
 case (Suc n)
 have ind: abc_list_crsp (lm1 @ 0 ↑ n) lm2 ⟹ abc_list_crsp lm1 lm2 by fact
 have h: abc_list_crsp (lm1 @ 0 ↑ Suc n) lm2 by fact
 then have abc_list_crsp (lm1 @ 0 ↑ n) lm2
  apply(auto simp only: exp_suc abc_list_crsp_def del: replicate_Suc)
   apply (metis Suc_pred append_eq_append_conv
     append_eq_append_conv2 butlast_append butlast_snoc length_replicate list.distinct(1)
     neq0_conv replicate_Suc replicate_Suc_iff_anywhere replicate_app_Cons_same
     replicate_empty self_append_conv self_append_conv2)
  apply (auto,metis replicate_Suc)
  .
 thus ?case
  using ind
  by auto
qed

lemma recursive_compile_correct_norm′:
 ⟦rec_ci f = (ap, arity, ft);
  terminate f args⟧
 ⟹ ∃ stp rl. (abc_steps_l (0, args) ap stp) = (length ap, rl) ∧ abc_list_crsp (args @ [rec_exec
f args]) rl
 using recursive_compile_correct[of f args ap arity ft []]
 apply(auto simp: abc_Hoare_halt_def)
 apply(rename_tac n)
 apply(rule_tac x = n in exI)
 apply(case_tac abc_steps_l (0, args @ 0 ↑ (ft − arity)) ap n, auto)
 apply(drule_tac abc_list_crsp_steps, auto)
 apply(rule_tac list_crsp_simp2, auto)
```

```
  done

lemma find_exponent_rec_exec[simp]:
  assumes a: args @ [rec_exec f args] = lm @ 0 ↑ n
    and b: length args < length lm
  shows ∃m. lm = args @ rec_exec f args # 0 ↑ m
  using assms
  apply(cases n, simp)
   apply(rule_tac x = 0 in exI, simp)
  apply(drule_tac length_equal, simp)
  done

lemma find_exponent_complex[simp]:
  ⟦args @ [rec_exec f args] = lm @ 0 ↑ n; ¬ length args < length lm⟧
  ⟹ ∃m. (lm @ 0 ↑ (length args − length lm) @ [Suc 0])[length args := 0] =
  args @ rec_exec f args # 0 ↑ m
  apply(cases n, simp_all add: exp_suc list_update_append list_update.simps del: replicate_Suc)
   apply(subgoal_tac length args = Suc (length lm), simp)
    apply(rule_tac x = Suc (Suc 0) in exI, simp)
   apply(drule_tac length_equal, simp, auto)
  done

lemma compile_append_dummy_correct:
  assumes compile: rec_ci f = (ap, ary, fp)
    and termi: terminate f args
  shows {λ nl. nl = args} (ap [+] dummy_abc (length args)) {λ nl. (∃ m. nl = args @ rec_exec
f args # 0↑m)}
proof(rule_tac abc_Hoare_plus_halt)
  show {λnl. nl = args} ap {λ nl. abc_list_crsp (args @ [rec_exec f args]) nl}
    using compile termi recursive_compile_correct_norm'[of f ap ary fp args]
    apply(auto simp: abc_Hoare_halt_def)
    by (metis abc_final.simps abc_holds_for.simps)
next
  show {abc_list_crsp (args @ [rec_exec f args])} dummy_abc (length args)
    {λnl. ∃m. nl = args @ rec_exec f args # 0 ↑ m}
    apply(auto simp: dummy_abc_def abc_Hoare_halt_def)
    apply(rule_tac x = 3 in exI)
   by(force simp: abc_steps_l.simps abc_list_crsp_def abc_step_l.simps numeral_3_eq_3 abc_fetch.simps
      abc_lm_v.simps nth_append abc_lm_s.simps)
qed

lemma cn_merge_gs_split:
  ⟦i < length gs; rec_ci (gs!i) = (ga, gb, gc)⟧ ⟹
  cn_merge_gs (map rec_ci gs) p = cn_merge_gs (map rec_ci (take i gs)) p [+] (ga [+]
     mv_box gb (p + i)) [+] cn_merge_gs (map rec_ci (drop (Suc i) gs)) (p + Suc i)
proof(induct i arbitrary: gs p)
  case 0
  then show ?case by(cases gs; simp)
next
case (Suc i)
```

```
  then show ?case
    by(cases gs, simp, cases rec_ci (hd gs),
       simp add: abc_comp_commute[THEN sym])
qed

lemma cn_unhalt_case:
  assumes compile1: rec_ci (Cn n f gs) = (ap, ar, ft) ∧ length args = ar
    and g: i < length gs
    and compile2: rec_ci (gs!i) = (gap, gar, gft) ∧ gar = length args
    and g_unhalt: ⋀ anything. {λ nl. nl = args @ 0↑(gft − gar) @ anything} gap ↑
    and g_ind: ⋀ apj arj ftj j anything. ⟦j < i; rec_ci (gs!j) = (apj, arj, ftj)⟧
  ⟹ {λ nl. nl = args @ 0↑(ftj − arj) @ anything} apj {λ nl. nl = args @ rec_exec (gs!j) args
# 0↑(ftj − Suc arj) @ anything}
    and all_termi: ∀ j<i. terminate (gs!j) args
  shows {λ nl. nl = args @ 0↑(ft − ar) @ anything} ap ↑
  using compile1
  apply(cases rec_ci f, auto simp: rec_ci.simps abc_comp_commute)
proof(rule_tac abc_Hoare_plus_unhalt1)
  fix fap far fft
  let ?ft = max (Suc (length args)) (Max (insert fft ((λ(aprog, p, n). n) ‘ rec_ci ‘ set gs)))
  let ?Q = λnl. nl = args @ 0↑ (?ft − length args) @ map (λi. rec_exec i args) (take i gs) @
    0↑(length gs − i) @ 0↑ Suc (length args) @ anything
  have cn_merge_gs (map rec_ci gs) ?ft =
    cn_merge_gs (map rec_ci (take i gs)) ?ft [+] (gap [+]
    mv_box gar (?ft + i)) [+]  cn_merge_gs (map rec_ci (drop (Suc i) gs)) (?ft + Suc i)
    using g compile2 cn_merge_gs_split by simp
  thus {λnl. nl = args @ 0 # 0 ↑ (?ft + length gs) @ anything} (cn_merge_gs (map rec_ci gs)
?ft) ↑
  proof(simp, rule_tac abc_Hoare_plus_unhalt1, rule_tac abc_Hoare_plus_unhalt2,
      rule_tac abc_Hoare_plus_unhalt1)
     let ?Q_tmp = λnl. nl = args @ 0↑ (gft − gar) @ 0↑(?ft − (length args) − (gft −gar)) @
map (λi. rec_exec i args) (take i gs) @
      0↑(length gs − i) @ 0↑ Suc (length args) @ anything
    have a: {?Q_tmp} gap ↑
      using g_unhalt[of 0 ↑ (?ft − (length args) − (gft − gar)) @
          map (λi. rec_exec i args) (take i gs) @ 0 ↑ (length gs − i) @ 0 ↑ Suc (length args) @
anything]
      by simp
    moreover have ?ft ≥ gft
      using g compile2
      apply(rule_tac max.coboundedI2, rule_tac Max_ge, simp, rule_tac insertI2)
      apply(rule_tac  x = rec_ci (gs ! i) in image_eqI, simp)
      by(rule_tac x = gs!i  in image_eqI, simp, simp)
    then have b:?Q_tmp = ?Q
      using compile2
      apply(rule_tac arg_cong)
      by(simp add: replicate_merge_anywhere)
    thus {?Q} gap ↑
      using a by simp
  next
```

```
  show {λnl. nl = args @ 0 # 0 ↑ (?ft + length gs) @ anything}
    cn_merge_gs (map rec_ci (take i gs)) ?ft
     {λnl. nl = args @ 0 ↑ (?ft − length args) @
      map (λi. rec_exec i args) (take i gs) @ 0 ↑ (length gs − i) @ 0 ↑ Suc (length args) @
anything}
    using all_termi
    by(rule_tac compile_cn_gs_correct′, auto simp: set_conv_nth intro:g_ind)
 qed
qed
```

```
lemma mn_unhalt_case′:
 assumes compile: rec_ci f = (a, b, c)
  and all_termi: ∀ i. terminate f (args @ [i]) ∧ 0 < rec_exec f (args @ [i])
  and B: B = [Dec (Suc (length args)) (length a + 5), Dec (Suc (length args)) (length a + 3),
 Goto (Suc (length a)), Inc (length args), Goto 0]
 shows {λnl. nl = args @ 0 ↑ (max (Suc (length args)) c − length args) @ anything}
 a @ B ↑
proof(rule_tac abc_Hoare_unhaltI, auto)
 fix n
 have a:  b = Suc (length args)
  using all_termi compile
  apply(erule_tac x = 0 in allE)
  by(auto, drule_tac param_pattern,auto)
 moreover have b: c > b
  using compile by(elim footprint_ge)
 ultimately have c: max (Suc (length args)) c = c by arith
 have ∃ stp > n. abc_steps_l (0, args @ 0 # 0↑(c − Suc (length args)) @ anything) (a @ B) stp
      = (0, args @ Suc n # 0↑(c − Suc (length args)) @ anything)
  using assms a b c
 proof(rule_tac mn_loop_correct′, auto)
  fix i xc
  show {λnl. nl = args @ i # 0 ↑ (c − Suc (length args)) @ xc} a
    {λnl. nl = args @ i # rec_exec f (args @ [i]) # 0 ↑ (c − Suc (Suc (length args))) @ xc}
    using all_termi recursive_compile_correct[of f args @ [i] a b c xc] compile a
    by(simp)
 qed
 then obtain stp where d: stp > n ∧ abc_steps_l (0, args @ 0 # 0↑(c − Suc (length args)) @
anything) (a @ B) stp
      = (0, args @ Suc n # 0↑(c − Suc (length args)) @ anything) ..
 then obtain d where e: stp = n + Suc d
  by (metis add_Suc_right less_iff_Suc_add)
 obtain s nl where f: abc_steps_l (0, args @ 0 # 0↑(c − Suc (length args)) @ anything) (a @
B) n = (s, nl)
  by (metis prod.exhaust)
 have g: s < length (a @ B)
  using d e f
  apply(rule_tac classical, simp only: abc_steps_add)
  by(simp add: halt_steps2 leI)
```

```
  from f g show abc_notfinal (abc_steps_l (0, args @ 0 ↑
    (max (Suc (length args)) c − length args) @ anything) (a @ B) n) (a @ B)
    using c b a
    by(simp add: replicate_Suc_iff_anywhere Suc_diff_Suc del: replicate_Suc)
qed

lemma mn_unhalt_case:
  assumes compile: rec_ci (Mn n f) = (ap, ar, ft) ∧ length args = ar
    and all_term: ∀ i. terminate f (args @ [i]) ∧ rec_exec f (args @ [i]) > 0
  shows {λ nl. nl = args @ 0↑(ft − ar) @ anything} ap ↑
  using assms
  apply(cases rec_ci f, auto simp: rec_ci.simps abc_comp_commute)
  by(rule_tac mn_unhalt_case′, simp_all)

fun tm_of_rec :: recf ⇒ instr list
  where tm_of_rec recf = (let (ap, k, fp) = rec_ci recf in
                      let tp = tm_of (ap [+] dummy_abc k) in
                        tp @ (shift (mopup k) (length tp div 2)))

lemma recursive_compile_to_tm_correct1:
  assumes  compile: rec_ci recf = (ap, ary, fp)
    and termi:  terminate recf args
    and tp: tp = tm_of (ap [+] dummy_abc (length args))
  shows ∃ stp m l. steps0 (Suc 0, Bk # Bk # ires, <args> @ Bk↑rn)
  (tp @ shift (mopup (length args)) (length tp div 2)) stp = (0, Bk↑m @ Bk # Bk # ires, Oc↑Suc
(rec_exec recf args) @ Bk↑l)
proof −
  have {λnl. nl = args} ap [+] dummy_abc (length args) {λnl. ∃m. nl = args @ rec_exec recf
args # 0 ↑ m}
    using compile termi compile
    by(rule_tac compile_append_dummy_correct, auto)
  then obtain stp m where h: abc_steps_l (0, args) (ap [+] dummy_abc (length args)) stp =
    (length (ap [+] dummy_abc (length args)), args @ rec_exec recf args # 0↑m)
    apply(simp add: abc_Hoare_halt_def, auto)
    apply(rename_tac n)
    by(case_tac abc_steps_l (0, args) (ap [+] dummy_abc (length args)) n, auto)
  thus ?thesis
    using assms tp compile_correct_halt[OF refl refl _ h _ _ refl]
    by(auto simp: crsp.simps start_of.simps abc_lm_v.simps)
qed

lemma recursive_compile_to_tm_correct2:
  assumes termi:  terminate recf args
  shows ∃ stp m l. steps0 (Suc 0, [Bk, Bk], <args>) (tm_of_rec recf) stp =
                    (0, Bk↑Suc (Suc m), Oc↑Suc (rec_exec recf args) @ Bk↑l)
proof(cases rec_ci recf, simp add: tm_of_rec.simps)
  fix ap ar fp
  assume rec_ci recf = (ap, ar, fp)
  thus ∃stp m l. steps0 (Suc 0, [Bk, Bk], <args>)
    (tm_of (ap [+] dummy_abc ar) @ shift (mopup ar) (sum_list (layout_of (ap [+] dummy_abc
```

```
ar)))) stp =
   (0, Bk # Bk # Bk ↑ m, Oc # Oc ↑ rec_exec recf args @ Bk ↑ l)
  using recursive_compile_to_tm_correct1[of recf ap ar fp args tm_of (ap [+] dummy_abc (length
args)) [] 0]
     assms param_pattern[of recf args ap ar fp]
   by(simp add: replicate_Suc[THEN sym] replicate_Suc_iff_anywhere del: replicate_Suc,
      simp add: exp_suc del: replicate_Suc)
qed

lemma recursive_compile_to_tm_correct3:
 assumes termi: terminate recf args
 shows {λ tp. tp =([Bk, Bk], <args>)} (tm_of_rec recf)
       {λ tp. ∃ k l. tp = (Bk↑ k, <rec_exec recf args> @ Bk ↑ l)}
 using recursive_compile_to_tm_correct2[OF assms]
 apply(auto simp add: Hoare_halt_def ) apply(rename_tac stp M l)
 apply(rule_tac x = stp in exI)
 apply(auto simp add: tape_of_nat_def)
 apply(rule_tac x = Suc (Suc M) in exI)
 apply(simp)
 done

lemma list_all_suc_many[simp]:
 list_all (λ(acn, s). s ≤ Suc (Suc (Suc (Suc (Suc (Suc (2 * n))))))) xs ⟹
 list_all (λ(acn, s). s ≤ Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (2 * n))))))))) xs
proof(induct xs)
 case (Cons a xs)
 then show ?case by(cases a, simp)
qed simp


lemma shift_append: shift (xs @ ys) n = shift xs n @ shift ys n
 apply(simp add: shift.simps)
 done

lemma length_shift_mopup[simp]: length (shift (mopup n) ss) = 4 * n + 12
 apply(auto simp: mopup.simps shift_append mopup_b_def)
 done

lemma length_tm_even[intro]: length (tm_of ap) mod 2 = 0
 apply(simp add: tm_of.simps)
 done

lemma tms_of_at_index[simp]: k < length ap ⟹ tms_of ap ! k  =
 ci (layout_of ap) (start_of (layout_of ap) k) (ap ! k)
 apply(simp add: tms_of.simps tpairs_of.simps)
 done

lemma start_of_suc_inc:
 ⟦k < length ap; ap ! k = Inc n⟧ ⟹ start_of (layout_of ap) (Suc k) =
                 start_of (layout_of ap) k + 2 * n + 9
```

```
 apply(rule_tac start_of_Suc1, auto simp: abc_fetch.simps)
 done

lemma start_of_suc_dec:
 ⟦k < length ap; ap ! k = (Dec n e)⟧ ⟹ start_of (layout_of ap) (Suc k) =
                  start_of (layout_of ap) k + 2 ∗ n + 16
 apply(rule_tac start_of_Suc2, auto simp: abc_fetch.simps)
 done

lemma inc_state_all_le:
 ⟦k < length ap; ap ! k = Inc n;
     (a, b) ∈ set (shift (shift tinc_b (2 ∗ n))
                  (start_of (layout_of ap) k − Suc 0))⟧
     ⟹ b ≤ start_of (layout_of ap) (length ap)
 apply(subgoal_tac b ≤ start_of (layout_of ap) (Suc k))
  apply(subgoal_tac start_of (layout_of ap) (Suc k) ≤ start_of (layout_of ap) (length ap) )
   apply(arith)
  apply(cases Suc k = length ap, simp)
  apply(rule_tac start_of_less, simp)
 apply(auto simp: tinc_b_def shift.simps start_of_suc_inc length_of.simps )
 done

lemma findnth_le[elim]:
 (a, b) ∈ set (shift (findnth n) (start_of (layout_of ap) k − Suc 0))
 ⟹ b ≤ Suc (start_of (layout_of ap) k + 2 ∗ n)
 apply(induct n, force simp add: shift.simps)
 apply(simp add: shift_append, auto)
 apply(auto simp: shift.simps)
 done

lemma findnth_state_all_le1:
 ⟦k < length ap; ap ! k = Inc n;
 (a, b) ∈
 set (shift (findnth n) (start_of (layout_of ap) k − Suc 0))⟧
 ⟹ b ≤ start_of (layout_of ap) (length ap)
 apply(subgoal_tac b ≤ start_of (layout_of ap) (Suc k))
  apply(subgoal_tac start_of (layout_of ap) (Suc k) ≤ start_of (layout_of ap) (length ap) )
   apply(arith)
  apply(cases Suc k = length ap, simp)
  apply(rule_tac start_of_less, simp)
 apply(subgoal_tac b ≤ start_of (layout_of ap) k + 2∗n + 1 ∧
   start_of (layout_of ap) k + 2∗n + 1 ≤  start_of (layout_of ap) (Suc k), auto)
 apply(auto simp: tinc_b_def shift.simps length_of.simps  start_of_suc_inc)
 done

lemma start_of_eq: length ap < as ⟹ start_of (layout_of ap) as = start_of (layout_of ap)
(length ap)
proof(induct as)
 case (Suc as)
 then show ?case
```

```
   apply(cases length ap < as, simp add: start_of.simps)
   apply(subgoal_tac as = length ap)
    apply(simp add: start_of.simps)
   apply arith
   done
qed simp

lemma start_of_all_le: start_of (layout_of ap) as ≤ start_of (layout_of ap) (length ap)
  apply(subgoal_tac as > length ap ∨ as = length ap ∨ as < length ap,
     auto simp: start_of_eq start_of_less)
  done

lemma findnth_state_all_le2:
  ⟦k < length ap;
  ap ! k = Dec n e;
  (a, b) ∈ set (shift (findnth n) (start_of (layout_of ap) k − Suc 0))⟧
  ⟹ b ≤ start_of (layout_of ap) (length ap)
  apply(subgoal_tac b ≤ start_of (layout_of ap) k + 2∗n + 1 ∧
    start_of (layout_of ap) k + 2∗n + 1 ≤  start_of (layout_of ap) (Suc k) ∧
     start_of (layout_of ap) (Suc k) ≤ start_of (layout_of ap) (length ap), auto)
   apply(subgoal_tac start_of (layout_of ap) (Suc k) =
  start_of  (layout_of ap)  k + 2∗n + 16, simp)
   apply(simp add: start_of_suc_dec)
  apply(rule_tac start_of_all_le)
  done

lemma dec_state_all_le[simp]:
  ⟦k < length ap; ap ! k = Dec n e;
  (a, b) ∈ set (shift (shift tdec_b (2 ∗ n))
  (start_of (layout_of ap) k − Suc 0))⟧
     ⟹ b ≤ start_of (layout_of ap) (length ap)
  apply(subgoal_tac 2∗n + start_of (layout_of ap) k + 16 ≤ start_of (layout_of ap) (length ap)
∧ start_of (layout_of ap) k > 0)
   prefer 2
   apply(subgoal_tac start_of (layout_of ap) (Suc k) = start_of (layout_of ap) k + 2∗n + 16
             ∧ start_of (layout_of ap) (Suc k) ≤ start_of (layout_of ap) (length ap))
    apply(simp, rule_tac conjI)
    apply(simp add: start_of_suc_dec)
   apply(rule_tac start_of_all_le)
  apply(auto simp: tdec_b_def shift.simps)
  done

lemma tms_any_less:
  ⟦k < length ap; (a, b) ∈ set (tms_of ap ! k)⟧ ⟹
  b ≤ start_of (layout_of ap) (length ap)
  apply(cases ap!k, auto simp: tms_of.simps tpairs_of.simps ci.simps shift_append adjust.simps)
     apply(erule_tac findnth_state_all_le1, simp_all)
    apply(erule_tac inc_state_all_le, simp_all)
   apply(erule_tac findnth_state_all_le2, simp_all)
  apply(rule_tac start_of_all_le)
```

apply(*rule_tac start_of_all_le*)
done

lemma *concat_in*: $i < length\ (concat\ xs) \Longrightarrow$
$\exists k < length\ xs.\ concat\ xs\ !\ i \in set\ (xs\ !\ k)$
proof(*induct xs rule*: *rev_induct*)
case (*snoc x xs*)
then show *?case*
apply(*cases* $i < length\ (concat\ xs)$, *simp*)
apply(*erule_tac exE*, *rule_tac* $x = k$ **in** *exI*)
apply(*simp add*: *nth_append*)
apply(*rule_tac* $x = length\ xs$ **in** *exI*, *simp*)
apply(*simp add*: *nth_append*)
done
qed *auto*

declare *length_concat*[*simp*]

lemma *in_tms*: $i < length\ (tm\_of\ ap) \Longrightarrow \exists\ k < length\ ap.\ (tm\_of\ ap\ !\ i) \in set\ (tms\_of\ ap\ !\ k)$
apply(*simp only*: *tm_of.simps*)
using *concat_in*[*of i tms_of ap*]
apply(*auto*)
done

lemma *all_le_start_of*: *list_all* $(\lambda(acn, s).$
$s \le start\_of\ (layout\_of\ ap)\ (length\ ap))\ (tm\_of\ ap)$
apply(*simp only*: *list_all_length*)
apply(*rule_tac allI*, *rule_tac impI*)
apply(*drule_tac in_tms*, *auto elim*: *tms_any_less*)
done

lemma *length_ci*:
$[\![k < length\ ap;\ length\ (ci\ ly\ y\ (ap\ !\ k)) = 2 * qa]\!]$
$\Longrightarrow layout\_of\ ap\ !\ k = qa$
apply(*cases ap* ! *k*)
apply(*auto simp*: *layout_of.simps ci.simps*
length_of.simps tinc_b_def tdec_b_def length_findnth adjust.simps)
done

lemma *ci_even*[*intro*]: $length\ (ci\ ly\ y\ i)\ mod\ 2 = 0$
apply(*cases i*, *auto simp*: *ci.simps length_findnth*
tinc_b_def adjust.simps tdec_b_def)
done

lemma *sum_list_ci_even*[*intro*]: $sum\_list\ (map\ (length \circ (\lambda(x, y).\ ci\ ly\ x\ y))\ zs)\ mod\ 2 = 0$
proof(*induct zs rule*: *rev_induct*)
case (*snoc x xs*)
then show *?case*
apply(*cases x*, *simp*)
apply(*subgoal_tac* $length\ (ci\ ly\ (fst\ x)\ (snd\ x))\ mod\ 2 = 0$)

apply(*auto*)
done
qed (*simp*)

lemma *zip_pre*:
($length\ ys$) $\leq$ $length\ ap \Longrightarrow$
$zip\ ys\ ap = zip\ ys\ (take\ (length\ ys)\ (ap{::}'a\ list))$
proof(*induct ys arbitrary*: *ap*)
case (*Cons a ys*)
from *Cons*(2) **have** $z{:}ap = aa \# list \Longrightarrow zip\ (a \# ys)\ ap = zip\ (a \# ys)\ (take\ (length\ (a \# ys))\ ap)$
for *aa list* **using** *Cons*(*1*)[*of list*] **by** *simp*
thus *?case* **by** (*cases ap*;*simp*)
qed *simp*

lemma *length_start_of_tm*: $start\_of\ (layout\_of\ ap)\ (length\ ap) = Suc\ (length\ (tm\_of\ ap)\ div\ 2)$
using *tpa_states*[*of tm_of ap length ap ap*]
by(*simp add*: *tm_of.simps*)

lemma *list_all_add_6E*[*elim*]: $list\_all\ (\lambda(acn, s).\ s \leq Suc\ q)\ xs$
$\Longrightarrow list\_all\ (\lambda(acn, s).\ s \leq q + (2 * n + 6))\ xs$
by(*auto simp*: *list_all_length*)

lemma *mopup_b_12*[*simp*]: $length\ mopup\_b = 12$
by(*simp add*: *mopup_b_def*)

lemma *mp_up_all_le*: $list\_all\ (\lambda(acn, s).\ s \leq q + (2 * n + 6))$
$[(R, Suc\ (Suc\ (2 * n + q))), (R, Suc\ (2 * n + q)),$
$(L, 5 + 2 * n + q), (W0, Suc\ (Suc\ (Suc\ (2 * n + q)))), (R, 4 + 2 * n + q),$
$(W0, Suc\ (Suc\ (Suc\ (2 * n + q)))), (R, Suc\ (Suc\ (2 * n + q))),$
$(W0, Suc\ (Suc\ (Suc\ (2 * n + q)))), (L, 5 + 2 * n + q),$
$(L, 6 + 2 * n + q), (R, 0), (L, 6 + 2 * n + q)]$
by(*auto*)

lemma *mopup_le6*[*simp*]: $(a, b) \in set\ (mopup\_a\ n) \Longrightarrow b \leq 2 * n + 6$
by(*induct n*, *auto simp*: *mopup_a.simps*)

lemma *shift_le2*[*simp*]: $(a, b) \in set\ (shift\ (mopup\ n)\ x)$
$\Longrightarrow b \leq (2 * x + length\ (mopup\ n))\ div\ 2$
apply(*auto simp*: *mopup.simps shift_append shift.simps*)
apply(*auto simp*: *mopup_b_def*)
done

lemma *mopup_ge2*[*intro*]: $2 \leq x + length\ (mopup\ n)$
apply(*simp add*: *mopup.simps*)
done

lemma *mopup_even*[*intro*]: $(2 * x + length\ (mopup\ n))\ mod\ 2 = 0$
by(*auto simp*: *mopup.simps*)

lemma *mopup_div_2*[*simp*]: $b \le Suc\ x$
$\Longrightarrow b \le (2 * x + length\ (mopup\ n))\ div\ 2$
by(*auto simp*: *mopup.simps*)

lemma *wf_tm_from_abacus*: **assumes** $tp = tm\_of\ ap$
shows *tm_wf0* (*tp* @ *shift* (*mopup n*) (*length tp div 2*))
proof −
have *is_even* (*length* (*mopup n*)) **for** *n* **using** *tm_wf.simps* **by** *blast*
moreover have $(aa, ba) \in set\ (mopup\ n) \Longrightarrow ba \le length\ (mopup\ n)\ div\ 2$ **for** *aa ba*
by (*metis* (*no_types*, *lifting*) *add_cancel_left_right case_prodD tm_wf.simps wf_mopup*)
moreover have $(\forall x{\in}set\ (tm\_of\ ap).\ case\ x\ of\ (acn, s) \Rightarrow s \le Suc\ (sum\_list\ (layout\_of\ ap)))$
$\Longrightarrow$
$(a, b) \in set\ (tm\_of\ ap) \Longrightarrow b \le sum\_list\ (layout\_of\ ap) + length\ (mopup\ n)\ div\ 2$
for *a b s*
by (*metis* (*no_types*, *lifting*) *add_Suc add_cancel_left_right case_prodD div_mult_mod_eq le_SucE mult_2_right not_numeral_le_zero tm_wf.simps trans_le_add1 wf_mopup*)
ultimately show *?thesis* **unfolding** *assms*
using *length_start_of_tm*[*of ap*] *all_le_start_of*[*of ap*] *tm_wf.simps*
by(*auto simp*: *List.list_all_iff shift.simps*)
qed

lemma *wf_tm_from_recf*:
assumes *compile*: $tp = tm\_of\_rec\ recf$
shows *tm_wf0 tp*
proof −
obtain *a b c* **where** $rec\_ci\ recf = (a, b, c)$
by (*metis prod_cases3*)
thus *?thesis*
using *compile*
using *wf_tm_from_abacus*[*of tm_of* (*a* [+] *dummy_abc b*) (*a* [+] *dummy_abc b*) *b*]
by *simp*
qed

end

11 Bijections between natural numbers and other types

theory *Nat_Bijection*
imports *Main*
begin

11.1 Type $nat \times nat$

Triangle numbers: 0, 1, 3, 6, 10, 15, ...

definition *triangle* :: $nat \Rightarrow nat$
where $triangle\ n = (n * Suc\ n)\ div\ 2$

lemma *triangle_0* [*simp*]: $triangle\ 0 = 0$

by (*simp add*: *triangle_def*)

lemma *triangle_Suc* [*simp*]: *triangle* (*Suc n*) = *triangle n* + *Suc n*
by (*simp add*: *triangle_def*)

definition *prod_encode* :: *nat* × *nat* ⇒ *nat*
where *prod_encode* = (λ(*m*, *n*). *triangle* (*m* + *n*) + *m*)

In this auxiliary function, *triangle k* + *m* is an invariant.

fun *prod_decode_aux* :: *nat* ⇒ *nat* ⇒ *nat* × *nat*
where *prod_decode_aux k m* =
(*if m* ≤ *k then* (*m*, *k* − *m*) *else prod_decode_aux* (*Suc k*) (*m* − *Suc k*))

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

definition *prod_decode* :: *nat* ⇒ *nat* × *nat*
where *prod_decode* = *prod_decode_aux 0*

lemma *prod_encode_prod_decode_aux*: *prod_encode* (*prod_decode_aux k m*) = *triangle k* + *m*
apply (*induct k m rule*: *prod_decode_aux*.*induct*)
apply (*subst prod_decode_aux*.*simps*)
apply (*simp add*: *prod_encode_def*)
done

lemma *prod_decode_inverse* [*simp*]: *prod_encode* (*prod_decode n*) = *n*
by (*simp add*: *prod_decode_def prod_encode_prod_decode_aux*)

lemma *prod_decode_triangle_add*: *prod_decode* (*triangle k* + *m*) = *prod_decode_aux k m*
apply (*induct k arbitrary*: *m*)
apply (*simp add*: *prod_decode_def*)
apply (*simp only*: *triangle_Suc add*.*assoc*)
apply (*subst prod_decode_aux*.*simps*)
apply *simp*
done

lemma *prod_encode_inverse* [*simp*]: *prod_decode* (*prod_encode x*) = *x*
unfolding *prod_encode_def*
apply (*induct x*)
apply (*simp add*: *prod_decode_triangle_add*)
apply (*subst prod_decode_aux*.*simps*)
apply *simp*
done

lemma *inj_prod_encode*: *inj_on prod_encode A*
by (*rule inj_on_inverseI*) (*rule prod_encode_inverse*)

lemma *inj_prod_decode*: *inj_on prod_decode A*
by (*rule inj_on_inverseI*) (*rule prod_decode_inverse*)

lemma *surj_prod_encode*: *surj prod_encode*

by (*rule surjI*) (*rule prod_decode_inverse*)

lemma *surj_prod_decode*: *surj prod_decode*
by (*rule surjI*) (*rule prod_encode_inverse*)

lemma *bij_prod_encode*: *bij prod_encode*
by (*rule bijI* [*OF inj_prod_encode surj_prod_encode*])

lemma *bij_prod_decode*: *bij prod_decode*
by (*rule bijI* [*OF inj_prod_decode surj_prod_decode*])

lemma *prod_encode_eq*: *prod_encode* x = *prod_encode* $y \longleftrightarrow x = y$
by (*rule inj_prod_encode* [*THEN inj_eq*])

lemma *prod_decode_eq*: *prod_decode* x = *prod_decode* $y \longleftrightarrow x = y$
by (*rule inj_prod_decode* [*THEN inj_eq*])

Ordering properties

lemma *le_prod_encode_1*: $a \leq$ *prod_encode* (a, b)
by (*simp add*: *prod_encode_def*)

lemma *le_prod_encode_2*: $b \leq$ *prod_encode* (a, b)
by (*induct b*) (*simp_all add*: *prod_encode_def*)

11.2 Type *nat* + *nat*

definition *sum_encode* :: *nat* + *nat* ⇒ *nat*
where *sum_encode x* = (*case x of Inl a* ⇒ $2 * a$ | *Inr b* ⇒ *Suc* $(2 * b)$)

definition *sum_decode* :: *nat* ⇒ *nat* + *nat*
where *sum_decode n* = (*if even n then Inl* (*n div* 2) *else Inr* (*n div* 2))

lemma *sum_encode_inverse* [*simp*]: *sum_decode* (*sum_encode x*) = *x*
by (*induct x*) (*simp_all add*: *sum_decode_def sum_encode_def*)

lemma *sum_decode_inverse* [*simp*]: *sum_encode* (*sum_decode n*) = *n*
by (*simp add*: *even_two_times_div_two sum_decode_def sum_encode_def*)

lemma *inj_sum_encode*: *inj_on sum_encode A*
by (*rule inj_on_inverseI*) (*rule sum_encode_inverse*)

lemma *inj_sum_decode*: *inj_on sum_decode A*
by (*rule inj_on_inverseI*) (*rule sum_decode_inverse*)

lemma *surj_sum_encode*: *surj sum_encode*
by (*rule surjI*) (*rule sum_decode_inverse*)

lemma *surj_sum_decode*: *surj sum_decode*
by (*rule surjI*) (*rule sum_encode_inverse*)

lemma *bij_sum_encode*: *bij sum_encode*
by (*rule bijI* [*OF inj_sum_encode surj_sum_encode*])

lemma *bij_sum_decode*: *bij sum_decode*
by (*rule bijI* [*OF inj_sum_decode surj_sum_decode*])

lemma *sum_encode_eq*: *sum_encode* x = *sum_encode* $y \longleftrightarrow x = y$
by (*rule inj_sum_encode* [*THEN inj_eq*])

lemma *sum_decode_eq*: *sum_decode* x = *sum_decode* $y \longleftrightarrow x = y$
by (*rule inj_sum_decode* [*THEN inj_eq*])

11.3 Type *int*

definition *int_encode* :: *int* $\Rightarrow$ *nat*
where *int_encode i* = *sum_encode* (*if* $0 \leq i$ *then Inl* (*nat i*) *else Inr* (*nat* $(- i - 1)$))

definition *int_decode* :: *nat* $\Rightarrow$ *int*
where *int_decode n* = (*case sum_decode n of Inl a* $\Rightarrow$ *int a* | *Inr b* $\Rightarrow$ $-$ *int b* $- 1$)

lemma *int_encode_inverse* [*simp*]: *int_decode* (*int_encode x*) = *x*
by (*simp add*: *int_decode_def int_encode_def*)

lemma *int_decode_inverse* [*simp*]: *int_encode* (*int_decode n*) = *n*
unfolding *int_decode_def int_encode_def*
using *sum_decode_inverse* [*of n*] **by** (*cases sum_decode n*) *simp_all*

lemma *inj_int_encode*: *inj_on int_encode A*
by (*rule inj_on_inverseI*) (*rule int_encode_inverse*)

lemma *inj_int_decode*: *inj_on int_decode A*
by (*rule inj_on_inverseI*) (*rule int_decode_inverse*)

lemma *surj_int_encode*: *surj int_encode*
by (*rule surjI*) (*rule int_decode_inverse*)

lemma *surj_int_decode*: *surj int_decode*
by (*rule surjI*) (*rule int_encode_inverse*)

lemma *bij_int_encode*: *bij int_encode*
by (*rule bijI* [*OF inj_int_encode surj_int_encode*])

lemma *bij_int_decode*: *bij int_decode*
by (*rule bijI* [*OF inj_int_decode surj_int_decode*])

lemma *int_encode_eq*: *int_encode* x = *int_encode* $y \longleftrightarrow x = y$
by (*rule inj_int_encode* [*THEN inj_eq*])

lemma *int_decode_eq*: *int_decode* x = *int_decode* $y \longleftrightarrow x = y$
by (*rule inj_int_decode* [*THEN inj_eq*])

11.4 Type *nat list*

```
fun list_encode :: nat list ⇒ nat
  where
    list_encode [] = 0
  | list_encode (x # xs) = Suc (prod_encode (x, list_encode xs))

function list_decode :: nat ⇒ nat list
  where
    list_decode 0 = []
  | list_decode (Suc n) = (case prod_decode n of (x, y) ⇒ x # list_decode y)
  by pat_completeness auto

termination list_decode
  apply (relation measure id)
   apply simp_all
  apply (drule arg_cong [where f=prod_encode])
  apply (drule sym)
  apply (simp add: le_imp_less_Suc le_prod_encode_2)
  done

lemma list_encode_inverse [simp]: list_decode (list_encode x) = x
  by (induct x rule: list_encode.induct) simp_all

lemma list_decode_inverse [simp]: list_encode (list_decode n) = n
  apply (induct n rule: list_decode.induct)
   apply simp
  apply (simp split: prod.split)
  apply (simp add: prod_decode_eq [symmetric])
  done

lemma inj_list_encode: inj_on list_encode A
  by (rule inj_on_inverseI) (rule list_encode_inverse)

lemma inj_list_decode: inj_on list_decode A
  by (rule inj_on_inverseI) (rule list_decode_inverse)

lemma surj_list_encode: surj list_encode
  by (rule surjI) (rule list_decode_inverse)

lemma surj_list_decode: surj list_decode
  by (rule surjI) (rule list_encode_inverse)

lemma bij_list_encode: bij list_encode
  by (rule bijI [OF inj_list_encode surj_list_encode])

lemma bij_list_decode: bij list_decode
  by (rule bijI [OF inj_list_decode surj_list_decode])

lemma list_encode_eq: list_encode x = list_encode y ⟷ x = y
```

by (*rule inj_list_encode* [*THEN inj_eq*])

lemma *list_decode_eq*: *list_decode x* = *list_decode y* ⟷ *x* = *y*
by (*rule inj_list_decode* [*THEN inj_eq*])

11.5 Finite sets of naturals

11.5.1 Preliminaries

lemma *finite_vimage_Suc_iff*: *finite* (*Suc* −' *F*) ⟷ *finite F*
 apply (*safe intro*!: *finite_vimageI inj_Suc*)
 apply (*rule finite_subset* [**where** *B*=*insert 0* (*Suc* ' *Suc* −' *F*)])
  apply (*rule subsetI*)
  apply (*case_tac x*)
   apply *simp*
  apply *simp*
 apply (*rule finite_insert* [*THEN iffD2*])
 apply (*erule finite_imageI*)
 done

lemma *vimage_Suc_insert_0*: *Suc* −' *insert 0 A* = *Suc* −' *A*
 by *auto*

lemma *vimage_Suc_insert_Suc*: *Suc* −' *insert* (*Suc n*) *A* = *insert n* (*Suc* −' *A*)
 by *auto*

lemma *div2_even_ext_nat*:
 fixes *x y* :: *nat*
 assumes *x div 2* = *y div 2*
  and *even x* ⟷ *even y*
 shows *x* = *y*
proof −
 from ⟨*even x* ⟷ *even y*⟩ **have** *x mod 2* = *y mod 2*
  by (*simp only*: *even_iff_mod_2_eq_zero*) *auto*
 with *assms* **have** *x div 2* ∗ *2* + *x mod 2* = *y div 2* ∗ *2* + *y mod 2*
  by *simp*
 then show *?thesis*
  by *simp*
qed

11.5.2 From sets to naturals

definition *set_encode* :: *nat set* ⇒ *nat*
 where *set_encode* = *sum* ((^) *2*)

lemma *set_encode_empty* [*simp*]: *set_encode* {} = *0*
 by (*simp add*: *set_encode_def*)

lemma *set_encode_inf*: ¬ *finite A* ⟹ *set_encode A* = *0*
 by (*simp add*: *set_encode_def*)

lemma *set_encode_insert* [*simp*]: *finite A* $\Longrightarrow$ *n* $\notin$ *A* $\Longrightarrow$ *set_encode* (*insert n A*) = *2^n* + *set_encode A*
 by (*simp add*: *set_encode_def*)

lemma *even_set_encode_iff*: *finite A* $\Longrightarrow$ *even* (*set_encode A*) $\longleftrightarrow$ *0* $\notin$ *A*
 by (*induct set*: *finite*) (*auto simp*: *set_encode_def*)

lemma *set_encode_vimage_Suc*: *set_encode* (*Suc* −' *A*) = *set_encode A div 2*
 apply (*cases finite A*)
  apply (*erule finite_induct*)
   apply *simp*
  apply (*case_tac x*)
   apply (*simp add*: *even_set_encode_iff vimage_Suc_insert_0*)
  apply (*simp add*: *finite_vimageI add.commute vimage_Suc_insert_Suc*)
 apply (*simp add*: *set_encode_def finite_vimage_Suc_iff*)
 done

lemmas *set_encode_div_2* = *set_encode_vimage_Suc* [*symmetric*]

11.5.3 From naturals to sets

definition *set_decode* :: *nat* ⇒ *nat set*
 where *set_decode x* = {*n*. *odd* (*x div 2* ^ *n*)}

lemma *set_decode_0* [*simp*]: *0* ∈ *set_decode x* $\longleftrightarrow$ *odd x*
 by (*simp add*: *set_decode_def*)

lemma *set_decode_Suc* [*simp*]: *Suc n* ∈ *set_decode x* $\longleftrightarrow$ *n* ∈ *set_decode* (*x div 2*)
 by (*simp add*: *set_decode_def div_mult2_eq*)

lemma *set_decode_zero* [*simp*]: *set_decode 0* = {}
 by (*simp add*: *set_decode_def*)

lemma *set_decode_div_2*: *set_decode* (*x div 2*) = *Suc* −' *set_decode x*
 by *auto*

lemma *set_decode_plus_power_2*:
 n $\notin$ *set_decode z* $\Longrightarrow$ *set_decode* (2 ^ *n* + *z*) = *insert n* (*set_decode z*)
proof (*induct n arbitrary*: *z*)
 case *0*
 show *?case*
 proof (*rule set_eqI*)
  show *q* ∈ *set_decode* (2 ^ *0* + *z*) $\longleftrightarrow$ *q* ∈ *insert 0* (*set_decode z*) **for** *q*
   by (*induct q*) (*use 0* **in** *simp_all*)
 qed
next
 case (*Suc n*)
 show *?case*
 proof (*rule set_eqI*)

```
    show q ∈ set_decode (2 ^ Suc n + z) ⟷ q ∈ insert (Suc n) (set_decode z) for q
      by (induct q) (use Suc in simp_all)
  qed
qed

lemma finite_set_decode [simp]: finite (set_decode n)
  apply (induct n rule: nat_less_induct)
  apply (case_tac n = 0)
   apply simp
  apply (drule_tac x=n div 2 in spec)
  apply simp
  apply (simp add: set_decode_div_2)
  apply (simp add: finite_vimage_Suc_iff)
  done
```

11.5.4 Proof of isomorphism

```
lemma set_decode_inverse [simp]: set_encode (set_decode n) = n
  apply (induct n rule: nat_less_induct)
  apply (case_tac n = 0)
   apply simp
  apply (drule_tac x=n div 2 in spec)
  apply simp
  apply (simp add: set_decode_div_2 set_encode_vimage_Suc)
  apply (erule div2_even_ext_nat)
  apply (simp add: even_set_encode_iff)
  done

lemma set_encode_inverse [simp]: finite A ⟹ set_decode (set_encode A) = A
  apply (erule finite_induct)
   apply simp_all
  apply (simp add: set_decode_plus_power_2)
  done

lemma inj_on_set_encode: inj_on set_encode (Collect finite)
  by (rule inj_on_inverseI [where g = set_decode]) simp

lemma set_encode_eq: finite A ⟹ finite B ⟹ set_encode A = set_encode B ⟷ A = B
  by (rule iffI) (simp_all add: inj_onD [OF inj_on_set_encode])

lemma subset_decode_imp_le:
  assumes set_decode m ⊆ set_decode n
  shows m ≤ n
proof −
  have n = m + set_encode (set_decode n − set_decode m)
  proof −
    obtain A B where
      m = set_encode A finite A
      n = set_encode B finite B
      by (metis finite_set_decode set_decode_inverse)
```

```
  with assms show ?thesis
    by auto (simp add: set_encode_def add.commute sum.subset_diff)
  qed
  then show ?thesis
    by (metis le_add1)
qed

end
```

12 Common discrete functions

```
theory Discrete
imports Complex_Main
begin
```

12.1 Discrete logarithm

```
context
begin

qualified fun log :: nat ⇒ nat
  where [simp del]: log n = (if n < 2 then 0 else Suc (log (n div 2)))

lemma log_induct [consumes 1, case_names one double]:
  fixes n :: nat
  assumes n > 0
  assumes one: P 1
  assumes double: ⋀n. n ≥ 2 ⟹ P (n div 2) ⟹ P n
  shows P n
using ‹n > 0› proof (induct n rule: log.induct)
  fix n
  assume ¬ n < 2 ⟹
        0 < n div 2 ⟹ P (n div 2)
  then have *: n ≥ 2 ⟹ P (n div 2) by simp
  assume n > 0
  show P n
  proof (cases n = 1)
    case True
    with one show ?thesis by simp
  next
    case False
    with ‹n > 0› have n ≥ 2 by auto
    with * have P (n div 2) .
    with ‹n ≥ 2› show ?thesis by (rule double)
  qed
qed

lemma log_zero [simp]: log 0 = 0
  by (simp add: log.simps)
```

lemma *log_one* [*simp*]: *log 1* = *0*
 by (*simp add*: *log*.*simps*)

lemma *log_Suc_zero* [*simp*]: *log* (*Suc 0*) = *0*
 using *log_one* **by** *simp*

lemma *log_rec*: $n \geq 2 \Longrightarrow log\ n = Suc\ (log\ (n\ div\ 2))$
 by (*simp add*: *log*.*simps*)

lemma *log_twice* [*simp*]: $n \neq 0 \Longrightarrow log\ (2 * n) = Suc\ (log\ n)$
 by (*simp add*: *log_rec*)

lemma *log_half* [*simp*]: $log\ (n\ div\ 2) = log\ n - 1$
proof (*cases* $n < 2$)
 case *True*
 then have $n = 0 \vee n = 1$ **by** *arith*
 then show *?thesis* **by** (*auto simp del*: *One_nat_def*)
next
 case *False*
 then show *?thesis* **by** (*simp add*: *log_rec*)
qed

lemma *log_exp* [*simp*]: $log\ (2\ \hat{}\ n) = n$
 by (*induct n*) *simp_all*

lemma *log_mono*: *mono log*
proof
 fix *m n* :: *nat*
 assume $m \leq n$
 then show $log\ m \leq log\ n$
 proof (*induct m arbitrary*: *n rule*: *log*.*induct*)
  case (*1 m*)
  then have *mn2*: $m\ div\ 2 \leq n\ div\ 2$ **by** *arith*
  show $log\ m \leq log\ n$
  proof (*cases* $m \geq 2$)
   case *False*
   then have $m = 0 \vee m = 1$ **by** *arith*
   then show *?thesis* **by** (*auto simp del*: *One_nat_def*)
  next
   case *True* **then have** $\neg\ m < 2$ **by** *simp*
   with *mn2* **have** $n \geq 2$ **by** *arith*
   from *True* **have** *m2_0*: $m\ div\ 2 \neq 0$ **by** *arith*
   with *mn2* **have** *n2_0*: $n\ div\ 2 \neq 0$ **by** *arith*
   from $\langle \neg\ m < 2 \rangle$ *1.hyps mn2* **have** $log\ (m\ div\ 2) \leq log\ (n\ div\ 2)$ **by** *blast*
   with *m2_0 n2_0* **have** $log\ (2 * (m\ div\ 2)) \leq log\ (2 * (n\ div\ 2))$ **by** *simp*
   with *m2_0 n2_0* $\langle m \geq 2 \rangle$ $\langle n \geq 2 \rangle$ **show** *?thesis* **by** (*simp only*: *log_rec* [*of m*] *log_rec* [*of n*])
simp
  qed
 qed

```
qed

lemma log_exp2_le:
  assumes n > 0
  shows 2 ^ log n ≤ n
  using assms
proof (induct n rule: log_induct)
  case one
  then show ?case by simp
next
  case (double n)
  with log_mono have log n ≥ Suc 0
    by (simp add: log.simps)
  assume 2 ^ log (n div 2) ≤ n div 2
  with ⟨n ≥ 2⟩ have 2 ^ (log n − Suc 0) ≤ n div 2 by simp
  then have 2 ^ (log n − Suc 0) ∗ 2 ^ 1 ≤ n div 2 ∗ 2 by simp
  with ⟨log n ≥ Suc 0⟩ have 2 ^ log n ≤ n div 2 ∗ 2
    unfolding power_add [symmetric] by simp
  also have n div 2 ∗ 2 ≤ n by (cases even n) simp_all
  finally show ?case .
qed

lemma log_exp2_gt: 2 ∗ 2 ^ log n > n
proof (cases n > 0)
  case True
  thus ?thesis
  proof (induct n rule: log_induct)
    case (double n)
    thus ?case
      by (cases even n) (auto elim!: evenE oddE simp: field_simps log.simps)
  qed simp_all
qed simp_all

lemma log_exp2_ge: 2 ∗ 2 ^ log n ≥ n
  using log_exp2_gt[of n] by simp

lemma log_le_iff: m ≤ n ⟹ log m ≤ log n
  by (rule monoD [OF log_mono])

lemma log_eqI:
  assumes n > 0 2^k ≤ n n < 2 ∗ 2^k
  shows   log n = k
proof (rule antisym)
  from ⟨n > 0⟩ have 2 ^ log n ≤ n by (rule log_exp2_le)
  also have . . . < 2 ^ Suc k using assms by simp
  finally have log n < Suc k by (subst (asm) power_strict_increasing_iff) simp_all
  thus log n ≤ k by simp
next
  have 2^k ≤ n by fact
  also have . . . < 2^(Suc (log n)) by (simp add: log_exp2_gt)
```

```
  finally have k < Suc (log n) by (subst (asm) power_strict_increasing_iff) simp_all
  thus k ≤ log n by simp
qed

lemma log_altdef: log n = (if n = 0 then 0 else nat ⌊Transcendental.log 2 (real_of_nat n)⌋)
proof (cases n = 0)
  case False
  have ⌊Transcendental.log 2 (real_of_nat n)⌋ = int (log n)
  proof (rule floor_unique)
    from False have 2 powr (real (log n)) ≤ real n
      by (simp add: powr_realpow log_exp2_le)
    hence Transcendental.log 2 (2 powr (real (log n))) ≤ Transcendental.log 2 (real n)
      using False by (subst Transcendental.log_le_cancel_iff) simp_all
    also have Transcendental.log 2 (2 powr (real (log n))) = real (log n) by simp
    finally show real_of_int (int (log n)) ≤ Transcendental.log 2 (real n) by simp
  next
    have real n < real (2 * 2 ^ log n)
      by (subst of_nat_less_iff) (rule log_exp2_gt)
    also have ... = 2 powr (real (log n) + 1)
      by (simp add: powr_add powr_realpow)
    finally have Transcendental.log 2 (real n) < Transcendental.log 2 ...
      using False by (subst Transcendental.log_less_cancel_iff) simp_all
    also have ... = real (log n) + 1 by simp
    finally show Transcendental.log 2 (real n) < real_of_int (int (log n)) + 1 by simp
  qed
  thus ?thesis by simp
qed simp_all
```

12.2 Discrete square root

```
qualified definition sqrt :: nat ⇒ nat
  where sqrt n = Max {m. m^2 ≤ n}

lemma sqrt_aux:
  fixes n :: nat
  shows finite {m. m^2 ≤ n} and {m. m^2 ≤ n} ≠ {}
proof −
  { fix m
    assume m^2 ≤ n
    then have m ≤ n
      by (cases m) (simp_all add: power2_eq_square)
  } note ∗∗ = this
  then have {m. m^2 ≤ n} ⊆ {m. m ≤ n} by auto
  then show finite {m. m^2 ≤ n} by (rule finite_subset) rule
  have 0^2 ≤ n by simp
  then show ∗: {m. m^2 ≤ n} ≠ {} by blast
qed

lemma sqrt_unique:
  assumes m^2 ≤ n n < (Suc m)^2
```

shows *Discrete.sqrt n* $= m$
proof $-$
have $m' \leq m$ **if** m'^$2 \leq n$ **for** m'
proof $-$
note *that*
also note *assms*(2)
finally have $m' <$ *Suc m* **by** (*rule power_less_imp_less_base*) *simp_all*
thus $m' \leq m$ **by** *simp*
qed
with ⟨m^$2 \leq n$⟩ *sqrt_aux*[*of n*] **show** *?thesis* **unfolding** *Discrete.sqrt_def*
by (*intro antisym Max.boundedI Max.coboundedI*) *simp_all*
qed

lemma *sqrt_code*[*code*]: *sqrt n* = *Max* (*Set.filter* ($\lambda m.\ m^2 \leq n$) {*0..n*})
proof $-$
from *power2_nat_le_imp_le* [*of _ n*] **have** $\{m.\ m \leq n \wedge m^2 \leq n\} = \{m.\ m^2 \leq n\}$ **by** *auto*
then show *?thesis* **by** (*simp add*: *sqrt_def Set.filter_def*)
qed

lemma *sqrt_inverse_power2* [*simp*]: *sqrt* $(n^2) = n$
proof $-$
have $\{m.\ m \leq n\} \neq \{\}$ **by** *auto*
then have *Max* $\{m.\ m \leq n\} \leq n$ **by** *auto*
then show *?thesis*
by (*auto simp add*: *sqrt_def power2_nat_le_eq_le intro*: *antisym*)
qed

lemma *sqrt_zero* [*simp*]: *sqrt 0* = *0*
using *sqrt_inverse_power2* [*of 0*] **by** *simp*

lemma *sqrt_one* [*simp*]: *sqrt 1* = *1*
using *sqrt_inverse_power2* [*of 1*] **by** *simp*

lemma *mono_sqrt*: *mono sqrt*
proof
fix *m n* :: *nat*
have $*$: $0 * 0 \leq m$ **by** *simp*
assume $m \leq n$
then show *sqrt m* $\leq$ *sqrt n*
by (*auto intro*!: *Max_mono* ⟨$0 * 0 \leq m$⟩ *finite_less_ub simp add*: *power2_eq_square sqrt_def*)
qed

lemma *mono_sqrt'*: $m \leq n \Longrightarrow$ *Discrete.sqrt m* $\leq$ *Discrete.sqrt n*
using *mono_sqrt* **unfolding** *mono_def* **by** *auto*

lemma *sqrt_greater_zero_iff* [*simp*]: *sqrt n* $> 0 \longleftrightarrow n > 0$
proof $-$
have $*$: $0 <$ *Max* $\{m.\ m^2 \leq n\} \longleftrightarrow (\exists a \in \{m.\ m^2 \leq n\}.\ 0 < a)$
by (*rule Max_gr_iff*) (*fact sqrt_aux*)+

```
  show ?thesis
  proof
    assume 0 < sqrt n
    then have 0 < Max {m. m² ≤ n} by (simp add: sqrt_def)
    with ∗ show 0 < n by (auto dest: power2_nat_le_imp_le)
  next
    assume 0 < n
    then have 1² ≤ n ∧ 0 < (1::nat) by simp
    then have ∃ q. q² ≤ n ∧ 0 < q ..
    with ∗ have 0 < Max {m. m² ≤ n} by blast
    then show 0 < sqrt n by  (simp add: sqrt_def)
  qed
qed

lemma sqrt_power2_le [simp]: (sqrt n)² ≤ n
proof (cases n > 0)
  case False then show ?thesis by simp
next
  case True then have sqrt n > 0 by simp
   then have mono (times (Max {m. m² ≤ n})) by (auto intro: mono_times_nat simp add: sqrt_def)
  then have ∗: Max {m. m² ≤ n} ∗ Max {m. m² ≤ n} = Max (times (Max {m. m² ≤ n}) ‘ {m. m² ≤ n})
    using sqrt_aux [of n] by (rule mono_Max_commute)
  have ⋀a. a ∗ a ≤ n ⟹ Max {m. m ∗ m ≤ n} ∗ a ≤ n
  proof −
    fix q
    assume q ∗ q ≤ n
    show Max {m. m ∗ m ≤ n} ∗ q ≤ n
    proof (cases q > 0)
      case False then show ?thesis by simp
    next
      case True then have mono (times q) by (rule mono_times_nat)
      then have q ∗ Max {m. m ∗ m ≤ n} = Max (times q ‘ {m. m ∗ m ≤ n})
        using sqrt_aux [of n] by (auto simp add: power2_eq_square intro: mono_Max_commute)
       then have Max {m. m ∗ m ≤ n} ∗ q = Max (times q ‘ {m. m ∗ m ≤ n}) by (simp add: ac_simps)
      moreover have finite (( ∗ ) q ‘ {m. m ∗ m ≤ n})
        by (metis (mono_tags) finite_imageI finite_less_ub le_square)
      moreover have ∃x. x ∗ x ≤ n
        by (metis ‹q ∗ q ≤ n›)
      ultimately show ?thesis
        by simp (metis ‹q ∗ q ≤ n› le_cases mult_le_mono1 mult_le_mono2 order_trans)
    qed
  qed
  then have Max (( ∗ ) (Max {m. m ∗ m ≤ n}) ‘ {m. m ∗ m ≤ n}) ≤ n
    apply (subst Max_le_iff)
      apply (metis (mono_tags) finite_imageI finite_less_ub le_square)
     apply auto
    apply (metis le0 mult_0_right)
```

done
with $*$ **show** *?thesis* **by** (*simp add*: *sqrt_def power2_eq_square*)
qed

lemma *sqrt_le*: $sqrt\ n \leq n$
using *sqrt_aux* [*of n*] **by** (*auto simp add*: *sqrt_def intro*: *power2_nat_le_imp_le*)

Additional facts about the discrete square root, thanks to Julian Biendarra, Manuel Eberl

lemma *Suc_sqrt_power2_gt*: $n < (Suc\ (Discrete.sqrt\ n))\hat{}2$
using *Max_ge*[*OF Discrete.sqrt_aux*(*1*), *of Discrete.sqrt n + 1 n*]
by (*cases* $n < (Suc\ (Discrete.sqrt\ n))\hat{}2$) (*simp_all add*: *Discrete.sqrt_def*)

lemma *le_sqrt_iff*: $x \leq Discrete.sqrt\ y \longleftrightarrow x\hat{}2 \leq y$
proof $-$
have $x \leq Discrete.sqrt\ y \longleftrightarrow (\exists z.\ z^2 \leq y \wedge x \leq z)$
using *Max_ge_iff* [*OF Discrete.sqrt_aux*, *of x y*] **by** (*simp add*: *Discrete.sqrt_def*)
also have $\ldots \longleftrightarrow x\hat{}2 \leq y$
proof *safe*
fix *z* **assume** $x \leq z\ z\ \hat{}\ 2 \leq y$
thus $x\hat{}2 \leq y$ **by** (*intro le_trans*[*of* $x\hat{}2\ z\hat{}2\ y$]) (*simp_all add*: *power2_nat_le_eq_le*)
qed *auto*
finally show *?thesis* **.**
qed

lemma *le_sqrtI*: $x\hat{}2 \leq y \Longrightarrow x \leq Discrete.sqrt\ y$
by (*simp add*: *le_sqrt_iff*)

lemma *sqrt_le_iff*: $Discrete.sqrt\ y \leq x \longleftrightarrow (\forall z.\ z\hat{}2 \leq y \longrightarrow z \leq x)$
using *Max.bounded_iff* [*OF Discrete.sqrt_aux*] **by** (*simp add*: *Discrete.sqrt_def*)

lemma *sqrt_leI*:
$(\bigwedge z.\ z\hat{}2 \leq y \Longrightarrow z \leq x) \Longrightarrow Discrete.sqrt\ y \leq x$
by (*simp add*: *sqrt_le_iff*)

lemma *sqrt_Suc*:
$Discrete.sqrt\ (Suc\ n) = (if\ \exists m.\ Suc\ n = m\hat{}2\ then\ Suc\ (Discrete.sqrt\ n)\ else\ Discrete.sqrt\ n)$
proof *cases*
assume $\exists\ m.\ Suc\ n = m\hat{}2$
then obtain *m* **where** *m_def*: $Suc\ n = m\hat{}2$ **by** *blast*
then have *lhs*: $Discrete.sqrt\ (Suc\ n) = m$ **by** *simp*
from *m_def sqrt_power2_le*[*of n*]
have $(Discrete.sqrt\ n)\hat{}2 < m\hat{}2$ **by** *linarith*
with *power2_less_imp_less* **have** *lt_m*: $Discrete.sqrt\ n < m$ **by** *blast*
from *m_def Suc_sqrt_power2_gt*[*of n*]
have $m\hat{}2 \leq (Suc(Discrete.sqrt\ n))\hat{}2$ **by** *simp*
with *power2_nat_le_eq_le* **have** $m \leq Suc\ (Discrete.sqrt\ n)$ **by** *blast*
with *lt_m* **have** $m = Suc\ (Discrete.sqrt\ n)$ **by** *simp*
with *lhs m_def* **show** *?thesis* **by** *fastforce*
next

assume *asm*: $\neg$ ($\exists$ *m*. *Suc n* = *m*ˆ2)
hence *Suc n* $\neq$ (*Discrete.sqrt* (*Suc n*))ˆ2 **by** *simp*
with *sqrt_power2_le*[*of Suc n*]
have *Discrete.sqrt* (*Suc n*) $\leq$ *Discrete.sqrt n* **by** (*intro le_sqrtI*) *linarith*
moreover have *Discrete.sqrt* (*Suc n*) $\geq$ *Discrete.sqrt n*
by (*intro monoD*[*OF mono_sqrt*]) *simp_all*
ultimately show *?thesis* **using** *asm* **by** *simp*
qed

end

end

theory *Recs*
imports *Main*
~~/*src*/*HOL*/*Library*/*Nat_Bijection*
~~/*src*/*HOL*/*Library*/*Discrete*
begin

A more streamlined and cleaned-up version of Recursive Functions following
A Course in Formal Languages, Automata and Groups I. M. Chiswell
and
Lecture on Undecidability Michael M. Wolf

declare *One_nat_def*[*simp del*]

lemma *if_zero_one* [*simp*]:
(*if P then 1 else 0*) = (*0*::*nat*) $\longleftrightarrow$ $\neg$ *P*
(*0*::*nat*) < (*if P then 1 else 0*) = *P*
(*if P then 0 else 1*) = (*if* $\neg$*P then 1 else* (*0*::*nat*))
by (*simp_all*)

lemma *nth*:
(*x* # *xs*) ! *0* = *x*
(*x* # *y* # *xs*) ! *1* = *y*
(*x* # *y* # *z* # *xs*) ! *2* = *z*
(*x* # *y* # *z* # *u* # *xs*) ! *3* = *u*
by (*simp_all*)

13 Some auxiliary lemmas about $\sum$ and $\prod$

lemma *setprod_atMost_Suc*[*simp*]:
($\prod i \leq Suc\ n.\ f\ i$) = ($\prod i \leq n.\ f\ i$) $*$ *f*(*Suc n*)
by(*simp add*:*atMost_Suc mult_ac*)

lemma *setprod_lessThan_Suc*[*simp*]:
($\prod i < Suc\ n.\ f\ i$) = ($\prod i < n.\ f\ i$) $*$ *f n*
by (*simp add*:*lessThan_Suc mult_ac*)

```
lemma setsum_add_nat_ivl2: n ≤ p ⟹
  sum f {..<n} + sum f {n..p} = sum f {..p::nat}
  apply(subst sum.union_disjoint[symmetric])
    apply(auto simp add: ivl_disj_un_one)
  done

lemma setsum_eq_zero [simp]:
  fixes f::nat ⇒ nat
  shows (∑ i < n. f i) = 0 ⟷ (∀ i < n. f i = 0)
    (∑ i ≤ n. f i) = 0 ⟷ (∀ i ≤ n. f i = 0)
  by (auto)

lemma setprod_eq_zero [simp]:
  fixes f::nat ⇒ nat
  shows (∏ i < n. f i) = 0 ⟷ (∃ i < n. f i = 0)
    (∏ i ≤ n. f i) = 0 ⟷ (∃ i ≤ n. f i = 0)
  by (auto)

lemma setsum_one_less:
  fixes n::nat
  assumes ∀ i < n. f i ≤ 1
  shows (∑ i < n. f i) ≤ n
  using assms
  by (induct n) (auto)

lemma setsum_one_le:
  fixes n::nat
  assumes ∀ i ≤ n. f i ≤ 1
  shows (∑ i ≤ n. f i) ≤ Suc n
  using assms
  by (induct n) (auto)

lemma setsum_eq_one_le:
  fixes n::nat
  assumes ∀ i ≤ n. f i = 1
  shows (∑ i ≤ n. f i) = Suc n
  using assms
  by (induct n) (auto)

lemma setsum_least_eq:
  fixes f::nat ⇒ nat
  assumes h0: p ≤ n
  assumes h1: ∀ i ∈ {..<p}. f i = 1
  assumes h2: ∀ i ∈ {p..n}. f i = 0
  shows (∑ i ≤ n. f i) = p
proof -
  have eq_p: (∑ i ∈ {..<p}. f i) = p
    using h1 by (induct p) (simp_all)
  have eq_zero: (∑ i ∈ {p..n}. f i) = 0
    using h2 by auto
```

```
  have (∑ i ≤ n. f i) = (∑ i ∈ {..<p}. f i) + (∑ i ∈ {p..n}. f i)
    using h0 by (simp add: setsum_add_nat_ivl2)
  also have ... = (∑ i ∈ {..<p}. f i) using eq_zero by simp
  finally show (∑ i ≤ n. f i) = p using eq_p by simp
qed

lemma nat_mult_le_one:
  fixes m n::nat
  assumes m ≤ 1 n ≤ 1
  shows m ∗ n ≤ 1
  using assms by (induct n) (auto)

lemma setprod_one_le:
  fixes f::nat ⇒ nat
  assumes ∀ i ≤ n. f i ≤ 1
  shows (∏ i ≤ n. f i) ≤ 1
  using assms
  by (induct n) (auto intro: nat_mult_le_one)

lemma setprod_greater_zero:
  fixes f::nat ⇒ nat
  assumes ∀ i ≤ n. f i ≥ 0
  shows (∏ i ≤ n. f i) ≥ 0
  using assms by (induct n) (auto)

lemma setprod_eq_one:
  fixes f::nat ⇒ nat
  assumes ∀ i ≤ n. f i = Suc 0
  shows (∏ i ≤ n. f i) = Suc 0
  using assms by (induct n) (auto)

lemma setsum_cut_off_less:
  fixes f::nat ⇒ nat
  assumes h1: m ≤ n
    and    h2: ∀ i ∈ {m..<n}. f i = 0
  shows (∑ i < n. f i) = (∑ i < m. f i)
proof −
  have eq_zero: (∑ i ∈ {m..<n}. f i) = 0
    using h2 by auto
  have (∑ i < n. f i) = (∑ i ∈ {..<m}. f i) + (∑ i ∈ {m..<n}. f i)
    using h1 by (metis atLeast0LessThan le0 sum_add_nat_ivl)
  also have ... = (∑ i ∈ {..<m}. f i) using eq_zero by simp
  finally show (∑ i < n. f i) = (∑ i < m. f i) by simp
qed

lemma setsum_cut_off_le:
  fixes f::nat ⇒ nat
  assumes h1: m ≤ n
    and    h2: ∀ i ∈ {m..n}. f i = 0
  shows (∑ i ≤ n. f i) = (∑ i < m. f i)
```

proof −
  have *eq_zero*: $(\sum i \in \{m..n\}.\ f\ i) = 0$
    using *h2* **by** *auto*
  have $(\sum i \le n.\ f\ i) = (\sum i \in \{..<m\}.\ f\ i) + (\sum i \in \{m..n\}.\ f\ i)$
    using *h1* **by** (*simp add*: *setsum_add_nat_ivl2*)
  also have ... $= (\sum i \in \{..<m\}.\ f\ i)$ **using** *eq_zero* **by** *simp*
  finally show $(\sum i \le n.\ f\ i) = (\sum i < m.\ f\ i)$ **by** *simp*
qed

lemma *setprod_one* [*simp*]:
  fixes *n*::*nat*
  shows $(\prod i < n.\ Suc\ 0) = Suc\ 0$
    $(\prod i \le n.\ Suc\ 0) = Suc\ 0$
  by (*induct n*) (*simp_all*)

14 Recursive Functions

datatype *recf* = *Z*
  | *S*
  | *Id nat nat*
  | *Cn nat recf recf list*
  | *Pr nat recf recf*
  | *Mn nat recf*

fun *arity* :: *recf* ⇒ *nat*
  where
    arity Z = *1*
  | *arity S* = *1*
  | *arity* (*Id m n*) = *m*
  | *arity* (*Cn n f gs*) = *n*
  | *arity* (*Pr n f g*) = *Suc n*
  | *arity* (*Mn n f*) = *n*

Abbreviations for calculating the arity of the constructors

abbreviation
  CN f gs $\stackrel{def}{=}$ *Cn* (*arity* (*hd gs*)) *f gs*

abbreviation
  PR f g $\stackrel{def}{=}$ *Pr* (*arity f*) *f g*

abbreviation
  MN f $\stackrel{def}{=}$ *Mn* (*arity f* − *1*) *f*

the evaluation function and termination relation

fun *rec_eval* :: *recf* ⇒ *nat list* ⇒ *nat*
  where
    rec_eval Z xs = *0*
  | *rec_eval S xs* = *Suc* (*xs* ! *0*)

| *rec_eval* $(Id\ m\ n)\ xs = xs\ !\ n$
| *rec_eval* $(Cn\ n\ f\ gs)\ xs =$ *rec_eval* $f\ (map\ (\lambda x.\ rec\_eval\ x\ xs)\ gs)$
| *rec_eval* $(Pr\ n\ f\ g)\ (0\ \#\ xs) =$ *rec_eval* $f\ xs$
| *rec_eval* $(Pr\ n\ f\ g)\ (Suc\ x\ \#\ xs) =$
rec_eval $g\ (x\ \#\ (rec\_eval\ (Pr\ n\ f\ g)\ (x\ \#\ xs))\ \#\ xs)$
| *rec_eval* $(Mn\ n\ f)\ xs = (LEAST\ x.\ rec\_eval\ f\ (x\ \#\ xs) = 0)$

inductive
terminates $:: recf \Rightarrow nat\ list \Rightarrow bool$
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*: $[\![terminates\ f\ (map\ (\lambda g.\ rec\_eval\ g\ xs)\ gs);$
$\forall g \in set\ gs.\ terminates\ g\ xs;\ length\ xs = n]\!] \Longrightarrow terminates\ (Cn\ n\ f\ gs)\ xs$
| *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*: $[\![length\ xs = n;\ terminates\ f\ (r\ \#\ xs);$
$rec\_eval\ f\ (r\ \#\ xs) = 0;$
$\forall\ i < r.\ terminates\ f\ (i\ \#\ xs) \wedge rec\_eval\ f\ (i\ \#\ xs) > 0]\!] \Longrightarrow terminates\ (Mn\ n\ f)\ xs$

15 Arithmetic Functions

constn n is the recursive function which computes natural number n.

fun *constn* $:: nat \Rightarrow recf$
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*
by (*induct n*) (*simp_all*)

lemma *swap_lemma* [*simp*]:
rec_eval (*rec_swap f*) [*x*, *y*] = *rec_eval f* [*y*, *x*]
by (*simp add*: *rec_swap_def*)

lemma *add_lemma* [*simp*]:
rec_eval rec_add [*x*, *y*] = *x* + *y*
by (*induct x*) (*simp_all add*: *rec_add_def*)

lemma *mult_lemma* [*simp*]:
rec_eval rec_mult [*x*, *y*] = *x* ∗ *y*
by (*induct x*) (*simp_all add*: *rec_mult_def*)

lemma *power_lemma* [*simp*]:
rec_eval rec_power [*x*, *y*] = *x* ^ *y*
by (*induct y*) (*simp_all add*: *rec_power_def*)

lemma *fact_aux_lemma* [*simp*]:
rec_eval rec_fact_aux [*x*, *y*] = *fact x*
by (*induct x*) (*simp_all add*: *rec_fact_aux_def*)

lemma *fact_lemma* [*simp*]:
rec_eval rec_fact [*x*] = *fact x*
by (*simp add*: *rec_fact_def*)

lemma *pred_lemma* [*simp*]:
rec_eval rec_predecessor [*x*] = *x* − *1*
by (*induct x*) (*simp_all add*: *rec_predecessor_def*)

lemma *minus_lemma* [*simp*]:
rec_eval rec_minus [*x*, *y*] = *x* − *y*
by (*induct y*) (*simp_all add*: *rec_minus_def*)

16 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*)
by (*simp add*: *rec_sign_def*)

lemma *not_lemma* [*simp*]:
rec_eval rec_not [*x*] = (*if* $x = 0$ *then 1 else 0*)
by (*simp add*: *rec_not_def*)

lemma *eq_lemma* [*simp*]:
rec_eval rec_eq [*x*, *y*] = (*if* $x = y$ *then 1 else 0*)
by (*simp add*: *rec_eq_def*)

lemma *noteq_lemma* [*simp*]:
rec_eval rec_noteq [*x*, *y*] = (*if* $x \neq y$ *then 1 else 0*)
by (*simp add*: *rec_noteq_def*)

lemma *conj_lemma* [*simp*]:
rec_eval rec_conj [*x*, *y*] = (*if* $x = 0 \vee y = 0$ *then 0 else 1*)
by (*simp add*: *rec_conj_def*)

lemma *disj_lemma* [*simp*]:

rec_eval rec_disj $[x, y] = ($*if* $x = 0 \wedge y = 0$ *then 0 else 1*$)$
by (*simp add*: *rec_disj_def*)

lemma *imp_lemma* [*simp*]:
rec_eval rec_imp $[x, y] = ($*if* $0 < x \wedge y = 0$ *then 0 else 1*$)$
by (*simp add*: *rec_imp_def*)

lemma *ifz_lemma* [*simp*]:
rec_eval rec_ifz $[z, x, y] = ($*if* $z = 0$ *then x else y*$)$
by (*cases z*) (*simp_all add*: *rec_ifz_def*)

lemma *if_lemma* [*simp*]:
rec_eval rec_if $[z, x, y] = ($*if* $0 < z$ *then x else y*$)$
by (*simp add*: *rec_if_def*)

17 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*$)$
by (*simp add*: *rec_less_def*)

lemma *le_lemma* [*simp*]:
rec_eval rec_le $[x, y] = ($*if* $(x \leq y)$ *then 1 else 0*$)$
by(*simp add*: *rec_le_def*)

18 Summation and Product Functions

definition
rec_sigma1 f = *PR* (*CN f* [*CN Z* [*Id 1 0*], *Id 1 0*])
    (*CN rec_add* [*Id 3 1*, *CN f* [*CN S* [*Id 3 0*], *Id 3 2*]])

definition
rec_sigma2 f = *PR* (*CN f* [*CN Z* [*Id 2 0*], *Id 2 0*, *Id 2 1*])
    (*CN rec_add* [*Id 4 1*, *CN f* [*CN S* [*Id 4 0*], *Id 4 2*, *Id 4 3*]])

definition
rec_accum1 f = *PR* (*CN f* [*CN Z* [*Id 1 0*], *Id 1 0*])
    (*CN rec_mult* [*Id 3 1*, *CN f* [*CN S* [*Id 3 0*], *Id 3 2*]])

definition

rec_accum2 f = *PR* (*CN f* [*CN Z* [*Id 2 0*], *Id 2 0*, *Id 2 1*])
(*CN rec_mult* [*Id 4 1*, *CN f* [*CN S* [*Id 4 0*], *Id 4 2*, *Id 4 3*]]])

definition
rec_accum3 f = *PR* (*CN f* [*CN Z* [*Id 3 0*], *Id 3 0*, *Id 3 1*, *Id 3 2*])
(*CN rec_mult* [*Id 5 1*, *CN f* [*CN S* [*Id 5 0*], *Id 5 2*, *Id 5 3*, *Id 5 4*]]])

lemma *sigma1_lemma* [*simp*]:
shows *rec_eval* (*rec_sigma1 f*) [*x*, *y*] = ($\sum$ *z* ≤ *x*. *rec_eval f* [*z*, *y*])
by (*induct x*) (*simp_all add*: *rec_sigma1_def*)

lemma *sigma2_lemma* [*simp*]:
shows *rec_eval* (*rec_sigma2 f*) [*x*, *y1*, *y2*] = ($\sum$ *z* ≤ *x*. *rec_eval f* [*z*, *y1*, *y2*])
by (*induct x*) (*simp_all add*: *rec_sigma2_def*)

lemma *accum1_lemma* [*simp*]:
shows *rec_eval* (*rec_accum1 f*) [*x*, *y*] = ($\prod$ *z* ≤ *x*. *rec_eval f* [*z*, *y*])
by (*induct x*) (*simp_all add*: *rec_accum1_def*)

lemma *accum2_lemma* [*simp*]:
shows *rec_eval* (*rec_accum2 f*) [*x*, *y1*, *y2*] = ($\prod$ *z* ≤ *x*. *rec_eval f* [*z*, *y1*, *y2*])
by (*induct x*) (*simp_all add*: *rec_accum2_def*)

lemma *accum3_lemma* [*simp*]:
shows *rec_eval* (*rec_accum3 f*) [*x*, *y1*, *y2*, *y3*] = ($\prod$ *z* ≤ *x*. (*rec_eval f*) [*z*, *y1*, *y2*, *y3*])
by (*induct x*) (*simp_all add*: *rec_accum3_def*)

19 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* ($\exists z \leq x$. *0* < *rec_eval f* [*z*, *y*]) *then 1 else 0*)
  by (*simp add*: *rec_ex1_def*)

lemma *ex2_lemma* [*simp*]:
  rec_eval (*rec_ex2 f*) [*x*, *y1*, *y2*] = (*if* ($\exists z \leq x$. *0* < *rec_eval f* [*z*, *y1*, *y2*]) *then 1 else 0*)
  by (*simp add*: *rec_ex2_def*)

lemma *all1_lemma* [*simp*]:
  rec_eval (*rec_all1 f*) [*x*, *y*] = (*if* ($\forall z \leq x$. *0* < *rec_eval f* [*z*, *y*]) *then 1 else 0*)
  by (*simp add*: *rec_all1_def*)

lemma *all2_lemma* [*simp*]:
  rec_eval (*rec_all2 f*) [*x*, *y1*, *y2*] = (*if* ($\forall z \leq x$. *0* < *rec_eval f* [*z*, *y1*, *y2*]) *then 1 else 0*)
  by (*simp add*: *rec_all2_def*)

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*)
  by (*simp add*: *rec_all3_def*)

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*)
  apply(*auto simp add*: *Let_def rec_all1_less_def*)
   apply (*metis nat_less_le*)+
  done

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*)
  apply(*auto simp add*: *Let_def rec_all2_less_def*)
   apply(*metis nat_less_le*)+
  done

20 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 $Quo\ 0\ y = 0$
by (*induct y*) (*auto*)

lemma *Quo1*:
$x * (Quo\ x\ y) \le y$
by (*induct y*) (*simp_all*)

lemma *Quo2*:
$b * (Quo\ b\ a) + a\ mod\ b = a$
by (*induct a*) (*auto simp add*: *mod_Suc*)

lemma *Quo3*:
$n * (Quo\ n\ m) = m - m\ mod\ n$
using *Quo2*[*of n m*] **by** (*auto*)

lemma *Quo4*:
assumes *h*: $0 < x$
shows $y < x + x * Quo\ x\ y$
proof −
have $x - (y\ mod\ x) > 0$ **using** *mod_less_divisor assms* **by** *auto*
then have $y < y + (x - (y\ mod\ x))$ **by** *simp*
then have $y < x + (y - (y\ mod\ x))$ **by** *simp*
then show $y < x + x * (Quo\ x\ y)$ **by** (*simp add*: *Quo3*)
qed

lemma *Quo_div*:
shows $Quo\ x\ y = y\ div\ x$
by (*metis Quo0 Quo1 Quo4 div_by_0 div_nat_eqI mult_Suc_right neq0_conv*)

lemma *Quo_rec_quo*:
shows *rec_eval rec_quo* $[y, x] = Quo\ x\ y$
by (*induct y*) (*simp_all add*: *rec_quo_def*)

lemma *quo_lemma* [*simp*]:
shows *rec_eval rec_quo* $[y, x] = y\ div\ x$
by (*simp add*: *Quo_div Quo_rec_quo*)

21 Iteration

definition
rec_iter f $= PR\ (Id\ 1\ 0)\ (CN\ f\ [Id\ 3\ 1])$

fun *Iter* **where**
$Iter\ f\ 0 = id$
| $Iter\ f\ (Suc\ n) = f \circ (Iter\ f\ n)$

lemma *Iter_comm*:
 $(Iter\ f\ n)\ (f\ x) = f\ ((Iter\ f\ n)\ x)$
 by (*induct n*) (*simp_all*)

lemma *iter_lemma* [*simp*]:
 $rec\_eval\ (rec\_iter\ f)\ [n, x] = Iter\ (\lambda x.\ rec\_eval\ f\ [x])\ n\ x$
 by (*induct n*) (*simp_all add*: *rec_iter_def*)

22 Bounded Maximisation

fun *BMax_rec* **where**
 $BMax\_rec\ R\ 0 = 0$
$|\ BMax\_rec\ R\ (Suc\ n) = (if\ R\ (Suc\ n)\ then\ (Suc\ n)\ else\ BMax\_rec\ R\ n)$

definition
 $BMax\_set :: (nat \Rightarrow bool) \Rightarrow nat \Rightarrow nat$
 where
  $BMax\_set\ R\ x = Max\ (\{z.\ z \leq x \wedge R\ z\} \cup \{0\})$

lemma *BMax_rec_eq1*:
 $BMax\_rec\ R\ x = (GREATEST\ z.\ (R\ z \wedge z \leq x) \vee z = 0)$
 apply(*induct x*)
  apply(*auto intro*: *Greatest_equality Greatest_equality*[*symmetric*])
 apply(*simp add*: *le_Suc_eq*)
 by *metis*

lemma *BMax_rec_eq2*:
 $BMax\_rec\ R\ x = Max\ (\{z.\ z \leq x \wedge R\ z\} \cup \{0\})$
 apply(*induct x*)
  apply(*auto intro*: *Max_eqI Max_eqI*[*symmetric*])
 apply(*simp add*: *le_Suc_eq*)
 by *metis*

lemma *BMax_rec_eq3*:
 $BMax\_rec\ R\ x = Max\ (Set.filter\ (\lambda z.\ R\ z)\ \{..x\} \cup \{0\})$
 by (*simp add*: *BMax_rec_eq2 Set.filter_def*)

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*]:
 $rec\_eval\ (rec\_max1\ f)\ [x, y] = BMax\_rec\ (\lambda u.\ rec\_eval\ f\ [u, y] = 0)\ x$
 by (*induct x*) (*simp_all add*: *rec_max1_def*)

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* (λ*u*. *rec_eval f* [*u*, *y1*, *y2*] = *0*) *x*
by (*induct x*) (*simp_all add*: *rec_max2_def*)

23 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*)
by (*subst* (*3*) *prod_decode_inverse*[*symmetric*])
(*simp add*: *prod_encode_def prod_decode_def split*: *prod.split*)

lemma *snd_prod_decode*:
pdec2 z = *Max_triangle z* − *pdec1 z*
by (*simp only*: *prod_decode_def*)

lemma *le_triangle*:
m ≤ *triangle* (*n* + *m*)
by (*induct m*) (*simp_all*)

lemma *Max_triangle_triangle_le*:
triangle (*Max_triangle z*) ≤ *z*
by (*subst* (*9*) *prod_decode_inverse*[*symmetric*])
(*simp add*: *prod_decode_def prod_encode_def split*: *prod.split*)

lemma *Max_triangle_le*:
Max_triangle z ≤ *z*
proof −
have *Max_triangle z* ≤ *triangle* (*Max_triangle z*)
using *le_triangle*[*of _ 0*, *simplified*] **by** *simp*
also have ... ≤ *z* **by** (*rule Max_triangle_triangle_le*)

```
  finally show Max_triangle z ≤ z .
qed

lemma w_aux:
  Max_triangle (triangle k + m) = Max_triangle_aux k m
  by (simp add: prod_decode_def[symmetric] prod_decode_triangle_add)

lemma y_aux: y ≤ Max_triangle_aux y k
  apply(induct k arbitrary: y rule: nat_less_induct)
  apply(subst (1 2) prod_decode_aux.simps)
  by(auto dest!:spec mp elim:Suc_leD)

lemma Max_triangle_greatest:
  Max_triangle z = (GREATEST k. (triangle k ≤ z ∧ k ≤ z) ∨ k = 0)
  apply(rule Greatest_equality[symmetric])
   apply(rule disjI1)
   apply(rule conjI)
    apply(rule Max_triangle_triangle_le)
   apply(rule Max_triangle_le)
  apply(erule disjE)
   apply(erule conjE)
   apply(subst (asm) (1) le_iff_add)
   apply(erule exE)
   apply(clarify)
   apply(simp only: w_aux)
   apply(rule y_aux)
  apply(simp)
  done


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
  by (simp add: rec_triangle_def triangle_def)

lemma max_triangle_lemma [simp]:
  rec_eval rec_max_triangle [x] = Max_triangle x
  by (simp add: Max_triangle_greatest rec_max_triangle_def Let_def BMax_rec_eq1)
```

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*
by (*simp add*: *rec_pdec1_def fst_prod_decode*)

lemma *pdec2_lemma* [*simp*]:
rec_eval rec_pdec2 [*z*] = *pdec2 z*
by (*simp add*: *rec_pdec2_def snd_prod_decode*)

lemma *penc_lemma* [*simp*]:
rec_eval rec_penc [*m*, *n*] = *penc m n*
by (*simp add*: *rec_penc_def prod_encode_def*)

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*
by (*simp_all add*: *prod_decode_def prod_decode_aux.simps*)

lemma *ldec_zero*:
ldec 0 n = *0*
by (*induct n*) (*simp_all add*: *prod_decode_def prod_decode_aux.simps*)

lemma *list_encode_inverse*:
ldec (*lenc xs*) *n* = (*if n* < *length xs then xs* ! *n else 0*)
by (*induct xs arbitrary*: *n rule*: *lenc.induct*)
(*auto simp add*: *ldec_zero nth_Cons split*: *nat.splits*)

lemma *lenc_length_le*:
length xs ≤ *lenc xs*
by (*induct xs*) (*simp_all add*: *prod_encode_def*)

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 (λx. inside z (x − 1)) z

lemma inside_False [simp]:
  inside 0 n = False
  by (induct n) (simp_all)

lemma inside_length [simp]:
  inside (lenc xs) s = (s < length xs)
proof(induct s arbitrary: xs)
  case 0
  then show ?case by (cases xs) (simp_all add: prod_encode_def)
next
  case (Suc s)
  then show ?case by (cases xs;auto)
qed
```

Length of Encoded Lists

```
lemma enclen_length [simp]:
  enclen (lenc xs) = length xs
  unfolding enclen_def
  apply(simp add: BMax_rec_eq1)
  apply(rule Greatest_equality)
   apply(auto simp add: lenc_length_le)
  done

lemma enclen_penc [simp]:
  enclen (penc (Suc x) (lenc xs)) = Suc (enclen (lenc xs))
  by (simp only: lenc.simps[symmetric] enclen_length) (simp)

lemma enclen_zero [simp]:
  enclen 0 = 0
  by (simp add: enclen_def)
```

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*
by (*induct n arbitrary*: *z*) (*simp* | *subst Iter_comm*)+

lemma *inside_iter*:
inside z n = (*0* < *Iter pdec2 n z*)
by (*induct n arbitrary*: *z*) (*simp* | *subst Iter_comm*)+

lemma *lenc_lemma* [*simp*]:
rec_eval (*rec_lenc fs*) *xs* = *lenc* (*map* (λ*f*. *rec_eval f xs*) *fs*)
by (*induct fs*) (*simp_all*)

lemma *ldec_lemma* [*simp*]:
rec_eval rec_ldec [*z*, *n*] = *ldec z n*
by (*simp add*: *ldec_iter rec_ldec_def*)

lemma *inside_lemma* [*simp*]:
rec_eval rec_inside [*z*, *n*] = (*if inside z n then 1 else 0*)
by (*simp add*: *inside_iter rec_inside_def*)

lemma *enclen_lemma* [*simp*]:
rec_eval rec_enclen [*z*] = *enclen z*
by (*simp add*: *rec_enclen_def enclen_def*)

end

24 Construction of a Universal Function

theory *UF*
imports *Rec_Def HOL.GCD Abacus*
begin

This theory file constructs the Universal Function *rec_F*, which is the UTM defined in terms of recursive functions. This *rec_F* is essentially an interpreter of Turing Machines. Once the correctness of *rec_F* is established, UTM can easil be obtained by compling *rec_F* into the corresponding Turing Machine.

25 Universal Function

25.1 The construction of component functions

The recursive function used to do arithmetic addition.

definition *rec_add* :: *recf*
 where
  $rec\_add \stackrel{def}{=} Pr\ 1\ (id\ 1\ 0)\ (Cn\ 3\ s\ [id\ 3\ 2])$

The recursive function used to do arithmetic multiplication.

definition *rec_mult* :: *recf*
 where
  $rec\_mult = Pr\ 1\ z\ (Cn\ 3\ rec\_add\ [id\ 3\ 0,\ id\ 3\ 2])$

The recursive function used to do arithmetic precede.

definition *rec_pred* :: *recf*
 where
  $rec\_pred = Cn\ 1\ (Pr\ 1\ z\ (id\ 3\ 1))\ [id\ 1\ 0,\ id\ 1\ 0]$

The recursive function used to do arithmetic subtraction.

definition *rec_minus* :: *recf*
 where
  $rec\_minus = Pr\ 1\ (id\ 1\ 0)\ (Cn\ 3\ rec\_pred\ [id\ 3\ 2])$

constn n is the recursive function which computes nature number *n*.

fun *constn* :: *nat* ⇒ *recf*
 where
  $constn\ 0 = z\ \ |$
  $constn\ (Suc\ n) = Cn\ 1\ s\ [constn\ n]$

Sign function, which returns 1 when the input argument is greater than *0*.

definition *rec_sg* :: *recf*
 where
  $rec\_sg = Cn\ 1\ rec\_minus\ [constn\ 1,$
    $Cn\ 1\ rec\_minus\ [constn\ 1,\ id\ 1\ 0]]$

rec_less compares its two arguments, returns *1* if the first is less than the second; otherwise returns *0*.

definition *rec_less* :: *recf*
 where
  $rec\_less = Cn\ 2\ rec\_sg\ [Cn\ 2\ rec\_minus\ [id\ 2\ 1,\ id\ 2\ 0]]$

rec_not inverse its argument: returns *1* when the argument is *0*; returns *0* otherwise.

definition *rec_not* :: *recf*
 where
  $rec\_not = Cn\ 1\ rec\_minus\ [constn\ 1,\ id\ 1\ 0]$

rec_eq compares its two arguments: returns *1* if they are equal; return *0* otherwise.

definition *rec_eq* :: *recf*
where
$rec\_eq = Cn\ 2\ rec\_minus\ [Cn\ 2\ (constn\ 1)\ [id\ 2\ 0],$
$\quad Cn\ 2\ rec\_add\ [Cn\ 2\ rec\_minus\ [id\ 2\ 0,\ id\ 2\ 1],$
$\quad\quad Cn\ 2\ rec\_minus\ [id\ 2\ 1,\ id\ 2\ 0]]]$

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* :: $recf \Rightarrow nat$
where
$arity\ z = 1$
$|\ arity\ s = 1$
$|\ arity\ (id\ m\ n) = m$
$|\ arity\ (Cn\ n\ f\ gs) = n$
$|\ arity\ (Pr\ n\ f\ g) = Suc\ n$
$|\ arity\ (Mn\ n\ f) = n$

get_fstn_args $n\ (Suc\ k)$ returns $[id\ n\ 0,\ id\ n\ 1,\ id\ n\ 2,\ \ldots,\ id\ n\ k]$, the effect of which is to take out the first *Suc k* arguments out of the *n* input arguments.

fun *get_fstn_args* :: $nat \Rightarrow nat \Rightarrow recf\ list$
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*:

$$(rec\_sigmaf)(x, y) = f(x, 0) + f(x, 1) + \cdots + f(x, y)$$

fun *rec_sigma* :: $recf \Rightarrow recf$
where
$rec\_sigma\ rf =$
$\quad (let\ vl = arity\ rf\ in$
$\quad\quad Pr\ (vl - 1)\ (Cn\ (vl - 1)\ rf\ (get\_fstn\_args\ (vl - 1)\ (vl - 1)\ @$
$\quad\quad\quad [Cn\ (vl - 1)\ (constn\ 0)\ [id\ (vl - 1)\ 0]]))$
$\quad\quad (Cn\ (Suc\ vl)\ rec\_add\ [id\ (Suc\ vl)\ vl,$
$\quad\quad\quad Cn\ (Suc\ vl)\ rf\ (get\_fstn\_args\ (Suc\ vl)\ (vl - 1)$
$\quad\quad\quad\quad @\ [Cn\ (Suc\ vl)\ s\ [id\ (Suc\ vl)\ (vl - 1)]]])))$

rec_exec is the interpreter function for reursive functions. The function is defined such that it always returns meaningful results for primitive recursive functions.

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

Correctness of *rec_add*.

lemma *add_lemma*: $\bigwedge x\ y.\ rec\_exec\ rec\_add\ [x, y] = x + y$
by(*induct_tac y*, *auto simp*: *rec_add_def rec_exec.simps*)

Correctness of *rec_mult*.

lemma *mult_lemma*: $\bigwedge x\ y.\ rec\_exec\ rec\_mult\ [x, y] = x * y$
by(*induct_tac y*, *auto simp*: *rec_mult_def rec_exec.simps add_lemma*)

Correctness of *rec_pred*.

lemma *pred_lemma*: $\bigwedge x.\ rec\_exec\ rec\_pred\ [x] = x - 1$
by(*induct_tac x*, *auto simp*: *rec_pred_def rec_exec.simps*)

Correctness of *rec_minus*.

lemma *minus_lemma*: $\bigwedge x\ y.\ rec\_exec\ rec\_minus\ [x, y] = x - y$
by(*induct_tac y*, *auto simp*: *rec_exec.simps rec_minus_def pred_lemma*)

Correctness of *rec_sg*.

lemma *sg_lemma*: $\bigwedge x.\ rec\_exec\ rec\_sg\ [x] = (\mathit{if}\ x = 0\ \mathit{then}\ 0\ \mathit{else}\ 1)$
by(*auto simp*: *rec_sg_def minus_lemma rec_exec.simps constn.simps*)

Correctness of *constn*.

lemma *constn_lemma*: $rec\_exec\ (constn\ n)\ [x] = n$
by(*induct n*, *auto simp*: *rec_exec.simps constn.simps*)

Correctness of *rec_less*.

lemma *less_lemma*: $\bigwedge x\ y.\ rec\_exec\ rec\_less\ [x, y] =$
$(\mathit{if}\ x < y\ \mathit{then}\ 1\ \mathit{else}\ 0)$
by(*induct_tac y*, *auto simp*: *rec_exec.simps*
rec_less_def minus_lemma sg_lemma)

Correctness of *rec_not*.

lemma *not_lemma*:
$\bigwedge x.\ rec\_exec\ rec\_not\ [x] = (\mathit{if}\ x = 0\ \mathit{then}\ 1\ \mathit{else}\ 0)$
by(*induct_tac x*, *auto simp*: *rec_exec.simps rec_not_def*
constn_lemma minus_lemma)

Correctness of *rec_eq*.

lemma *eq_lemma*: $\bigwedge x\ y.\ rec\_exec\ rec\_eq\ [x, y] = (\mathit{if}\ x = y\ \mathit{then}\ 1\ \mathit{else}\ 0)$
by(*induct_tac y*, *auto simp*: *rec_exec.simps rec_eq_def constn_lemma add_lemma minus_lemma*)

Correctness of *rec_conj*.

lemma *conj_lemma*: $\bigwedge x\ y.\ rec\_exec\ rec\_conj\ [x, y] = (\mathit{if}\ x = 0 \lor y = 0\ \mathit{then}\ 0$
$\mathit{else}\ 1)$
by(*induct_tac y*, *auto simp*: *rec_exec.simps sg_lemma rec_conj_def mult_lemma*)

Correctness of *rec_disj*.

```
lemma disj_lemma: ⋀ x y. rec_exec rec_disj [x, y] = (if x = 0 ∧ y = 0 then 0
                                                      else 1)
 by(induct_tac y, auto simp: rec_disj_def sg_lemma add_lemma rec_exec.simps)
```

primrec recf n is true iff *recf* is a primitive recursive function with arity *n*.

```
inductive primerec :: recf ⇒ nat ⇒ bool
 where
  prime_z[intro]:  primerec z (Suc 0) |
  prime_s[intro]:  primerec s (Suc 0) |
  prime_id[intro!]: ⟦n < m⟧ ⟹ primerec (id m n) m |
  prime_cn[intro!]: ⟦primerec f k; length gs = k;
 ∀ i < length gs. primerec (gs ! i) m; m = n⟧
 ⟹ primerec (Cn n f gs) m |
  prime_pr[intro!]: ⟦primerec f n;
 primerec g (Suc (Suc n)); m = Suc n⟧
 ⟹ primerec (Pr n f g) m

inductive-cases prime_cn_reverse'[elim]: primerec (Cn n f gs) n
inductive-cases prime_mn_reverse: primerec (Mn n f) m
inductive-cases prime_z_reverse[elim]: primerec z n
inductive-cases prime_s_reverse[elim]: primerec s n
inductive-cases prime_id_reverse[elim]: primerec (id m n) k
inductive-cases prime_cn_reverse[elim]: primerec (Cn n f gs) m
inductive-cases prime_pr_reverse[elim]: primerec (Pr n f g) m

declare mult_lemma[simp] add_lemma[simp] pred_lemma[simp]
 minus_lemma[simp] sg_lemma[simp] constn_lemma[simp]
 less_lemma[simp] not_lemma[simp] eq_lemma[simp]
 conj_lemma[simp] disj_lemma[simp]
```

Sigma is the logical specification of the recursive function *rec_sigma*.

```
function Sigma :: (nat list ⇒ nat) ⇒ nat list ⇒ nat
 where
  Sigma g xs = (if last xs = 0 then g xs
            else (Sigma g (butlast xs @ [last xs − 1]) +
               g xs))
 by pat_completeness auto
termination
proof
 show wf (measure (λ (f, xs). last xs)) by auto
next
 fix g xs
 assume last (xs::nat list) ≠ 0
 thus ((g, butlast xs @ [last xs − 1]), g, xs)
             ∈ measure (λ(f, xs). last xs)
  by auto
qed
```

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*])])
by(*simp add*: *rec_exec.simps*)

lemma *Sigma_0_simp_rewrite*:
Sigma f (*xs* @ [*0*]) = *f* (*xs* @ [*0*])
by(*simp add*: *Sigma.simps*)

lemma *Sigma_Suc_simp_rewrite*:
Sigma f (*xs* @ [*Suc x*]) = *Sigma f* (*xs* @ [*x*]) + *f* (*xs* @ [*Suc x*])
by(*simp add*: *Sigma.simps*)

lemma *append_access_1*[*simp*]: (*xs* @ *ys*) ! (*Suc* (*length xs*)) = *ys* ! *1*
by(*simp add*: *nth_append*)

lemma *get_fstn_args_take*: $[\![$*length xs* = *m*; *n* $\leq$ *m*$]\!] \Longrightarrow$
map (λ *f*. *rec_exec f xs*) (*get_fstn_args m n*)= *take n xs*
proof(*induct n*)
case *0* **thus** *?case*
by(*simp add*: *get_fstn_args.simps*)
next
case (*Suc n*) **thus** *?case*
by(*simp add*: *get_fstn_args.simps rec_exec.simps*
take_Suc_conv_app_nth)
qed

lemma *arity_primerec*[*simp*]: *primerec f n* $\Longrightarrow$ *arity f* = *n*
apply(*cases f*)
apply(*auto simp*: *arity.simps*)
apply(*erule_tac prime_mn_reverse*)
done

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*])
apply(*induct x*)
apply(*auto simp*: *rec_sigma.simps Let_def rec_pr_Suc_simp_rewrite*
rec_exec.simps get_fstn_args_take)
done

The correctness of *rec_sigma* with respect to its specification.

lemma *sigma_lemma*:
primerec rg (*Suc* (*length xs*))

```
  ⟹ rec_exec (rec_sigma rg) (xs @ [x]) = Sigma (rec_exec rg) (xs @ [x])
 apply(induct x)
  apply(auto simp: rec_exec.simps rec_sigma.simps Let_def
    get_fstn_args_take Sigma_0_simp_rewrite
    Sigma_Suc_simp_rewrite)
 done
```

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))
 by pat_completeness auto
termination
proof
 show wf (measure (λ (f, xs). last xs))
  by auto
next
 fix f xs
 assume last xs ≠ (0::nat)
 thus ((f, butlast xs @ [last xs − 1]), f, xs) ∈
        measure (λ(f, xs). last xs)
  by auto
qed

lemma rec_accum_Suc_simp_rewrite:
 primerec f (Suc (length xs))
  ⟹ rec_exec (rec_accum f) (xs @ [Suc x]) =
  rec_exec (rec_accum f) (xs @ [x]) * rec_exec f (xs @ [Suc x])
 apply(induct x)
  apply(auto simp: rec_sigma.simps Let_def rec_pr_Suc_simp_rewrite
    rec_exec.simps get_fstn_args_take)
 done
```

The correctness of *rec_accum* with respect to its specification.

```
lemma accum_lemma :
```

```
primerec rg (Suc (length xs))
   ⟹ rec_exec (rec_accum rg) (xs @ [x]) = Accum (rec_exec rg) (xs @ [x])
 apply(induct x)
  apply(auto simp: rec_exec.simps rec_sigma.simps Let_def
    get_fstn_args_take)
 done

declare rec_accum.simps [simp del]
```

rec_all t f $(x1, x2, \ldots, xn)$ computes the charactrization function of the following FOL formula: $(\forall\ x \leq t(x1, x2, \ldots, xn).\ (f(x1, x2, \ldots, xn, x) > 0))$

```
fun rec_all :: recf ⇒ recf ⇒ recf
 where
  rec_all rt rf =
  (let vl = arity rf in
    Cn (vl − 1) rec_sg [Cn (vl − 1) (rec_accum rf)
           (get_fstn_args (vl − 1) (vl − 1) @ [rt])])

lemma rec_accum_ex:
 assumes primerec rf (Suc (length xs))
 shows (rec_exec (rec_accum rf) (xs @ [x]) = 0) =
       (∃ t ≤ x. rec_exec rf (xs @ [t]) = 0)
proof(induct x)
 case (Suc x)
 with assms show ?case
  apply(auto simp add: rec_exec.simps rec_accum.simps get_fstn_args_take)
  apply(rename_tac t ta)
  apply(rule_tac x = ta in exI, simp)
 apply(case_tac t = Suc x, simp_all)
 apply(rule_tac x = t in exI, simp) done
qed (insert assms,auto simp add: rec_exec.simps rec_accum.simps get_fstn_args_take)
```

The correctness of *rec_all*.

```
lemma all_lemma:
 ⟦primerec rf (Suc (length xs));
  primerec rt (length xs)⟧
 ⟹ rec_exec (rec_all rt rf) xs = (if (∀ x ≤ (rec_exec rt xs). 0 < rec_exec rf (xs @ [x])) then 1
                                                                                    else 0)
 apply(auto simp: rec_all.simps)
  apply(simp add: rec_exec.simps map_append get_fstn_args_take split: if_splits)
  apply(drule_tac x = rec_exec rt xs in rec_accum_ex)
  apply(cases rec_exec (rec_accum rf) (xs @ [rec_exec rt xs]) = 0, simp_all)
  apply force
 apply(simp add: rec_exec.simps map_append get_fstn_args_take)
 apply(drule_tac x = rec_exec rt xs in rec_accum_ex)
 apply(cases rec_exec (rec_accum rf) (xs @ [rec_exec rt xs]) = 0)
 apply force+
 done
```

rec_ex t f $(x1, x2, \ldots, xn)$ computes the charactrization function of the following

FOL formula: $(\exists\ x \leq t(x1, x2, \ldots, xn).\ (f(x1, x2, \ldots, xn, x) > 0))$

```
fun rec_ex :: recf ⇒ recf ⇒ recf
 where
  rec_ex rt rf =
    (let vl = arity rf in
      Cn (vl − 1) rec_sg [Cn (vl − 1) (rec_sigma rf)
            (get_fstn_args (vl − 1) (vl − 1) @ [rt])])

lemma rec_sigma_ex:
 assumes primerec rf (Suc (length xs))
 shows (rec_exec (rec_sigma rf) (xs @ [x]) = 0) =
                    (∀ t ≤ x. rec_exec rf (xs @ [t]) = 0)
proof(induct x)
 case (Suc x)
 from Suc assms show ?case
  by(auto simp add: rec_exec.simps rec_sigma.simps
      get_fstn_args_take elim:le_SucE)
qed (insert assms,auto simp: get_fstn_args_take rec_exec.simps rec_sigma.simps)
```

The correctness of *ex_lemma*.

```
lemma ex_lemma:
 ⟦primerec rf (Suc (length xs));
  primerec rt (length xs)⟧
⟹ (rec_exec (rec_ex rt rf) xs =
   (if (∃ x ≤ (rec_exec rt xs). 0 <rec_exec rf (xs @ [x])) then 1
    else 0))
 apply(auto simp: rec_exec.simps get_fstn_args_take split: if_splits)
  apply(drule_tac x = rec_exec rt xs in rec_sigma_ex, simp)
 apply(drule_tac x = rec_exec rt xs in rec_sigma_ex, simp)
 done
```

Definition of *Min*[*R*] on page 77 of Boolos's book.

```
fun Minr :: (nat list ⇒ bool) ⇒ nat list ⇒ nat ⇒ nat
 where Minr Rr xs w = (let setx = {y | y. (y ≤ w) ∧ Rr (xs @ [y])} in
                  if (setx = {}) then (Suc w)
                            else (Min setx))

declare Minr.simps[simp del] rec_all.simps[simp del]
```

The following is a set of auxilliary lemmas about *Minr*.

```
lemma Minr_range: Minr Rr xs w ≤ w ∨ Minr Rr xs w = Suc w
 apply(auto simp: Minr.simps)
 apply(subgoal_tac Min {x. x ≤ w ∧ Rr (xs @ [x])} ≤ x)
  apply(erule_tac order_trans, simp)
 apply(rule_tac Min_le, auto)
 done

lemma expand_conj_in_set: {x. x ≤ Suc w ∧ Rr (xs @ [x])}
  = (if Rr (xs @ [Suc w]) then insert (Suc w)
```

```
                    {x. x ≤ w ∧ Rr (xs @ [x])}
    else {x. x ≤ w ∧ Rr (xs @ [x])})
  by (auto elim:le_SucE)

lemma Minr_strip_Suc[simp]: Minr Rr xs w ≤ w ⟹ Minr Rr xs (Suc w) = Minr Rr xs w
  by(cases ∀ x≤w. ¬ Rr (xs @ [x]),auto simp add: Minr.simps expand_conj_in_set)

lemma x_empty_set[simp]: ∀ x≤w. ¬ Rr (xs @ [x]) ⟹
                    {x. x ≤ w ∧ Rr (xs @ [x])} = {}
  by auto

lemma Minr_is_Suc[simp]: ⟦Minr Rr xs w = Suc w; Rr (xs @ [Suc w])⟧ ⟹
                        Minr Rr xs (Suc w) = Suc w
  apply(simp add: Minr.simps expand_conj_in_set)
  apply(cases ∀ x≤w. ¬ Rr (xs @ [x]), auto)
  done

lemma Minr_is_Suc_Suc[simp]: ⟦Minr Rr xs w = Suc w; ¬ Rr (xs @ [Suc w])⟧ ⟹
                        Minr Rr xs (Suc w) = Suc (Suc w)
  apply(simp add: Minr.simps expand_conj_in_set)
  apply(cases ∀ x≤w. ¬ Rr (xs @ [x]), auto)
  apply(subgoal_tac Min {x. x ≤ w ∧ Rr (xs @ [x])} ∈
                    {x. x ≤ w ∧ Rr (xs @ [x])}, simp)
  apply(rule_tac Min_in, auto)
  done

lemma Minr_Suc_simp:
  Minr Rr xs (Suc w) =
    (if Minr Rr xs w ≤ w then Minr Rr xs w
     else if (Rr (xs @ [Suc w])) then (Suc w)
     else Suc (Suc w))
  by(insert Minr_range[of Rr xs w], auto)
```

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
  by(induct n, auto simp: get_fstn_args.simps)

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*))
 apply(*simp*)
 done

lemma *primerec_accum*: *primerec* (*rec_accum rf*) *n* $\Longrightarrow$ *primerec rf n*
 apply(*auto simp*: *rec_accum.simps Let_def*)
 apply(*erule_tac prime_pr_reverse*, *simp*)
 apply(*erule_tac prime_cn_reverse*, *simp only*: *length_app*)
 done

lemma *primerec_all*: *primerec* (*rec_all rt rf*) *n* $\Longrightarrow$
                primerec rt n $\wedge$ *primerec rf* (*Suc n*)
 apply(*simp add*: *rec_all.simps Let_def*)
 apply(*erule_tac prime_cn_reverse*, *simp*)
 apply(*erule_tac prime_cn_reverse*, *simp*)
 apply(*erule_tac x* = *n* **in** *allE*, *simp add*: *nth_append primerec_accum*)
 done

declare *numeral_3_eq_3*[*simp*]

lemma *primerec_rec_pred_1*[*intro*]: *primerec rec_pred* (*Suc 0*)
 apply(*simp add*: *rec_pred_def*)
 apply(*rule_tac prime_cn*, *auto dest*:*less_2_cases*[*unfolded numeral One_nat_def*])
 done

lemma *primerec_rec_minus_2*[*intro*]: *primerec rec_minus* (*Suc* (*Suc 0*))
 apply(*auto simp*: *rec_minus_def*)
 done

lemma *primerec_constn_1*[*intro*]: *primerec* (*constn n*) (*Suc 0*)
 apply(*induct n*)
  apply(*auto simp*: *constn.simps*)
 done

lemma *primerec_rec_sg_1*[*intro*]: *primerec rec_sg* (*Suc 0*)
 apply(*simp add*: *rec_sg_def*)
 apply(*rule_tac k* = *Suc* (*Suc 0*) **in** *prime_cn*)
   apply(*auto*)
 apply(*auto dest*!:*less_2_cases*[*unfolded numeral One_nat_def*])
 apply(*auto*)
 done

lemma *primerec_getpren*[*elim*]: $[\![i < n; n \le m]\!] \Longrightarrow$ *primerec* (*get_fstn_args m n* ! *i*) *m*
 apply(*induct n*, *auto simp*: *get_fstn_args.simps*)
 apply(*cases i* = *n*, *auto simp*: *nth_append intro*: *prime_id*)
 done

```
lemma primerec_rec_add_2[intro]: primerec rec_add (Suc (Suc 0))
 apply(simp add: rec_add_def)
 apply(rule_tac prime_pr, auto)
 done

lemma primerec_rec_mult_2[intro]:primerec rec_mult (Suc (Suc 0))
 apply(simp add: rec_mult_def )
 apply(rule_tac prime_pr, auto)
 using less_2_cases numeral_2_eq_2 by fastforce

lemma primerec_ge_2_elim[elim]: ⟦primerec rf n; n ≥ Suc (Suc 0)⟧ ⟹
              primerec (rec_accum rf) n
 apply(auto simp: rec_accum.simps)
  apply(simp add: nth_append, auto dest!:less_2_cases[unfolded numeral One_nat_def])
   apply force
  apply force
 apply(auto simp: nth_append)
 done

lemma primerec_all_iff:
 ⟦primerec rt n; primerec rf (Suc n); n > 0⟧ ⟹
                 primerec (rec_all rt rf) n
 apply(simp add: rec_all.simps, auto)
   apply(auto, simp add: nth_append, auto)
 done

lemma primerec_rec_not_1[intro]: primerec rec_not (Suc 0)
 apply(simp add: rec_not_def)
 apply(rule prime_cn, auto dest!:less_2_cases[unfolded numeral One_nat_def])
 done

lemma Min_false1[simp]: ⟦¬ Min {uu. uu ≤ w ∧ 0 < rec_exec rf (xs @ [uu])} ≤ w;
     x ≤ w; 0 < rec_exec rf (xs @ [x])⟧
    ⟹ False
 apply(subgoal_tac finite {uu. uu ≤ w ∧ 0 < rec_exec rf (xs @ [uu])})
  apply(subgoal_tac {uu. uu ≤ w ∧ 0 < rec_exec rf (xs @ [uu])} ≠ {})
   apply(simp add: Min_le_iff, simp)
  apply(rule_tac x = x in exI, simp)
 apply(simp)
 done

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 (λargs. 0 < rec_exec rf args) xs w
```

```
proof(induct w)
  let ?rt = (recf.id (Suc (length xs)) ((length xs)))
  let ?rf = (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)))])])
  let ?rq = (rec_all ?rt ?rf)
  have prrf: primerec ?rf (Suc (length (xs @ [0]))) ∧
        primerec ?rt (length (xs @ [0]))
    apply(auto simp: prrf nth_append)+
    done
  show Sigma (rec_exec (rec_all ?rt ?rf)) (xs @ [0])
       = Minr (λargs. 0 < rec_exec rf args) xs 0
    apply(simp add: Sigma.simps)
    apply(simp only: prrf all_lemma,
        auto simp: rec_exec.simps get_fstn_args_take Minr.simps)
    apply(rule_tac Min_eqI, auto)
    done
next
  fix w
  let ?rt = (recf.id (Suc (length xs)) ((length xs)))
  let ?rf = (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)))])])
  let ?rq = (rec_all ?rt ?rf)
  assume ind:
    Sigma (rec_exec (rec_all ?rt ?rf)) (xs @ [w]) = Minr (λargs. 0 < rec_exec rf args) xs w
  have prrf: primerec ?rf (Suc (length (xs @ [Suc w]))) ∧
        primerec ?rt (length (xs @ [Suc w]))
    apply(auto simp: prrf nth_append)+
    done
  show UF.Sigma (rec_exec (rec_all ?rt ?rf))
         (xs @ [Suc w]) =
        Minr (λargs. 0 < rec_exec rf args) xs (Suc w)
    apply(auto simp: Sigma_Suc_simp_rewrite ind Minr_Suc_simp)
      apply(simp_all only: prrf all_lemma)
      apply(auto simp: rec_exec.simps get_fstn_args_take Let_def Minr.simps split: if_splits)
      apply(drule_tac Min_false1, simp, simp, simp)
     apply (metis le_SucE neq0_conv)
    apply(drule_tac Min_false1, simp, simp, simp)
    apply(drule_tac Min_false1, simp, simp, simp)
    done
qed
```

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*
proof −
let *?rt* = (*recf.id* (*Suc* (*length xs*)) ((*length xs*)))
let *?rf* = (*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*)))])])
let *?rq* = (*rec_all ?rt ?rf*)
assume *h*: *primerec rf* (*Suc* (*length xs*))
have *h1*: *primerec ?rq* (*Suc* (*length xs*))
apply(*rule_tac primerec_all_iff*)
apply(*auto simp*: *h nth_append*)+
done
moreover have *arity rf* = *Suc* (*length xs*)
using *h* **by** *auto*
ultimately show *rec_exec* (*rec_Minr rf*) (*xs* @ [*w*]) =
Minr (λ *args*. (*0* < *rec_exec rf args*)) *xs w*
apply(*simp add*: *arity.simps Let_def sigma_lemma all_lemma*)
apply(*rule_tac sigma_minr_lemma*)
apply(*simp add*: *h*)
done
qed

rec_le is the comparasion function which compares its two arguments, testing whether the first is less or equal to the second.

definition *rec_le* :: *recf*
where
rec_le = *Cn* (*Suc* (*Suc 0*)) *rec_disj* [*rec_less*, *rec_eq*]

The correctness of *rec_le*.

lemma *le_lemma*:
$\bigwedge$*x y*. *rec_exec rec_le* [*x*, *y*] = (*if* (*x* $\leq$ *y*) *then 1 else 0*)
by(*auto simp*: *rec_le_def rec_exec.simps*)

Definition of *Max*[*Rr*] on page 77 of Boolos's book.

fun *Maxr* :: (*nat list* $\Rightarrow$ *bool*) $\Rightarrow$ *nat list* $\Rightarrow$ *nat* $\Rightarrow$ *nat*
where
Maxr Rr xs w = (*let setx* = {*y*. *y* $\leq$ *w* $\wedge$ *Rr* (*xs* @[*y*])} *in*
if setx = {} *then 0*
else Max setx)

rec_maxr is the recursive function used to implementation *Maxr*.

fun *rec_maxr* :: *recf* $\Rightarrow$ *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))
 apply(simp add: rec_disj_def, auto)
  apply(auto dest!:less_2_cases[unfolded numeral One_nat_def])
 done

lemma primerec_rec_less_2[intro]: primerec rec_less (Suc (Suc 0))
 apply(simp add: rec_less_def, auto)
  apply(auto dest!:less_2_cases[unfolded numeral One_nat_def])
 done

lemma primerec_rec_eq_2[intro]: primerec rec_eq (Suc (Suc 0))
 apply(simp add: rec_eq_def)
 apply(rule_tac prime_cn, auto dest!:less_2_cases[unfolded numeral One_nat_def])
 apply force+
 done

lemma primerec_rec_le_2[intro]: primerec rec_le (Suc (Suc 0))
 apply(simp add: rec_le_def)
 apply(rule_tac prime_cn, auto dest!:less_2_cases[unfolded numeral One_nat_def])
 done

lemma Sigma_0: ∀ i ≤ n. (f (xs @ [i]) = 0) ⟹
                 Sigma f (xs @ [n]) = 0
 apply(induct n, simp add: Sigma.simps)
 apply(simp add: Sigma_Suc_simp_rewrite)
 done

lemma Sigma_Suc[elim]: ∀ k<Suc w. f (xs @ [k]) = Suc 0
     ⟹ Sigma f (xs @ [w]) = Suc w
 apply(induct w)
  apply(simp add: Sigma.simps, simp)
 apply(simp add: Sigma.simps)
 done
```

```
lemma Sigma_max_point: ⟦∀ k < ma. f (xs @ [k]) = 1;
    ∀ k ≥ ma. f (xs @ [k]) = 0; ma ≤ w⟧
  ⟹ Sigma f (xs @ [w]) = ma
 apply(induct w, auto)
  apply(rule_tac Sigma_0, simp)
 apply(simp add: Sigma_Suc_simp_rewrite)
 using Sigma_Suc by fastforce

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 (λargs. 0 < rec_exec rf args) xs w
proof −
 let ?rt = (recf.id (Suc (Suc (length xs))) ((length xs)))
 let ?rf1 = 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)))]
 let ?rf2 = 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))))])
 let ?rf3 = Cn (Suc (Suc (Suc (length xs)))) rec_not [?rf2]
 let ?rf = Cn (Suc (Suc (Suc (length xs)))) rec_disj [?rf1, ?rf3]
 let ?rq = rec_all ?rt ?rf
 let ?notrq = Cn (Suc (Suc (length xs))) rec_not [?rq]
 show ?thesis
 proof(auto simp: Maxr.simps)
  assume h: ∀x≤w. rec_exec rf (xs @ [x]) = 0
  have primerec ?rf (Suc (length (xs @ [w, i]))) ∧
      primerec ?rt (length (xs @ [w, i]))
   using prrf
   apply(auto dest!:less_2_cases[unfolded numeral One_nat_def])
       apply force+
   apply(case_tac ia, auto simp: h nth_append primerec_getpren)
   done
  hence Sigma (rec_exec ?notrq) ((xs@[w])@[w]) = 0
   apply(rule_tac Sigma_0)
```

```
    apply(auto simp: rec_exec.simps all_lemma
      get_fstn_args_take nth_append h)
    done
  thus UF.Sigma (rec_exec ?notrq)
   (xs @ [w, w]) = 0
   by simp
 next
  fix x
  assume h: x ≤ w 0 < rec_exec rf (xs @ [x])
  hence ∃ ma. Max {y. y ≤ w ∧ 0 < rec_exec rf (xs @ [y])} = ma
   by auto
  from this obtain ma where k1:
   Max {y. y ≤ w ∧ 0 < rec_exec rf (xs @ [y])} = ma ..
  hence k2: ma ≤ w ∧ 0 < rec_exec rf (xs @ [ma])
   using h
   apply(subgoal_tac
     Max {y. y ≤ w ∧ 0 < rec_exec rf (xs @ [y])} ∈ {y. y ≤ w ∧ 0 < rec_exec rf (xs @ [y])})
    apply(erule_tac CollectE, simp)
   apply(rule_tac Max_in, auto)
   done
  hence k3: ∀ k < ma. (rec_exec ?notrq (xs @ [w, k]) = 1)
   apply(auto simp: nth_append)
   apply(subgoal_tac primerec ?rf (Suc (length (xs @ [w, k]))) ∧
    primerec ?rt (length (xs @ [w, k])))
    apply(auto simp: rec_exec.simps all_lemma get_fstn_args_take nth_append
      dest!:less_2_cases[unfolded numeral One_nat_def])
   using prrf
   apply force+
   done
  have k4: ∀ k ≥ ma. (rec_exec ?notrq (xs @ [w, k]) = 0)
   apply(auto)
   apply(subgoal_tac primerec ?rf (Suc (length (xs @ [w, k]))) ∧
    primerec ?rt (length (xs @ [w, k])))
    apply(auto simp: rec_exec.simps all_lemma get_fstn_args_take nth_append)
    apply(subgoal_tac x ≤ Max {y. y ≤ w ∧ 0 < rec_exec rf (xs @ [y])},
      simp add: k1)
    apply(rule_tac Max_ge, auto dest!:less_2_cases[unfolded numeral One_nat_def])
   using prrf apply force+
   apply(auto simp: h nth_append)
   done
  from k3 k4 k1 have Sigma (rec_exec ?notrq) ((xs @ [w]) @ [w]) = ma
   apply(rule_tac Sigma_max_point, simp, simp, simp add: k2)
   done
  from k1 and this show Sigma (rec_exec ?notrq) (xs @ [w, w]) =
   Max {y. y ≤ w ∧ 0 < rec_exec rf (xs @ [y])}
   by simp
 qed
qed
```

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 -
 from h have arity rf = Suc (length xs)
  by auto
 thus ?thesis
 proof(simp add: rec_exec.simps rec_maxr.simps nth_append get_fstn_args_take)
  let ?rt = (recf.id (Suc (Suc (length xs))) ((length xs)))
  let ?rf1 = 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)))]
  let ?rf2 = 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))))])
  let ?rf3 = Cn (Suc (Suc (Suc (length xs)))) rec_not [?rf2]
  let ?rf = Cn (Suc (Suc (Suc (length xs)))) rec_disj [?rf1, ?rf3]
  let ?rq = rec_all ?rt ?rf
  let ?notrq = Cn (Suc (Suc (length xs))) rec_not [?rq]
  have prt: primerec ?rt (Suc (Suc (length xs)))
   by(auto intro: prime_id)
  have prrf: primerec ?rf (Suc (Suc (Suc (length xs))))
   apply(auto dest!:less_2_cases[unfolded numeral One_nat_def])
      apply force+
    apply(auto intro: prime_id)
    apply(simp add: h)
   apply(auto simp add: nth_append)
   done
  from prt and prrf have prrq: primerec ?rq
                          (Suc (Suc (length xs)))
   by(erule_tac primerec_all_iff, auto)
  hence prnotrp: primerec ?notrq (Suc (length ((xs @ [w]))))
   by(rule_tac prime_cn, auto)
  have g1: rec_exec (rec_sigma ?notrq) ((xs @ [w]) @ [w])
   = Maxr (λargs. 0 < rec_exec rf args) xs w
   using prnotrp
   using sigma_lemma
   apply(simp only: sigma_lemma)
   apply(rule_tac Sigma_Max_lemma)
   apply(simp add: h)
   done
  thus rec_exec (rec_sigma ?notrq)
   (xs @ [w, w]) =
  Maxr (λargs. 0 < rec_exec rf args) xs w
   apply(simp)
   done
```

qed
qed

quo is the formal specification of division.

fun $quo :: nat\ list \Rightarrow nat$
where
$quo\ [x, y] = (let\ Rr =$
$(\lambda\ zs.\ ((zs\ !\ (Suc\ 0) * zs\ !\ (Suc\ (Suc\ 0))$
$\leq zs\ !\ 0) \wedge zs\ !\ Suc\ 0 \neq (0::nat)))$
$in\ Maxr\ Rr\ [x, y]\ x)$

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

The following lemmas shows more directly the menaing of *quo*:

lemma *quo_is_div*: $y > 0 \Longrightarrow quo\ [x, y] = x\ div\ y$
proof $-$
{
fix *xa ya*
assume h: $y * ya \leq x \ \ y > 0$
hence $(y * ya)\ div\ y \leq x\ div\ y$
by(*insert div_le_mono*[*of* $y * ya\ x\ y$], *simp*)
from *this* **and** *h* **have** $ya \leq x\ div\ y$ **by** *simp*}
thus *?thesis* **by**(*simp add*: *quo.simps Maxr.simps*, *auto*,
rule_tac Max_eqI, *simp*, *auto*)
qed

lemma *quo_zero*[*intro*]: $quo\ [x, 0] = 0$
by(*simp add*: *quo.simps Maxr.simps*)

lemma *quo_div*: $quo\ [x, y] = x\ div\ y$
by(*cases* y=0, *auto elim*!:*quo_is_div*)

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*:
$\bigwedge x\ y.\ rec\_exec\ rec\_noteq\ [x, y] =$
$(if\ x \neq y\ then\ 1\ else\ 0)$
by(*simp add*: *rec_exec.simps rec_noteq_def*)

declare *noteq_lemma*[*simp*]

rec_quo is the recursive function used to implement *quo*

definition *rec_quo* :: *recf*

```
where
  rec_quo = (let rR = Cn (Suc (Suc (Suc 0))) rec_conj
          [Cn (Suc (Suc (Suc 0))) rec_le
           [Cn (Suc (Suc (Suc 0))) rec_mult
             [id (Suc (Suc (Suc 0))) (Suc 0),
               id (Suc (Suc (Suc 0))) ((Suc (Suc 0)))],
            id (Suc (Suc (Suc 0))) (0)],
            Cn (Suc (Suc (Suc 0))) rec_noteq
                 [id (Suc (Suc (Suc 0))) (Suc (0)),
            Cn (Suc (Suc (Suc 0))) (constn 0)
                     [id (Suc (Suc (Suc 0))) (0)]]]
          in Cn (Suc (Suc 0)) (rec_maxr rR)) [id (Suc (Suc 0))
                   (0),id (Suc (Suc 0)) (Suc (0)),
                         id (Suc (Suc 0)) (0)]

lemma primerec_rec_conj_2[intro]: primerec rec_conj (Suc (Suc 0))
 apply(simp add: rec_conj_def)
 apply(rule_tac prime_cn, auto dest!:less_2_cases[unfolded numeral One_nat_def])
 done

lemma primerec_rec_noteq_2[intro]: primerec rec_noteq (Suc (Suc 0))
 apply(simp add: rec_noteq_def)
 apply(rule_tac prime_cn, auto dest!:less_2_cases[unfolded numeral One_nat_def])
 done


lemma quo_lemma1: rec_exec rec_quo [x, y] = quo [x, y]
proof(simp add: rec_exec.simps rec_quo_def)
 let ?rR = (Cn (Suc (Suc (Suc 0))) rec_conj
           [Cn (Suc (Suc (Suc 0))) rec_le
              [Cn (Suc (Suc (Suc 0))) rec_mult
          [recf.id (Suc (Suc (Suc 0))) (Suc (0)),
           recf.id (Suc (Suc (Suc 0))) (Suc (Suc (0)))],
            recf.id (Suc (Suc (Suc 0))) (0)],
       Cn (Suc (Suc (Suc 0))) rec_noteq
                     [recf.id (Suc (Suc (Suc 0)))
         (Suc (0)), Cn (Suc (Suc (Suc 0))) (constn 0)
                [recf.id (Suc (Suc (Suc 0))) (0)]]])
 have rec_exec (rec_maxr ?rR) ([x, y]@ [ x]) = Maxr (λ args. 0 < rec_exec ?rR args) [x, y] x
 proof(rule_tac Maxr_lemma, simp)
  show primerec ?rR (Suc (Suc (Suc 0)))
   apply(auto dest!:less_2_cases[unfolded numeral One_nat_def])
       apply force+
   done
 qed
 hence g1: rec_exec (rec_maxr ?rR) ([x, y,  x]) =
         Maxr (λ args. if rec_exec ?rR args = 0 then False
                  else True) [x, y] x
  by simp
 have g2: Maxr (λ args. if rec_exec ?rR args = 0 then False
```

else True) [*x*, *y*] *x* = *quo* [*x*, *y*]
apply(*simp add*: *rec_exec*.*simps*)
apply(*simp add*: *Maxr*.*simps quo*.*simps*, *auto*)
done
from *g1* **and** *g2* **show**
rec_exec (*rec_maxr ?rR*) ([*x*, *y*, *x*]) = *quo* [*x*, *y*]
by *simp*
qed

The correctness of *quo*.

lemma *quo_lemma2*: *rec_exec rec_quo* [*x*, *y*] = *x div y*
using *quo_lemma1*[*of x y*] *quo_div*[*of x y*]
by *simp*

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*)
by(*simp add*: *rec_exec*.*simps rec_mod_def quo_lemma2 minus_div_mult_eq_mod*)

lemmas for embranch function

type-synonym *ftype* = *nat list* ⇒ *nat*
type-synonym *rtype* = *nat list* ⇒ *bool*

The specifation of the mutli-way branching statement on page 79 of Boolos's book.

fun *Embranch* :: (*ftype* ∗ *rtype*) *list* ⇒ *nat list* ⇒ *nat*
where
Embranch [] *xs* = *0* |
Embranch (*gc* # *gcs*) *xs* = (
let (*g*, *c*) = *gc in*
if c xs then g xs else Embranch gcs xs)

fun *rec_embranch′* :: (*recf* ∗ *recf*) *list* ⇒ *nat* ⇒ *recf*
where
rec_embranch′ [] *vl* = *Cn vl z* [*id vl* (*vl* − *1*)] |
rec_embranch′ ((*rg*, *rc*) # *rgcs*) *vl* = *Cn vl rec_add*
[*Cn vl rec_mult* [*rg*, *rc*], *rec_embranch′ rgcs vl*]

rec_embrach is the recursive function used to implement *Embranch*.

fun *rec_embranch* :: (*recf* ∗ *recf*) *list* ⇒ *recf*
where
rec_embranch ((*rg*, *rc*) # *rgcs*) =
(*let vl* = *arity rg in*
rec_embranch′ ((*rg*, *rc*) # *rgcs*) *vl*)

```
declare Embranch.simps[simp del] rec_embranch.simps[simp del]

lemma embranch_all0:
  ⟦∀ j < length rcs. rec_exec (rcs ! j) xs = 0;
    length rgs = length rcs;
  rcs ≠ [];
  list_all (λ rf. primerec rf (length xs)) (rgs @ rcs)⟧ ⟹
  rec_exec (rec_embranch (zip rgs rcs)) xs = 0
proof(induct rcs arbitrary: rgs)
  case (Cons a rcs)
  then show ?case proof(cases rgs, simp)  fix a rcs rgs aa list
    assume ind:
      ⋀rgs. ⟦∀j<length rcs. rec_exec (rcs ! j) xs = 0;
           length rgs = length rcs; rcs ≠ [];
          list_all (λrf. primerec rf (length xs)) (rgs @ rcs)⟧ ⟹
                rec_exec (rec_embranch (zip rgs rcs)) xs = 0
      and h:  ∀j<length (a # rcs). rec_exec ((a # rcs) ! j) xs = 0
      length rgs = length (a # rcs)
      a # rcs ≠ []
      list_all (λrf. primerec rf (length xs)) (rgs @ a # rcs)
      rgs = aa # list
    have g: rcs ≠ [] ⟹ rec_exec (rec_embranch (zip list rcs)) xs = 0
      using h by(rule_tac ind, auto)
    show rec_exec (rec_embranch (zip rgs (a # rcs))) xs = 0
    proof(cases rcs = [], simp)
      show rec_exec (rec_embranch (zip rgs [a])) xs = 0
        using h by (auto simp add: rec_embranch.simps rec_exec.simps)
    next
      assume rcs ≠ []
      hence rec_exec (rec_embranch (zip list rcs)) xs = 0
        using g by simp
      thus rec_exec (rec_embranch (zip rgs (a # rcs))) xs = 0
        using h
        by(cases rcs;cases list, auto simp add: rec_embranch.simps rec_exec.simps)
    qed
  qed
qed simp


lemma embranch_exec_0: ⟦rec_exec aa xs = 0; zip rgs list ≠ [];
       list_all (λ rf. primerec rf (length xs)) ([a, aa] @ rgs @ list)⟧
       ⟹ rec_exec (rec_embranch ((a, aa) # zip rgs list)) xs
         = rec_exec (rec_embranch (zip rgs list)) xs
  apply(auto simp add: rec_exec.simps rec_embranch.simps)
  apply(cases zip rgs list, force)
  apply(cases hd (zip rgs list), simp add: rec_embranch.simps rec_exec.simps)
  apply(subgoal_tac arity a = length xs)
   apply(cases rgs;cases list;force)
  by force
```

lemma *zip_null_iff*: $[\![$*length xs* $=$ *k*; *length ys* $=$ *k*; *zip xs ys* $=$ []$]\!] \Longrightarrow$ *xs* $=$ [] $\wedge$ *ys* $=$ []
 apply(*cases xs*, *simp*, *simp*)
 apply(*cases ys*, *simp*, *simp*)
 done

lemma *zip_null_gr*: $[\![$*length xs* $=$ *k*; *length ys* $=$ *k*; *zip xs ys* $\neq$ []$]\!] \Longrightarrow$ *0* $<$ *k*
 apply(*cases xs*, *simp*, *simp*)
 done

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* (λ*r args*. *0* $<$ *rec_exec r args*) *rcs*)) *xs* $=$ *0*
proof(*induct rgs arbitrary*: *rcs k*)
 case (*Cons a rgs rcs k*)
 then show *?case*
  apply(*cases rcs*, *simp*, *cases rgs* $=$ [])
   apply(*simp add*: *Embranch.simps*)
   apply(*erule_tac x* $=$ *0* **in** *allE*)
   apply (*auto simp add*: *Embranch.simps intro*!: *Cons*(*1*))**.**
qed *simp*

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* (λ *rf*. *primerec rf* (*length xs*)) (*rgs* @ *rcs*)
 shows *rec_exec* (*rec_embranch* (*zip rgs rcs*)) *xs* $=$
     Embranch (*zip* (*map rec_exec rgs*)
      (*map* (λ *r args*. *0* $<$ *rec_exec r args*) *rcs*)) *xs*
 using *branch_num partition prime_all*
proof(*induct rgs arbitrary*: *rcs n*, *simp*)
 fix *a rgs rcs n*
 assume *ind*:
  $\bigwedge$*rcs n*. $[\![$*length rgs* $=$ *n*; *length rcs* $=$ *n*; *0* $<$ *n*;
  $\exists$*i*$<$*n*. *rec_exec* (*rcs* ! *i*) *xs* $=$ *1* $\wedge$ ($\forall$*j*$<$*n*. *j* $\neq$ *i* $\longrightarrow$ *rec_exec* (*rcs* ! *j*) *xs* $=$ *0*);
  list_all (λ*rf*. *primerec rf* (*length xs*)) (*rgs* @ *rcs*)$]\!]$
  $\Longrightarrow$ *rec_exec* (*rec_embranch* (*zip rgs rcs*)) *xs* $=$
  Embranch (*zip* (*map rec_exec rgs*) (*map* (λ*r args*. *0* $<$ *rec_exec r args*) *rcs*)) *xs*
  and *h*: *length* (*a* # *rgs*) $=$ *n length* (*rcs*::*recf list*) $=$ *n 0* $<$ *n*
  $\exists$*i*$<$*n*. *rec_exec* (*rcs* ! *i*) *xs* $=$ *1* $\wedge$
   ($\forall$*j*$<$*n*. *j* $\neq$ *i* $\longrightarrow$ *rec_exec* (*rcs* ! *j*) *xs* $=$ *0*)
  list_all (λ*rf*. *primerec rf* (*length xs*)) ((*a* # *rgs*) @ *rcs*)
 from *h* **show** *rec_exec* (*rec_embranch* (*zip* (*a* # *rgs*) *rcs*)) *xs* $=$
  Embranch (*zip* (*map rec_exec* (*a* # *rgs*)) (*map* (λ*r args*.
      0 $<$ *rec_exec r args*) *rcs*)) *xs*

```
 apply(cases rcs, simp, simp)
 apply(cases rec_exec (hd rcs) xs = 0)
  apply(case_tac [!] zip rgs (tl rcs) = [], simp)
     apply(subgoal_tac rgs = [] ∧ (tl rcs) = [], simp add: Embranch.simps rec_exec.simps
rec_embranch.simps)
   apply(rule_tac zip_null_iff, simp, simp, simp)
proof -
 fix aa list
 assume rcs = aa # list
 assume g:
  Suc (length rgs) = n Suc (length list) = n
  ∃ i<n. rec_exec ((aa # list) ! i) xs = Suc 0 ∧
    (∀ j<n. j ≠ i ⟶ rec_exec ((aa # list) ! j) xs = 0)
  primerec a (length xs) ∧
  list_all (λrf. primerec rf (length xs)) rgs ∧
  primerec aa (length xs) ∧
  list_all (λrf. primerec rf (length xs)) list
  rec_exec (hd rcs) xs = 0 rcs = aa # list zip rgs (tl rcs) ≠ []
 hence rec_exec aa xs = 0 zip rgs list ≠ [] by auto
 note g = g(1,2,3,4,6) this
 have rec_exec (rec_embranch ((a, aa) # zip rgs list)) xs
    = rec_exec (rec_embranch (zip rgs list)) xs
  apply(rule embranch_exec_0, simp_all add: g)
  done
 from g and this show rec_exec (rec_embranch ((a, aa) # zip rgs list)) xs =
     Embranch ((rec_exec a, λargs. 0 < rec_exec aa args) #
      zip (map rec_exec rgs) (map (λr args. 0 < rec_exec r args) list)) xs
  apply(simp add: Embranch.simps)
  apply(rule_tac n = n − Suc 0 in ind)
     apply(cases n;force)
    apply(cases n;force)
   apply(cases n;force simp add: zip_null_gr)
   apply(auto)
  apply(rename_tac i)
  apply(case_tac i, force, simp)
  apply(rule_tac x = i − 1 in exI, simp)
  by auto
next
 fix aa list
 assume g: Suc (length rgs) = n Suc (length list) = n
  ∃ i<n. rec_exec ((aa # list) ! i) xs = Suc 0 ∧
  (∀ j<n. j ≠ i ⟶ rec_exec ((aa # list) ! j) xs = 0)
  primerec a (length xs) ∧ list_all (λrf. primerec rf (length xs)) rgs ∧
  primerec aa (length xs) ∧ list_all (λrf. primerec rf (length xs)) list
  rcs = aa # list rec_exec (hd rcs) xs ≠ 0 zip rgs (tl rcs) = []
 thus rec_exec (rec_embranch ((a, aa) # zip rgs list)) xs =
    Embranch ((rec_exec a, λargs. 0 < rec_exec aa args) #
   zip (map rec_exec rgs) (map (λr args. 0 < rec_exec r args) list)) xs
  apply(subgoal_tac rgs = [] ∧ list = [], simp)
   prefer 2
```

```
   apply(rule_tac zip_null_iff, simp, simp, simp)
  apply(simp add: rec_exec.simps rec_embranch.simps Embranch.simps, auto)
  done
next
 fix aa list
 assume g: Suc (length rgs) = n Suc (length list) = n
  ∃i<n. rec_exec ((aa # list) ! i) xs = Suc 0 ∧
      (∀j<n. j ≠ i ⟶ rec_exec ((aa # list) ! j) xs = 0)
  primerec a (length xs) ∧ list_all (λrf. primerec rf (length xs)) rgs
  ∧ primerec aa (length xs) ∧ list_all (λrf. primerec rf (length xs)) list
  rcs = aa # list rec_exec (hd rcs) xs ≠ 0 zip rgs (tl rcs) ≠ []
 have rec_exec aa xs =  Suc 0
  using g
  apply(cases rec_exec aa xs, simp, auto)
  done
 moreover have rec_exec (rec_embranch' (zip rgs list) (length xs)) xs = 0
 proof -
  have rec_embranch' (zip rgs list) (length xs) = rec_embranch (zip rgs list)
   using g
   apply(cases zip rgs list, force)
   apply(cases hd (zip rgs list))
   apply(simp add: rec_embranch.simps)
   apply(cases rgs, simp, simp, cases list, simp, auto)
   done
  moreover have rec_exec (rec_embranch (zip rgs list)) xs = 0
  proof(rule embranch_all0)
   show  ∀j<length list. rec_exec (list ! j) xs = 0
    using g
    apply(auto)
    apply(rename_tac i j)
    apply(case_tac i, simp)
     apply(erule_tac x = Suc j in allE, simp)
    apply(simp)
    apply(erule_tac x = 0 in allE, simp)
    done
  next
   show length rgs = length list
    using g by(cases n;force)
  next
   show list ≠ []
    using g by(cases list; force)
  next
   show list_all (λrf. primerec rf (length xs)) (rgs @ list)
    using g by auto
  qed
  ultimately show rec_exec (rec_embranch' (zip rgs list) (length xs)) xs = 0
   by simp
 qed
 moreover have
  Embranch (zip (map rec_exec rgs)
```

```
      (map (λr args. 0 < rec_exec r args) list)) xs = 0
    using g
    apply(rule_tac k = length rgs in Embranch_0)
      apply(simp, cases n, simp, simp)
     apply(cases rgs, simp, simp)
    apply(auto)
    apply(rename_tac i j)
    apply(case_tac i, simp)
     apply(erule_tac x = Suc j in allE, simp)
    apply(simp)
    apply(rule_tac x = 0 in allE, auto)
    done
  moreover have arity a = length xs
    using g
    apply(auto)
    done
  ultimately show rec_exec (rec_embranch ((a, aa) # zip rgs list)) xs =
    Embranch ((rec_exec a, λargs. 0 < rec_exec aa args) #
        zip (map rec_exec rgs) (map (λr args. 0 < rec_exec r args) list)) xs
    apply(simp add: rec_exec.simps rec_embranch.simps Embranch.simps)
    done
 qed
qed
```

prime n means *n* is a prime number.

```
fun Prime :: nat ⇒ bool
 where
  Prime x = (1 < x ∧ (∀ u < x. (∀ v < x. u ∗ v ≠ x)))

declare Prime.simps [simp del]

lemma primerec_all1:
 primerec (rec_all rt rf) n ⟹ primerec rt n
 by (simp add: primerec_all)

lemma primerec_all2: primerec (rec_all rt rf) n ⟹
 primerec rf (Suc n)
 by(insert primerec_all[of rt rf n], simp)
```

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))
 apply(rule_tac all_lemma)
  apply(auto simp:numeral)
  apply (metis (no_types, lifting) Suc_mono length_Cons less_2_cases list.size(3) nth_Cons_0
    nth_Cons_Suc numeral_2_eq_2 prime_cn prime_id primerec_rec_mult_2 zero_less_Suc)
 by (metis (no_types, lifting) One_nat_def length_Cons less_2_cases nth_Cons_0 nth_Cons_Suc
    prime_cn_reverse primerec_rec_eq_2 rec_eq_def zero_less_Suc)
```

The correctness of *Prime*.

```
lemma prime_lemma: rec_exec rec_prime [x] = (if Prime x then 1 else 0)
proof(simp add: rec_exec.simps rec_prime_def)
 let ?rt1 = (Cn 2 rec_minus [recf.id 2 0,
  Cn 2 (constn (Suc 0)) [recf.id 2 0]])
 let ?rf1 = (Cn 3 rec_noteq [Cn 3 rec_mult
  [recf.id 3 (Suc 0), recf.id 3 2], recf.id 3 (0)])
 let ?rt2 = (Cn (Suc 0) rec_minus
  [recf.id (Suc 0) 0, constn (Suc 0)])
 let ?rf2 = rec_all ?rt1 ?rf1
 have h1: rec_exec (rec_all ?rt2 ?rf2) ([x]) =
     (if (∀ k≤rec_exec ?rt2 ([x]). 0 < rec_exec ?rf2 ([x] @ [k])) then 1 else 0)
 proof(rule_tac all_lemma, simp_all)
  show primerec ?rf2 (Suc (Suc 0))
    apply(rule_tac primerec_all_iff)
     apply(auto simp: numeral)
   apply (metis (no_types, lifting) One_nat_def length_Cons less_2_cases nth_Cons_0 nth_Cons_Suc
      prime_cn_reverse primerec_rec_eq_2 rec_eq_def zero_less_Suc)
    by (metis (no_types, lifting) Suc_mono length_Cons less_2_cases list.size(3) nth_Cons_0
      nth_Cons_Suc numeral_2_eq_2 prime_cn prime_id primerec_rec_mult_2 zero_less_Suc)
 next
  show primerec (Cn (Suc 0) rec_minus
        [recf.id (Suc 0) 0, constn (Suc 0)]) (Suc 0)
    using less_2_cases numeral by fastforce
 qed
 from h1 show
  (Suc 0 < x ⟶  (rec_exec (rec_all ?rt2 ?rf2) [x] = 0 ⟶
  ¬ Prime x) ∧
   (0 < rec_exec (rec_all ?rt2 ?rf2) [x] ⟶ Prime x)) ∧
  (¬ Suc 0 < x ⟶ ¬ Prime x ∧ (rec_exec (rec_all ?rt2 ?rf2) [x] = 0
  ⟶ ¬ Prime x))
  apply(auto simp:rec_exec.simps)
    apply(simp add: exec_tmp rec_exec.simps)
 proof −
  assume ∗:∀ k≤x − Suc 0. (0::nat) < (if ∀ w≤x − Suc 0.
      0 < (if k ∗ w ≠ x then 1 else (0 :: nat)) then 1 else 0) Suc 0 < x
```

```
  thus Prime x
    apply(simp add: rec_exec.simps split: if_splits)
    apply(simp add: Prime.simps, auto)
    apply(rename_tac u v)
    apply(erule_tac x = u in allE, auto)
    apply(case_tac u, simp)
    apply(case_tac u - 1, simp, simp)
    apply(case_tac v, simp)
    apply(case_tac v - 1, simp, simp)
    done
 next
  assume ¬ Suc 0 < x Prime x
  thus False
    apply(simp add: Prime.simps)
    done
 next
  fix k
  assume rec_exec (rec_all ?rt1 ?rf1)
   [x, k] = 0 k ≤ x − Suc 0 Prime x
  thus False
    apply(simp add: exec_tmp rec_exec.simps Prime.simps split: if_splits)
    done
 next
  fix k
  assume rec_exec (rec_all ?rt1 ?rf1)
   [x, k] = 0 k ≤ x − Suc 0 Prime x
  thus False
    apply(simp add: exec_tmp rec_exec.simps Prime.simps split: if_splits)
    done
 qed
qed

definition rec_dummyfac :: recf
 where
  rec_dummyfac = Pr 1 (constn 1)
 (Cn 3 rec_mult [id 3 2, Cn 3 s [id 3 1]])
```

The recursive function used to implment factorization.

```
definition rec_fac :: recf
 where
  rec_fac = Cn 1 rec_dummyfac [id 1 0, id 1 0]
```

Formal specification of factorization.

```
fun fac :: nat ⇒ nat  (_! [100] 99)
 where
  fac 0 = 1 |
  fac (Suc x) = (Suc x) ∗ fac x

lemma fac_dummy: rec_exec rec_dummyfac [x, y] = y !
 apply(induct y)
```

```
 apply(auto simp: rec_dummyfac_def rec_exec.simps)
 done
```

The correctness of *rec_fac*.

```
lemma fac_lemma: rec_exec rec_fac [x] = x!
 apply(simp add: rec_fac_def rec_exec.simps fac_dummy)
 done

declare fac.simps[simp del]
```

Np x returns the first prime number after *x*.

```
fun Np ::nat ⇒ nat
  where
  Np x = Min {y. y ≤ Suc (x!) ∧ x < y ∧ 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!)
proof(induct n)
  case (Suc n)
  then show ?case  apply(simp add: fac.simps)
    apply(cases n, auto simp: fac.simps)
    done
qed simp

lemma divsor_ex:
  ⟦¬ Prime x; x > Suc 0⟧ ⟹ (∃ u > Suc 0. (∃ v > Suc 0. u ∗ v = x))
  by(auto simp: Prime.simps)

lemma divsor_prime_ex: ⟦¬ Prime x; x > Suc 0⟧ ⟹
  ∃ p. Prime p ∧ p dvd x
  apply(induct x rule: wf_induct[where r = measure (λ y. y)], simp)
  apply(drule_tac divsor_ex, simp, auto)
  apply(rename_tac u v)
  apply(erule_tac x = u in allE, simp)
  apply(case_tac Prime u, simp)
   apply(rule_tac x = u in exI, simp, auto)
  done

lemma fact_pos[intro]: 0 < n!
  apply(induct n)
   apply(auto simp: fac.simps)
  done
```

```
lemma fac_Suc: Suc n! = (Suc n) ∗ (n!) by(simp add: fac.simps)

lemma fac_dvd: ⟦0 < q; q ≤ n⟧ ⟹ q dvd n!
proof(induct n)
 case (Suc n)
 then show ?case
  apply(cases q ≤ n, simp add: fac_Suc)
  apply(subgoal_tac q = Suc n, simp only: fac_Suc)
   apply(rule_tac dvd_mult2, simp, simp)
  done
qed simp

lemma fac_dvd2: ⟦Suc 0 < q; q dvd n!; q ≤ n⟧ ⟹ ¬ q dvd Suc (n!)
proof(auto simp: dvd_def)
 fix k ka
 assume h1: Suc 0 < q q ≤ n
  and h2: Suc (q ∗ k) = q ∗ ka
 have k < ka
 proof −
  have q ∗ k < q ∗ ka
   using h2 by arith
  thus k < ka
   using h1
   by(auto)
 qed
 hence ∃ d. d > 0 ∧ ka = d + k
  by(rule_tac x = ka − k in exI, simp)
 from this obtain d where d > 0 ∧ ka = d + k ..
 from h2 and this and h1 show False
  by(simp add: add_mult_distrib2)
qed

lemma prime_ex: ∃ p. n < p ∧ p ≤ Suc (n!) ∧ Prime p
proof(cases Prime (n! + 1))
 case True thus ?thesis
  by(rule_tac x = Suc (n!) in exI, simp)
next
 assume h: ¬ Prime (n! + 1)
 hence ∃ p. Prime p ∧ p dvd (n! + 1)
  by(erule_tac divsor_prime_ex, auto)
 from this obtain q where k: Prime q ∧ q dvd (n! + 1) ..
 thus ?thesis
 proof(cases q > n)
  case True thus ?thesis
   using k by(auto intro:dvd_imp_le)
 next
  case False thus ?thesis
  proof −
   assume g: ¬ n < q
```

```
    have j: q > Suc 0
      using k by(cases q, auto simp: Prime.simps)
    hence q dvd n!
      using g
      apply(rule_tac fac_dvd, auto)
      done
    hence ¬ q dvd Suc (n!)
      using g j
      by(rule_tac fac_dvd2, auto)
    thus ?thesis
      using k by simp
  qed
 qed
qed

lemma Suc_Suc_induct[elim!]: ⟦i < Suc (Suc 0);
 primerec (ys ! 0) n; primerec (ys ! 1) n⟧ ⟹ primerec (ys ! i) n
 by(cases i, auto)

lemma primerec_rec_prime_1[intro]: primerec rec_prime (Suc 0)
 apply(auto simp: rec_prime_def, auto)
 apply(rule_tac primerec_all_iff, auto, auto)
 apply(rule_tac primerec_all_iff, auto, auto simp:
   numeral_2_eq_2 numeral_3_eq_3)
 done
```

The correctness of *rec_np*.

```
lemma np_lemma: rec_exec rec_np [x] = Np x
proof(auto simp: rec_np_def rec_exec.simps Let_def fac_lemma)
 let ?rr = (Cn 2 rec_conj [Cn 2 rec_less [recf.id 2 0,
  recf.id 2 (Suc 0)], Cn 2 rec_prime [recf.id 2 (Suc 0)]])
 let ?R = λ zs. zs ! 0 < zs ! 1 ∧ Prime (zs ! 1)
 have g1: rec_exec (rec_Minr ?rr) ([x] @ [Suc (x!)]) =
  Minr (λ args. 0 < rec_exec ?rr args) [x] (Suc (x!))
  by(rule_tac Minr_lemma, auto simp: rec_exec.simps
    prime_lemma, auto simp:  numeral_2_eq_2 numeral_3_eq_3)
 have g2: Minr (λ args. 0 < rec_exec ?rr args) [x] (Suc (x!)) = Np x
  using prime_ex[of x]
  apply(auto simp: Minr.simps Np.simps rec_exec.simps prime_lemma)
  apply(subgoal_tac
    {uu. (Prime uu ⟶ (x < uu ⟶ uu ≤ Suc (x!)) ∧ x < uu) ∧ Prime uu}
  = {y. y ≤ Suc (x!) ∧ x < y ∧ Prime y}, auto)
  done
 from g1 and g2 show rec_exec (rec_Minr ?rr) ([x, Suc (x!)]) = Np x
  by simp
qed
```

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
 by(induct y, auto simp: rec_exec.simps rec_power_def)
```

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
 apply(induct y)
 by(auto simp: rec_exec.simps rec_dummy_pi_def Pi.simps np_lemma)
```

The correctness of *rec_pi*.

```
lemma pi_lemma: rec_exec rec_pi [x] = Pi x
 apply(simp add: rec_pi_def rec_exec.simps pi_dummy_lemma)
 done

fun loR :: nat list ⇒ bool
 where
  loR [x, y, u] = (x mod (y^u) = 0)

declare loR.simps[simp del]
```

Lo specifies the *lo* function given on page 79 of Boolos's book. It is one of the two notions of integeral logarithmetic operation on that page. The other is *lg*.

```
fun lo ::  nat ⇒ nat ⇒ nat
 where
  lo x y  = (if x > 1 ∧ y > 1 ∧ {u. loR [x, y, u]} ≠ {} then Max {u. loR [x, y, u]}
                                              else 0)

declare lo.simps[simp del]

lemma primerec_sigma[intro!]:
 ⟦n > Suc 0; primerec rf n⟧ ⟹
 primerec (rec_sigma rf) n
 apply(simp add: rec_sigma.simps)
```

```
 apply(auto, auto simp: nth_append)
 done

lemma primerec_rec_maxr[intro!]: ⟦primerec rf n; n > 0⟧ ⟹ primerec (rec_maxr rf) n
 apply(simp add: rec_maxr.simps)
 apply(rule_tac prime_cn, auto)
  apply(rule_tac primerec_all_iff, auto, auto simp: nth_append)
 done

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
 apply(cases i, auto)
 apply(cases i−1, simp, simp add: numeral_2_eq_2)
 done

lemma primerec_2[intro]:
 primerec rec_quo (Suc (Suc 0)) primerec rec_mod (Suc (Suc 0))
 primerec rec_power (Suc (Suc 0))
 by(force simp: prime_cn prime_id rec_mod_def rec_quo_def rec_power_def prime_pr numeral)+
```

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(auto simp: rec_exec.simps rec_lo_def Let_def
   numeral_2_eq_2 numeral_3_eq_3)
 let ?rR = (Cn (Suc (Suc (Suc 0))) rec_eq
   [Cn (Suc (Suc (Suc 0))) rec_mod [recf.id (Suc (Suc (Suc 0))) 0,
   Cn (Suc (Suc (Suc 0))) rec_power [recf.id (Suc (Suc (Suc 0)))
   (Suc 0), recf.id (Suc (Suc (Suc 0))) (Suc (Suc 0))]],
   Cn (Suc (Suc (Suc 0))) (constn 0) [recf.id (Suc (Suc (Suc 0))) (Suc 0)]])
 have h: rec_exec (rec_maxr ?rR) ([x, y] @ [x]) =
  Maxr (λ args. 0 < rec_exec ?rR args) [x, y] x
```

```
  by(rule_tac Maxr_lemma, auto simp: rec_exec.simps
    mod_lemma power_lemma, auto simp: numeral_2_eq_2 numeral_3_eq_3)
 have Maxr loR [x, y] x = Maxr (λ args. 0 < rec_exec ?rR args) [x, y] x
  apply(simp add: rec_exec.simps mod_lemma power_lemma)
  apply(simp add: Maxr.simps loR.simps)
  done
 from h and this show rec_exec (rec_maxr ?rR) [x, y, x] =
  Maxr loR [x, y] x
  apply(simp)
  done
qed

lemma x_less_exp: ⟦y > Suc 0⟧ ⟹ x < y^x
proof(induct x)
 case (Suc x)
 then show ?case
  apply(cases x, simp, auto)
  apply(rule_tac y = y∗ y^(x−1) in le_less_trans, auto)
  done
qed simp


lemma uplimit_loR:
 assumes Suc 0 < x Suc 0 < y loR [x, y, xa]
 shows xa ≤ x
proof −
 have Suc 0 < x ⟹ Suc 0 < y ⟹ y ^ xa dvd x ⟹ xa ≤ x
  by (meson Suc_lessD le_less_trans nat_dvd_not_less nat_le_linear x_less_exp)
 thus ?thesis using assms by(auto simp: loR.simps)
qed

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]}
 apply(rule_tac Collect_cong, auto)
 apply(erule_tac uplimit_loR, simp, simp)
 done

lemma Maxr_lo: ⟦Suc 0 < x; Suc 0 < y⟧ ⟹
 Maxr loR [x, y] x = lo x y
 apply(simp add: Maxr.simps lo.simps, auto simp: uplimit_loR)
 by (meson uplimit_loR)+

lemma lo_lemma': ⟦Suc 0 < x; Suc 0 < y⟧ ⟹
 rec_exec rec_lo [x, y] = lo x y
 by(simp add: Maxr_lo rec_lo_Maxr_lor)

lemma lo_lemma'': ⟦¬ Suc 0 < x⟧ ⟹ rec_exec rec_lo [x, y] = lo x y
 apply(cases x, auto simp: rec_exec.simps rec_lo_def
   Let_def lo.simps)
 done
```

```
lemma lo_lemma''': ⟦¬ Suc 0 < y⟧ ⟹ rec_exec rec_lo [x, y] = lo x y
 apply(cases y, auto simp: rec_exec.simps rec_lo_def
   Let_def lo.simps)
 done
```

The correctness of *rec_lo*:

```
lemma lo_lemma: rec_exec rec_lo [x, y] = lo x y
 apply(cases Suc 0 < x ∧ Suc 0 < y)
  apply(auto simp: lo_lemma' lo_lemma'' lo_lemma''')
 done

fun lgR :: nat list ⇒ bool
 where
  lgR [x, y, u] = (y^u ≤ x)
```

lg specifies the *lg* function given on page 79 of Boolos's book. It is one of the two notions of integeral logarithmetic operation on that page. The other is *lo*.

```
fun lg :: nat ⇒ nat ⇒ nat
 where
  lg x y = (if x > 1 ∧ y > 1 ∧ {u. lgR [x, y, u]} ≠ {} then
             Max {u. lgR [x, y, u]}
          else 0)

declare lg.simps[simp del] lgR.simps[simp del]
```

rec_lg is the recursive function used to implement *lg*.

```
definition rec_lg :: recf
 where
  rec_lg = (let rec_lgR = Cn 3 rec_le
 [Cn 3 rec_power [id 3 1, id 3 2], id 3 0] in
 let conR1 = Cn 2 rec_conj [Cn 2 rec_less
               [Cn 2 (constn 1) [id 2 0], id 2 0],
                    Cn 2 rec_less [Cn 2 (constn 1)
                        [id 2 0], id 2 1]] in
 let conR2 = Cn 2 rec_not [conR1] in
      Cn 2 rec_add [Cn 2 rec_mult
           [conR1, Cn 2 (rec_maxr rec_lgR)
                  [id 2 0, id 2 1, id 2 0]],
                  Cn 2 rec_mult [conR2, Cn 2 (constn 0)
                         [id 2 0]]])

lemma lg_maxr: ⟦Suc 0 < x; Suc 0 < y⟧ ⟹
                rec_exec rec_lg [x, y] = Maxr lgR [x, y] x
proof(simp add: rec_exec.simps rec_lg_def Let_def)
 assume h: Suc 0 < x Suc 0 < y
 let ?rR = (Cn 3 rec_le [Cn 3 rec_power
          [recf.id 3 (Suc 0), recf.id 3 2], recf.id 3 0])
 have rec_exec (rec_maxr ?rR) ([x, y] @ [x])
```

$= Maxr\ ((\lambda\ args.\ 0 < rec\_exec\ ?rR\ args))\ [x, y]\ x$
proof(*rule Maxr_lemma*)
show *primerec* (*Cn 3 rec_le* [*Cn 3 rec_power*
[*recf.id 3* (*Suc 0*), *recf.id 3 2*], *recf.id 3 0*]) (*Suc* (*length* [*x*, *y*]))
apply(*auto simp*: *numeral_3_eq_3*)+
done
qed
moreover have *Maxr lgR* [*x*, *y*] *x* = *Maxr* ((λ *args*. *0* < *rec_exec ?rR args*)) [*x*, *y*] *x*
apply(*simp add*: *rec_exec.simps power_lemma*)
apply(*simp add*: *Maxr.simps lgR.simps*)
done
ultimately show *rec_exec* (*rec_maxr ?rR*) [*x*, *y*, *x*] = *Maxr lgR* [*x*, *y*] *x*
by *simp*
qed

lemma *lgR_ok*: ⟦*Suc 0* < *y*; *lgR* [*x*, *y*, *xa*]⟧ ⟹ *xa* ≤ *x*
apply(*auto simp add*: *lgR.simps*)
apply(*subgoal_tac y^xa* > *xa*, *simp*)
apply(*erule x_less_exp*)
done

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*]}
apply(*rule_tac Collect_cong*, *auto simp*:*lgR_ok*)
done

lemma *maxr_lg*: ⟦*Suc 0* < *x*; *Suc 0* < *y*⟧ ⟹ *Maxr lgR* [*x*, *y*] *x* = *lg x y*
apply(*auto simp add*: *lg.simps Maxr.simps*)
using *lgR_ok* **by** *blast*

lemma *lg_lemma′*: ⟦*Suc 0* < *x*; *Suc 0* < *y*⟧ ⟹ *rec_exec rec_lg* [*x*, *y*] = *lg x y*
apply(*simp add*: *maxr_lg lg_maxr*)
done

lemma *lg_lemma″*: ¬ *Suc 0* < *x* ⟹ *rec_exec rec_lg* [*x*, *y*] = *lg x y*
apply(*simp add*: *rec_exec.simps rec_lg_def Let_def lg.simps*)
done

lemma *lg_lemma‴*: ¬ *Suc 0* < *y* ⟹ *rec_exec rec_lg* [*x*, *y*] = *lg x y*
apply(*simp add*: *rec_exec.simps rec_lg_def Let_def lg.simps*)
done

The correctness of *rec_lg*.

lemma *lg_lemma*: *rec_exec rec_lg* [*x*, *y*] = *lg x y*
apply(*cases Suc 0* < *x* ∧ *Suc 0* < *y*, *auto simp*:
lg_lemma′ lg_lemma″ lg_lemma‴)
done

Entry sr i returns the *i*-th entry of a list of natural numbers encoded by number *sr* using Godel's coding.

fun *Entry* :: *nat* ⇒ *nat* ⇒ *nat*
where
Entry sr i = *lo sr* (*Pi* (*Suc i*))

rec_entry is the recursive function used to implement *Entry*.

definition *rec_entry*:: *recf*
where
rec_entry = *Cn 2 rec_lo* [*id 2 0*, *Cn 2 rec_pi* [*Cn 2 s* [*id 2 1*]]]

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

The correctness of *rec_entry*.

lemma *entry_lemma*: *rec_exec rec_entry* [*str*, *i*] = *Entry str i*
by(*simp add*: *rec_entry_def rec_exec.simps lo_lemma pi_lemma*)

25.2 The construction of F

Using the auxilliary functions obtained in last section, we are going to contruct the function *F*, which is an interpreter of Turing Machines.

fun *listsum2* :: *nat list* ⇒ *nat* ⇒ *nat*
where
listsum2 xs 0 = *0*
| *listsum2 xs* (*Suc n*) = *listsum2 xs n* + *xs* ! *n*

fun *rec_listsum2* :: *nat* ⇒ *nat* ⇒ *recf*
where
rec_listsum2 vl 0 = *Cn vl z* [*id vl 0*]
| *rec_listsum2 vl* (*Suc n*) = *Cn vl rec_add* [*rec_listsum2 vl n*, *id vl n*]

declare *listsum2.simps*[*simp del*] *rec_listsum2.simps*[*simp del*]

lemma *listsum2_lemma*: ⟦*length xs* = *vl*; *n* ≤ *vl*⟧ ⟹
rec_exec (*rec_listsum2 vl n*) *xs* = *listsum2 xs n*
apply(*induct n*, *simp_all*)
apply(*simp_all add*: *rec_exec.simps rec_listsum2.simps listsum2.simps*)
done

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*ˆ(*xs* ! *n* + *dbound*) − *2*ˆ*dbound*))

fun *rec_strt′* :: *nat* ⇒ *nat* ⇒ *recf*
where
rec_strt′ vl 0 = *Cn vl z* [*id vl 0*]
| *rec_strt′ vl* (*Suc n*) = (*let rec_dbound* =
Cn vl rec_add [*rec_listsum2 vl n*, *Cn vl* (*constn n*) [*id vl 0*]]
in Cn vl rec_add [*rec_strt′ vl n*, *Cn vl rec_minus*

```
  [Cn vl rec_power [Cn vl (constn 2) [id vl 0], Cn vl rec_add
  [id vl (n), rec_dbound]],
  Cn vl rec_power [Cn vl (constn 2) [id vl 0], rec_dbound]]])

declare strt'.simps[simp del] rec_strt'.simps[simp del]

lemma strt'_lemma: ⟦length xs = vl; n ≤ vl⟧ ⟹
  rec_exec (rec_strt' vl n) xs = strt' xs n
  apply(induct n)
   apply(simp_all add: rec_exec.simps rec_strt'.simps strt'.simps
    Let_def power_lemma listsum2_lemma)
  done
```

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
  apply(induct vl arbitrary: xs, simp, auto simp: rec_exec.simps)
  apply(rename_tac vl xs)
  apply(subgoal_tac ∃ ys y. xs = ys @ [y], auto)
proof -
  fix ys y
  assume ind: ⋀xs. length xs = length (ys::nat list) ⟹
    map ((λa. rec_exec a xs) ∘ (λi. Cn (length ys) s
      [recf.id (length ys) (i)])) [0..<length ys] = map Suc xs
  show
   map ((λa. rec_exec a (ys @ [y])) ∘ (λi. Cn (Suc (length ys)) s
  [recf.id (Suc (length ys)) (i)])) [0..<length ys] = map Suc ys
  proof -
   have map ((λa. rec_exec a ys) ∘ (λi. Cn (length ys) s
      [recf.id (length ys) (i)])) [0..<length ys] = map Suc ys
    apply(rule_tac ind, simp)
    done
   moreover have
```

```
    map ((λa. rec_exec a (ys @ [y])) ∘ (λi. Cn (Suc (length ys)) s
        [recf.id (Suc (length ys)) (i)])) [0..<length ys]
      = map ((λa. rec_exec a ys) ∘ (λi. Cn (length ys) s
            [recf.id (length ys) (i)])) [0..<length ys]
    apply(rule_tac map_ext, auto simp: rec_exec.simps nth_append)
    done
   ultimately show ?thesis
    by simp
 qed
next
 fix vl xs
 assume length xs = Suc vl
 thus ∃ ys y. xs = ys @ [y]
  apply(rule_tac x = butlast xs in exI, rule_tac x = last xs in exI)
  apply(subgoal_tac xs ≠ [], auto)
  done
qed
```

The correctness of *rec_strt*.

```
lemma strt_lemma: length xs = vl ⟹
 rec_exec (rec_strt vl) xs = strt xs
 apply(simp add: strt.simps rec_exec.simps strt'_lemma)
 apply(subgoal_tac (map ((λa. rec_exec a xs) ∘ (λi. Cn vl s [recf.id vl (i)])) [0..<vl])
            = map Suc xs, auto)
 apply(rule map_s_lemma, simp)
 done
```

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
 by(simp add: rec_exec.simps rec_scan_def mod_lemma)

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*
 apply(*cases i*, *force*)
 apply(*cases i* − *1*, *force*)
 apply(*cases i* − *1* − *1*, *force*)
 by(*cases i* − *1* − *1* − *1*, *auto simp*:*numeral*)

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(simp only: rec_newleft_def Let_def)
  let ?rgs = [Cn 3 rec_newleft0 [recf.id 3 0, recf.id 3 1], Cn 3 rec_newleft2
       [recf.id 3 0, recf.id 3 1], Cn 3 rec_newleft3 [recf.id 3 0, recf.id 3 1], recf.id 3 0]
  let ?rrs =
    [Cn 3 rec_disj [Cn 3 rec_eq [recf.id 3 2, Cn 3 (constn 0)
     [recf.id 3 0]], Cn 3 rec_eq [recf.id 3 2, Cn 3 (constn 1) [recf.id 3 0]]]],
     Cn 3 rec_eq [recf.id 3 2, Cn 3 (constn 2) [recf.id 3 0]],
     Cn 3 rec_eq [recf.id 3 2, Cn 3 (constn 3) [recf.id 3 0]],
     Cn 3 rec_less [Cn 3 (constn 3) [recf.id 3 0], recf.id 3 2]]
  have k1: rec_exec (rec_embranch (zip ?rgs ?rrs)) [p, r, a]
                  = Embranch (zip (map rec_exec ?rgs) (map (λr args. 0 < rec_exec r args) ?rrs))
[p, r, a]
    apply(rule_tac embranch_lemma )
      apply(auto simp: numeral_3_eq_3 numeral_2_eq_2 rec_newleft0_def
      rec_newleft1_def rec_newleft2_def rec_newleft3_def)+
    apply(cases a = 0 ∨ a = 1, rule_tac x = 0 in exI)
     prefer 2
     apply(cases a = 2, rule_tac x = Suc 0 in exI)
      prefer 2
      apply(cases a = 3, rule_tac x = 2 in exI)
       prefer 2
       apply(cases a > 3, rule_tac x = 3 in exI, auto)
           apply(auto simp: rec_exec.simps)
      apply(erule_tac [!] Suc_Suc_Suc_Suc_induct, auto simp: rec_exec.simps)
    done
  have k2: Embranch (zip (map rec_exec ?rgs) (map (λr args. 0 < rec_exec r args) ?rrs)) [p, r,
a] = newleft p r a
    apply(simp add: Embranch.simps)
    apply(simp add: rec_exec.simps)
    apply(auto simp: newleft.simps rec_newleft0_def rec_exec.simps
      rec_newleft1_def rec_newleft2_def rec_newleft3_def)
    done
  from k1 and k2 show
    rec_exec (rec_embranch (zip ?rgs ?rrs)) [p, r, a] = newleft p r a
    by simp
qed
```

The *newrght* function is one similar to *newleft*, but used to compute the right number.

```
fun newrght :: nat ⇒ nat ⇒ nat ⇒ nat
  where
    newrght p r a  = (if a = 0 then newrgt0 [p, r]
               else if a = 1 then newrgt1 [p, r]
               else if a = 2 then newrgt2 [p, r]
```

else if a = 3 then newrgt3 [p, r]
else r)

rec_newrght is the recursive function used to implement *newrgth*.

definition *rec_newrght* :: *recf*
where
rec_newrght =
(let g0 = Cn 3 rec_newrgt0 [id 3 0, id 3 1] in
let g1 = Cn 3 rec_newrgt1 [id 3 0, id 3 1] in
let g2 = Cn 3 rec_newrgt2 [id 3 0, id 3 1] in
let g3 = Cn 3 rec_newrgt3 [id 3 0, id 3 1] in
let g4 = id 3 1 in
let r0 = Cn 3 rec_eq [id 3 2, Cn 3 (constn 0) [id 3 0]] in
let r1 = Cn 3 rec_eq [id 3 2, Cn 3 (constn 1) [id 3 0]] in
let r2 = Cn 3 rec_eq [id 3 2, Cn 3 (constn 2) [id 3 0]] in
let r3 = Cn 3 rec_eq [id 3 2, Cn 3 (constn 3) [id 3 0]] in
let r4 = Cn 3 rec_less [Cn 3 (constn 3) [id 3 0], id 3 2] in
let gs = [g0, g1, g2, g3, g4] in
let rs = [r0, r1, r2, r3, r4] in
rec_embranch (zip gs rs))
declare *newrght.simps[simp del]*

lemma *numeral_4_eq_4*: *4 = Suc 3*
by *auto*

lemma *Suc_5_induct*:
$[\![i < Suc\ (Suc\ (Suc\ (Suc\ (Suc\ 0))))$; $i = 0 \Longrightarrow P\ 0$;
$i = 1 \Longrightarrow P\ 1$; $i = 2 \Longrightarrow P\ 2$; $i = 3 \Longrightarrow P\ 3$; $i = 4 \Longrightarrow P\ 4]\!] \Longrightarrow P\ i$
apply(*cases i, force*)
apply(*cases i−1, force*)
apply(*cases i−1−1*)
using *less_2_cases numeral* **by** *auto*

lemma *primerec_rec_scan_1[intro]*: *primerec rec_scan (Suc 0)*
apply(*auto simp*: *rec_scan_def*, *auto*)
done

The correctness of *rec_newrght*.

lemma *newrght_lemma*: *rec_exec rec_newrght [p, r, a] = newrght p r a*
proof(*simp only*: *rec_newrght_def Let_def*)
let *?gs′ = [newrgt0, newrgt1, newrgt2, newrgt3, λ zs. zs ! 1]*
let *?r0 = λ zs. zs ! 2 = 0*
let *?r1 = λ zs. zs ! 2 = 1*
let *?r2 = λ zs. zs ! 2 = 2*
let *?r3 = λ zs. zs ! 2 = 3*
let *?r4 = λ zs. zs ! 2 > 3*
let *?gs = map (λ g. (λ zs. g [zs ! 0, zs ! 1])) ?gs′*
let *?rs = [?r0, ?r1, ?r2, ?r3, ?r4]*
let *?rgs =*

```
  [Cn 3 rec_newrgt0 [recf.id 3 0, recf.id 3 1],
  Cn 3 rec_newrgt1 [recf.id 3 0, recf.id 3 1],
   Cn 3 rec_newrgt2 [recf.id 3 0, recf.id 3 1],
    Cn 3 rec_newrgt3 [recf.id 3 0, recf.id 3 1], recf.id 3 1]
 let ?rrs =
  [Cn 3 rec_eq [recf.id 3 2, Cn 3 (constn 0) [recf.id 3 0]], Cn 3 rec_eq [recf.id 3 2,
  Cn 3 (constn 1) [recf.id 3 0]], Cn 3 rec_eq [recf.id 3 2, Cn 3 (constn 2) [recf.id 3 0]],
   Cn 3 rec_eq [recf.id 3 2, Cn 3 (constn 3) [recf.id 3 0]],
     Cn 3 rec_less [Cn 3 (constn 3) [recf.id 3 0], recf.id 3 2]]

 have k1: rec_exec (rec_embranch (zip ?rgs ?rrs)) [p, r, a]
  = Embranch (zip (map rec_exec ?rgs) (map (λr args. 0 < rec_exec r args) ?rrs)) [p, r, a]
  apply(rule_tac embranch_lemma)
      apply(auto simp: numeral_3_eq_3 numeral_2_eq_2 rec_newrgt0_def
      rec_newrgt1_def rec_newrgt2_def rec_newrgt3_def)+
  apply(cases a = 0, rule_tac x = 0 in exI)
   prefer 2
   apply(cases a = 1, rule_tac x = Suc 0 in exI)
    prefer 2
    apply(cases a = 2, rule_tac x = 2 in exI)
     prefer 2
     apply(cases a = 3, rule_tac x = 3 in exI)
      prefer 2
      apply(cases a > 3, rule_tac x = 4 in exI, auto simp: rec_exec.simps)
      apply(erule_tac [!] Suc_5_induct, auto simp: rec_exec.simps)
  done
 have k2: Embranch (zip (map rec_exec ?rgs)
  (map (λr args. 0 < rec_exec r args) ?rrs)) [p, r, a] = newrght p r a
  apply(auto simp:Embranch.simps rec_exec.simps)
      apply(auto simp: newrght.simps rec_newrgt3_def rec_newrgt2_def
      rec_newrgt1_def rec_newrgt0_def rec_exec.simps
      scan_lemma)
  done
 from k1 and k2 show
  rec_exec (rec_embranch (zip ?rgs ?rrs)) [p, r, a] =
                          newrght p r a by simp
qed

declare Entry.simps[simp del]
```

The *actn* function given on page 92 of B book, which is used to fetch Turing Machine intructions. In *actn m q r*, *m* is the Godel coding of a Turing Machine, *q* is the current state of Turing Machine, *r* is the right number of Turing Machine tape.

```
fun actn :: nat ⇒ nat ⇒ nat ⇒ nat
 where
  actn m q r = (if q ≠ 0 then Entry m (4∗(q − 1) + 2 ∗ scan r)
          else 4)
```

rec_actn is the recursive function used to implement *actn*

```
definition rec_actn :: recf
```

```
 where
  rec_actn =
 Cn 3 rec_add [Cn 3 rec_mult
     [Cn 3 rec_entry [id 3 0, Cn 3 rec_add [Cn 3 rec_mult
                     [Cn 3 (constn 4) [id 3 0],
           Cn 3 rec_minus [id 3 1, Cn 3 (constn 1) [id 3 0]]],
             Cn 3 rec_mult [Cn 3 (constn 2) [id 3 0],
               Cn 3 rec_scan [id 3 2]]]],
        Cn 3 rec_noteq [id 3 1, Cn 3 (constn 0) [id 3 0]]],
                    Cn 3 rec_mult [Cn 3 (constn 4) [id 3 0],
         Cn 3 rec_eq [id 3 1, Cn 3 (constn 0) [id 3 0]]]]
```

The correctness of *actn*.

```
lemma actn_lemma: rec_exec rec_actn [m, q, r] = actn m q r
 by(auto simp: rec_actn_def rec_exec.simps entry_lemma scan_lemma)

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
 by(auto simp:  rec_exec.simps entry_lemma scan_lemma rec_newstat_def)

declare newstat.simps[simp del] actn.simps[simp del]
```

code the configuration

```
fun trpl :: nat ⇒ nat ⇒ nat ⇒ nat
 where
  trpl p q r = (Pi 0)^p ∗ (Pi 1)^q ∗ (Pi 2)^r

definition rec_trpl :: recf
 where
  rec_trpl = Cn 3 rec_mult [Cn 3 rec_mult
     [Cn 3 rec_power [Cn 3 (constn (Pi 0)) [id 3 0], id 3 0],
      Cn 3 rec_power [Cn 3 (constn (Pi 1)) [id 3 0], id 3 1]],
      Cn 3 rec_power [Cn 3 (constn (Pi 2)) [id 3 0], id 3 2]]
declare trpl.simps[simp del]
```

```
lemma trpl_lemma: rec_exec rec_trpl [p, q, r] = trpl p q r
  by(auto simp: rec_trpl_def rec_exec.simps power_lemma trpl.simps)
```

left, stat, rght: decode func

```
fun left :: nat ⇒ nat
  where
    left c = lo c (Pi 0)

fun stat :: nat ⇒ nat
  where
    stat c = lo c (Pi 1)

fun rght :: nat ⇒ nat
  where
    rght c = lo c (Pi 2)

fun inpt :: nat ⇒ nat list ⇒ nat
  where
    inpt m xs = trpl 0 1 (strt xs)

fun newconf :: nat ⇒ nat ⇒ nat
  where
    newconf m c = trpl (newleft (left c) (rght c)
                     (actn m (stat c) (rght c)))
                     (newstat m (stat c) (rght c))
                     (newrght (left c) (rght c)
                          (actn m (stat c) (rght c)))

declare left.simps[simp del] stat.simps[simp del] rght.simps[simp del]
  inpt.simps[simp del] newconf.simps[simp del]

definition rec_left :: recf
  where
    rec_left = Cn 1 rec_lo [id 1 0, constn (Pi 0)]

definition rec_right :: recf
  where
    rec_right = Cn 1 rec_lo [id 1 0, constn (Pi 2)]

definition rec_stat :: recf
  where
    rec_stat = Cn 1 rec_lo [id 1 0, constn (Pi 1)]

definition rec_inpt :: nat ⇒ recf
  where
    rec_inpt vl = Cn vl rec_trpl
               [Cn vl (constn 0) [id vl 0],
                Cn vl (constn 1) [id vl 0],
                Cn vl (rec_strt (vl − 1))
                     (map (λ i. id vl (i)) [1..<vl])]
```

```
lemma left_lemma: rec_exec rec_left [c] = left c
 by(simp add: rec_exec.simps rec_left_def left.simps lo_lemma)

lemma right_lemma: rec_exec rec_right [c] = rght c
 by(simp add: rec_exec.simps rec_right_def rght.simps lo_lemma)

lemma stat_lemma: rec_exec rec_stat [c] = stat c
 by(simp add: rec_exec.simps rec_stat_def stat.simps lo_lemma)

declare rec_strt.simps[simp del] strt.simps[simp del]

lemma map_cons_eq:
 (map ((λa. rec_exec a (m # xs)) ∘
  (λi. recf.id (Suc (length xs)) (i)))
      [Suc 0..<Suc (length xs)])
     = map (λ i. xs ! (i − 1)) [Suc 0..<Suc (length xs)]
 apply(rule map_ext, auto)
  apply(auto simp: rec_exec.simps nth_append nth_Cons split: nat.split)
 done

lemma list_map_eq:
 vl = length (xs::nat list) ⟹ map (λ i. xs ! (i − 1))
                              [Suc 0..<Suc vl] = xs
proof(induct vl arbitrary: xs)
 case (Suc vl)
 then show ?case
  apply(subgoal_tac ∃ ys y. xs = ys @ [y], auto)
 proof −
  fix ys y
  assume ind:
   ⋀xs. length (ys::nat list) = length (xs::nat list) ⟹
      map (λi. xs ! (i − Suc 0)) [Suc 0..<length xs] @
                 [xs ! (length xs − Suc 0)] = xs
   and h: Suc 0 ≤ length (ys::nat list)
  have map (λi. ys ! (i − Suc 0)) [Suc 0..<length ys] @
                   [ys ! (length ys − Suc 0)] = ys
   apply(rule_tac ind, simp)
   done
  moreover have
   map (λi. (ys @ [y]) ! (i − Suc 0)) [Suc 0..<length ys]
   = map (λi. ys ! (i − Suc 0)) [Suc 0..<length ys]
   apply(rule map_ext)
   using h
   apply(auto simp: nth_append)
   done
  ultimately show map (λi. (ys @ [y]) ! (i − Suc 0))
    [Suc 0..<length ys] @ [(ys @ [y]) ! (length ys − Suc 0)] = ys
   apply(simp del: map_eq_conv add: nth_append, auto)
   using h
```

```
    apply(simp)
    done
  next
   fix vl xs
   assume Suc vl = length (xs::nat list)
   thus ∃ys y. xs = ys @ [y]
    apply(rule_tac x = butlast xs in exI,
      rule_tac x = last xs in exI)
    apply(cases xs ≠ [], auto)
    done
  qed
qed simp

lemma nonempty_listE:
  Suc 0 ≤ length xs ⟹
    (map ((λa. rec_exec a (m # xs)) ∘
      (λi. recf.id (Suc (length xs)) (i)))
        [Suc 0..<length xs] @ [(m # xs) ! length xs]) = xs
  using map_cons_eq[of m xs]
  apply(simp del: map_eq_conv add: rec_exec.simps)
  using list_map_eq[of length xs xs]
  apply(simp)
  done

lemma inpt_lemma:
 ⟦Suc (length xs) = vl⟧ ⟹
        rec_exec (rec_inpt vl) (m # xs) = inpt m xs
  apply(auto simp: rec_exec.simps rec_inpt_def
    trpl_lemma inpt.simps strt_lemma)
   apply(subgoal_tac
    (map ((λa. rec_exec a (m # xs)) ∘
      (λi. recf.id (Suc (length xs)) (i)))
       [Suc 0..<length xs] @ [(m # xs) ! length xs]) = xs, simp)
   apply(auto elim:nonempty_listE, cases xs, auto)
  done

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
 by(auto simp: rec_newconf_def rec_exec.simps
   trpl_lemma newleft_lemma left_lemma
   right_lemma stat_lemma newrght_lemma actn_lemma
   newstat_lemma newconf.simps)

declare newconf_lemma[simp]
```

conf m r k computes the TM configuration after *k* steps of execution of TM coded as *m* starting from the initial configuration where the left number equals *0*, right number equals *r*.

```
fun conf :: nat ⇒ nat ⇒ nat ⇒ nat
 where
  conf m r 0 = trpl 0 (Suc 0) r
 | conf m r (Suc t) = newconf m (conf m r t)

declare conf.simps[simp del]
```

conf is implemented by the following recursive function *rec_conf*.

```
definition rec_conf :: recf
 where
  rec_conf = Pr 2 (Cn 2 rec_trpl [Cn 2 (constn 0) [id 2 0], Cn 2 (constn (Suc 0)) [id 2 0], id 2
1])
        (Cn 4 rec_newconf [id 4 0, id 4 3])

lemma conf_step:
 rec_exec rec_conf [m, r, Suc t] =
    rec_exec rec_newconf [m, rec_exec rec_conf [m, r, t]]
proof -
 have rec_exec rec_conf ([m, r] @ [Suc t]) =
     rec_exec rec_newconf [m, rec_exec rec_conf [m, r, t]]
  by(simp only: rec_conf_def rec_pr_Suc_simp_rewrite,
    simp add: rec_exec.simps)
 thus rec_exec rec_conf [m, r, Suc t] =
       rec_exec rec_newconf [m, rec_exec rec_conf [m, r, t]]
  by simp
qed
```

The correctness of *rec_conf*.

```
lemma conf_lemma:
 rec_exec rec_conf [m, r, t] = conf m r t
 by (induct t)
  (auto simp add: rec_conf_def rec_exec.simps conf.simps inpt_lemma trpl_lemma)
```

NSTD c returns true if the configureation coded by *c* is no a stardard final configuration.

```
fun NSTD :: nat ⇒ bool
  where
   NSTD c = (stat c ≠ 0 ∨ left c ≠ 0 ∨
         rght c ≠ 2^(lg (rght c + 1) 2) − 1 ∨ rght c = 0)
```

rec_NSTD is the recursive function implementing *NSTD*.

```
definition rec_NSTD :: recf
  where
   rec_NSTD =
    Cn 1 rec_disj [
       Cn 1 rec_disj [
         Cn 1 rec_disj
           [Cn 1 rec_noteq [rec_stat, constn 0],
            Cn 1 rec_noteq [rec_left, constn 0]] ,
          Cn 1 rec_noteq [rec_right,
                    Cn 1 rec_minus [Cn 1 rec_power
                      [constn 2, Cn 1 rec_lg
                        [Cn 1 rec_add
                         [rec_right, constn 1],
                              constn 2]], constn 1]]],
          Cn 1 rec_eq [rec_right, constn 0]]

lemma NSTD_lemma1: rec_exec rec_NSTD [c] = Suc 0 ∨
               rec_exec rec_NSTD [c] = 0
  by(simp add: rec_exec.simps rec_NSTD_def)

declare NSTD.simps[simp del]
lemma NSTD_lemma2': (rec_exec rec_NSTD [c] = Suc 0) ⟹ NSTD c
  apply(simp add: rec_exec.simps rec_NSTD_def stat_lemma left_lemma
     lg_lemma right_lemma power_lemma NSTD.simps)
  apply(auto)
  apply(cases 0 < left c, simp, simp)
  done

lemma NSTD_lemma2'':
  NSTD c ⟹ (rec_exec rec_NSTD [c] = Suc 0)
  apply(simp add: rec_exec.simps rec_NSTD_def stat_lemma
     left_lemma lg_lemma right_lemma power_lemma NSTD.simps)
  apply(auto split: if_splits)
  done
```

The correctness of *NSTD*.

```
lemma NSTD_lemma2: (rec_exec rec_NSTD [c] = Suc 0) = NSTD c
  using NSTD_lemma1
  apply(auto intro: NSTD_lemma2' NSTD_lemma2'')
  done

fun nstd :: nat ⇒ nat
  where
   nstd c = (if NSTD c then 1 else 0)
```

lemma *nstd_lemma*: *rec_exec rec_NSTD* $[c]$ = *nstd c*
using *NSTD_lemma1*
apply(*simp add*: *NSTD_lemma2*, *auto*)
done

nonstep m r t means afer *t* steps of execution, the TM coded by *m* is not at a stardard final configuration.

fun *nonstop* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *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*
apply(*simp add*: *rec_exec.simps rec_nonstop_def nstd_lemma conf_lemma*)
done

rec_halt is the recursive function calculating the steps a TM needs to execute before to reach a stardard final configuration. This recursive function is the only one using *Mn* combinator. So it is the only non-primitive recursive function needs to be used in the construction of the universal function *F*.

definition *rec_halt* :: *recf*
where
rec_halt = *Mn* (*Suc* (*Suc 0*)) (*rec_nonstop*)

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

The lemma relates the interpreter of primitive functions with the calculation relation of general recursive functions.

declare *numeral_2_eq_2*[*simp*] *numeral_3_eq_3*[*simp*]

lemma *primerec_rec_right_1*[*intro*]: *primerec rec_right* (*Suc 0*)
by(*auto simp*: *rec_right_def rec_lo_def Let_def*;*force*)

lemma *primerec_rec_pi_helper*:
∀ *i*<*Suc* (*Suc 0*). *primerec* ([*recf.id* (*Suc 0*) *0*, *recf.id* (*Suc 0*) *0*] ! *i*) (*Suc 0*)
by *fastforce*

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)
 by(auto simp: rec_dummyfac_def;force)

lemma primerec_rec_pi_1[intro]:  primerec rec_pi (Suc 0)
 apply(simp add: rec_pi_def rec_dummy_pi_def
    rec_np_def rec_fac_def rec_prime_def
    rec_Minr.simps Let_def get_fstn_args.simps
    arity.simps
    rec_all.simps rec_sigma.simps rec_accum.simps)
 apply(tactic ⟪ resolve_tac @{context} [@{thm prime_cn},  @{thm prime_pr}] 1⟫
    ;(simp add:primerec_rec_pi_helpers primrec_dummyfac)?)+
 by fastforce+

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)))
      apply(simp_all add: rec_newleft_def rec_embranch.simps rec_left_def rec_lo_def rec_entry_def
    rec_actn_def Let_def arity.simps rec_newleft0_def rec_stat_def rec_newstat_def
    rec_newleft1_def rec_newleft2_def rec_newleft3_def rec_trpl_def)
        apply(tactic ⟪ resolve_tac @{context} [@{thm prime_cn},
  @{thm prime_id}, @{thm prime_pr}] 1⟫;force)+
 done

lemma primerec_rec_newrght[intro]: primerec rec_newrght (Suc (Suc (Suc 0)))
 apply(simp add: rec_newrght_def rec_embranch.simps
    Let_def arity.simps rec_newrgt0_def
    rec_newrgt1_def rec_newrgt2_def rec_newrgt3_def)
 apply(tactic ⟪ resolve_tac @{context} [@{thm prime_cn},
  @{thm prime_id}, @{thm prime_pr}] 1⟫;force)+
 done

lemma primerec_rec_newconf[intro]: primerec rec_newconf (Suc (Suc 0))
 apply(simp add: rec_newconf_def)
 by(tactic ⟪ resolve_tac @{context} [@{thm prime_cn},
  @{thm prime_id}, @{thm prime_pr}] 1⟫;force)
```

```
lemma primerec_rec_conf[intro]: primerec rec_conf (Suc (Suc (Suc 0)))
 apply(simp add: rec_conf_def)
 by(tactic ⟪ resolve_tac @{context} [@{thm prime_cn},
   @{thm prime_id}, @{thm prime_pr}] 1⟫;force simp: numeral)

lemma primerec_recs2[intro]:
 primerec rec_lg (Suc (Suc 0))
 primerec rec_nonstop (Suc (Suc (Suc 0)))
  apply(simp_all add: rec_lg_def rec_nonstop_def rec_NSTD_def rec_stat_def
    rec_lo_def Let_def rec_left_def rec_right_def rec_newconf_def
    rec_newstat_def)
 by(tactic ⟪ resolve_tac @{context} [@{thm prime_cn},
   @{thm prime_id}, @{thm prime_pr}] 1⟫;fastforce)+

lemma primerec_terminate:
 ⟦primerec f x; length xs = x⟧ ⟹ terminate f xs
proof(induct arbitrary: xs rule: primerec.induct)
 fix xs
 assume length (xs::nat list) = Suc 0  thus terminate z xs
  by(cases xs, auto intro: termi_z)
next
 fix xs
 assume length (xs::nat list) = Suc 0 thus terminate s xs
  by(cases xs, auto intro: termi_s)
next
 fix n m xs
 assume n < m length (xs::nat list) = m  thus terminate (id m n) xs
  by(erule_tac termi_id, simp)
next
 fix f k gs m n xs
 assume ind: ∀ i<length gs. primerec (gs ! i) m ∧ (∀ x. length x = m ⟶ terminate (gs ! i) x)
  and ind2: ⋀ xs. length xs = k ⟹ terminate f xs
  and h: primerec f k  length gs = k m = n length (xs::nat list) = m
 have terminate f (map (λg. rec_exec g xs) gs)
  using ind2[of (map (λg. rec_exec g xs) gs)] h
  by simp
 moreover have ∀ g∈set gs. terminate g xs
  using ind h
  by(auto simp: set_conv_nth)
 ultimately show terminate (Cn n f gs) xs
  using h
  by(rule_tac termi_cn, auto)
next
 fix f n g m xs
 assume ind1: ⋀xs. length xs = n ⟹ terminate f xs
  and ind2: ⋀xs. length xs = Suc (Suc n) ⟹ terminate g xs
  and h: primerec f n  primerec g (Suc (Suc n))  m = Suc n length (xs::nat list) = m
 have ∀ y<last xs. terminate g (butlast xs @ [y, rec_exec (Pr n f g) (butlast xs @ [y])])
  using h ind2 by(auto)
 moreover have terminate f (butlast xs)
```

```
  using ind1[of butlast xs] h
  by simp
 moreover have length (butlast xs) = n
  using h by simp
 ultimately have terminate (Pr n f g) (butlast xs @ [last xs])
  by(rule_tac termi_pr, simp_all)
 thus terminate (Pr n f g) xs
  using h
  by(cases xs = [], auto)
qed
```

The following lemma gives the correctness of *rec_halt*. It says: if *rec_halt* calculates that the TM coded by *m* will reach a standard final configuration after *t* steps of execution, then it is indeed so.

F: universal machine

valu r extracts computing result out of the right number *r*.

```
fun valu :: nat ⇒ nat
 where
  valu r = (lg (r + 1) 2) − 1
```

rec_valu is the recursive function implementing *valu*.

```
definition rec_valu :: recf
 where
  rec_valu = Cn 1 rec_minus [Cn 1 rec_lg [s, constn 2], constn 1]
```

The correctness of *rec_valu*.

```
lemma value_lemma: rec_exec rec_valu [r] = valu r
 by(simp add: rec_exec.simps rec_valu_def lg_lemma)

lemma primerec_rec_valu_1[intro]: primerec rec_valu (Suc 0)
 unfolding rec_valu_def
 apply(rule prime_cn[of _ Suc (Suc 0)])
 by auto auto

declare valu.simps[simp del]
```

The definition of the universal function *rec_F*.

```
definition rec_F :: recf
 where
  rec_F = Cn (Suc (Suc 0)) rec_valu [Cn (Suc (Suc 0)) rec_right [Cn (Suc (Suc 0))
 rec_conf ([id (Suc (Suc 0)) 0, id (Suc (Suc 0)) (Suc 0), rec_halt])]]

lemma terminate_halt_lemma:
 ⟦rec_exec rec_nonstop ([m, r] @ [t]) = 0;
  ∀ i<t. 0 < rec_exec rec_nonstop ([m, r] @ [i])⟧ ⟹ terminate rec_halt [m, r]
 apply(simp add: rec_halt_def)
 apply(rule termi_mn, auto)
 by(rule primerec_terminate; auto)+
```

The correctness of *rec_F*, halt case.

```
lemma F_lemma: rec_exec rec_halt [m, r] = t ⟹ rec_exec rec_F [m, r] = (valu (rght (conf m r
t)))
 by(simp add: rec_F_def rec_exec.simps value_lemma right_lemma conf_lemma halt_lemma)

lemma terminate_F_lemma: terminate rec_halt [m, r] ⟹ terminate rec_F [m, r]
 apply(simp add: rec_F_def)
 apply(rule termi_cn, auto)
  apply(rule primerec_terminate, auto)
 apply(rule termi_cn, auto)
  apply(rule primerec_terminate, auto)
 apply(rule termi_cn, auto)
   apply(rule primerec_terminate, auto)
  apply(rule termi_id;force)
 apply(rule termi_id;force)
 done
```

The correctness of *rec_F*, nonhalt case.

25.3 Coding function of TMs

The purpose of this section is to get the coding function of Turing Machine, which is going to be named *code*.

```
fun bl2nat :: cell list ⇒ nat ⇒ nat
 where
  bl2nat [] n = 0
 | bl2nat (Bk#bl) n = bl2nat bl (Suc n)
 | bl2nat (Oc#bl) n = 2^n + bl2nat bl (Suc n)

fun bl2wc :: cell list ⇒ nat
 where
  bl2wc xs = bl2nat xs 0

fun trpl_code :: config ⇒ nat
 where
  trpl_code (st, l, r) = trpl (bl2wc l) st (bl2wc r)

declare bl2nat.simps[simp del] bl2wc.simps[simp del]
 trpl_code.simps[simp del]

fun action_map :: action ⇒ nat
 where
  action_map W0 = 0
 | action_map W1 = 1
 | action_map L = 2
 | action_map R = 3
 | action_map Nop = 4

fun action_map_iff :: nat ⇒ action
```

```
  where
    action_map_iff (0::nat) = W0
  | action_map_iff (Suc 0) = W1
  | action_map_iff (Suc (Suc 0)) = L
  | action_map_iff (Suc (Suc (Suc 0))) = R
  | action_map_iff n = Nop

fun block_map :: cell ⇒ nat
  where
    block_map Bk = 0
  | block_map Oc = 1

fun godel_code' :: nat list ⇒ nat ⇒ nat
  where
    godel_code' [] n = 1
  | godel_code' (x#xs) n = (Pi n)^x * godel_code' xs (Suc n)

fun godel_code :: nat list ⇒ nat
  where
    godel_code xs = (let lh = length xs in
               2^lh * (godel_code' xs (Suc 0)))

fun modify_tprog :: instr list ⇒ nat list
  where
    modify_tprog [] =  []
  | modify_tprog ((ac, ns)#nl) = action_map ac # ns # modify_tprog nl
```

code tp gives the Godel coding of TM program *tp*.

```
fun code :: instr list ⇒ nat
  where
    code tp = (let nl = modify_tprog tp in
           godel_code nl)
```

25.4 Relating interperter functions to the execution of TMs

```
lemma bl2wc_0[simp]: bl2wc [] = 0 by(simp add: bl2wc.simps bl2nat.simps)

lemma fetch_action_map_4[simp]: ⟦fetch tp 0 b = (nact, ns)⟧ ⟹ action_map nact = 4
  apply(simp add: fetch.simps)
  done

lemma Pi_gr_1[simp]: Pi n > Suc 0
proof(induct n, auto simp: Pi.simps Np.simps)
  fix n
  let ?setx = {y. y ≤ Suc (Pi n!) ∧ Pi n < y ∧ Prime y}
  have finite ?setx by auto
  moreover have ?setx ≠ {}
    using prime_ex[of Pi n]
    apply(auto)
    done
```

```
  ultimately show Suc 0 < Min ?setx
    apply(simp add: Min_gr_iff)
    apply(auto simp: Prime.simps)
    done
qed

lemma Pi_not_0[simp]: Pi n > 0
  using Pi_gr_1[of n]
  by arith

declare godel_code.simps[simp del]

lemma godel_code'_nonzero[simp]: 0 < godel_code' nl n
  apply(induct nl arbitrary: n)
   apply(auto simp: godel_code'.simps)
  done

lemma godel_code_great: godel_code nl > 0
  apply(simp add: godel_code.simps)
  done

lemma godel_code_eq_1: (godel_code nl = 1) = (nl = [])
  apply(auto simp: godel_code.simps)
  done

lemma godel_code_1_iff[elim]:
  ⟦i < length nl; ¬ Suc 0 < godel_code nl⟧ ⟹ nl ! i = 0
  using godel_code_great[of nl] godel_code_eq_1[of nl]
  apply(simp)
  done

lemma prime_coprime: ⟦Prime x; Prime y; x≠y⟧ ⟹ coprime x y
proof (simp only: Prime.simps coprime_def, auto simp: dvd_def,
    rule_tac classical, simp)
  fix d k ka
  assume case_ka: ∀u<d * ka. ∀v<d * ka. u * v ≠ d * ka
    and case_k: ∀u<d * k. ∀v<d * k. u * v ≠ d * k
    and h: (0::nat) < d d ≠ Suc 0 Suc 0 < d * ka
    ka ≠ k Suc 0 < d * k
  from h have k > Suc 0 ∨ ka >Suc 0
    by (cases ka;cases k;force+)
  from this show False
  proof(erule_tac disjE)
    assume (Suc 0::nat) < k
    hence k < d*k ∧ d < d*k
      using h
      by(auto)
    thus ?thesis
      using case_k
      apply(erule_tac x = d in allE)
```

```
    apply(simp)
    apply(erule_tac x = k in allE)
    apply(simp)
    done
  next
   assume (Suc 0::nat) < ka
   hence ka < d ∗ ka ∧ d < d∗ka
    using h by auto
   thus ?thesis
    using case_ka
    apply(erule_tac x = d in allE)
    apply(simp)
    apply(erule_tac x = ka in allE)
    apply(simp)
    done
  qed
qed

lemma Pi_inc: Pi (Suc i) > Pi i
proof(simp add: Pi.simps Np.simps)
  let ?setx = {y. y ≤ Suc (Pi i!) ∧ Pi i < y ∧ Prime y}
  have finite ?setx by simp
  moreover have ?setx ≠ {}
   using prime_ex[of Pi i]
   apply(auto)
   done
  ultimately show Pi i < Min ?setx
   apply(simp)
   done
qed

lemma Pi_inc_gr: i < j ⟹ Pi i < Pi j
proof(induct j, simp)
  fix j
  assume ind: i < j ⟹ Pi i < Pi j
   and h: i < Suc j
  from h show Pi i < Pi (Suc j)
  proof(cases i < j)
   case True thus ?thesis
   proof −
    assume i < j
    hence Pi i < Pi j by(erule_tac ind)
    moreover have Pi j < Pi (Suc j)
     apply(simp add: Pi_inc)
     done
    ultimately show ?thesis
     by simp
   qed
  next
   assume i < Suc j ¬ i < j
```

```
    hence i = j
      by arith
    thus Pi i < Pi (Suc j)
      apply(simp add: Pi_inc)
      done
  qed
qed

lemma Pi_notEq: i ≠ j ⟹ Pi i ≠ Pi j
  apply(cases i < j)
  using Pi_inc_gr[of i j]
   apply(simp)
  using Pi_inc_gr[of j i]
  apply(simp)
  done

lemma prime_2[intro]: Prime (Suc (Suc 0))
  apply(auto simp: Prime.simps)
  using less_2_cases by fastforce

lemma Prime_Pi[intro]: Prime (Pi n)
proof(induct n, auto simp: Pi.simps Np.simps)
  fix n
  let ?setx = {y. y ≤ Suc (Pi n!) ∧ Pi n < y ∧ Prime y}
  show Prime (Min ?setx)
  proof -
    have finite ?setx by simp
    moreover have ?setx ≠ {}
      using prime_ex[of Pi n]
      apply(simp)
      done
    ultimately show ?thesis
      apply(drule_tac Min_in, simp, simp)
      done
  qed
qed

lemma Pi_coprime: i ≠ j ⟹ coprime (Pi i) (Pi j)
  using Prime_Pi[of i]
  using Prime_Pi[of j]
  apply(rule_tac prime_coprime, simp_all add: Pi_notEq)
  done

lemma Pi_power_coprime: i ≠ j ⟹ coprime ((Pi i)^m) ((Pi j)^n)
  unfolding coprime_power_right_iff coprime_power_left_iff using Pi_coprime by auto

lemma coprime_dvd_mult_nat2: ⟦coprime (k::nat) n; k dvd n * m⟧ ⟹ k dvd m
  unfolding coprime_dvd_mult_right_iff.

declare godel_code'.simps[simp del]
```

lemma *godel_code′_butlast_last_id′* :
  godel_code′ (*ys* @ [*y*]) (*Suc j*) = *godel_code′ ys* (*Suc j*) ∗
                              Pi (*Suc* (*length ys* + *j*)) ^ *y*
proof(*induct ys arbitrary*: *j*, *simp_all add*: *godel_code′.simps*)
qed

lemma *godel_code′_butlast_last_id*:
  xs ≠ [] ⟹ *godel_code′ xs* (*Suc j*) =
  godel_code′ (*butlast xs*) (*Suc j*) ∗ *Pi* (*length xs* + *j*)^(*last xs*)
  apply(*subgoal_tac* ∃ *ys y*. *xs* = *ys* @ [*y*])
   apply(*erule_tac exE*, *erule_tac exE*, *simp add*:
     godel_code′_butlast_last_id′)
  apply(*rule_tac x* = *butlast xs* **in** *exI*)
  apply(*rule_tac x* = *last xs* **in** *exI*, *auto*)
  done

lemma *godel_code′_not0*: *godel_code′ xs n* ≠ *0*
  apply(*induct xs*, *auto simp*: *godel_code′.simps*)
  done

lemma *godel_code_append_cons*:
  length xs = *i* ⟹ *godel_code′* (*xs*@*y*#*ys*) (*Suc 0*)
    = *godel_code′ xs* (*Suc 0*) ∗ *Pi* (*Suc i*)^*y* ∗ *godel_code′ ys* (*i* + 2)
proof(*induct length xs arbitrary*: *i y ys xs*, *simp add*: *godel_code′.simps*,*simp*)
  fix *x xs i y ys*
  assume *ind*:
    ⋀*xs i y ys*. ⟦*x* = *i*; *length xs* = *i*⟧ ⟹
      godel_code′ (*xs* @ *y* # *ys*) (*Suc 0*)
    = *godel_code′ xs* (*Suc 0*) ∗ *Pi* (*Suc i*) ^ *y* ∗
                        godel_code′ ys (*Suc* (*Suc i*))
    and *h*: *Suc x* = *i*
    length (*xs*::*nat list*) = *i*
  have
    godel_code′ (*butlast xs* @ *last xs* # ((*y*::*nat*)#*ys*)) (*Suc 0*) =
      godel_code′ (*butlast xs*) (*Suc 0*) ∗ *Pi* (*Suc* (*i* − *1*))^(*last xs*)
          ∗ *godel_code′* (*y*#*ys*) (*Suc* (*Suc* (*i* − *1*)))
    apply(*rule_tac ind*)
    using *h*
    by(*auto*)
  moreover have
    godel_code′ xs (*Suc 0*)= *godel_code′* (*butlast xs*) (*Suc 0*) ∗
                                    Pi (*i*)^(*last xs*)
    using *godel_code′_butlast_last_id*[*of xs*] *h*
    apply(*cases xs* = [], *simp*, *simp*)
    done
  moreover have *butlast xs* @ *last xs* # *y* # *ys* = *xs* @ *y* # *ys*
    using *h*
    apply(*cases xs*, *auto*)
    done

```
 ultimately show
  godel_code′ (xs @ y # ys) (Suc 0) =
        godel_code′ xs (Suc 0) ∗ Pi (Suc i) ^ y ∗
            godel_code′ ys (Suc (Suc i))
  using h
  apply(simp add: godel_code′_not0 Pi_not_0)
  apply(simp add: godel_code′.simps)
  done
qed

lemma Pi_coprime_pre:
 length ps ≤ i ⟹ coprime (Pi (Suc i)) (godel_code′ ps (Suc 0))
proof(induct length ps arbitrary: ps)
 fix x ps
 assume ind:
  ⋀ps. ⟦x = length ps; length ps ≤ i⟧ ⟹
         coprime (Pi (Suc i)) (godel_code′ ps (Suc 0))
  and h: Suc x = length ps
  length (ps::nat list) ≤ i
 have g: coprime (Pi (Suc i)) (godel_code′ (butlast ps) (Suc 0))
  apply(rule_tac ind)
  using h by auto
 have k: godel_code′ ps (Suc 0) =
     godel_code′ (butlast ps) (Suc 0) ∗ Pi (length ps)^(last ps)
  using godel_code′_butlast_last_id[of ps 0] h
  by(cases ps, simp, simp)
 from g have coprime (Pi (Suc i)) (Pi (length ps) ^ last ps)
  unfolding coprime_power_right_iff using Pi_coprime h(2) by auto
 with g have
  coprime (Pi (Suc i)) (godel_code′ (butlast ps) (Suc 0) ∗
                  Pi (length ps)^(last ps))
  unfolding coprime_mult_right_iff coprime_power_right_iff by auto

 from this and k show coprime (Pi (Suc i)) (godel_code′ ps (Suc 0))
  by simp
qed (auto simp add: godel_code′.simps)

lemma Pi_coprime_suf: i < j ⟹ coprime (Pi i) (godel_code′ ps j)
proof(induct length ps arbitrary: ps)
 fix x ps
 assume ind:
  ⋀ps. ⟦x = length ps; i < j⟧ ⟹
          coprime (Pi i) (godel_code′ ps j)
  and h: Suc x = length (ps::nat list) i < j
 have g: coprime (Pi i) (godel_code′ (butlast ps) j)
  apply(rule ind) using h by auto
 have k: (godel_code′ ps j) = godel_code′ (butlast ps) j ∗
                Pi (length ps + j − 1)^last ps
  using h godel_code′_butlast_last_id[of ps j − 1]
  apply(cases ps = [], simp, simp)
```

```
  done
 from g have
  coprime (Pi i) (godel_code' (butlast ps) j *
            Pi (length ps + j - 1)^last ps)
  using Pi_power_coprime[of i length ps + j - 1 1 last ps] h
  by(auto)
 from k and this show coprime (Pi i) (godel_code' ps j)
  by auto
qed (simp add: godel_code'.simps)

lemma godel_finite:
 finite {u. Pi (Suc i) ^ u dvd godel_code' nl (Suc 0)}
proof(rule bounded_nat_set_is_finite[of _ godel_code' nl (Suc 0),rule_format],goal_cases)
 case (1 ia)
 then show ?case proof(cases ia < godel_code' nl (Suc 0))
  case False
  hence g1: Pi (Suc i) ^ ia dvd godel_code' nl (Suc 0)
   and g2: ¬ ia < godel_code' nl (Suc 0)
   and Pi (Suc i)^ia ≤ godel_code' nl (Suc 0)
   using godel_code'_not0[of nl Suc 0] using 1 by (auto elim:dvd_imp_le)
  moreover have ia < Pi (Suc i)^ia
   by(rule x_less_exp[OF Pi_gr_1])
  ultimately show ?thesis
   using g2 by(auto)
 qed auto
qed

lemma godel_code_in:
 i < length nl ⟹ nl ! i ∈ {u. Pi (Suc i) ^ u dvd
                godel_code' nl (Suc 0)}
proof -
 assume h: i<length nl
 hence godel_code' (take i nl@(nl!i)#drop (Suc i) nl) (Suc 0)
     = godel_code' (take i nl) (Suc 0) * Pi (Suc i)^(nl!i) *
              godel_code' (drop (Suc i) nl) (i + 2)
  by(rule_tac godel_code_append_cons, simp)
 moreover from h have take i nl @ (nl ! i) # drop (Suc i) nl = nl
  using upd_conv_take_nth_drop[of i nl nl ! i]
  by simp
 ultimately show
  nl ! i ∈ {u. Pi (Suc i) ^ u dvd godel_code' nl (Suc 0)}
  by(simp)
qed

lemma godel_code'_get_nth:
 i < length nl ⟹ Max {u. Pi (Suc i) ^ u dvd
            godel_code' nl (Suc 0)} = nl ! i
proof(rule_tac Max_eqI)
 let ?gc = godel_code' nl (Suc 0)
 assume h: i < length nl thus finite {u. Pi (Suc i) ^ u dvd ?gc}
```

```
  by (simp add: godel_finite)
next
 fix y
 let ?suf =godel_code′ (drop (Suc i) nl) (i + 2)
 let ?pref = godel_code′ (take i nl) (Suc 0)
 assume h: i < length nl
   y ∈ {u. Pi (Suc i) ^ u dvd godel_code′ nl (Suc 0)}
 moreover hence
   godel_code′ (take i nl@(nl!i)#drop (Suc i) nl) (Suc 0)
   = ?pref ∗ Pi (Suc i)^(nl!i) ∗ ?suf
   by(rule_tac godel_code_append_cons, simp)
 moreover from h have take i nl @ (nl!i) # drop (Suc i) nl = nl
   using upd_conv_take_nth_drop[of i nl nl!i]
   by simp
 ultimately show y≤nl!i
 proof(simp)
   let ?suf′ = godel_code′ (drop (Suc i) nl) (Suc (Suc i))
   assume mult_dvd:
     Pi (Suc i) ^ y dvd ?pref ∗  Pi (Suc i) ^ nl ! i ∗ ?suf′
   hence Pi (Suc i) ^ y dvd ?pref ∗ Pi (Suc i) ^ nl ! i
   proof −
     have coprime (Pi (Suc i)^y) ?suf′ by (simp add: Pi_coprime_suf)
     thus ?thesis using coprime_dvd_mult_left_iff mult_dvd by blast
   qed
   hence Pi (Suc i) ^ y dvd Pi (Suc i) ^ nl ! i
   proof(rule_tac coprime_dvd_mult_nat2)
     have coprime (Pi (Suc i)^y) (?pref^Suc 0) using Pi_coprime_pre by simp
     thus coprime (Pi (Suc i) ^ y) ?pref by simp
   qed
   hence Pi (Suc i) ^ y ≤  Pi (Suc i) ^ nl ! i
     apply(rule_tac dvd_imp_le, auto)
     done
   thus y ≤ nl ! i
     apply(rule_tac power_le_imp_le_exp, auto)
     done
 qed
next
 assume h: i<length nl

 thus nl ! i ∈ {u. Pi (Suc i) ^ u dvd godel_code′ nl (Suc 0)}
   by(rule_tac godel_code_in, simp)
qed

lemma godel_code′_set[simp]:
 {u. Pi (Suc i) ^ u dvd (Suc (Suc 0)) ^ length nl ∗
                              godel_code′ nl (Suc 0)} =
   {u. Pi (Suc i) ^ u dvd  godel_code′ nl (Suc 0)}
 apply(rule_tac Collect_cong, auto)
 apply(rule_tac n =  (Suc (Suc 0)) ^ length nl in
     coprime_dvd_mult_nat2)
```

```
proof -
  have Pi 0 = (2::nat) by(simp add: Pi.simps)
  show coprime (Pi (Suc i) ^ u) ((Suc (Suc 0)) ^ length nl) for u
    using Pi_coprime Pi.simps(1) by force
qed

lemma godel_code_get_nth:
  i < length nl ⟹
        Max {u. Pi (Suc i) ^ u dvd godel_code nl} = nl ! i
  by(simp add: godel_code.simps godel_code'_get_nth)

lemma mod_dvd_simp: (x mod y = (0::nat)) = (y dvd x)
  by(simp add: dvd_def, auto)

lemma dvd_power_le: ⟦a > Suc 0; a ^ y dvd a ^ l⟧ ⟹ y ≤ l
  apply(cases y ≤ l, simp, simp)
  apply(subgoal_tac ∃ d. y = l + d, auto simp: power_add)
  apply(rule_tac x = y − l in exI, simp)
  done


lemma Pi_nonzeroE[elim]: Pi n = 0 ⟹ RR
  using Pi_not_0[of n] by simp

lemma Pi_not_oneE[elim]: Pi n = Suc 0 ⟹ RR
  using Pi_gr_1[of n] by simp

lemma finite_power_dvd:
  ⟦(a::nat) > Suc 0; y ≠ 0⟧ ⟹ finite {u. a^u dvd y}
  apply(auto simp: dvd_def simp:gr0_conv_Suc intro!:bounded_nat_set_is_finite[of _ y])
  by (metis le_less_trans mod_less mod_mult_self1_is_0 not_le Suc_lessD less_trans_Suc
      mult.right_neutral n_less_n_mult_m x_less_exp
      zero_less_Suc zero_less_mult_pos)

lemma conf_decode1: ⟦m ≠ n; m ≠ k; k ≠ n⟧ ⟹
  Max {u. Pi m ^ u dvd Pi m ^ l ∗ Pi n ^ st ∗ Pi k ^ r} = l
proof -
  let ?setx = {u. Pi m ^ u dvd Pi m ^ l ∗ Pi n ^ st ∗ Pi k ^ r}
  assume g: m ≠ n m ≠ k k ≠ n
  show Max ?setx = l
  proof(rule_tac Max_eqI)
    show finite ?setx
      apply(rule_tac finite_power_dvd, auto)
      done
  next
    fix y
    assume h: y ∈ ?setx
    have Pi m ^ y dvd Pi m ^ l
    proof -
      have Pi m ^ y dvd Pi m ^ l ∗ Pi n ^ st
```

```
      using h g Pi_power_coprime
      by (simp add: coprime_dvd_mult_left_iff)
    thus Pi m^y dvd Pi m^l using g Pi_power_coprime coprime_dvd_mult_left_iff by blast
  qed
  thus y ≤ (l::nat)
    apply(rule_tac a = Pi m in power_le_imp_le_exp)
     apply(simp_all)
    apply(rule_tac dvd_power_le, auto)
    done
 next
  show l ∈ ?setx by simp
 qed
qed

lemma left_trpl_fst[simp]: left (trpl l st r) = l
 apply(simp add: left.simps trpl.simps lo.simps loR.simps mod_dvd_simp)
 apply(auto simp: conf_decode1)
  apply(cases Pi 0 ^ l * Pi (Suc 0) ^ st * Pi (Suc (Suc 0)) ^ r)
   apply(auto)
 apply(erule_tac x = l in allE, auto)
 done

lemma stat_trpl_snd[simp]: stat (trpl l st r) = st
 apply(simp add: stat.simps trpl.simps lo.simps
    loR.simps mod_dvd_simp, auto)
   apply(subgoal_tac Pi 0 ^ l * Pi (Suc 0) ^ st * Pi (Suc (Suc 0)) ^ r
          = Pi (Suc 0)^st * Pi 0 ^ l *  Pi (Suc (Suc 0)) ^ r)
    apply(simp (no_asm_simp) add: conf_decode1, simp)
  apply(cases Pi 0 ^ l * Pi (Suc 0) ^ st *
                          Pi (Suc (Suc 0)) ^ r, auto)
 apply(erule_tac x = st in allE, auto)
 done

lemma rght_trpl_trd[simp]: rght (trpl l st r) = r
 apply(simp add: rght.simps trpl.simps lo.simps
    loR.simps mod_dvd_simp, auto)
   apply(subgoal_tac Pi 0 ^ l * Pi (Suc 0) ^ st * Pi (Suc (Suc 0)) ^ r
          = Pi (Suc (Suc 0))^r * Pi 0 ^ l *  Pi (Suc 0) ^ st)
    apply(simp (no_asm_simp) add: conf_decode1, simp)
  apply(cases Pi 0 ^ l * Pi (Suc 0) ^ st * Pi (Suc (Suc 0)) ^ r,
    auto)
 apply(erule_tac x = r in allE, auto)
 done

lemma max_lor:
 i < length nl ⟹ Max {u. loR [godel_code nl, Pi (Suc i), u]}
             = nl ! i
 apply(simp add: loR.simps godel_code_get_nth mod_dvd_simp)
 done
```

```
lemma godel_decode:
  i < length nl ⟹ Entry (godel_code nl) i = nl ! i
  apply(auto simp: Entry.simps lo.simps max_lor)
  apply(erule_tac x = nl!i in allE)
  using max_lor[of i nl] godel_finite[of i nl]
  apply(simp)
  apply(drule_tac Max_in, auto simp: loR.simps
    godel_code.simps mod_dvd_simp)
  using godel_code_in[of i nl]
  apply(simp)
  done

lemma Four_Suc: 4 = Suc (Suc (Suc (Suc 0)))
  by auto

declare numeral_2_eq_2[simp del]

lemma modify_tprog_fetch_even:
  ⟦st ≤ length tp div 2; st > 0⟧ ⟹
  modify_tprog tp ! (4 * (st − Suc 0) ) =
  action_map (fst (tp ! (2 * (st − Suc 0))))
proof(induct st arbitrary: tp, simp)
  fix tp st
  assume ind:
    ⋀tp. ⟦st ≤ length tp div 2; 0 < st⟧ ⟹
     modify_tprog tp ! (4 * (st − Suc 0)) =
          action_map (fst ((tp::instr list) ! (2 * (st − Suc 0))))
    and h: Suc st ≤ length (tp::instr list) div 2 0 < Suc st
  thus modify_tprog tp ! (4 * (Suc st − Suc 0)) =
        action_map (fst (tp ! (2 * (Suc st − Suc 0))))
  proof(cases st = 0)
    case True thus ?thesis
      using h by(cases tp, auto)
  next
    case False
    assume g: st ≠ 0
    hence ∃ aa ab ba bb tp'. tp = (aa, ab) # (ba, bb) # tp'
      using h by(cases tp; cases tl tp, auto)
    from this obtain aa ab ba bb tp' where g1:
      tp = (aa, ab) # (ba, bb) # tp' by blast
    hence g2:
      modify_tprog tp' ! (4 * (st − Suc 0)) =
      action_map (fst ((tp'::instr list) ! (2 * (st − Suc 0))))
      using h g by (auto intro:ind)
    thus ?thesis
      using g1 g
      by(cases st, auto simp add: Four_Suc)
  qed
qed
```

```
lemma modify_tprog_fetch_odd:
  ⟦st ≤ length tp div 2; st > 0⟧ ⟹
     modify_tprog tp ! (Suc (Suc (4 ∗ (st − Suc 0)))) =
     action_map (fst (tp ! (Suc (2 ∗ (st − Suc 0)))))
proof(induct st arbitrary: tp, simp)
  fix tp st
  assume ind:
    ⋀tp. ⟦st ≤ length tp div 2; 0 < st⟧ ⟹
      modify_tprog tp ! Suc (Suc (4 ∗ (st − Suc 0))) =
        action_map (fst (tp ! Suc (2 ∗ (st − Suc 0))))
    and h: Suc st ≤ length (tp::instr list) div 2 0 < Suc st
  thus modify_tprog tp ! Suc (Suc (4 ∗ (Suc st − Suc 0)))
    = action_map (fst (tp ! Suc (2 ∗ (Suc st − Suc 0))))
  proof(cases st = 0)
    case True thus ?thesis
      using h
      apply(cases tp, force)
      by(cases tl tp, auto)
  next
    case False
    assume g: st ≠ 0
    hence ∃ aa ab ba bb tp′. tp = (aa, ab) # (ba, bb) # tp′
      using h
      apply(cases tp, simp, cases tl tp, simp, simp)
      done
    from this obtain aa ab ba bb tp′ where g1:
      tp = (aa, ab) # (ba, bb) # tp′ by blast
    hence g2: modify_tprog tp′ ! Suc (Suc (4 ∗ (st  − Suc 0))) =
        action_map (fst (tp′ ! Suc (2 ∗ (st − Suc 0))))
      apply(rule_tac ind)
      using h g by auto
    thus ?thesis
      using g1 g
      apply(cases st, simp, simp add: Four_Suc)
      done
  qed
qed

lemma modify_tprog_fetch_action:
  ⟦st ≤ length tp div 2; st > 0; b = 1 ∨ b = 0⟧ ⟹
      modify_tprog tp ! (4 ∗ (st − Suc 0) + 2∗ b) =
      action_map (fst (tp ! ((2 ∗ (st − Suc 0)) + b)))
  apply(erule_tac disjE, auto elim: modify_tprog_fetch_odd
      modify_tprog_fetch_even)
  done

lemma length_modify: length (modify_tprog tp) = 2 ∗ length tp
  apply(induct tp, auto)
  done
```

```
declare fetch.simps[simp del]

lemma fetch_action_eq:
  ⟦block_map b = scan r; fetch tp st b = (nact, ns);
   st ≤ length tp div 2⟧ ⟹ actn (code tp) st r = action_map nact
proof(simp add: actn.simps, auto)
 let ?i = 4 ∗ (st − Suc 0) + 2 ∗ (r mod 2)
 assume h: block_map b = r mod 2 fetch tp st b = (nact, ns)
   st ≤ length tp div 2 0 < st
 have ?i < length (modify_tprog tp)
 proof −
   have length (modify_tprog tp) = 2 ∗ length tp
     by(simp add: length_modify)
   thus ?thesis
     using h
     by(auto)
 qed
 hence
   Entry (godel_code (modify_tprog tp))?i =
                          (modify_tprog tp) ! ?i
   by(erule_tac godel_decode)
 moreover have
   modify_tprog tp ! ?i =
          action_map (fst (tp ! (2 ∗ (st − Suc 0) + r mod 2)))
   apply(rule_tac  modify_tprog_fetch_action)
   using h
   by(auto)
 moreover have (fst (tp ! (2 ∗ (st − Suc 0) + r mod 2))) = nact
   using h
   apply(cases st, simp_all add: fetch.simps nth_of.simps)
   apply(cases b, auto simp: block_map.simps nth_of.simps fetch.simps
      split: if_splits)
   apply(cases r mod 2, simp, simp)
   done
 ultimately show
   Entry (godel_code (modify_tprog tp))
                 (4 ∗ (st − Suc 0) + 2 ∗ (r mod 2))
          = action_map nact
   by simp
qed

lemma fetch_zero_zero[simp]: fetch tp 0 b = (nact, ns) ⟹ ns = 0
 by(simp add: fetch.simps)

lemma modify_tprog_fetch_state:
  ⟦st ≤ length tp div 2; st > 0; b = 1 ∨ b = 0⟧ ⟹
    modify_tprog tp ! Suc (4 ∗ (st − Suc 0) + 2 ∗ b) =
  (snd (tp ! (2 ∗ (st − Suc 0) + b)))
proof(induct st arbitrary: tp, simp)
 fix st tp
```

```
  assume ind:
    ⋀tp. ⟦st ≤ length tp div 2; 0 < st; b = 1 ∨ b = 0⟧ ⟹
    modify_tprog tp ! Suc (4 * (st − Suc 0) + 2 * b) =
                      snd (tp ! (2 * (st − Suc 0) + b))
    and h:
    Suc st ≤ length (tp::instr list) div 2
    0 < Suc st
    b = 1 ∨ b = 0
  show modify_tprog tp ! Suc (4 * (Suc st − Suc 0) + 2 * b) =
                      snd (tp ! (2 * (Suc st − Suc 0) + b))
  proof(cases st = 0)
    case True
    thus ?thesis
      using h
      apply(cases tp, force)
      apply(cases tl tp, auto)
      done
  next
    case False
    assume g: st ≠ 0
    hence ∃ aa ab ba bb tp′. tp = (aa, ab) # (ba, bb) # tp′
      using h
      by(cases tp, force, cases tl tp, auto)
    from this obtain aa ab ba bb tp′ where g1:
      tp = (aa, ab) # (ba, bb) # tp′ by blast
    hence g2:
      modify_tprog tp′ ! Suc (4 * (st − Suc 0) + 2 * b) =
                      snd (tp′ ! (2 * (st − Suc 0) + b))
      apply(intro ind)
      using h g by auto
    thus ?thesis
      using g1 g
      by(cases st;force)
  qed
qed

lemma fetch_state_eq:
  ⟦block_map b = scan r;
  fetch tp st b = (nact, ns);
  st ≤ length tp div 2⟧ ⟹ newstat (code tp) st r = ns
proof(simp add: newstat.simps, auto)
  let ?i = Suc (4 * (st − Suc 0) + 2 * (r mod 2))
  assume h: block_map b = r mod 2 fetch tp st b =
          (nact, ns) st ≤ length tp div 2 0 < st
  have ?i < length (modify_tprog tp)
  proof −
    have length (modify_tprog tp) = 2 * length tp
      by(simp add: length_modify)
    thus ?thesis
      using h
```

```
    by(auto)
  qed
  hence Entry (godel_code (modify_tprog tp)) (?i) =
                  (modify_tprog tp) ! ?i
    by(erule_tac godel_decode)
  moreover have
    modify_tprog tp ! ?i =
          (snd (tp ! (2 * (st − Suc 0) + r mod 2)))
    apply(rule_tac  modify_tprog_fetch_state)
    using h
    by(auto)
  moreover have (snd (tp ! (2 * (st − Suc 0) + r mod 2))) = ns
    using h
    apply(cases st, simp)
    apply(cases b, auto simp: fetch.simps split: if_splits)
    apply(cases (2 * (st − r mod 2) + r mod 2) =
                (2 * (st − 1) + r mod 2);auto)
    by (metis diff_Suc_Suc diff_zero prod.sel(2))
  ultimately show Entry (godel_code (modify_tprog tp)) (?i)
       = ns
    by simp
qed


lemma tpl_eqI[intro!]:
  ⟦a = a′; b = b′; c = c′⟧ ⟹ trpl a b c = trpl a′ b′ c′
  by simp

lemma bl2nat_double: bl2nat xs (Suc n) = 2 * bl2nat xs n
proof(induct xs arbitrary: n)
  case Nil thus ?case
    by(simp add: bl2nat.simps)
next
  case (Cons x xs) thus ?case
  proof −
    assume ind: ⋀n. bl2nat xs (Suc n) = 2 * bl2nat xs n
    show bl2nat (x # xs) (Suc n) = 2 * bl2nat (x # xs) n
    proof(cases x)
      case Bk thus ?thesis
        apply(simp add: bl2nat.simps)
        using ind[of Suc n] by simp
    next
      case Oc thus ?thesis
        apply(simp add: bl2nat.simps)
        using ind[of Suc n] by simp
    qed
  qed
qed
```

```
lemma bl2wc_simps[simp]:
  bl2wc (Oc # tl c) = Suc (bl2wc c) − bl2wc c mod 2
  bl2wc (Bk # c) = 2∗bl2wc (c)
  2 ∗ bl2wc (tl c) = bl2wc c − bl2wc c mod 2
  bl2wc [Oc] = Suc 0
  c ≠ [] ⟹ bl2wc (tl c) = bl2wc c div 2
  c ≠ [] ⟹ bl2wc [hd c] = bl2wc c mod 2
  c ≠ [] ⟹ bl2wc (hd c # d) = 2 ∗ bl2wc d + bl2wc c mod 2
  2 ∗ (bl2wc c div 2) = bl2wc c − bl2wc c mod 2
  bl2wc (Oc # list) mod 2 = Suc 0
  by(cases c;cases hd c;force simp: bl2wc.simps bl2nat.simps bl2nat_double)+

declare code.simps[simp del]
declare nth_of.simps[simp del]
```

The lemma relates the one step execution of TMs with the interpreter function *rec_newconf*.

```
lemma rec_t_eq_step:
  (λ (s, l, r). s ≤ length tp div 2) c ⟹
  trpl_code (step0 c tp) =
  rec_exec rec_newconf [code tp, trpl_code c]
proof(cases c)
  case (fields s l r) assume case c of (s, l, r) ⇒ s ≤ length tp div 2
  with fields have s ≤ length tp div 2 by auto
  thus ?thesis unfolding fields
  proof(cases fetch tp s (read r),
      simp add: newconf.simps trpl_code.simps step.simps)
    fix a b ca aa ba
    assume h: (a::nat) ≤ length tp div 2
     fetch tp a (read ca) = (aa, ba)
    moreover hence actn (code tp) a (bl2wc ca) = action_map aa
      apply(rule_tac b = read ca
         in fetch_action_eq, auto)
      apply(cases hd ca;cases ca;force)
      done
    moreover from h have (newstat (code tp) a (bl2wc ca)) = ba
      apply(rule_tac b = read ca
         in fetch_state_eq, auto split: list.splits)
      apply(cases hd ca;cases ca;force)
      done
    ultimately show
      trpl_code (ba, update aa (b, ca)) =
         trpl (newleft (bl2wc b) (bl2wc ca) (actn (code tp) a (bl2wc ca)))
    (newstat (code tp) a (bl2wc ca)) (newrght (bl2wc b) (bl2wc ca) (actn (code tp) a (bl2wc
ca)))
      apply(cases aa)
         apply(auto simp: trpl_code.simps
         newleft.simps newrght.simps split: action.splits)
      done
  qed
```

```
qed

lemma bl2nat_simps[simp]: bl2nat (Oc # Oc↑x) 0 = (2 * 2 ^ x − Suc 0)
  bl2nat (Bk↑x) n = 0
  by(induct x;force simp: bl2nat.simps bl2nat_double exp_ind)+

lemma bl2nat_exp_zero[simp]: bl2nat (Oc↑y) 0 = 2^y − Suc 0
proof(induct y)
  case (Suc y)
  then show ?case by(cases (2::nat)^y, auto)
qed (auto simp: bl2nat.simps bl2nat_double)

lemma bl2nat_cons_bk: bl2nat (ks @ [Bk]) 0 = bl2nat ks 0
proof(induct ks)
  case (Cons a ks)
  then show ?case by (cases a, auto simp: bl2nat.simps bl2nat_double)
qed (auto simp: bl2nat.simps)

lemma bl2nat_cons_oc:
  bl2nat (ks @ [Oc]) 0 =  bl2nat ks 0 + 2 ^ length ks
proof(induct ks)
  case (Cons a ks)
  then show ?case
    by(cases a, auto simp: bl2nat.simps bl2nat_double)
qed (auto simp: bl2nat.simps)

lemma bl2nat_append:
  bl2nat (xs @ ys) 0 = bl2nat xs 0 + bl2nat ys (length xs)
proof(induct length xs arbitrary: xs ys, simp add: bl2nat.simps)
  fix x xs ys
  assume ind:
    ⋀xs ys. x = length xs ⟹
          bl2nat (xs @ ys) 0 = bl2nat xs 0 + bl2nat ys (length xs)
    and h: Suc x = length (xs::cell list)
  have ∃ ks k. xs = ks @ [k]
    apply(rule_tac x = butlast xs in exI,
      rule_tac x = last xs in exI)
    using h
    apply(cases xs, auto)
    done
  from this obtain ks k where xs = ks @ [k] by blast
  moreover hence
    bl2nat (ks @ (k # ys)) 0 = bl2nat ks 0 +
                        bl2nat (k # ys) (length ks)
    apply(rule_tac ind) using h by simp
  ultimately show bl2nat (xs @ ys) 0 =
              bl2nat xs 0 + bl2nat ys (length xs)
    apply(cases k, simp_all add: bl2nat.simps)
     apply(simp_all only: bl2nat_cons_bk bl2nat_cons_oc)
    done
```

```
qed

lemma trpl_code_simp[simp]:
  trpl_code (steps0 (Suc 0, Bk↑l, <lm>) tp 0) =
    rec_exec rec_conf [code tp, bl2wc (<lm>), 0]
  apply(simp add: steps.simps rec_exec.simps conf_lemma  conf.simps
      inpt.simps trpl_code.simps bl2wc.simps)
  done
```

The following lemma relates the multi-step interpreter function *rec_conf* with the multi-step execution of TMs.

```
lemma state_in_range_step
  : ⟦a ≤ length A div 2; step0 (a, b, c) A = (st, l, r); tm_wf (A,0)⟧
  ⟹ st ≤ length A div 2
  apply(simp add: step.simps fetch.simps tm_wf.simps
      split: if_splits list.splits)
   apply(case_tac [!] a, auto simp: list_all_length
      fetch.simps nth_of.simps)
   apply(erule_tac x = A ! (2*nat)  in ballE, auto)
  apply(cases hd c, auto simp: fetch.simps nth_of.simps)
   apply(erule_tac x = A !(2 * nat) in ballE, auto)
  apply(erule_tac x = A !Suc (2 * nat) in ballE, auto)
  done

lemma state_in_range: ⟦steps0 (Suc 0, tp) A stp = (st, l, r); tm_wf (A, 0)⟧
  ⟹ st ≤ length A div 2
proof(induct stp arbitrary: st l r)
case (Suc stp st l r)
  from Suc.prems show ?case
  proof(simp add: step_red, cases (steps0 (Suc 0, tp) A stp), simp)
    fix a b c
    assume h3: step0 (a, b, c) A = (st, l, r)
      and h4: steps0 (Suc 0, tp) A stp = (a, b, c)
    have a ≤ length A div 2 using Suc.prems h4 by (auto intro: Suc.hyps)
    thus ?thesis using h3 Suc.prems by (auto elim: state_in_range_step)
  qed
qed(auto simp: tm_wf.simps steps.simps)

lemma rec_t_eq_steps:
  tm_wf (tp,0) ⟹
  trpl_code (steps0 (Suc 0, Bk↑l, <lm>) tp stp) =
  rec_exec rec_conf [code tp, bl2wc (<lm>), stp]
proof(induct stp)
  case 0 thus ?case by(simp)
next
  case (Suc n) thus ?case
  proof -
    assume ind:
      tm_wf (tp,0) ⟹ trpl_code (steps0 (Suc 0, Bk↑ l, <lm>) tp n)
      = rec_exec rec_conf [code tp, bl2wc (<lm>), n]
```

```
    and h: tm_wf (tp, 0)
   show
    trpl_code (steps0 (Suc 0, Bk↑ l, <lm>) tp (Suc n)) =
    rec_exec rec_conf [code tp, bl2wc (<lm>), Suc n]
   proof(cases steps0 (Suc 0, Bk↑ l, <lm>) tp  n,
      simp only: step_red conf_lemma conf.simps)
    fix a b c
    assume g: steps0 (Suc 0, Bk↑ l, <lm>) tp n = (a, b, c)
    hence conf (code tp) (bl2wc (<lm>)) n= trpl_code (a, b, c)
     using ind h
     apply(simp add: conf_lemma)
     done
    moreover hence
     trpl_code (step0 (a, b, c) tp) =
     rec_exec rec_newconf [code tp, trpl_code (a, b, c)]
     apply(rule_tac rec_t_eq_step)
     using h g
     apply(simp add: state_in_range)
     done
    ultimately show
     trpl_code (step0 (a, b, c) tp) =
        newconf (code tp) (conf (code tp) (bl2wc (<lm>)) n)
     by(simp)
   qed
  qed
qed

lemma bl2wc_Bk_0[simp]: bl2wc (Bk↑ m) = 0
  apply(induct m)
   apply(simp, simp)
  done

lemma bl2wc_Oc_then_Bk[simp]: bl2wc (Oc↑ rs@Bk↑ n) = bl2wc (Oc↑ rs)
  apply(induct rs, simp,
      simp add: bl2wc.simps bl2nat.simps bl2nat_double)
  done

lemma lg_power: x > Suc 0 ⟹ lg (x ^ rs) x = rs
proof(simp add: lg.simps, auto)
  fix xa
  assume h: Suc 0 < x
  show Max {ya. ya ≤ x ^ rs ∧ lgR [x ^ rs, x, ya]} = rs
   apply(rule_tac Max_eqI, simp_all add: lgR.simps)
    apply(simp add: h)
   using x_less_exp[of x rs] h
   apply(simp)
   done
next
  assume ¬ Suc 0 < x ^ rs Suc 0 < x
  thus rs = 0
```

```
    apply(cases x ^ rs, simp, simp)
    done
next
  assume Suc 0 < x ∀xa. ¬ lgR [x ^ rs, x, xa]
  thus rs = 0
    apply(simp only:lgR.simps)
    apply(erule_tac x = rs in allE, simp)
    done
qed
```

The following lemma relates execution of TMs with the multi-step interpreter function *rec_nonstop*. Note, *rec_nonstop* is constructed using *rec_conf*.

```
declare tm_wf.simps[simp del]

lemma nonstop_t_eq:
  ⟦steps0 (Suc 0, Bk↑l, <lm>) tp stp = (0, Bk↑ m, Oc↑ rs @ Bk↑ n);
   tm_wf (tp, 0);
   rs > 0⟧
  ⟹ rec_exec rec_nonstop [code tp, bl2wc (<lm>), stp] = 0
proof(simp add: nonstop_lemma nonstop.simps )
  assume h: steps0 (Suc 0, Bk↑l, <lm>) tp stp = (0, Bk↑ m, Oc↑ rs @ Bk↑ n)
    and tc_t: tm_wf (tp, 0) rs > 0
  have g: rec_exec rec_conf [code tp,  bl2wc (<lm>), stp] =
                                    trpl_code (0, Bk↑ m, Oc↑ rs@Bk↑ n)
    using rec_t_eq_steps[of tp l lm stp] tc_t h
    by(simp)
  thus ¬ NSTD (conf (code tp) (bl2wc (<lm>)) stp)
  proof(auto simp: NSTD.simps)
    show stat (conf (code tp) (bl2wc (<lm>)) stp) = 0
      using g
      by(auto simp: conf_lemma trpl_code.simps)
  next
    show left (conf (code tp) (bl2wc (<lm>)) stp) = 0
      using g
      by(simp add: conf_lemma trpl_code.simps)
  next
    show rght (conf (code tp) (bl2wc (<lm>)) stp) =
          2 ^ lg (Suc (rght (conf (code tp) (bl2wc (<lm>)) stp))) 2 − Suc 0
      using g h
    proof(simp add: conf_lemma trpl_code.simps)
      have 2 ^ lg (Suc (bl2wc (Oc↑ rs))) 2 = Suc (bl2wc (Oc↑ rs))
        apply(simp add: bl2wc.simps lg_power)
        done
      thus bl2wc (Oc↑ rs) = 2 ^ lg (Suc (bl2wc (Oc↑ rs))) 2 − Suc 0
        apply(simp)
        done
    qed
  next
    show 0 < rght (conf (code tp) (bl2wc (<lm>)) stp)
      using g h tc_t
```

```
    apply(simp add: conf_lemma trpl_code.simps bl2wc.simps
      bl2nat.simps)
    apply(cases rs, simp, simp add: bl2nat.simps)
    done
  qed
qed

lemma actn_0_is_4[simp]: actn m 0 r = 4
  by(simp add: actn.simps)

lemma newstat_0_0[simp]: newstat m 0 r = 0
  by(simp add: newstat.simps)

declare step_red[simp del]

lemma halt_least_step:
  ⟦steps0 (Suc 0, Bk↑l, <lm>) tp stp =
      (0, Bk↑ m, Oc↑rs @ Bk↑n);
    tm_wf (tp, 0);
    0<rs⟧ ⟹
    ∃ stp. (nonstop (code tp) (bl2wc (<lm>)) stp = 0 ∧
      (∀ stp'. nonstop (code tp) (bl2wc (<lm>)) stp' = 0 ⟶ stp ≤ stp'))
proof(induct stp)
  case 0
  then show ?case by (simp add: steps.simps(1))
next
  case (Suc stp)
  hence ind:
    steps0 (Suc 0, Bk↑ l, <lm>) tp stp = (0, Bk↑ m, Oc↑ rs @ Bk↑ n) ⟹
    ∃ stp. nonstop (code tp) (bl2wc (<lm>)) stp = 0 ∧
        (∀ stp'. nonstop (code tp) (bl2wc (<lm>)) stp' = 0 ⟶ stp ≤ stp')
    and h:
    steps0 (Suc 0, Bk↑ l, <lm>) tp (Suc stp) = (0, Bk↑ m, Oc↑ rs @ Bk↑ n)
    tm_wf (tp, 0::nat)
    0 < rs by simp+
  {
    fix a b c nat
    assume steps0 (Suc 0, Bk↑ l, <lm>) tp stp = (a, b, c)
      a = Suc nat
    hence ∃ stp. nonstop (code tp) (bl2wc (<lm>)) stp = 0 ∧
      (∀ stp'. nonstop (code tp) (bl2wc (<lm>)) stp' = 0 ⟶ stp ≤ stp')
      using h
      apply(rule_tac x = Suc stp in exI, auto)
       apply(drule_tac  nonstop_t_eq, simp_all add: nonstop_lemma)
    proof –
      fix stp'
      assume g:steps0 (Suc 0, Bk↑ l, <lm>) tp stp = (Suc nat, b, c)
        nonstop (code tp) (bl2wc (<lm>)) stp' = 0
      thus  Suc stp ≤ stp'
      proof(cases Suc stp ≤ stp', simp, simp)
```

```
      assume ¬ Suc stp ≤ stp′
      hence stp′ ≤ stp by simp
      hence ¬ is_final (steps0 (Suc 0, Bk↑ l, <lm>) tp stp′)
        using g
        apply(cases steps0 (Suc 0, Bk↑ l, <lm>) tp stp′,auto, simp)
        apply(subgoal_tac ∃ n. stp = stp′ + n, auto)
         apply(cases fst (steps0 (Suc 0, Bk ↑ l, <lm>) tp stp′), simp_all add: steps.simps)
        apply(rule_tac x = stp − stp′ in exI, simp)
        done
      hence nonstop (code tp) (bl2wc (<lm>)) stp′ = 1
      proof(cases steps0 (Suc 0, Bk↑ l, <lm>) tp stp′,
          simp add: nonstop.simps)
        fix a b c
        assume k:
          0 < a steps0 (Suc 0, Bk↑ l, <lm>) tp stp′ = (a, b, c)
        thus  NSTD (conf (code tp) (bl2wc (<lm>)) stp′)
          using rec_t_eq_steps[of tp l lm stp′] h
        proof(simp add: conf_lemma)
          assume trpl_code (a, b, c) = conf (code tp) (bl2wc (<lm>)) stp′
          moreover have NSTD (trpl_code (a, b, c))
            using k
            apply(auto simp: trpl_code.simps NSTD.simps)
            done
          ultimately show NSTD (conf (code tp) (bl2wc (<lm>)) stp′) by simp
        qed
      qed
      thus False using g by simp
    qed qed
  }
  note [intro] = this
 from h show
   ∃ stp. nonstop (code tp) (bl2wc (<lm>)) stp = 0
   ∧ (∀ stp′. nonstop (code tp) (bl2wc (<lm>)) stp′ = 0 ⟶ stp ≤ stp′)
   by(simp add: step_red,
     cases steps0 (Suc 0, Bk↑ l, <lm>) tp stp, simp,
     cases fst (steps0 (Suc 0, Bk↑ l, <lm>) tp stp),
     auto simp add: nonstop_t_eq intro:ind dest:nonstop_t_eq)
qed

lemma conf_trpl_ex: ∃ p q r. conf m (bl2wc (<lm>)) stp = trpl p q r
 apply(induct stp, auto simp: conf.simps inpt.simps trpl.simps
     newconf.simps)
 apply(rule_tac x = 0 in exI, rule_tac x = 1 in exI,
     rule_tac x = bl2wc (<lm>) in exI)
 apply(simp)
 done

lemma nonstop_rgt_ex:
 nonstop m (bl2wc (<lm>)) stpa = 0 ⟹ ∃ r. conf m (bl2wc (<lm>)) stpa = trpl 0 0 r
 apply(auto simp: nonstop.simps NSTD.simps split: if_splits)
```

```
  using conf_trpl_ex[of m lm stpa]
  apply(auto)
  done

lemma max_divisors: x > Suc 0 ⟹ Max {u. x ^ u dvd x ^ r} = r
proof(rule_tac Max_eqI)
  assume x > Suc 0
  thus finite {u. x ^ u dvd x ^ r}
    apply(rule_tac finite_power_dvd, auto)
    done
next
  fix y
  assume Suc 0 < x y ∈ {u. x ^ u dvd x ^ r}
  thus y ≤ r
    apply(cases y≤ r, simp)
    apply(subgoal_tac ∃ d. y = r + d)
     apply(auto simp: power_add)
    apply(rule_tac x = y − r in exI, simp)
    done
next
  show r ∈ {u. x ^ u dvd x ^ r} by simp
qed

lemma lo_power:
  assumes x > Suc 0 shows lo (x ^ r) x = r
proof −
  have ¬ Suc 0 < x ^ r ⟹ r = 0 using assms
    by (metis Suc_lessD Suc_lessI nat_power_eq_Suc_0_iff zero_less_power)
  moreover have ∀ xa. ¬ x ^ xa dvd x ^ r ⟹ r = 0
    using dvd_refl assms by(cases x^r;blast)
  ultimately show ?thesis using assms
    by(auto simp: lo.simps loR.simps mod_dvd_simp elim:max_divisors)
qed

lemma lo_rgt: lo (trpl 0 0 r) (Pi 2) = r
  apply(simp add: trpl.simps lo_power)
  done

lemma conf_keep:
  conf m lm stp = trpl 0 0 r  ⟹
  conf m lm (stp + n) = trpl 0 0 r
  apply(induct n)
   apply(auto simp: conf.simps  newconf.simps newleft.simps
     newrght.simps rght.simps lo_rgt)
  done

lemma halt_state_keep_steps_add:
  ⟦nonstop m (bl2wc (<lm>)) stpa = 0⟧ ⟹
  conf m (bl2wc (<lm>)) stpa = conf m (bl2wc (<lm>)) (stpa + n)
  apply(drule_tac nonstop_rgt_ex, auto simp: conf_keep)
```

```
 done

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
 apply(cases stpa > stpb)
 using halt_state_keep_steps_add[of m lm stpb stpa − stpb]
  apply simp
 using halt_state_keep_steps_add[of m lm stpa stpb − stpa]
 apply(simp)
 done
```

The correntess of *rec_F* which relates the interpreter function *rec_F* with the execution of of TMs.

```
lemma terminate_halt:
 ⟦steps0 (Suc 0, Bk↑l, <lm>) tp stp = (0, Bk↑m, Oc↑rs@Bk↑n);
  tm_wf (tp,0); 0<rs⟧ ⟹ terminate rec_halt [code tp, (bl2wc (<lm>))]
 by(frule_tac halt_least_step;force simp:nonstop_lemma intro:terminate_halt_lemma)

lemma terminate_F:
 ⟦steps0 (Suc 0, Bk↑l, <lm>) tp stp = (0, Bk↑m, Oc↑rs@Bk↑n);
  tm_wf (tp,0); 0<rs⟧ ⟹ terminate rec_F [code tp, (bl2wc (<lm>))]
 apply(drule_tac terminate_halt, simp_all)
 apply(erule_tac terminate_F_lemma)
 done

lemma F_correct:
 ⟦steps0 (Suc 0, Bk↑l, <lm>) tp stp = (0, Bk↑m, Oc↑rs@Bk↑n);
  tm_wf (tp,0); 0<rs⟧
  ⟹ rec_exec rec_F [code tp, (bl2wc (<lm>))] = (rs − Suc 0)
 apply(frule_tac halt_least_step, auto)
 apply(frule_tac  nonstop_t_eq, auto simp: nonstop_lemma)
 using rec_t_eq_steps[of tp l lm stp]
 apply(simp add: conf_lemma)
proof −
 fix stpa
 assume h:
  nonstop (code tp) (bl2wc (<lm>)) stpa = 0
  ∀ stp′. nonstop (code tp) (bl2wc (<lm>)) stp′ = 0 ⟶ stpa ≤ stp′
  nonstop (code tp) (bl2wc (<lm>)) stp = 0
  trpl_code (0, Bk↑ m, Oc↑ rs @ Bk↑ n) = conf (code tp) (bl2wc (<lm>)) stp
  steps0 (Suc 0, Bk↑ l, <lm>) tp stp = (0, Bk↑ m, Oc↑ rs @ Bk↑ n)
 hence g1: conf (code tp) (bl2wc (<lm>)) stpa = trpl_code (0, Bk↑ m, Oc↑ rs @ Bk↑n)
  using halt_state_keep[of code tp lm stpa stp]
  by(simp)
 moreover have g2:
  rec_exec rec_halt [code tp, (bl2wc (<lm>))] = stpa
  using h
  by(auto simp: rec_exec.simps rec_halt_def nonstop_lemma intro!: Least_equality)
 show
```

```
  rec_exec rec_F [code tp, (bl2wc (<lm>))] = (rs − Suc 0)
 proof −
  have
    valu (rght (conf (code tp) (bl2wc (<lm>)) stpa)) = rs − Suc 0
    using g1
    apply(simp add: valu.simps trpl_code.simps
      bl2wc.simps  bl2nat_append lg_power)
    done
  thus ?thesis
    by(simp add: rec_exec.simps F_lemma g2)
 qed
qed

end
```

26 Construction of a Universal Turing Machine

```
theory UTM
 imports Recursive Abacus UF HOL.GCD Turing_Hoare
begin
```

27 Wang coding of input arguments

The direct compilation of the universal function *rec_F* can not give us UTM, because *rec_F* is of arity 2, where the first argument represents the Godel coding of the TM being simulated and the second argument represents the right number (in Wang's coding) of the TM tape. (Notice, left number is always *0* at the very beginning). However, UTM needs to simulate the execution of any TM which may very well take many input arguments. Therefore, a initialization TM needs to run before the TM compiled from *rec_F*, and the sequential composition of these two TMs will give rise to the UTM we are seeking. The purpose of this initialization TM is to transform the multiple input arguments of the TM being simulated into Wang's coding, so that it can be consumed by the TM compiled from *rec_F* as the second argument.

However, this initialization TM (named *t_wcode*) can not be constructed by compiling from any recursive function, because every recursive function takes a fixed number of input arguments, while *t_wcode* needs to take varying number of arguments and tranform them into Wang's coding. Therefore, this section give a direct construction of *t_wcode* with just some parts being obtained from recursive functions.

The TM used to generate the Wang's code of input arguments is divided into three TMs executed sequentially, namely $prepare$, $mainwork$ and $adjust$. According to the convention, the start state of ever TM is fixed to state 1 while the final state is fixed to 0.

The input and output of $prepare$ are illustrated respectively by Figure 1 and 2.

As shown in Figure 1, the input of $prepare$ is the same as the the input of UTM, where m is the Godel coding of the TM being interpreted and a_1 through a_n are the n input arguments of the TM under interpretation. The purpose of $purpose$ is to trans-

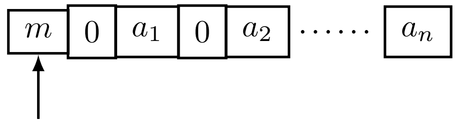

Figure 1: The input of TM $prepare$

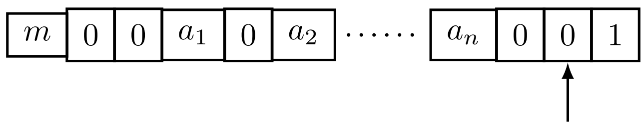

Figure 2: The output of TM $prepare$

form this initial tape layout to the one shown in Figure 2, which is convenient for the generation of Wang's codding of $a_1, \ldots, a_n$. The coding procedure starts from a_n and ends after a_1 is encoded. The coding result is stored in an accumulator at the end of the tape (initially represented by the 1 two blanks right to a_n in Figure 2). In Figure 2, arguments $a_1, \ldots, a_n$ are separated by two blanks on both ends with the rest so that movement conditions can be implemented conveniently in subsequent TMs, because, by convention, two consecutive blanks are usually used to signal the end or start of a large chunk of data. The diagram of $prepare$ is given in Figure 3.

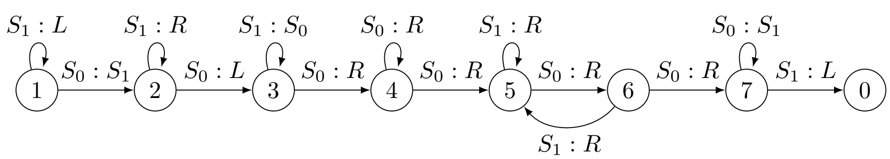

Figure 3: The diagram of TM $prepare$

The purpose of TM $mainwork$ is to compute the Wang's encoding of $a_1, \ldots, a_n$. Every bit of $a_1, \ldots, a_n$, including the separating bits, is processed from left to right. In order to detect the termination condition when the left most bit of a_1 is reached, TM $mainwork$ needs to look ahead and consider three different situations at the start of every iteration:

1. The TM configuration for the first situation is shown in Figure 4, where the accumulator is stored in r, both of the next two bits to be encoded are 1. The configuration at the end of the iteration is shown in Figure 5, where the first 1-bit has been encoded and cleared. Notice that the accumulator has been changed to $(r + 1) \times 2$ to reflect the encoded bit.

2. The TM configuration for the second situation is shown in Figure 6, where the accumulator is stored in r, the next two bits to be encoded are 1 and 0. After the first 1-bit was encoded and cleared, the second 0-bit is difficult to detect and process. To solve this problem, these two consecutive bits are encoded in

one iteration. In this situation, only the first 1-bit needs to be cleared since the second one is cleared by definition. The configuration at the end of the iteration is shown in Figure 7. Notice that the accumulator has been changed to $(r+1) \times 4$ to reflect the two encoded bits.

3. The third situation corresponds to the case when the last bit of a_1 is reached. The TM configurations at the start and end of the iteration are shown in Figure 8 and 9 respectively. For this situation, only the read write head needs to be moved to the left to prepare a initial configuration for TM $adjust$ to start with.

The diagram of $mainwork$ is given in Figure 10. The two rectangular nodes labeled with $2 \times x$ and $4 \times x$ are two TMs compiling from recursive functions so that we do not have to design and verify two quite complicated TMs.

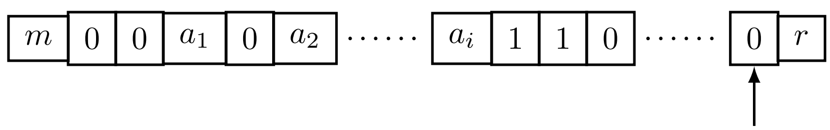

Figure 4: The first situation for TM $mainwork$ to consider

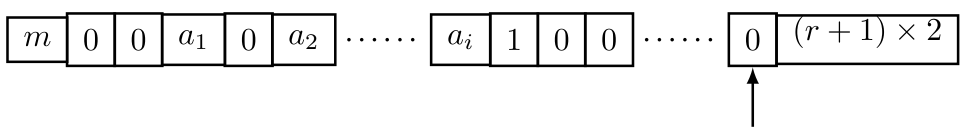

Figure 5: The output for the first case of TM $mainwork$'s processing

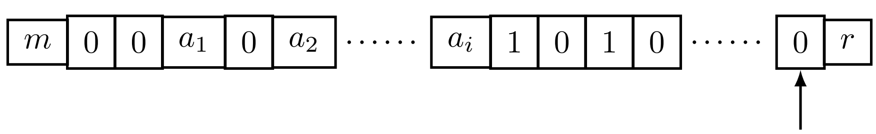

Figure 6: The second situation for TM $mainwork$ to consider

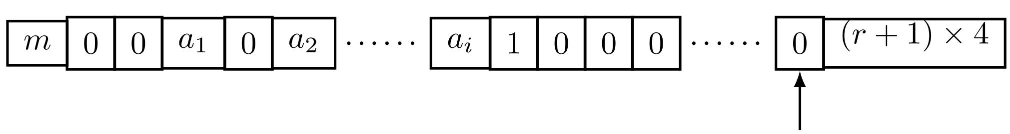

Figure 7: The output for the second case of TM $mainwork$'s processing

The purpose of TM $adjust$ is to encode the last bit of a_1. The initial and final configuration of this TM are shown in Figure 11 and 12 respectively. The diagram of TM $adjust$ is shown in Figure 13.

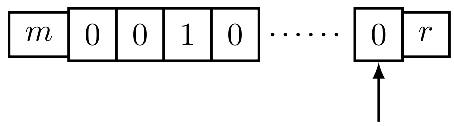

Figure 8: The third situation for TM $mainwork$ to consider

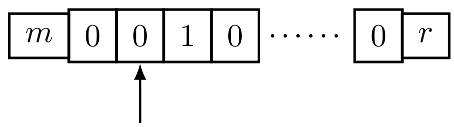

Figure 9: The output for the third case of TM $mainwork$'s processing

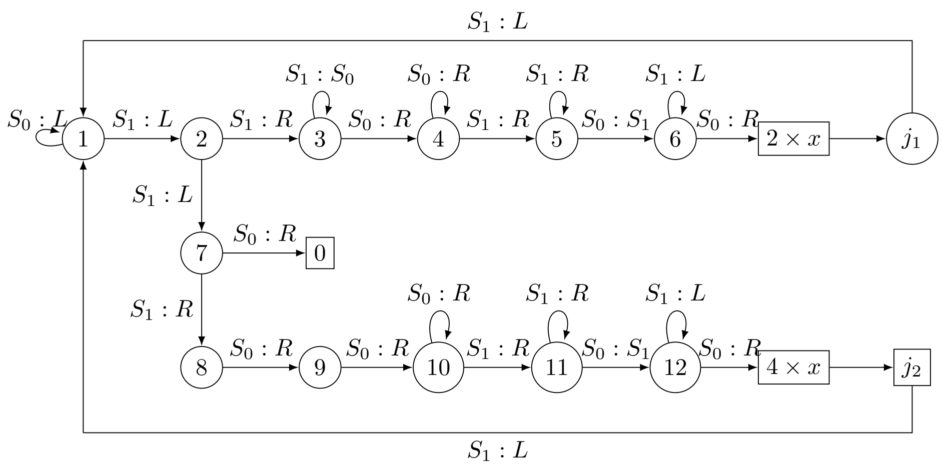

Figure 10: The diagram of TM $mainwork$

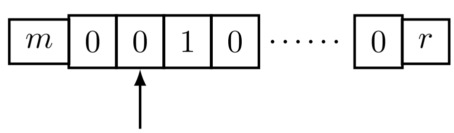

Figure 11: Initial configuration of TM $adjust$

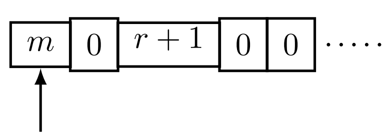

Figure 12: Final configuration of TM $adjust$

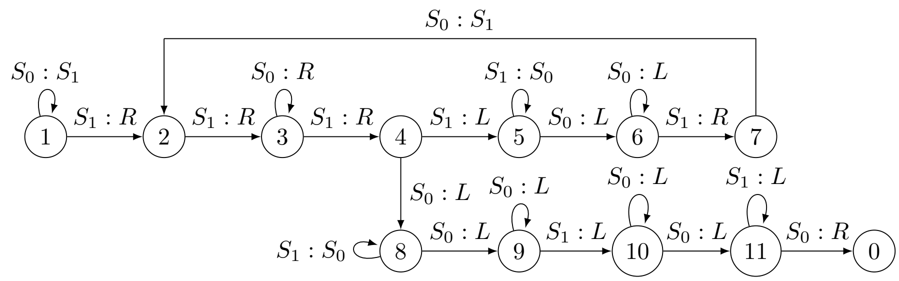

Figure 13: Diagram of TM $adjust$

definition *rec_twice* :: *recf*
 where
  rec_twice = *Cn 1 rec_mult* [*id 1 0*, *constn 2*]

definition *rec_fourtimes* :: *recf*
 where
  rec_fourtimes = *Cn 1 rec_mult* [*id 1 0*, *constn 4*]

definition *abc_twice* :: *abc_prog*
 where
  abc_twice = (*let* (*aprog*, *ary*, *fp*) = *rec_ci rec_twice in*
        aprog [+] *dummy_abc* ((*Suc 0*)))

definition *abc_fourtimes* :: *abc_prog*
 where
  abc_fourtimes = (*let* (*aprog*, *ary*, *fp*) = *rec_ci rec_fourtimes in*
        aprog [+] *dummy_abc* ((*Suc 0*)))

definition *twice_ly* :: *nat list*
 where
  twice_ly = *layout_of abc_twice*

definition *fourtimes_ly* :: *nat list*
 where
  fourtimes_ly = *layout_of abc_fourtimes*

definition *t_twice_compile* :: *instr list*
 where
  t_twice_compile= (*tm_of abc_twice* @ (*shift* (*mopup 1*) (*length* (*tm_of abc_twice*) *div 2*)))

definition *t_twice* :: *instr list*
 where
  t_twice = *adjust0 t_twice_compile*

definition *t_fourtimes_compile* :: *instr list*
 where

```
  t_fourtimes_compile= (tm_of abc_fourtimes @ (shift (mopup 1) (length (tm_of abc_fourtimes)
div 2)))

definition t_fourtimes :: instr list
  where
    t_fourtimes = adjust0 t_fourtimes_compile

definition t_twice_len :: nat
  where
    t_twice_len = length t_twice div 2

definition t_wcode_main_first_part:: instr list
  where
    t_wcode_main_first_part =def=
              [(L, 1), (L, 2), (L, 7), (R, 3),
               (R, 4), (W0, 3), (R, 4), (R, 5),
               (W1, 6), (R, 5), (R, 13), (L, 6),
               (R, 0), (R, 8), (R, 9), (Nop, 8),
               (R, 10), (W0, 9), (R, 10), (R, 11),
               (W1, 12), (R, 11), (R, t_twice_len + 14), (L, 12)]

definition t_wcode_main :: instr list
  where
    t_wcode_main = (t_wcode_main_first_part @ shift t_twice 12 @ [(L, 1), (L, 1)]
               @ shift t_fourtimes (t_twice_len + 13) @ [(L, 1), (L, 1)])

fun bl_bin :: cell list ⇒ nat
  where
    bl_bin [] = 0
  | bl_bin (Bk # xs) = 2 ∗ bl_bin xs
  | bl_bin (Oc # xs) = Suc (2 ∗ bl_bin xs)

declare bl_bin.simps[simp del]

type-synonym bin_inv_t = cell list ⇒ nat ⇒ tape ⇒ bool

fun wcode_before_double :: bin_inv_t
  where
    wcode_before_double ires rs (l, r) =
     (∃ ln rn. l = Bk # Bk # Bk↑(ln) @ Oc # ires ∧
           r = Oc↑((Suc (Suc rs))) @ Bk↑(rn ))

declare wcode_before_double.simps[simp del]

fun wcode_after_double :: bin_inv_t
  where
    wcode_after_double ires rs (l, r) =
     (∃ ln rn. l = Bk # Bk # Bk↑(ln) @ Oc # ires ∧
        r = Oc↑(Suc (Suc (Suc 2∗rs))) @ Bk↑(rn))
```

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

fun *wcode_on_left_moving_1_B* :: *bin_inv_t*
where
wcode_on_left_moving_1_B ires rs $(l, r) =$
$(\exists\ ml\ mr\ rn.\ l = Bk\uparrow(ml)\ @\ Oc\ \#\ Oc\ \#\ ires \wedge$
$r = Bk\uparrow(mr)\ @\ Oc\uparrow(Suc\ rs)\ @\ Bk\uparrow(rn) \wedge$
$ml + mr > Suc\ 0 \wedge mr > 0)$

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 $(l, r) =$
$(\exists\ ln\ rn.$
$l = Oc\ \#\ ires \wedge$
$r = Oc\ \#\ Bk\uparrow(ln)\ @\ Bk\ \#\ Bk\ \#\ Oc\uparrow(Suc\ rs)\ @\ Bk\uparrow(rn))$

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* $(l, r) \vee$ *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 $(l, r) =$
$(\exists\ ln\ rn.\ l = ires \wedge$
$r = Oc\ \#\ Oc\ \#\ Bk\uparrow(ln)\ @\ Bk\ \#\ Bk\ \#\ Oc\uparrow(Suc\ rs)\ @\ Bk\uparrow(rn))$

fun *wcode_erase1* :: *bin_inv_t*
where
wcode_erase1 ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l = Oc\ \#\ ires \wedge$
$tl\ r = Bk\uparrow(ln)\ @\ Bk\ \#\ Bk\ \#\ Oc\uparrow(Suc\ rs)\ @\ Bk\uparrow(rn))$

declare *wcode_erase1.simps* [*simp del*]

fun *wcode_on_right_moving_1* :: *bin_inv_t*
where
wcode_on_right_moving_1 ires rs $(l, r) =$
$(\exists\ ml\ mr\ rn.$
$l = Bk\uparrow(ml)\ @\ Oc\ \#\ ires \wedge$
$r = Bk\uparrow(mr)\ @\ Oc\uparrow(Suc\ rs)\ @\ Bk\uparrow(rn) \wedge$
$ml + mr > Suc\ 0)$

```
declare wcode_on_right_moving_1.simps [simp del]

declare wcode_on_right_moving_1.simps[simp del]

fun wcode_goon_right_moving_1 :: bin_inv_t
  where
    wcode_goon_right_moving_1 ires rs (l, r) =
      (∃ 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(induct xs)
  show  bl_bin [] = bl2wc []
    apply(simp add: bl_bin.simps)
    done
next
  fix a xs
  assume bl_bin xs = bl2wc xs
  thus  bl_bin (a # xs) = bl2wc (a # xs)
    apply(case_tac a, simp_all add: bl_bin.simps bl2wc.simps)
     apply(simp_all add: bl2nat.simps bl2nat_double)
```

```
  done
qed

lemma tape_of_nl_append_one: lm ≠ [] ⟹ <lm @ [a]> = <lm> @ Bk # Oc↑Suc a
 apply(induct lm, auto simp: tape_of_nl_cons split:if_splits)
 done

lemma tape_of_nl_rev: rev (<lm::nat list>) = (<rev lm>)
 apply(induct lm, simp, auto)
 apply(auto simp: tape_of_nl_cons tape_of_nl_append_one split: if_splits)
 apply(simp add: exp_ind[THEN sym])
 done

lemma exp_1[simp]: a↑(Suc 0) = [a]
 by(simp)

lemma tape_of_nl_cons_app1: (<a # xs @ [b]>) = (Oc↑(Suc a) @ Bk # (<xs@ [b]>))
 apply(case_tac xs; simp add: tape_of_list_def tape_of_nat_list.simps tape_of_nat_def)
 done

lemma bl_bin_bk_oc[simp]:
 bl_bin (xs @ [Bk, Oc]) =
 bl_bin xs + 2*2^(length xs)
 apply(simp add: bin_wc_eq)
 using bl2nat_cons_oc[of xs @ [Bk]]
 apply(simp add: bl2nat_cons_bk bl2wc.simps)
 done

lemma tape_of_nat[simp]: (<a::nat>) = Oc↑(Suc a)
 apply(simp add: tape_of_nat_def)
 done

lemma tape_of_nl_cons_app2: (<c # xs @ [b]>) = (<c # xs> @ Bk # Oc↑(Suc b))
proof(induct length xs arbitrary: xs c, simp add: tape_of_list_def)
 fix x xs c
 assume ind: ⋀xs c. x = length xs ⟹ <c # xs @ [b]> =
  <c # xs> @ Bk # Oc↑(Suc b)
  and h: Suc x = length (xs::nat list)
 show <c # xs @ [b]> = <c # xs> @ Bk # Oc↑(Suc b)
 proof(cases xs, simp add: tape_of_list_def)
  fix a list
  assume g: xs = a # list
  hence k: <a # list @ [b]> = <a # list> @ Bk # Oc↑(Suc b)
   apply(rule_tac ind)
   using h
   apply(simp)
   done
  from g and k show <c # xs @ [b]> = <c # xs> @ Bk # Oc↑(Suc b)
   apply(simp add: tape_of_list_def)
   done
```

```
  qed
qed

lemma length_2_elems[simp]: length (<aa # a # list>) = Suc (Suc aa) + length (<a # list>)
  apply(simp add: tape_of_list_def)
  done

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)))
  using bl_bin_bk_oc[of Oc↑(Suc aa) @ Bk # tape_of_nat_list (a # lista)]
  apply(simp)
  done

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]>)))
  apply(case_tac list, simp add: add_mult_distrib)
  apply(simp add: tape_of_nl_cons_app2 add_mult_distrib)
  apply(simp add: tape_of_list_def)
  done

lemma tape_of_nl_app_Suc: ((<list @ [Suc ab]>)) = (<list @ [ab]>) @ [Oc]
proof(induct list)
  case (Cons a list)
  then show ?case by(cases list;simp_all add:tape_of_list_def exp_ind)
qed (simp add: tape_of_list_def exp_ind)

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]>))
  apply(simp add: bin_wc_eq)
  apply(simp add: bl2nat_cons_oc bl2wc.simps)
  using bl2nat_cons_oc[of Oc # Oc↑(aa) @ Bk # <list @ [ab]>]
  apply(simp)
  done
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]>)))
  apply(simp add: tape_of_nl_app_Suc)
  done

declare tape_of_nat[simp del]

fun wcode_double_case_inv :: nat ⇒ bin_inv_t
```

where
wcode_double_case_inv st ires rs $(l, r) =$
($if\ st = Suc\ 0\ then\ wcode\_on\_left\_moving\_1\ ires\ rs\ (l, r)$
$else\ if\ st = Suc\ (Suc\ 0)\ then\ wcode\_on\_checking\_1\ ires\ rs\ (l, r)$
$else\ if\ st = 3\ then\ wcode\_erase1\ ires\ rs\ (l, r)$
$else\ if\ st = 4\ then\ wcode\_on\_right\_moving\_1\ ires\ rs\ (l, r)$
$else\ if\ st = 5\ then\ wcode\_goon\_right\_moving\_1\ ires\ rs\ (l, r)$
$else\ if\ st = 6\ then\ wcode\_backto\_standard\_pos\ ires\ rs\ (l, r)$
$else\ if\ st = 13\ then\ wcode\_before\_double\ ires\ rs\ (l, r)$
$else\ False$)

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

fun *wcode_double_case_state* :: *config* ⇒ *nat*
where
wcode_double_case_state $(st, l, r) =$
$13 - st$

fun *wcode_double_case_step* :: *config* ⇒ *nat*
where
wcode_double_case_step $(st, l, r) =$
($if\ st = Suc\ 0\ then\ (length\ l)$
$else\ if\ st = Suc\ (Suc\ 0)\ then\ (length\ r)$
$else\ if\ st = 3\ then$
$if\ hd\ r = Oc\ then\ 1\ else\ 0$
$else\ if\ st = 4\ then\ (length\ r)$
$else\ if\ st = 5\ then\ (length\ r)$
$else\ if\ st = 6\ then\ (length\ l)$
$else\ 0$)

fun *wcode_double_case_measure* :: *config* ⇒ *nat* × *nat*
where
wcode_double_case_measure $(st, l, r) =$
(*wcode_double_case_state* (st, l, r),
wcode_double_case_step (st, l, r))

definition *wcode_double_case_le* :: (*config* × *config*) *set*
where *wcode_double_case_le* $\stackrel{def}{=}$ (*inv_image lex_pair wcode_double_case_measure*)

lemma *wf_lex_pair*[*intro*]: *wf lex_pair*
by(*auto intro*:*wf_lex_prod simp*:*lex_pair_def*)

lemma *wf_wcode_double_case_le*[*intro*]: *wf wcode_double_case_le*
by(*auto intro*:*wf_inv_image simp*: *wcode_double_case_le_def*)

lemma *fetch_t_wcode_main*[*simp*]:
fetch t_wcode_main $(Suc\ 0)\ Bk = (L, Suc\ 0)$
fetch t_wcode_main $(Suc\ 0)\ Oc = (L, Suc\ (Suc\ 0))$
fetch t_wcode_main $(Suc\ (Suc\ 0))\ Oc = (R, 3)$

fetch t_wcode_main (Suc (Suc 0)) Bk = (*L*, *7*)
fetch t_wcode_main (Suc (Suc (Suc 0))) Bk = (*R*, *4*)
fetch t_wcode_main (Suc (Suc (Suc 0))) Oc = (*W0*, *3*)
fetch t_wcode_main 4 Bk = (*R*, *4*)
fetch t_wcode_main 4 Oc = (*R*, *5*)
fetch t_wcode_main 5 Oc = (*R*, *5*)
fetch t_wcode_main 5 Bk = (*W1*, *6*)
fetch t_wcode_main 6 Bk = (*R*, *13*)
fetch t_wcode_main 6 Oc = (*L*, *6*)
fetch t_wcode_main 7 Oc = (*R*, *8*)
fetch t_wcode_main 7 Bk = (*R*, *0*)
fetch t_wcode_main 8 Bk = (*R*, *9*)
fetch t_wcode_main 9 Bk = (*R*, *10*)
fetch t_wcode_main 9 Oc = (*W0*, *9*)
fetch t_wcode_main 10 Bk = (*R*, *10*)
fetch t_wcode_main 10 Oc = (*R*, *11*)
fetch t_wcode_main 11 Bk = (*W1*, *12*)
fetch t_wcode_main 11 Oc = (*R*, *11*)
fetch t_wcode_main 12 Oc = (*L*, *12*)
fetch t_wcode_main 12 Bk = (*R*, *t_twice_len* + *14*)
by(*auto simp*: *t_wcode_main_def t_wcode_main_first_part_def fetch.simps numeral*)

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*) $\Longrightarrow$ *b* $\neq$ []
by(*auto simp*: *wcode_on_left_moving_1.simps wcode_on_left_moving_1_B.simps*
wcode_on_left_moving_1_O.simps)

lemma *wcode_on_left_moving_1E*[*elim*]: ⟦*wcode_on_left_moving_1 ires rs* (*b*, *Bk* # *list*);
tl b = *aa* $\wedge$ *hd b* # *Bk* # *list* = *ba*⟧ $\Longrightarrow$
wcode_on_left_moving_1 ires rs (*aa*, *ba*)
apply(*simp only*: *wcode_on_left_moving_1.simps wcode_on_left_moving_1_O.simps*
wcode_on_left_moving_1_B.simps)
apply(*erule_tac disjE*)
apply(*erule_tac exE*)+
apply(*rename_tac ml mr rn*)
apply(*case_tac ml*, *simp*)
apply(*rule_tac x* = *mr* − *Suc* (*Suc 0*) **in** *exI*, *rule_tac x* = *rn* **in** *exI*)
apply (*smt One_nat_def Suc_diff_Suc append_Cons empty_replicate list.sel*(*3*) *neq0_conv replicate_Suc replicate_app_Cons_same tl_append2 tl_replicate*)
apply(*rule_tac disjI1*)

apply (*metis add_Suc_shift less_SucI list.exhaust_sel list.inject list.simps*(*3*) *replicate_Suc_iff_anywhere*)
by *simp*

declare *replicate_Suc*[*simp*]

lemma *wcode_on_moving_1_Elim*[*elim*]:
$[\![$*wcode_on_left_moving_1 ires rs* (*b*, *Oc* # *list*); *tl b* = *aa* $\wedge$ *hd b* # *Oc* # *list* = *ba*$]\!]$
$\Longrightarrow$ *wcode_on_checking_1 ires rs* (*aa*, *ba*)
apply(*simp only*: *wcode_double_case_inv_simps*)
apply(*erule_tac disjE*)
apply (*metis cell.distinct*(*1*) *empty_replicate hd_append2 hd_replicate list.sel*(*1*) *not_gr_zero*)
apply *force*.

lemma *wcode_on_checking_1_Elim*[*elim*]: $[\![$*wcode_on_checking_1 ires rs* (*b*, *Oc* # *ba*);*Oc* # *b* = *aa* $\wedge$ *list* = *ba*$]\!]$
$\Longrightarrow$ *wcode_erase1 ires rs* (*aa*, *ba*)
apply(*simp only*: *wcode_double_case_inv_simps*)
apply(*erule_tac exE*)+ **by** *auto*

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*
by(*auto simp*: *wcode_double_case_inv_simps*)

lemma *wcode_erase1_nonempty_snd*[*simp*]: *wcode_erase1 ires rs* (*b*, []) = *False*
apply(*simp add*: *wcode_double_case_inv_simps*)
done

lemma *wcode_on_right_moving_1_nonempty_snd*[*simp*]: *wcode_on_right_moving_1 ires rs* (*b*, []) = *False*
apply(*simp add*: *wcode_double_case_inv_simps*)
done

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*)
apply(*simp only*: *wcode_double_case_inv_simps*)
apply(*erule_tac exE*)+
apply(*rename_tac ml mr rn*)
apply(*rule_tac x* = *Suc ml* **in** *exI*, *rule_tac x* = *mr* − *Suc 0* **in** *exI*,
rule_tac x = *rn* **in** *exI*)
apply(*simp*)
apply(*case_tac mr*, *simp*, *simp*)
done

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*)
apply(*simp only*: *wcode_double_case_inv_simps*)
apply(*erule_tac exE*)+

apply(*rename_tac ml mr rn*)
apply(*rule_tac x* = *Suc 0* **in** *exI*, *rule_tac x* = *rs* **in** *exI*,
rule_tac x = *ml* − *Suc* (*Suc 0*) **in** *exI*, *rule_tac x* = *rn* **in** *exI*)
apply(*case_tac mr*, *simp_all*)
apply(*case_tac ml*, *simp*, *case_tac nat*, *simp*, *simp*)
done

lemma *wcode_erase1_BkE*[*elim*]:
assumes *wcode_erase1 ires rs* (*b*, *Bk* # *ba*) *Bk* # *b* = *aa* ∧ *list* = *ba c* = *Bk* # *ba*
shows *wcode_on_right_moving_1 ires rs* (*aa*, *ba*)
proof −
from *assms* **obtain** *rn ln* **where** *b* = *Oc* # *ires*
tl (*Bk* # *ba*) = *Bk* ↑ *ln* @ *Bk* # *Bk* # *Oc* ↑ *Suc rs* @ *Bk* ↑ *rn*
unfolding *wcode_double_case_inv_simps* **by** *auto*
thus *?thesis* **using** *assms*(*2*−) **unfolding** *wcode_double_case_inv_simps*
apply(*rule_tac x* = *Suc 0* **in** *exI*, *rule_tac x* = *Suc* (*Suc ln*) **in** *exI*,
rule_tac x = *rn* **in** *exI*, *simp add*: *exp_ind del*: *replicate_Suc*)
done
qed

lemma *wcode_erase1_OcE*[*elim*]: ⟦*wcode_erase1 ires rs* (*aa*, *Oc* # *list*); *b* = *aa* ∧ *Bk* # *list* = *ba*⟧ ⟹
wcode_erase1 ires rs (*aa*, *ba*)
unfolding *wcode_double_case_inv_simps*
by *auto auto*

lemma *wcode_goon_right_moving_1_emptyE*[*elim*]:
assumes *wcode_goon_right_moving_1 ires rs* (*aa*, []) *b* = *aa* ∧ [*Oc*] = *ba*
shows *wcode_backto_standard_pos ires rs* (*aa*, *ba*)
proof −
from *assms* **obtain** *ml ln rn mr* **where** *aa* = *Oc* ↑ *ml* @ *Bk* # *Bk* # *Bk* ↑ *ln* @ *Oc* # *ires*
[] = *Oc* ↑ *mr* @ *Bk* ↑ *rn ml* + *mr* = *Suc rs*
by(*auto simp*:*wcode_double_case_inv_simps*)
thus *?thesis* **using** *assms*(*2*)
apply(*simp only*: *wcode_double_case_inv_simps*)
apply(*rule_tac disjI2*)
apply(*simp only*:*wcode_backto_standard_pos_O.simps*)
apply(*rule_tac x* = *ml* **in** *exI*, *rule_tac x* = *Suc 0* **in** *exI*, *rule_tac x* = *ln* **in** *exI*,
rule_tac x = *rn* **in** *exI*, *simp*)
done
qed

lemma *wcode_goon_right_moving_1_BkE*[*elim*]:
assumes *wcode_goon_right_moving_1 ires rs* (*aa*, *Bk* # *list*) *b* = *aa* ∧ *Oc* # *list* = *ba*
shows *wcode_backto_standard_pos ires rs* (*aa*, *ba*)
proof −
from *assms* **obtain** *ln rn* **where** *aa* = *Oc* ↑ *Suc rs* @ *Bk* ↑ *Suc* (*Suc ln*) @ *Oc* # *ires*
Bk # *list* = *Bk* ↑ *rn b* = *Oc* ↑ *Suc rs* @ *Bk* ↑ *Suc* (*Suc ln*) @ *Oc* # *ires ba* = *Oc* # *list*
by(*auto simp*:*wcode_double_case_inv_simps*)
thus *?thesis* **using** *assms*(*2*)

apply(*simp only*: *wcode_double_case_inv_simps wcode_backto_standard_pos_O.simps*)
apply(*rule_tac disjI2*)
apply(*rule exI*[*of _ Suc rs*], *rule exI*[*of _ Suc 0*], *rule_tac x* = *ln* **in** *exI*,
rule_tac x = *rn* − *Suc 0* **in** *exI*, *simp*)
apply(*cases rn*;*auto*)
done
qed

lemma *wcode_goon_right_moving_1_OcE*[*elim*]:
assumes *wcode_goon_right_moving_1 ires rs* (*b*, *Oc* # *ba*) *Oc* # *b* = *aa* ∧ *list* = *ba*
shows *wcode_goon_right_moving_1 ires rs* (*aa*, *ba*)
proof −
from *assms* **obtain** *ml mr ln rn* **where**
b = *Oc* ↑ *ml* @ *Bk* # *Bk* # *Bk* ↑ *ln* @ *Oc* # *ires* ∧
Oc # *ba* = *Oc* ↑ *mr* @ *Bk* ↑ *rn* ∧ *ml* + *mr* = *Suc rs*
unfolding *wcode_double_case_inv_simps* **by** *auto*
with *assms*(*2*) **show** *?thesis* **unfolding** *wcode_double_case_inv_simps*
apply(*rule_tac x* = *Suc ml* **in** *exI*, *rule_tac x* = *mr* − *Suc 0* **in** *exI*,
rule_tac x = *ln* **in** *exI*, *rule_tac x* = *rn* **in** *exI*)
apply(*simp*)
apply(*case_tac mr*, *simp*, *case_tac rn*, *simp_all*)
done
qed

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*)
apply(*simp only*: *wcode_double_case_inv_simps wcode_backto_standard_pos_B.simps*
wcode_backto_standard_pos_O.simps wcode_before_double.simps)
apply(*erule_tac disjE*)
apply(*erule_tac exE*)+
by *auto*

lemma *wcode_backto_standard_pos_no_Oc*[*simp*]: *wcode_backto_standard_pos ires rs* ([], *Oc* # *list*) = *False*
apply(*auto simp*: *wcode_backto_standard_pos.simps wcode_backto_standard_pos_B.simps*
wcode_backto_standard_pos_O.simps)
done

lemma *wcode_backto_standard_pos_nonempty_snd*[*simp*]: *wcode_backto_standard_pos ires rs* (*b*, []) = *False*
apply(*auto simp*: *wcode_backto_standard_pos.simps wcode_backto_standard_pos_B.simps*
wcode_backto_standard_pos_O.simps)
done

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*)
apply(*simp only*: *wcode_backto_standard_pos.simps wcode_backto_standard_pos_B.simps*

```
    wcode_backto_standard_pos_O.simps)
  apply(erule_tac disjE)
   apply(simp)
  apply(erule_tac exE)+
  apply(simp)
  apply (rename_tac ml mr ln rn)
  apply(case_tac ml)
   apply(rule_tac disjI1, rule_tac conjI)
    apply(rule_tac x = ln  in exI, force, rule_tac x = rn in exI, force, force).

declare nth_of.simps[simp del] fetch.simps[simp del]
lemma wcode_double_case_first_correctness:
  let P = (λ (st, l, r). st = 13) in
       let Q = (λ (st, l, r). wcode_double_case_inv st ires rs (l, r)) in
         let f = (λ stp. steps0 (Suc 0, Bk # Bk↑(m) @ Oc # Oc # ires, Bk # Oc↑(Suc rs) @
Bk↑(n)) t_wcode_main stp) in
      ∃ n .P (f n) ∧ Q (f (n::nat))
proof −
  let ?P = (λ (st, l, r). st = 13)
  let ?Q = (λ (st, l, r). wcode_double_case_inv st ires rs (l, r))
  let ?f = (λ stp. steps0 (Suc 0, Bk # Bk↑(m) @ Oc # Oc # ires, Bk # Oc↑(Suc rs) @ Bk↑(n))
t_wcode_main stp)
  have ∃ n. ?P (?f n) ∧ ?Q (?f (n::nat))
  proof(rule_tac halt_lemma2)
    show wf wcode_double_case_le
      by auto
  next
    show ∀ na. ¬ ?P (?f na) ∧ ?Q (?f na) ⟶
                ?Q (?f (Suc na)) ∧ (?f (Suc na), ?f na) ∈ wcode_double_case_le
    proof(rule_tac allI, case_tac ?f na, simp)
      fix na a b c
      show a ≠ 13 ∧ wcode_double_case_inv a ires rs (b, c) ⟶
              (case step0 (a, b, c) t_wcode_main of (st, x) ⇒
                 wcode_double_case_inv st ires rs x) ∧
               (step0 (a, b, c) t_wcode_main, a, b, c) ∈ wcode_double_case_le
        apply(rule_tac impI, simp add: wcode_double_case_inv.simps)
        apply(auto split: if_splits simp: step.simps,
          case_tac [!] c, simp_all, case_tac [!] (c::cell list)!0)
                    apply(simp_all add: wcode_double_case_inv.simps wcode_double_case_le_def
          lex_pair_def)
                apply(auto split: if_splits)
        done
    qed
  next
    show ?Q (?f 0)
      apply(simp add: steps.simps wcode_double_case_inv.simps
        wcode_on_left_moving_1.simps
        wcode_on_left_moving_1_B.simps)
      apply(rule_tac disjI1)
      apply(rule_tac x = Suc m in exI, simp)
```

```
    apply(rule_tac x = Suc 0 in exI, simp)
    done
  next
   show ¬ ?P (?f 0)
    apply(simp add: steps.simps)
    done
  qed
  thus let P = λ(st, l, r). st = 13;
   Q = λ(st, l, r). wcode_double_case_inv st ires rs (l, r);
     f = steps0 (Suc 0, Bk # Bk↑(m) @ Oc # Oc # ires, Bk # Oc↑(Suc rs) @ Bk↑(n))
t_wcode_main
   in ∃ n. P (f n) ∧ Q (f n)
   apply(simp)
   done
qed

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 −
  assume h:
   steps0 (st, l, r) tp stp = (st', l', r')
   0 < st'
   length tp1 mod 2 = 0
  from h have
   steps (st + length tp1 div 2, l, r) (tp1 @ shift tp (length tp1 div 2), 0) stp =
                  (st' + length tp1 div 2, l', r')
   by(rule_tac tm_append_second_steps_eq, simp_all)
  then have steps (st + length tp1 div 2, l, r) ((tp1 @ shift tp (length tp1 div 2)) @ tp2, 0) stp =
                  (st' + length tp1 div 2, l', r')
   using h
   apply(rule_tac tm_append_first_steps_eq, simp_all)
   done
  thus ?thesis
   by simp
qed

declare start_of.simps[simp del]

lemma twice_lemma: rec_exec rec_twice [rs] = 2*rs
  by(auto simp: rec_twice_def rec_exec.simps)

lemma t_twice_correct:
  ∃ stp ln rn. steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n))
  (tm_of abc_twice @ shift (mopup (Suc 0)) ((length (tm_of abc_twice) div 2))) stp =
  (0, Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (2 ∗ rs)) @ Bk↑(rn))
```

proof(*case_tac rec_ci rec_twice*)
 fix *a b c*
 assume *h*: *rec_ci rec_twice* = (*a*, *b*, *c*)
 have ∃ *stp m l*. *steps0* (*Suc 0*, *Bk* # *Bk* # *ires*, <[*rs*]> @ *Bk*↑(*n*)) (*tm_of abc_twice* @ *shift* (*mopup* (*length* [*rs*]))
 (*length* (*tm_of abc_twice*) *div 2*)) *stp* = (*0*, *Bk*↑(*m*) @ *Bk* # *Bk* # *ires*, *Oc*↑(*Suc* (*rec_exec rec_twice* [*rs*])) @ *Bk*↑(*l*))
 thm *recursive_compile_to_tm_correct1*
 proof(*rule_tac recursive_compile_to_tm_correct1*)
 show *rec_ci rec_twice* = (*a*, *b*, *c*) **by** (*simp add*: *h*)
 next
 show *terminate rec_twice* [*rs*]
 apply(*rule_tac primerec_terminate*, *auto*)
 apply(*simp add*: *rec_twice_def*, *auto simp*: *constn*.*simps numeral_2_eq_2*)
 by(*auto*)
 next
 show *tm_of abc_twice* = *tm_of* (*a* [+] *dummy_abc* (*length* [*rs*]))
 using *h*
 by(*simp add*: *abc_twice_def*)
 qed
 thus *?thesis*
 apply(*simp add*: *tape_of_list_def tape_of_nat_def rec_exec*.*simps twice_lemma*)
 done
qed

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

lemma *adjust_fetch0*:
 ⟦*0* < *a*; *a* ≤ *length ap div 2*; *fetch ap a b* = (*aa*, *0*)⟧
 ⟹ *fetch* (*adjust0 ap*) *a b* = (*aa*, *Suc* (*length ap div 2*))
 apply(*case_tac b*, *auto simp*: *fetch*.*simps nth_of*.*simps nth_map*
 split: *if_splits*)
 apply(*case_tac* [!] *a*, *auto simp*: *fetch*.*simps nth_of*.*simps*)
 done

lemma *adjust_fetch_norm*:
 ⟦*st* > *0*; *st* ≤ *length tp div 2*; *fetch ap st b* = (*aa*, *ns*); *ns* ≠ *0*⟧
 ⟹ *fetch* (*adjust0 ap*) *st b* = (*aa*, *ns*)
 apply(*case_tac b*, *auto simp*: *fetch*.*simps nth_of*.*simps nth_map*
 split: *if_splits*)
 apply(*case_tac* [!] *st*, *auto simp*: *fetch*.*simps nth_of*.*simps*)
 done

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

lemma *adjust_step_eq*:
 assumes *exec*: *step0* (*st*,*l*,*r*) *ap* = (*st*′, *l*′, *r*′)
 and *wf_tm*: *tm_wf* (*ap*, *0*)
 and *notfinal*: *st*′ > *0*
 shows *step0* (*st*, *l*, *r*) (*adjust0 ap*) = (*st*′, *l*′, *r*′)

```
  using assms
proof -
  have st > 0
    using assms
    by(case_tac st, simp_all add: step.simps fetch.simps)
  moreover hence st ≤ (length ap) div 2
    using assms
    apply(case_tac st ≤ (length ap) div 2, simp)
    apply(case_tac st, auto simp: step.simps fetch.simps)
    apply(case_tac read r, simp_all add: fetch.simps
        nth_of.simps adjust.simps tm_wf.simps split: if_splits)
     apply(auto simp: mod_ex2)
    done
  ultimately have fetch (adjust0 ap) st (read r) = fetch ap st (read r)
    using assms
    apply(case_tac fetch ap st (read r))
    apply(drule_tac adjust_fetch_norm, simp_all)
    apply(simp add: step.simps)
    done
  thus ?thesis
    using exec
    by(simp add: step.simps)
qed

declare adjust.simps[simp del]

lemma adjust_steps_eq:
  assumes exec: steps0 (st,l,r) ap stp = (st', l', r')
    and wf_tm: tm_wf (ap, 0)
    and notfinal: st' > 0
  shows steps0 (st, l, r) (adjust0 ap) stp = (st', l', r')
  using exec notfinal
proof(induct stp arbitrary: st' l' r')
  case 0
  thus ?case
    by(simp add: steps.simps)
next
  case (Suc stp st' l' r')
  have ind: ⋀st' l' r'. ⟦steps0 (st, l, r) ap stp = (st', l', r'); 0 < st'⟧
    ⟹ steps0 (st, l, r) (adjust0 ap) stp = (st', l', r') by fact
  have h: steps0 (st, l, r) ap (Suc stp) = (st', l', r') by fact
  have g:   0 < st' by fact
  obtain st'' l'' r'' where a: steps0 (st, l, r) ap stp = (st'', l'', r'')
    by (metis prod_cases3)
  hence c:0 < st''
    using h g
    apply(simp add: step_red)
    apply(case_tac st'', auto)
    done
  hence b: steps0 (st, l, r) (adjust0 ap) stp = (st'', l'', r'')
```

```
    using a
    by(rule_tac ind, simp_all)
  thus ?case
    using assms a b h g
    apply(simp add: step_red)
    apply(rule_tac adjust_step_eq, simp_all)
    done
qed

lemma adjust_halt_eq:
  assumes exec: steps0 (1, l, r) ap stp = (0, l′, r′)
    and tm_wf: tm_wf (ap, 0)
  shows ∃ stp. steps0 (Suc 0, l, r) (adjust0 ap) stp =
        (Suc (length ap div 2), l′, r′)
proof −
  have ∃ stp. ¬ is_final (steps0 (1, l, r) ap stp) ∧ (steps0 (1, l, r) ap (Suc stp) = (0, l′, r′))
    using exec
    by(erule_tac before_final)
  then obtain stpa where a:
    ¬ is_final (steps0 (1, l, r) ap stpa) ∧ (steps0 (1, l, r) ap (Suc stpa) = (0, l′, r′)) ..
  obtain sa la ra where b:steps0 (1, l, r) ap stpa = (sa, la, ra)  by (metis prod_cases3)
  hence c: steps0 (Suc 0, l, r) (adjust0 ap) stpa = (sa, la, ra)
    using assms a
    apply(rule_tac adjust_steps_eq, simp_all)
    done
  have d: sa ≤ length ap div 2
    using steps_in_range[of (l, r) ap stpa] a tm_wf b
    by(simp)
  obtain ac ns where e: fetch ap sa (read ra) = (ac, ns)
    by (metis prod.exhaust)
  hence f: ns = 0
    using b a
    apply(simp add: step_red step.simps)
    done
  have k: fetch (adjust0 ap) sa (read ra) = (ac, Suc (length ap div 2))
    using a b c d e f
    apply(rule_tac adjust_fetch0, simp_all)
    done
  from a b e f k and c show ?thesis
    apply(rule_tac x = Suc stpa in exI)
    apply(simp add: step_red, auto)
    apply(simp add: step.simps)
    done
qed

declare tm_wf.simps[simp del]

lemma tm_wf_t_twice_compile [simp]: tm_wf (t_twice_compile, 0)
  apply(simp only: t_twice_compile_def)
  apply(rule_tac wf_tm_from_abacus, simp)
```

```
 done

lemma t_twice_change_term_state:
 ∃ stp ln rn. steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n)) t_twice stp
   = (Suc t_twice_len, Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (2 ∗ rs)) @ Bk↑(rn))
proof −
 have ∃ stp ln rn. steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n))
  (tm_of abc_twice @ shift (mopup (Suc 0)) ((length (tm_of abc_twice) div 2))) stp =
  (0, Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (2 ∗ rs)) @ Bk↑(rn))
  by(rule_tac t_twice_correct)
 then obtain stp ln rn where  steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n))
  (tm_of abc_twice @ shift (mopup (Suc 0)) ((length (tm_of abc_twice) div 2))) stp =
  (0, Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (2 ∗ rs)) @ Bk↑(rn)) by blast
 hence ∃ stp. steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n))
  (adjust0 t_twice_compile) stp
    = (Suc (length t_twice_compile div 2), Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (2 ∗ rs)) @
Bk↑(rn))
  apply(rule_tac stp = stp in adjust_halt_eq)
   apply(simp add: t_twice_compile_def, auto)
  done
 then obtain stpb where
  steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n))
  (adjust0 t_twice_compile) stpb
    = (Suc (length t_twice_compile div 2), Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (2 ∗ rs)) @
Bk↑(rn)) ..
 thus ?thesis
  apply(simp add: t_twice_def t_twice_len_def)
  by metis
qed

lemma length_t_wcode_main_first_part_even[intro]: length t_wcode_main_first_part mod 2 = 0
 apply(auto simp: t_wcode_main_first_part_def)
 done

lemma t_twice_append_pre:
 steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n)) t_twice stp
 = (Suc t_twice_len, Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (2 ∗ rs)) @ Bk↑(rn))
  =⇒ steps0 (Suc 0 + length t_wcode_main_first_part div 2, Bk # Bk # ires, Oc↑(Suc rs) @
Bk↑(n))
   (t_wcode_main_first_part @ shift t_twice (length t_wcode_main_first_part div 2) @
    ([(L, 1), (L, 1)] @ shift t_fourtimes (t_twice_len + 13) @ [(L, 1), (L, 1)])) stp
  = (Suc (t_twice_len) + length t_wcode_main_first_part div 2,
        Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (2 ∗ rs)) @ Bk↑(rn))
 by(rule_tac tm_append_shift_append_steps, auto)

lemma t_twice_append:
 ∃ stp ln rn. steps0 (Suc 0 + length t_wcode_main_first_part div 2, Bk # Bk # ires, Oc↑(Suc
rs) @ Bk↑(n))
   (t_wcode_main_first_part @ shift t_twice (length t_wcode_main_first_part div 2) @
    ([(L, 1), (L, 1)] @ shift t_fourtimes (t_twice_len + 13) @ [(L, 1), (L, 1)])) stp
```

$= (Suc\ (t\_twice\_len) + length\ t\_wcode\_main\_first\_part\ div\ 2,\ Bk\uparrow(ln)\ @\ Bk\ \#\ Bk\ \#\ ires,$
$Oc\uparrow(Suc\ (2 * rs))\ @\ Bk\uparrow(rn))$
using *t_twice_change_term_state*[*of ires rs n*]
apply(*erule_tac exE*)
apply(*erule_tac exE*)
apply(*erule_tac exE*)
apply(*drule_tac t_twice_append_pre*)
apply(*rename_tac stp ln rn*)
apply(*rule_tac x = stp* **in** *exI*, *rule_tac x = ln* **in** *exI*, *rule_tac x = rn* **in** *exI*)
apply(*simp*)
done

lemma *mopup_mod2*: $length\ (mopup\ k)\ mod\ 2\ = 0$
by(*auto simp*: *mopup.simps*)

lemma *fetch_t_wcode_main_Oc*[*simp*]: $fetch\ t\_wcode\_main\ (Suc\ (t\_twice\_len + length\ t\_wcode\_main\_first\_part$
$div\ 2))\ Oc$
$= (L,\ Suc\ 0)$
apply(*subgoal_tac* $length\ (t\_twice)\ mod\ 2 = 0$)
apply(*simp add*: *t_wcode_main_def nth_append fetch.simps t_wcode_main_first_part_def*
nth_of.simps t_twice_len_def, *auto*)
apply(*simp add*: *t_twice_def t_twice_compile_def*)
using *mopup_mod2*[*of 1*]
apply(*simp*)
done

lemma *wcode_jump1*:
$\exists\ stp\ ln\ rn.\ steps0\ (Suc\ (t\_twice\_len) + length\ t\_wcode\_main\_first\_part\ div\ 2,$
$Bk\uparrow(m)\ @\ Bk\ \#\ Bk\ \#\ ires,\ Oc\uparrow(Suc\ (2 * rs))\ @\ Bk\uparrow(n))$
$t\_wcode\_main\ stp$
$= (Suc\ 0,\ Bk\uparrow(ln)\ @\ Bk\ \#\ ires,\ Bk\ \#\ Oc\uparrow(Suc\ (2 * rs))\ @\ Bk\uparrow(rn))$
apply(*rule_tac x = Suc 0* **in** *exI*, *rule_tac x = m* **in** *exI*, *rule_tac x = n* **in** *exI*)
apply(*simp add*: *steps.simps step.simps exp_ind*)
apply(*case_tac m*, *simp_all*)
apply(*simp add*: *exp_ind*[*THEN sym*])
done

lemma *wcode_main_first_part_len*[*simp*]:
$length\ t\_wcode\_main\_first\_part = 24$
apply(*simp add*: *t_wcode_main_first_part_def*)
done

lemma *wcode_double_case*:
shows $\exists\ stp\ ln\ rn.\ steps0\ (Suc\ 0,\ Bk\ \#\ Bk\uparrow(m)\ @\ Oc\ \#\ Oc\ \#\ ires,\ Bk\ \#\ Oc\uparrow(Suc\ rs)\ @$
$Bk\uparrow(n))\ t\_wcode\_main\ stp =$
$(Suc\ 0,\ Bk\ \#\ Bk\uparrow(ln)\ @\ Oc\ \#\ ires,\ Bk\ \#\ Oc\uparrow(Suc\ (2 * rs + 2))\ @\ Bk\uparrow(rn))$
(**is** $\exists\ stp\ ln\ rn.\ ?tm\ stp\ ln\ rn$)
proof $-$
from *wcode_double_case_first_correctness*[*of ires rs m n*] **obtain** *na ln rn* **where**
$steps0\ (Suc\ 0,\ Bk\ \#\ Bk \uparrow m\ @\ Oc\ \#\ Oc\ \#\ ires,\ Bk\ \#\ Oc\ \#\ Oc \uparrow rs\ @\ Bk \uparrow n)\ t\_wcode\_main$

na
  $= (13, Bk \# Bk \# Bk \uparrow ln @ Oc \# ires, Oc \# Oc \# Oc \uparrow rs @ Bk \uparrow rn)$
 by(*auto simp*: *wcode_double_case_inv.simps wcode_before_double.simps*)
 hence $\exists stp\ ln\ rn.\ steps0\ (Suc\ 0, Bk \# Bk\uparrow(m) @ Oc \# Oc \# ires, Bk \# Oc\uparrow(Suc\ rs) @ Bk\uparrow(n))\ t\_wcode\_main\ stp =$
  $(13, Bk \# Bk \# Bk\uparrow(ln) @ Oc \# ires, Oc\uparrow(Suc\ (Suc\ rs)) @ Bk\uparrow(rn))$
 by(*case_tac steps0* $(Suc\ 0, Bk \# Bk\uparrow(m) @ Oc \# Oc \# ires,$
  $Bk \# Oc\uparrow(Suc\ rs) @ Bk\uparrow(n))$ *t_wcode_main na*, *auto*)
 from *this* **obtain** *stpa lna rna* **where** *stp1*:
 $steps0\ (Suc\ 0, Bk \# Bk\uparrow(m) @ Oc \# Oc \# ires, Bk \# Oc\uparrow(Suc\ rs) @ Bk\uparrow(n))\ t\_wcode\_main\ stpa =$
 $(13, Bk \# Bk \# Bk\uparrow(lna) @ Oc \# ires, Oc\uparrow(Suc\ (Suc\ rs)) @ Bk\uparrow(rna))$ **by** *blast*
 from *t_twice_append*[*of* $Bk\uparrow(lna) @ Oc \# ires$ *Suc rs rna*] **obtain** *stp ln rn*
 where $steps0\ (Suc\ 0 + length\ t\_wcode\_main\_first\_part\ div\ 2,$
  $Bk \# Bk \# Bk \uparrow lna @ Oc \# ires, Oc \uparrow Suc\ (Suc\ rs) @ Bk \uparrow rna)$
  $(t\_wcode\_main\_first\_part @ shift\ t\_twice\ (length\ t\_wcode\_main\_first\_part\ div\ 2) @$
  $[(L, 1), (L, 1)] @ shift\ t\_fourtimes\ (t\_twice\_len + 13) @ [(L, 1), (L, 1)])\ stp =$
  $(Suc\ t\_twice\_len + length\ t\_wcode\_main\_first\_part\ div\ 2,$
  $Bk \uparrow ln @ Bk \# Bk \# Bk \uparrow lna @ Oc \# ires, Oc \uparrow Suc\ (2 * Suc\ rs) @ Bk \uparrow rn)$ **by** *blast*
 hence $\exists\ stp\ ln\ rn.\ steps0\ (13, Bk \# Bk \# Bk\uparrow(lna) @ Oc \# ires, Oc\uparrow(Suc\ (Suc\ rs)) @ Bk\uparrow(rna))\ t\_wcode\_main\ stp =$
 $(13 + t\_twice\_len, Bk \# Bk \# Bk\uparrow(ln) @ Oc \# ires, Oc\uparrow(Suc\ (Suc\ (Suc\ (2 *rs)))) @ Bk\uparrow(rn))$
 using *t_twice_append*[*of* $Bk\uparrow(lna) @ Oc \# ires$ *Suc rs rna*]
 apply(*simp*)
 apply(*rule_tac* $x = stp$ **in** *exI*, *rule_tac* $x = ln + lna$ **in** *exI*,
  rule_tac $x = rn$ **in** *exI*)
 apply(*simp add*: *t_wcode_main_def*)
 apply(*simp add*: *replicate_Suc*[*THEN sym*] *replicate_add* [*THEN sym*] *del*: *replicate_Suc*)
 done
 from *this* **obtain** *stpb lnb rnb* **where** *stp2*:
 $steps0\ (13, Bk \# Bk \# Bk\uparrow(lna) @ Oc \# ires, Oc\uparrow(Suc\ (Suc\ rs)) @ Bk\uparrow(rna))\ t\_wcode\_main\ stpb =$
 $(13 + t\_twice\_len, Bk \# Bk \# Bk\uparrow(lnb) @ Oc \# ires, Oc\uparrow(Suc\ (Suc\ (Suc\ (2 *rs)))) @ Bk\uparrow(rnb))$ **by** *blast*
 from *wcode_jump1*[*of lnb* $Oc \# ires$ *Suc rs rnb*] **obtain** *stp ln rn* **where**
 $steps0\ (Suc\ t\_twice\_len + length\ t\_wcode\_main\_first\_part\ div\ 2,$
  $Bk \uparrow lnb @ Bk \# Bk \# Oc \# ires, Oc \uparrow Suc\ (2 * Suc\ rs) @ Bk \uparrow rnb)\ t\_wcode\_main\ stp =$
 $(Suc\ 0, Bk \uparrow ln @ Bk \# Oc \# ires, Bk \# Oc \uparrow Suc\ (2 * Suc\ rs) @ Bk \uparrow rn)$ **by** *metis*
 hence $steps0\ (13 + t\_twice\_len, Bk \# Bk \# Bk\uparrow(lnb) @ Oc \# ires,$
 $Oc\uparrow(Suc\ (Suc\ (Suc\ (2 *rs)))) @ Bk\uparrow(rnb))\ t\_wcode\_main\ stp =$
 $(Suc\ 0, Bk \# Bk\uparrow(ln) @ Oc \# ires, Bk \# Oc\uparrow(Suc\ (Suc\ (Suc\ (2 *rs)))) @ Bk\uparrow(rn))$
 apply(*auto simp add*: *t_wcode_main_def*)
 apply(*subgoal_tac* $Bk\uparrow(lnb) @ Bk \# Bk \# Oc \# ires = Bk \# Bk \# Bk\uparrow(lnb) @ Oc \# ires$, *simp*)
 apply(*simp add*: *replicate_Suc*[*THEN sym*] *exp_ind*[*THEN sym*] *del*: *replicate_Suc*)
 apply(*simp*)
 apply(*simp add*: *replicate_Suc*[*THEN sym*] *exp_ind del*: *replicate_Suc*)
 done
 hence $\exists stp\ ln\ rn.\ steps0\ (13 + t\_twice\_len, Bk \# Bk \# Bk\uparrow(lnb) @ Oc \# ires,$

Oc↑(Suc (Suc (Suc (2 ∗rs)))) @ Bk↑(rnb)) t_wcode_main stp =
(Suc 0, Bk # Bk↑(ln) @ Oc # ires, Bk # Oc↑(Suc (Suc (Suc (2 ∗rs)))) @ Bk↑(rn))
by blast
from this **obtain** stpc lnc rnc **where** stp3:
steps0 (13 + t_twice_len, Bk # Bk # Bk↑(lnb) @ Oc # ires,
Oc↑(Suc (Suc (Suc (2 ∗rs)))) @ Bk↑(rnb)) t_wcode_main stpc =
(Suc 0, Bk # Bk↑(lnc) @ Oc # ires, Bk # Oc↑(Suc (Suc (Suc (2 ∗rs)))) @ Bk↑(rnc))
by blast
from stp1 stp2 stp3 **have** ?tm (stpa + stpb + stpc) lnc rnc **by** simp
thus ?thesis **by** blast
qed

fun wcode_on_left_moving_2_B :: bin_inv_t
where
wcode_on_left_moving_2_B ires rs (l, r) =
(∃ ml mr rn. l = Bk↑(ml) @ Oc # Bk # Oc # ires ∧
r = Bk↑(mr) @ Oc↑(Suc rs) @ Bk↑(rn) ∧
ml + mr > Suc 0 ∧ mr > 0)

fun wcode_on_left_moving_2_O :: bin_inv_t
where
wcode_on_left_moving_2_O ires rs (l, r) =
(∃ ln rn. l = Bk # Oc # ires ∧
r = Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun wcode_on_left_moving_2 :: bin_inv_t
where
wcode_on_left_moving_2 ires rs (l, r) =
(wcode_on_left_moving_2_B ires rs (l, r) ∨
wcode_on_left_moving_2_O ires rs (l, r))

fun wcode_on_checking_2 :: bin_inv_t
where
wcode_on_checking_2 ires rs (l, r) =
(∃ ln rn. l = Oc#ires ∧
r = Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun wcode_goon_checking :: bin_inv_t
where
wcode_goon_checking ires rs (l, r) =
(∃ ln rn. l = ires ∧
r = Oc # Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun wcode_right_move :: bin_inv_t
where
wcode_right_move ires rs (l, r) =
(∃ ln rn. l = Oc # ires ∧
r = Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))

fun *wcode_erase2* :: *bin_inv_t*
where
wcode_erase2 ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l = Bk \# Oc \# ires \wedge$
$tl\ r = Bk\uparrow(ln)\ @\ Bk \# Bk \# Oc\uparrow(Suc\ rs)\ @\ Bk\uparrow(rn))$

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 $(l, r) =$
$(\exists\ ln\ rn.\ l = Bk \# Bk\uparrow(ln)\ @\ Oc \# ires \wedge$
$r = Bk \# Oc\uparrow(Suc\ (Suc\ rs))\ @\ Bk\uparrow(rn))$

fun *wcode_backto_standard_pos_2_O* :: *bin_inv_t*
where
wcode_backto_standard_pos_2_O ires rs $(l, r) =$
$(\exists\ ml\ mr\ ln\ rn.\ l = Oc\uparrow(ml\ )@\ Bk \# Bk \# Bk\uparrow(ln)\ @\ Oc \# ires \wedge$
$r = Oc\uparrow(mr)\ @\ Bk\uparrow(rn) \wedge$
$ml + mr = (Suc\ (Suc\ rs)) \wedge mr > 0)$

fun *wcode_backto_standard_pos_2* :: *bin_inv_t*
where
wcode_backto_standard_pos_2 ires rs $(l, r) =$
(*wcode_backto_standard_pos_2_O ires rs* $(l, r) \vee$
wcode_backto_standard_pos_2_B ires rs $(l, r))$

fun *wcode_before_fourtimes* :: *bin_inv_t*
where
wcode_before_fourtimes ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l = Bk \# Bk \# Bk\uparrow(ln)\ @\ Oc \# ires \wedge$
$r = Oc\uparrow(Suc\ (Suc\ rs))\ @\ Bk\uparrow(rn))$

declare *wcode_on_left_moving_2_B.simps*[*simp del*] *wcode_on_left_moving_2.simps*[*simp del*]
wcode_on_left_moving_2_O.simps[*simp del*] *wcode_on_checking_2.simps*[*simp del*]
wcode_goon_checking.simps[*simp del*] *wcode_right_move.simps*[*simp del*]
wcode_erase2.simps[*simp del*]
wcode_on_right_moving_2.simps[*simp del*] *wcode_goon_right_moving_2.simps*[*simp del*]
wcode_backto_standard_pos_2_B.simps[*simp del*] *wcode_backto_standard_pos_2_O.simps*[*simp*

del]
wcode_backto_standard_pos_2.simps[*simp del*]

lemmas *wcode_fourtimes_invs* =
wcode_on_left_moving_2_B.simps wcode_on_left_moving_2.simps
wcode_on_left_moving_2_O.simps wcode_on_checking_2.simps
wcode_goon_checking.simps wcode_right_move.simps
wcode_erase2.simps
wcode_on_right_moving_2.simps wcode_goon_right_moving_2.simps
wcode_backto_standard_pos_2_B.simps wcode_backto_standard_pos_2_O.simps
wcode_backto_standard_pos_2.simps

fun *wcode_fourtimes_case_inv* :: *nat* ⇒ *bin_inv_t*
where
wcode_fourtimes_case_inv st ires rs (*l*, *r*) =
(*if st* = *Suc 0 then wcode_on_left_moving_2 ires rs* (*l*, *r*)
else if st = *Suc* (*Suc 0*) *then wcode_on_checking_2 ires rs* (*l*, *r*)
else if st = *7 then wcode_goon_checking ires rs* (*l*, *r*)
else if st = *8 then wcode_right_move ires rs* (*l*, *r*)
else if st = *9 then wcode_erase2 ires rs* (*l*, *r*)
else if st = *10 then wcode_on_right_moving_2 ires rs* (*l*, *r*)
else if st = *11 then wcode_goon_right_moving_2 ires rs* (*l*, *r*)
else if st = *12 then wcode_backto_standard_pos_2 ires rs* (*l*, *r*)
else if st = *t_twice_len* + *14 then wcode_before_fourtimes ires rs* (*l*, *r*)
else False)

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

fun *wcode_fourtimes_case_state* :: *config* ⇒ *nat*
where
wcode_fourtimes_case_state (*st*, *l*, *r*) = *13* − *st*

fun *wcode_fourtimes_case_step* :: *config* ⇒ *nat*
where
wcode_fourtimes_case_step (*st*, *l*, *r*) =
(*if st* = *Suc 0 then length l*
else if st = *9 then*
(*if hd r* = *Oc then 1*
else 0)
else if st = *10 then length r*
else if st = *11 then length r*
else if st = *12 then length l*
else 0)

fun *wcode_fourtimes_case_measure* :: *config* ⇒ *nat* × *nat*
where
wcode_fourtimes_case_measure (*st*, *l*, *r*) =
(*wcode_fourtimes_case_state* (*st*, *l*, *r*),
wcode_fourtimes_case_step (*st*, *l*, *r*))

definition *wcode_fourtimes_case_le* :: (*config* × *config*) *set*
where *wcode_fourtimes_case_le* $\stackrel{def}{=}$ (*inv_image lex_pair wcode_fourtimes_case_measure*)

lemma *wf_wcode_fourtimes_case_le*[*intro*]: *wf wcode_fourtimes_case_le*
by(*auto simp*: *wcode_fourtimes_case_le_def*)

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*
by(*auto simp*: *wcode_fourtimes_invs*)

lemma *wcode_on_left_moving_2*[*simp*]:
wcode_on_left_moving_2 ires rs (*b*, *Bk* # *list*) ⟹ *wcode_on_left_moving_2 ires rs* (*tl b*, *hd b* # *Bk* # *list*)
apply(*simp only*: *wcode_fourtimes_invs*)
apply(*erule_tac disjE*)
apply(*erule_tac exE*)+
apply(*simp add*: *gr1_conv_Suc exp_ind replicate_app_Cons_same split*:*hd_repeat_cases*)
apply (*auto simp add*: *gr0_conv_Suc*[*symmetric*] *replicate_app_Cons_same split*:*hd_repeat_cases*)
by *force*+

lemma *wcode_goon_checking_via_2* [*simp*]: *wcode_on_checking_2 ires rs* (*b*, *Bk* # *list*)
⟹ *wcode_goon_checking ires rs* (*tl b*, *hd b* # *Bk* # *list*)
unfolding *wcode_fourtimes_invs* **by** *auto*

lemma *wcode_erase2_via_move* [*simp*]: *wcode_right_move ires rs* (*b*, *Bk* # *list*) ⟹ *wcode_erase2 ires rs* (*Bk* # *b*, *list*)
by (*auto simp*:*wcode_fourtimes_invs*) *auto*

lemma *wcode_on_right_moving_2_via_erase2*[*simp*]:
wcode_erase2 ires rs (*b*, *Bk* # *list*) ⟹ *wcode_on_right_moving_2 ires rs* (*Bk* # *b*, *list*)
apply(*auto simp*:*wcode_fourtimes_invs*)
apply(*rule_tac x* = *Suc* (*Suc 0*) **in** *exI*, *simp add*: *exp_ind*)
by (*metis replicate_Suc_iff_anywhere replicate_app_Cons_same*)

lemma *wcode_on_right_moving_2_move_Bk*[*simp*]: *wcode_on_right_moving_2 ires rs* (*b*, *Bk* # *list*)
⟹ *wcode_on_right_moving_2 ires rs* (*Bk* # *b*, *list*)
apply(*auto simp*: *wcode_fourtimes_invs*) **apply**(*rename_tac ml mr rn*)
apply(*rule_tac x* = *Suc ml* **in** *exI*, *simp*)
apply(*rule_tac x* = *mr* − *1* **in** *exI*, *case_tac mr*,*auto*)
done

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*)
apply(*simp add*: *wcode_fourtimes_invs*, *auto*)
by (*metis add.right_neutral add_Suc_shift append_Cons list.sel*(*3*)
replicate.simps(*1*) *replicate_Suc replicate_Suc_iff_anywhere self_append_conv2*
tl_replicate zero_less_Suc)

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*)
by(*auto simp*: *wcode_fourtimes_invs*)

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*])
apply(*simp only*: *wcode_fourtimes_invs*)
apply(*erule_tac exE*)+
by(*rule_tac disjI1*,*auto*)

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*))
apply(*simp only*: *wcode_fourtimes_invs*)
apply(*erule_tac exE*)+
apply(*auto*)
done

lemma *wcode_right_move_no_Oc*[*simp*]: *wcode_right_move ires rs* (*b*, *Oc* # *list*) = *False*
apply(*auto simp*: *wcode_fourtimes_invs*)
done

lemma *wcode_erase2_Bk_via_Oc*[*simp*]: *wcode_erase2 ires rs* (*b*, *Oc* # *list*)
$\Longrightarrow$ *wcode_erase2 ires rs* (*b*, *Bk* # *list*)
apply(*auto simp*: *wcode_fourtimes_invs*)
done

lemma *wcode_goon_right_moving_2_Oc_move*[*simp*]:
wcode_on_right_moving_2 ires rs (*b*, *Oc* # *list*)
$\Longrightarrow$ *wcode_goon_right_moving_2 ires rs* (*Oc* # *b*, *list*)
apply(*auto simp*: *wcode_fourtimes_invs*)
apply(*rule_tac x* = *Suc 0* **in** *exI*, *auto*)
apply(*rule_tac x* = *ml* − *2* **in** *exI*)
apply(*case_tac ml*, *simp*, *case_tac ml* − *1*, *simp_all*)
done

lemma *wcode_backto_standard_pos_2_exists*[*simp*]: *wcode_backto_standard_pos_2 ires rs* (*b*, *Bk* # *list*)
$\Longrightarrow$ ($\exists$ *ln*. *b* = *Bk* # *Bk*↑(*ln*) @ *Oc* # *ires*) $\wedge$ ($\exists$ *rn*. *list* = *Oc*↑(*Suc* (*Suc rs*)) @ *Bk*↑(*rn*))
by(*simp add*: *wcode_fourtimes_invs*)

lemma *wcode_goon_right_moving_2_move_Oc*[*simp*]: *wcode_goon_right_moving_2 ires rs* (*b*, *Oc* # *list*) $\Longrightarrow$
wcode_goon_right_moving_2 ires rs (*Oc* # *b*, *list*)
apply(*simp only*:*wcode_fourtimes_invs*, *auto*)
apply(*rename_tac ml ln mr rn*)
apply(*case_tac mr*;*force*)
done

lemma *wcode_backto_standard_pos_2_Oc_mv_hd*[*simp*]:
wcode_backto_standard_pos_2 ires rs (*b*, *Oc* # *list*)
$\Longrightarrow$ *wcode_backto_standard_pos_2 ires rs* (*tl b*, *hd b* # *Oc* # *list*)
apply(*simp only*: *wcode_fourtimes_invs*)
apply(*erule_tac disjE*)
apply(*erule_tac exE*)+ **apply**(*rename_tac ml mr ln rn*)
by (*case_tac ml*, *force*,*force*,*force*)

lemma *nonempty_fst*[*simp*]:
wcode_on_left_moving_2 ires rs (*b*, *Bk* # *list*) $\Longrightarrow$ *b* $\neq$ []
wcode_on_checking_2 ires rs (*b*, *Bk* # *list*) $\Longrightarrow$ *b* $\neq$ []
wcode_goon_checking ires rs (*b*, *Bk* # *list*) = *False*
wcode_right_move ires rs (*b*, *Bk* # *list*) $\Longrightarrow$ *b* $\neq$ []
wcode_erase2 ires rs (*b*, *Bk* # *list*) $\Longrightarrow$ *b* $\neq$ []
wcode_on_right_moving_2 ires rs (*b*, *Bk* # *list*) $\Longrightarrow$ *b* $\neq$ []
wcode_goon_right_moving_2 ires rs (*b*, *Bk* # *list*) $\Longrightarrow$ *b* $\neq$ []
wcode_backto_standard_pos_2 ires rs (*b*, *Bk* # *list*) $\Longrightarrow$ *b* $\neq$ []
wcode_on_left_moving_2 ires rs (*b*, *Oc* # *list*) $\Longrightarrow$ *b* $\neq$ []
wcode_goon_right_moving_2 ires rs (*b*, []) $\Longrightarrow$ *b* $\neq$ []
wcode_erase2 ires rs (*b*, *Oc* # *list*) $\Longrightarrow$ *b* $\neq$ []
wcode_on_right_moving_2 ires rs (*b*, *Oc* # *list*) $\Longrightarrow$ *b* $\neq$ []
wcode_goon_right_moving_2 ires rs (*b*, *Oc* # *list*) $\Longrightarrow$ *b* $\neq$ []
wcode_backto_standard_pos_2 ires rs (*b*, *Oc* # *list*) $\Longrightarrow$ *b* $\neq$ []
by(*auto simp*: *wcode_fourtimes_invs*)

lemma *wcode_fourtimes_case_first_correctness*:
shows *let P* = (λ (*st*, *l*, *r*). *st* = *t_twice_len* + *14*) *in*
let Q = (λ (*st*, *l*, *r*). *wcode_fourtimes_case_inv st ires rs* (*l*, *r*)) *in*
let f = (λ *stp*. *steps0* (*Suc 0*, *Bk* # *Bk*↑(*m*) @ *Oc* # *Bk* # *Oc* # *ires*, *Bk* # *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *t_wcode_main stp*) *in*
$\exists$ *n* .*P* (*f n*) $\wedge$ *Q* (*f* (*n*::*nat*))
proof −
let *?P* = (λ (*st*, *l*, *r*). *st* = *t_twice_len* + *14*)
let *?Q* = (λ (*st*, *l*, *r*). *wcode_fourtimes_case_inv st ires rs* (*l*, *r*))
let *?f* = (λ *stp*. *steps0* (*Suc 0*, *Bk* # *Bk*↑(*m*) @ *Oc* # *Bk* # *Oc* # *ires*, *Bk* # *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *t_wcode_main stp*)
have $\exists$ *n* . *?P* (*?f n*) $\wedge$ *?Q* (*?f* (*n*::*nat*))
proof(*rule_tac halt_lemma2*)
show *wf wcode_fourtimes_case_le*

```
    by auto
  next
    have ¬ ?P (?f na) ∧ ?Q (?f na) ⟶
              ?Q (?f (Suc na)) ∧ (?f (Suc na), ?f na) ∈ wcode_fourtimes_case_le for na
      apply(cases ?f na, rule_tac impI)
      apply(simp add: step_red step.simps)
      apply(case_tac snd (snd (?f na)), simp, case_tac [2] hd (snd (snd (?f na))), simp_all)
        apply(simp_all add: wcode_fourtimes_case_inv.simps
          wcode_fourtimes_case_le_def lex_pair_def split: if_splits)
      by(auto simp: wcode_backto_standard_pos_2.simps wcode_backto_standard_pos_2_O.simps
          wcode_backto_standard_pos_2_B.simps gr0_conv_Suc)
    thus ∀ na. ¬ ?P (?f na) ∧ ?Q (?f na) ⟶
              ?Q (?f (Suc na)) ∧ (?f (Suc na), ?f na) ∈ wcode_fourtimes_case_le by auto
  next
    show ?Q (?f 0)
      apply(simp add: steps.simps wcode_fourtimes_case_inv.simps)
      apply(simp add: wcode_on_left_moving_2.simps wcode_on_left_moving_2_B.simps
          wcode_on_left_moving_2_O.simps)
      apply(rule_tac x = Suc m in exI, simp )
      apply(rule_tac x =Suc 0 in exI, auto)
      done
  next
    show ¬ ?P (?f 0)
      apply(simp add: steps.simps)
      done
  qed
  thus ?thesis
    apply(erule_tac exE, simp)
    done
qed

definition t_fourtimes_len :: nat
  where
    t_fourtimes_len = (length t_fourtimes div 2)

lemma primerec_rec_fourtimes_1[intro]: primerec rec_fourtimes (Suc 0)
  apply(auto simp: rec_fourtimes_def numeral_4_eq_4 constn.simps)
  by auto

lemma fourtimes_lemma: rec_exec rec_fourtimes [rs] = 4 ∗ rs
  by(simp add: rec_exec.simps rec_fourtimes_def)

lemma t_fourtimes_correct:
  ∃ stp ln rn. steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n))
    (tm_of abc_fourtimes @ shift (mopup 1) (length (tm_of abc_fourtimes) div 2)) stp =
      (0, Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (4 ∗ rs)) @ Bk↑(rn))
proof(case_tac rec_ci rec_fourtimes)
  fix a b c
  assume h: rec_ci rec_fourtimes = (a, b, c)
  have ∃ stp m l. steps0 (Suc 0, Bk # Bk # ires, <[rs]> @ Bk↑(n)) (tm_of abc_fourtimes @ shift
```

```
(mopup (length [rs]))
  (length (tm_of abc_fourtimes) div 2)) stp = (0, Bk↑(m) @ Bk # Bk # ires, Oc↑(Suc (rec_exec
rec_fourtimes [rs])) @ Bk↑(l))
  thm recursive_compile_to_tm_correct1
 proof(rule_tac recursive_compile_to_tm_correct1)
  show rec_ci rec_fourtimes = (a, b, c) by (simp add: h)
 next
  show terminate rec_fourtimes [rs]
   apply(rule_tac primerec_terminate)
   by auto
 next
  show tm_of abc_fourtimes = tm_of (a [+] dummy_abc (length [rs]))
   using h
   by(simp add: abc_fourtimes_def)
 qed
 thus ?thesis
  apply(simp add: tape_of_list_def tape_of_nat_def fourtimes_lemma)
  done
qed

lemma wf_fourtimes[intro]: tm_wf (t_fourtimes_compile, 0)
 apply(simp only: t_fourtimes_compile_def)
 apply(rule_tac wf_tm_from_abacus, simp)
 done

lemma t_fourtimes_change_term_state:
 ∃ stp ln rn. steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n)) t_fourtimes stp
   = (Suc t_fourtimes_len, Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (4 * rs)) @ Bk↑(rn))
proof −
 have ∃ stp ln rn. steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n))
  (tm_of abc_fourtimes @ shift (mopup 1) ((length (tm_of abc_fourtimes) div 2))) stp =
  (0, Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (4 * rs)) @ Bk↑(rn))
  by(rule_tac t_fourtimes_correct)
 then obtain stp ln rn where
  steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n))
  (tm_of abc_fourtimes @ shift (mopup 1) ((length (tm_of abc_fourtimes) div 2))) stp =
  (0, Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (4 * rs)) @ Bk↑(rn)) by blast
 hence ∃ stp. steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n))
  (adjust0 t_fourtimes_compile) stp
   = (Suc (length t_fourtimes_compile div 2), Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (4 * rs)) @
Bk↑(rn))
  apply(rule_tac stp = stp in adjust_halt_eq)
   apply(simp add: t_fourtimes_compile_def, auto)
  done
 then obtain stpb where
  steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n))
  (adjust0 t_fourtimes_compile) stpb
   = (Suc (length t_fourtimes_compile div 2), Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (4 * rs)) @
Bk↑(rn)) ..
 thus ?thesis
```

```
  apply(simp add: t_fourtimes_def t_fourtimes_len_def)
  by metis
qed

lemma length_t_twice_even[intro]: is_even (length t_twice)
 by(auto simp: t_twice_def t_twice_compile_def intro!:mopup_mod2)

lemma t_fourtimes_append_pre:
 steps0 (Suc 0, Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n)) t_fourtimes stp
 = (Suc t_fourtimes_len, Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (4 ∗ rs)) @ Bk↑(rn))
  ⟹ steps0 (Suc 0 + length (t_wcode_main_first_part @
        shift t_twice (length t_wcode_main_first_part div 2) @ [(L, 1), (L, 1)]) div 2,
    Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n))
   ((t_wcode_main_first_part @
 shift t_twice (length t_wcode_main_first_part div 2) @ [(L, 1), (L, 1)]) @
 shift t_fourtimes (length (t_wcode_main_first_part @
 shift t_twice (length t_wcode_main_first_part div 2) @ [(L, 1), (L, 1)]) div 2) @ ([(L, 1), (L,
1)])) stp
 = ((Suc t_fourtimes_len) + length (t_wcode_main_first_part @
 shift t_twice (length t_wcode_main_first_part div 2) @ [(L, 1), (L, 1)]) div 2,
 Bk↑(ln) @ Bk # Bk # ires, Oc↑(Suc (4 ∗ rs)) @ Bk↑(rn))
 using length_t_twice_even
 by(intro tm_append_shift_append_steps, auto)

lemma split_26_even[simp]: (26 + l::nat) div 2 = l div 2 + 13 by(simp)

lemma t_twice_len_plust_14[simp]: t_twice_len + 14 =  14 + length (shift t_twice 12) div 2
 apply(simp add: t_twice_def t_twice_len_def)
 done

lemma t_fourtimes_append:
 ∃ stp ln rn.
 steps0 (Suc 0 + length (t_wcode_main_first_part @ shift t_twice
 (length t_wcode_main_first_part div 2) @ [(L, 1), (L, 1)]) div 2,
 Bk # Bk # ires, Oc↑(Suc rs) @ Bk↑(n))
 ((t_wcode_main_first_part @ shift t_twice (length t_wcode_main_first_part div 2) @
 [(L, 1), (L, 1)]) @ shift t_fourtimes (t_twice_len + 13) @ [(L, 1), (L, 1)]) stp
 = (Suc t_fourtimes_len + length (t_wcode_main_first_part @ shift t_twice
 (length t_wcode_main_first_part div 2) @ [(L, 1), (L, 1)]) div 2, Bk↑(ln) @ Bk # Bk # ires,
                                        Oc↑(Suc (4 ∗ rs)) @ Bk↑(rn))
 using t_fourtimes_change_term_state[of ires rs n]
 apply(erule_tac exE)
 apply(erule_tac exE)
 apply(erule_tac exE)
 apply(drule_tac t_fourtimes_append_pre)
 apply(rule_tac x = stp in exI, rule_tac x = ln in exI, rule_tac x = rn in exI)
 apply(simp add: t_twice_len_def)
 done

lemma even_fourtimes_len: length t_fourtimes mod 2 = 0
```

apply(*auto simp*: *t_fourtimes_def t_fourtimes_compile_def*)
by (*metis mopup_mod2*)

lemma *t_twice_even*[*simp*]: $2 * (length\ t\_twice\ div\ 2) = length\ t\_twice$
using *length_t_twice_even* **by** *arith*

lemma *t_fourtimes_even*[*simp*]: $2 * (length\ t\_fourtimes\ div\ 2) = length\ t\_fourtimes$
using *even_fourtimes_len*
by *arith*

lemma *fetch_t_wcode_14_Oc*: *fetch t_wcode_main* $(14 + length\ t\_twice\ div\ 2 + t\_fourtimes\_len)$ *Oc*
$= (L, Suc\ 0)$
apply(*subgoal_tac* $14 = Suc\ 13$)
apply(*simp only*: *fetch.simps add_Suc nth_of .simps t_wcode_main_def*)
apply(*simp add*:*length_t_twice_even t_fourtimes_len_def nth_append*)
by *arith*

lemma *fetch_t_wcode_14_Bk*: *fetch t_wcode_main* $(14 + length\ t\_twice\ div\ 2 + t\_fourtimes\_len)$ *Bk*
$= (L, Suc\ 0)$
apply(*subgoal_tac* $14 = Suc\ 13$)
apply(*simp only*: *fetch.simps add_Suc nth_of .simps t_wcode_main_def*)
apply(*simp add*:*length_t_twice_even t_fourtimes_len_def nth_append*)
by *arith*

lemma *fetch_t_wcode_14* [*simp*]: *fetch t_wcode_main* $(14 + length\ t\_twice\ div\ 2 + t\_fourtimes\_len)$ *b*
$= (L, Suc\ 0)$
apply(*case_tac b*, *simp_all add*:*fetch_t_wcode_14_Bk fetch_t_wcode_14_Oc*)
done

lemma *wcode_jump2*:
$\exists\ stp\ ln\ rn.\ steps0\ (t\_twice\_len + 14 + t\_fourtimes\_len$
$, Bk\ \#\ Bk\ \#\ Bk\uparrow(lnb)\ @\ Oc\ \#\ ires, Oc\uparrow(Suc\ (4 * rs + 4))\ @\ Bk\uparrow(rnb))\ t\_wcode\_main\ stp =$
$(Suc\ 0, Bk\ \#\ Bk\uparrow(ln)\ @\ Oc\ \#\ ires, Bk\ \#\ Oc\uparrow(Suc\ (4 * rs + 4))\ @\ Bk\uparrow(rn))$
apply(*rule_tac* $x = Suc\ 0$ **in** *exI*)
apply(*simp add*: *steps.simps*)
apply(*rule_tac* $x = lnb$ **in** *exI*, *rule_tac* $x = rnb$ **in** *exI*)
apply(*simp add*: *step.simps*)
done

lemma *wcode_fourtimes_case*:
shows $\exists\ stp\ ln\ rn.$
$steps0\ (Suc\ 0, Bk\ \#\ Bk\uparrow(m)\ @\ Oc\ \#\ Bk\ \#\ Oc\ \#\ ires, Bk\ \#\ Oc\uparrow(Suc\ rs)\ @\ Bk\uparrow(n))$ $t\_wcode\_main\ stp =$
$(Suc\ 0, Bk\ \#\ Bk\uparrow(ln)\ @\ Oc\ \#\ ires, Bk\ \#\ Oc\uparrow(Suc\ (4*rs + 4))\ @\ Bk\uparrow(rn))$
proof −
have $\exists\ stp\ ln\ rn.$
$steps0\ (Suc\ 0, Bk\ \#\ Bk\uparrow(m)\ @\ Oc\ \#\ Bk\ \#\ Oc\ \#\ ires, Bk\ \#\ Oc\uparrow(Suc\ rs)\ @\ Bk\uparrow(n))$

t_wcode_main stp =
(*t_twice_len* + *14*, *Bk* # *Bk* # *Bk*↑(*ln*) @ *Oc* # *ires*, *Oc*↑(*Suc* (*rs* + *1*)) @ *Bk*↑(*rn*))
using *wcode_fourtimes_case_first_correctness*[*of ires rs m n*]
by (*auto simp add*: *wcode_fourtimes_case_inv.simps*) *auto*
from *this* **obtain** *stpa lna rna* **where** *stp1*:
steps0 (*Suc 0*, *Bk* # *Bk*↑(*m*) @ *Oc* # *Bk* # *Oc* # *ires*, *Bk* # *Oc*↑(*Suc rs*) @ *Bk*↑(*n*)) *t_wcode_main stpa* =
(*t_twice_len* + *14*, *Bk* # *Bk* # *Bk*↑(*lna*) @ *Oc* # *ires*, *Oc*↑(*Suc* (*rs* + *1*)) @ *Bk*↑(*rna*)) **by** *blast*
have ∃ *stp ln rn*. *steps0* (*t_twice_len* + *14*, *Bk* # *Bk* # *Bk*↑(*lna*) @ *Oc* # *ires*, *Oc*↑(*Suc* (*rs* + *1*)) @ *Bk*↑(*rna*))
t_wcode_main stp =
(*t_twice_len* + *14* + *t_fourtimes_len*, *Bk* # *Bk* # *Bk*↑(*ln*) @ *Oc* # *ires*, *Oc*↑(*Suc* (*4*∗*rs* + *4*)) @ *Bk*↑(*rn*))
using *t_fourtimes_append*[*of Bk*↑(*lna*) @ *Oc* # *ires rs* + *1 rna*]
apply(*erule_tac exE*)
apply(*erule_tac exE*)
apply(*erule_tac exE*)
apply(*simp add*: *t_wcode_main_def*) **apply**(*rename_tac stp ln rn*)
apply(*rule_tac x* = *stp* **in** *exI*,
rule_tac x = *ln* + *lna* **in** *exI*,
rule_tac x = *rn* **in** *exI*, *simp*)
apply(*simp add*: *replicate_Suc*[*THEN sym*] *replicate_add*[*THEN sym*] *del*: *replicate_Suc*)
done
from *this* **obtain** *stpb lnb rnb* **where** *stp2*:
steps0 (*t_twice_len* + *14*, *Bk* # *Bk* # *Bk*↑(*lna*) @ *Oc* # *ires*, *Oc*↑(*Suc* (*rs* + *1*)) @ *Bk*↑(*rna*))
t_wcode_main stpb =
(*t_twice_len* + *14* + *t_fourtimes_len*, *Bk* # *Bk* # *Bk*↑(*lnb*) @ *Oc* # *ires*, *Oc*↑(*Suc* (*4*∗*rs* + *4*)) @ *Bk*↑(*rnb*))
by *blast*
have ∃ *stp ln rn*. *steps0* (*t_twice_len* + *14* + *t_fourtimes_len*,
Bk # *Bk* # *Bk*↑(*lnb*) @ *Oc* # *ires*, *Oc*↑(*Suc* (*4*∗*rs* + *4*)) @ *Bk*↑(*rnb*))
t_wcode_main stp =
(*Suc 0*, *Bk* # *Bk*↑(*ln*) @ *Oc* # *ires*, *Bk* # *Oc*↑(*Suc* (*4*∗*rs* + *4*)) @ *Bk*↑(*rn*))
apply(*rule wcode_jump2*)
done
from *this* **obtain** *stpc lnc rnc* **where** *stp3*:
steps0 (*t_twice_len* + *14* + *t_fourtimes_len*,
Bk # *Bk* # *Bk*↑(*lnb*) @ *Oc* # *ires*, *Oc*↑(*Suc* (*4*∗*rs* + *4*)) @ *Bk*↑(*rnb*))
t_wcode_main stpc =
(*Suc 0*, *Bk* # *Bk*↑(*lnc*) @ *Oc* # *ires*, *Bk* # *Oc*↑(*Suc* (*4*∗*rs* + *4*)) @ *Bk*↑(*rnc*))
by *blast*
from *stp1 stp2 stp3* **show** *?thesis*
apply(*rule_tac x* = *stpa* + *stpb* + *stpc* **in** *exI*,
rule_tac x = *lnc* **in** *exI*, *rule_tac x* = *rnc* **in** *exI*)
apply(*simp*)
done
qed

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 ≝ (inv_image lex_pair wcode_halt_case_measure)

lemma wf_wcode_halt_case_le[intro]: wf wcode_halt_case_le
  by(auto simp: wcode_halt_case_le_def)

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)
  apply(simp only: wcode_halt_invs)
  apply(erule_tac disjE)
   apply(erule_tac exE)+ apply(rename_tac ml mr rn)
   apply(case_tac ml, simp)
    apply(rule_tac x = mr − 2 in exI, rule_tac x = rn in exI)
    apply(case_tac mr, force, simp add: exp_ind del: replicate_Suc)
    apply(case_tac mr − 1, force, simp add: exp_ind del: replicate_Suc)
   apply force
  apply force
  done

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))
  apply(auto simp: wcode_halt_invs)
```

```
  done

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)
  by(simp add:wcode_halt_invs)

lemma wcode_3_nonempty[simp]:
  wcode_on_left_moving_3 ires rs (b, []) = False
  wcode_on_checking_3 ires rs (b, []) = False
  wcode_goon_checking_3 ires rs (b, []) = False
  wcode_on_left_moving_3 ires rs (b, Oc # list) ⟹ b ≠ []
  wcode_on_checking_3 ires rs (b, Oc # list) = False
  wcode_on_left_moving_3 ires rs (b, Bk # list) ⟹ b ≠ []
  wcode_on_checking_3 ires rs (b, Bk # list) ⟹ b ≠ []
  wcode_goon_checking_3 ires rs (b, Oc # list) = False
  by(auto simp: wcode_halt_invs)

lemma wcode_goon_checking_3_mv_Bk[simp]: wcode_on_checking_3 ires rs (b, Bk # list) ⟹
  wcode_goon_checking_3 ires rs (tl b, hd b # Bk # list)
  apply(auto simp: wcode_halt_invs)
  done

lemma t_halt_case_correctness:
  shows let P = (λ (st, l, r). st = 0) in
     let Q = (λ (st, l, r). wcode_halt_case_inv st ires rs (l, r)) in
     let f = (λ stp. steps0 (Suc 0, Bk # Bk↑(m) @ Oc # Bk # Bk # ires, Bk # Oc↑(Suc rs) @
Bk↑(n)) t_wcode_main stp) in
     ∃ n .P (f n) ∧ Q (f (n::nat))
proof −
  let ?P = (λ (st, l, r). st = 0)
  let ?Q = (λ (st, l, r). wcode_halt_case_inv st ires rs (l, r))
  let ?f = (λ stp. steps0 (Suc 0, Bk # Bk↑(m) @ Oc # Bk # Bk # ires, Bk # Oc↑(Suc rs) @
Bk↑(n)) t_wcode_main stp)
  have ∃ n. ?P (?f n) ∧ ?Q (?f (n::nat))
  proof(rule_tac halt_lemma2)
   show wf wcode_halt_case_le by auto
  next
   { fix na
    obtain a b c where abc:?f na = (a,b,c) by(cases ?f na,auto)
    hence ¬ ?P (?f na) ∧ ?Q (?f na) ⟹
           ?Q (?f (Suc na)) ∧ (?f (Suc na), ?f na) ∈ wcode_halt_case_le
     apply(simp add: step.simps)
     apply(cases c;cases hd c)
      apply(auto simp: wcode_halt_case_le_def lex_pair_def split: if_splits)
     done
   }
   thus ∀ na. ¬ ?P (?f na) ∧ ?Q (?f na) ⟶
           ?Q (?f (Suc na)) ∧ (?f (Suc na), ?f na) ∈ wcode_halt_case_le by blast
  next
   show ?Q (?f 0)
```

```
    apply(simp add: steps.simps wcode_halt_invs)
    apply(rule_tac x = Suc m in exI, simp)
    apply(rule_tac x = Suc 0 in exI, auto)
    done
  next
   show ¬ ?P (?f 0)
    apply(simp add: steps.simps)
    done
  qed
  thus ?thesis
   apply(auto)
   done
qed

declare wcode_halt_case_inv.simps[simp del]
lemma leading_Oc[intro]: ∃ xs. (<rev list @ [aa::nat]> :: cell list) = Oc # xs
 apply(case_tac rev list, simp)
 apply(simp add: tape_of_nl_cons)
 done

lemma wcode_halt_case:
 ∃stp ln rn. steps0 (Suc 0, Bk # Bk↑(m) @ Oc # Bk # Bk # ires, Bk # Oc↑(Suc rs) @ Bk↑(n))
 t_wcode_main stp = (0, Bk # ires, Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(Suc rs) @ Bk↑(rn))
proof −
 let ?P = λ(st, l, r). st = 0
 let ?Q = λ(st, l, r). wcode_halt_case_inv st ires rs (l, r)
 let ?f = steps0 (Suc 0, Bk # Bk ↑ m @ Oc # Bk # Bk # ires, Bk # Oc ↑ Suc rs @ Bk ↑ n)
t_wcode_main
 from t_halt_case_correctness[of ires rs m n] obtain n where ?P (?f n) ∧ ?Q (?f n) by metis
 thus ?thesis
  apply(simp add: wcode_halt_case_inv.simps wcode_stop.simps)
  apply(case_tac steps0 (Suc 0, Bk # Bk↑(m) @ Oc # Bk # Bk # ires,
           Bk # Oc↑(Suc rs) @ Bk↑(n)) t_wcode_main n)
  apply(auto simp: wcode_halt_case_inv.simps wcode_stop.simps)
  by auto
qed

lemma bl_bin_one[simp]: bl_bin [Oc] = 1
 apply(simp add: bl_bin.simps)
 done

lemma twice_power[intro]: 2 ∗ 2 ˆ a = Suc (Suc (2 ∗ bl_bin (Oc ↑ a)))
 apply(induct a, auto simp: bl_bin.simps)
 done
declare replicate_Suc[simp del]

lemma t_wcode_main_lemma_pre:
 ⟦args ≠ []; lm = <args::nat list>⟧ =⇒
    ∃ stp ln rn. steps0 (Suc 0, Bk # Bk↑(m) @ rev lm @ Bk # Bk # ires, Bk # Oc↑(Suc rs) @
Bk↑(n)) t_wcode_main
```

stp
= (*0*, *Bk* # *ires*, *Bk* # *Oc* # *Bk*↑(*ln*) @ *Bk* # *Bk* # *Oc*↑(*bl_bin lm* + *rs* ∗ 2ˆ(*length lm* − *1*)
) @ *Bk*↑(*rn*))
proof(*induct length args arbitrary*: *args lm rs m n*, *simp*)
fix *x args lm rs m n*
assume *ind*:
⋀*args lm rs m n*.
⟦*x* = *length args*; (*args*::*nat list*) ≠ []; *lm* = <*args*>⟧
⟹ ∃ *stp ln rn*.
steps0 (*Suc 0*, *Bk* # *Bk*↑(*m*) @ *rev lm* @ *Bk* # *Bk* # *ires*, *Bk* # *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))
t_wcode_main stp =
(*0*, *Bk* # *ires*, *Bk* # *Oc* # *Bk*↑(*ln*) @ *Bk* # *Bk* # *Oc*↑(*bl_bin lm* + *rs* ∗ *2* ˆ (*length lm* − *1*))
@ *Bk*↑(*rn*))
and *h*: *Suc x* = *length args* (*args*::*nat list*) ≠ [] *lm* = <*args*>
from *h* **have** ∃ (*a*::*nat*) *xs*. *args* = *xs* @ [*a*]
apply(*rule_tac x* = *last args* **in** *exI*)
apply(*rule_tac x* = *butlast args* **in** *exI*, *auto*)
done
from *this* **obtain** *a xs* **where** *args* = *xs* @ [*a*] **by** *blast*
from *h* **and** *this* **show**
∃ *stp ln rn*.
steps0 (*Suc 0*, *Bk* # *Bk*↑(*m*) @ *rev lm* @ *Bk* # *Bk* # *ires*, *Bk* # *Oc*↑(*Suc rs*) @ *Bk*↑(*n*))
t_wcode_main stp =
(*0*, *Bk* # *ires*, *Bk* # *Oc* # *Bk*↑(*ln*) @ *Bk* # *Bk* # *Oc*↑(*bl_bin lm* + *rs* ∗ *2* ˆ (*length lm* − *1*))
@ *Bk*↑(*rn*))
proof(*case_tac xs*::*nat list*, *simp*)
show ∃ *stp ln rn*.
steps0 (*Suc 0*, *Bk* # *Bk* ↑ *m* @ *Oc* ↑ *Suc a* @ *Bk* # *Bk* # *ires*, *Bk* # *Oc* ↑ *Suc rs* @ *Bk* ↑
n) *t_wcode_main stp* =
(*0*, *Bk* # *ires*, *Bk* # *Oc* # *Bk* ↑ *ln* @ *Bk* # *Bk* # *Oc* ↑ (*bl_bin* (*Oc* ↑ *Suc a*) + *rs* ∗ *2* ˆ *a*)
@ *Bk* ↑ *rn*)
proof(*induct a arbitrary*: *m n rs ires*, *simp*)
fix *m n rs ires*
show ∃ *stp ln rn*.
steps0 (*Suc 0*, *Bk* # *Bk* ↑ *m* @ *Oc* # *Bk* # *Bk* # *ires*, *Bk* # *Oc* ↑ *Suc rs* @ *Bk* ↑ *n*)
t_wcode_main stp =
(*0*, *Bk* # *ires*, *Bk* # *Oc* # *Bk* ↑ *ln* @ *Bk* # *Bk* # *Oc* ↑ *Suc rs* @ *Bk* ↑ *rn*)
apply(*rule_tac wcode_halt_case*)
done
next
fix *a m n rs ires*
assume *ind2*:
⋀*m n rs ires*.
∃ *stp ln rn*.
steps0 (*Suc 0*, *Bk* # *Bk* ↑ *m* @ *Oc* ↑ *Suc a* @ *Bk* # *Bk* # *ires*, *Bk* # *Oc* ↑ *Suc rs* @ *Bk* ↑
n) *t_wcode_main stp* =
(*0*, *Bk* # *ires*, *Bk* # *Oc* # *Bk* ↑ *ln* @ *Bk* # *Bk* # *Oc* ↑ (*bl_bin* (*Oc* ↑ *Suc a*) + *rs* ∗ *2* ˆ
a) @ *Bk* ↑ *rn*)
show ∃ *stp ln rn*.
steps0 (*Suc 0*, *Bk* # *Bk* ↑ *m* @ *Oc* ↑ *Suc* (*Suc a*) @ *Bk* # *Bk* # *ires*, *Bk* # *Oc* ↑ *Suc rs* @

Bk ↑ n) t_wcode_main stp =
(0, Bk # ires, Bk # Oc # Bk ↑ ln @ Bk # Bk # Oc ↑ (bl_bin (Oc ↑ Suc (Suc a)) + rs ∗ 2 ^ Suc a) @ Bk ↑ rn)
proof −
have ∃ stp ln rn.
steps0 (Suc 0, Bk # Bk↑(m) @ rev (<Suc a>) @ Bk # Bk # ires, Bk # Oc↑(Suc rs) @ Bk↑(n)) t_wcode_main stp =
(Suc 0, Bk # Bk↑(ln) @ rev (<a>) @ Bk # Bk # ires, Bk # Oc↑(Suc (2 ∗ rs + 2)) @ Bk↑(rn))
apply(simp add: tape_of_nat)
using wcode_double_case[of m Oc↑(a) @ Bk # Bk # ires rs n]
apply(simp add: replicate_Suc)
done
from this **obtain** stpa lna rna **where** stp1:
steps0 (Suc 0, Bk # Bk↑(m) @ rev (<Suc a>) @ Bk # Bk # ires, Bk # Oc↑(Suc rs) @ Bk↑(n)) t_wcode_main stpa =
(Suc 0, Bk # Bk↑(lna) @ rev (<a>) @ Bk # Bk # ires, Bk # Oc↑(Suc (2 ∗ rs + 2)) @ Bk↑(rna)) **by** blast
moreover have
∃ stp ln rn.
steps0 (Suc 0, Bk # Bk↑(lna) @ rev (<a::nat>) @ Bk # Bk # ires, Bk # Oc↑(Suc (2 ∗ rs + 2)) @ Bk↑(rna)) t_wcode_main stp =
(0, Bk # ires, Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(bl_bin (<a>) + (2∗rs + 2) ∗ 2 ^ a) @ Bk↑(rn))
using ind2[of lna ires 2∗rs + 2 rna] **by**(simp add: tape_of_list_def tape_of_nat_def)
from this **obtain** stpb lnb rnb **where** stp2:
steps0 (Suc 0, Bk # Bk↑(lna) @ rev (<a>) @ Bk # Bk # ires, Bk # Oc↑(Suc (2 ∗ rs + 2)) @ Bk↑(rna)) t_wcode_main stpb =
(0, Bk # ires, Bk # Oc # Bk↑(lnb) @ Bk # Bk # Oc↑(bl_bin (<a>) + (2∗rs + 2) ∗ 2 ^ a) @ Bk↑(rnb))
by blast
from stp1 **and** stp2 **show** ?thesis
apply(rule_tac x = stpa + stpb **in** exI,
rule_tac x = lnb **in** exI, rule_tac x = rnb **in** exI, simp add: tape_of_nat_def)
apply(simp add: bl_bin.simps replicate_Suc)
apply(auto)
done
qed
qed
next
fix aa list
assume g: Suc x = length args args ≠ [] lm = <args> args = xs @ [a::nat] xs = (aa::nat) # list
thus ∃ stp ln rn. steps0 (Suc 0, Bk # Bk↑(m) @ rev lm @ Bk # Bk # ires, Bk # Oc↑(Suc rs) @ Bk↑(n)) t_wcode_main stp =
(0, Bk # ires, Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(bl_bin lm + rs ∗ 2 ^ (length lm − 1)) @ Bk↑(rn))
proof(induct a arbitrary: m n rs args lm, simp_all add: tape_of_nl_rev,
simp only: tape_of_nl_cons_app1, simp)
fix m n rs args lm

```
    have ∃ stp ln rn.
      steps0 (Suc 0, Bk # Bk↑(m) @ Oc # Bk # rev (<(aa::nat) # list>) @ Bk # Bk # ires,
      Bk # Oc↑(Suc rs) @ Bk↑(n)) t_wcode_main stp =
      (Suc 0, Bk # Bk↑(ln) @ rev (<aa # list>) @ Bk # Bk # ires,
      Bk # Oc↑(Suc (4*rs + 4)) @ Bk↑(rn))
    proof(simp add: tape_of_nl_rev)
      have ∃ xs. (<rev list @ [aa]>) = Oc # xs by auto
      from this obtain xs where (<rev list @ [aa]>) = Oc # xs ..
      thus ∃ stp ln rn.
          steps0 (Suc 0, Bk # Bk↑(m) @ Oc # Bk # <rev list @ [aa]> @ Bk # Bk # ires,
          Bk # Oc↑(Suc rs) @ Bk↑(n)) t_wcode_main stp =
          (Suc 0, Bk # Bk↑(ln) @ <rev list @ [aa]> @ Bk # Bk # ires, Bk # Oc↑(5 + 4 * rs) @
Bk↑(rn))
        apply(simp)
        using wcode_fourtimes_case[of m xs @ Bk # Bk # ires rs n]
        apply(simp)
        done
    qed
    from this obtain stpa lna rna where stp1:
      steps0 (Suc 0, Bk # Bk↑(m) @ Oc # Bk # rev (<aa # list>) @ Bk # Bk # ires,
      Bk # Oc↑(Suc rs) @ Bk↑(n)) t_wcode_main stpa =
      (Suc 0, Bk # Bk↑(lna) @ rev (<aa # list>) @ Bk # Bk # ires,
      Bk # Oc↑(Suc (4*rs + 4)) @ Bk↑(rna)) by blast
    from g have
      ∃ stp ln rn. steps0 (Suc 0, Bk # Bk↑(lna) @ rev (<(aa::nat) # list>) @ Bk # Bk # ires,
      Bk # Oc↑(Suc (4*rs + 4)) @ Bk↑(rna)) t_wcode_main stp = (0, Bk # ires,
       Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(bl_bin (<aa#list>)+ (4*rs + 4) * 2^(length
(<aa#list>) − 1) ) @ Bk↑(rn))
      apply(rule_tac args = (aa::nat)#list in ind, simp_all)
      done
    from this obtain stpb lnb rnb where stp2:
      steps0 (Suc 0, Bk # Bk↑(lna) @ rev (<(aa::nat) # list>) @ Bk # Bk # ires,
       Bk # Oc↑(Suc (4*rs + 4)) @ Bk↑(rna)) t_wcode_main stpb = (0, Bk # ires,
       Bk # Oc # Bk↑(lnb) @ Bk # Bk # Oc↑(bl_bin (<aa#list>)+ (4*rs + 4) * 2^(length
(<aa#list>) − 1) ) @ Bk↑(rnb))
      by blast
    from stp1 and stp2 and h
    show ∃ stp ln rn.
       steps0 (Suc 0, Bk # Bk↑(m) @ Oc # Bk # <rev list @ [aa]> @ Bk # Bk # ires,
       Bk # Oc↑(Suc rs) @ Bk↑(n)) t_wcode_main stp =
       (0, Bk # ires, Bk # Oc # Bk↑(ln) @ Bk #
       Bk # Oc↑(bl_bin (Oc↑(Suc aa) @ Bk # <list @ [0]>) + rs * (2 * 2 ^ (aa + length (<list
@ [0]>)))) @ Bk↑(rn))
      apply(rule_tac x = stpa + stpb in exI, rule_tac x = lnb in exI,
        rule_tac x = rnb in exI, simp add: steps_add tape_of_nl_rev)
      done
  next
    fix ab m n rs args lm
    assume ind2:
      ⋀ m n rs args lm.
```

```
⟦lm = <aa # list @ [ab]>; args = aa # list @ [ab]⟧
⟹ ∃ stp ln rn.
steps0 (Suc 0, Bk # Bk↑(m) @ <ab # rev list @ [aa]> @ Bk # Bk # ires,
Bk # Oc↑(Suc rs) @ Bk↑(n)) t_wcode_main stp =
(0, Bk # ires, Bk # Oc # Bk↑(ln) @ Bk #
 Bk # Oc↑(bl_bin (<aa # list @ [ab]>) + rs * 2 ^ (length (<aa # list @ [ab]>) − Suc
0)) @ Bk↑(rn))
  and k: args = aa # list @ [Suc ab] lm = <aa # list @ [Suc ab]>
 show ∃ stp ln rn.
  steps0 (Suc 0, Bk # Bk↑(m) @ <Suc ab # rev list @ [aa]> @ Bk # Bk # ires,
  Bk # Oc↑(Suc rs) @ Bk↑(n)) t_wcode_main stp =
  (0, Bk # ires,Bk # Oc # Bk↑(ln) @ Bk #
  Bk # Oc↑(bl_bin (<aa # list @ [Suc ab]>) + rs * 2 ^ (length (<aa # list @ [Suc ab]>)
− Suc 0)) @ Bk↑(rn))
 proof(simp add: tape_of_nl_cons_app1)
  have ∃ stp ln rn.
    steps0 (Suc 0, Bk # Bk↑(m) @ Oc↑(Suc (Suc ab)) @ Bk # <rev list @ [aa]> @ Bk #
Bk # ires,
   Bk # Oc # Oc↑(rs) @ Bk↑(n)) t_wcode_main stp
   = (Suc 0, Bk # Bk↑(ln) @ Oc↑(Suc ab) @ Bk # <rev list @ [aa]> @ Bk # Bk # ires,
   Bk # Oc↑(Suc (2*rs + 2)) @ Bk↑(rn))
   using wcode_double_case[of m Oc↑(ab) @ Bk # <rev list @ [aa]> @ Bk # Bk # ires
     rs n]
   apply(simp add: replicate_Suc)
   done
  from this obtain stpa lna rna where stp1:
   steps0 (Suc 0, Bk # Bk↑(m) @ Oc↑(Suc (Suc ab)) @ Bk # <rev list @ [aa]> @ Bk # Bk
# ires,
   Bk # Oc # Oc↑(rs) @ Bk↑(n)) t_wcode_main stpa
   = (Suc 0, Bk # Bk↑(lna) @ Oc↑(Suc ab) @ Bk # <rev list @ [aa]> @ Bk # Bk # ires,
   Bk # Oc↑(Suc (2*rs + 2)) @ Bk↑(rna)) by blast
  from k have
   ∃ stp ln rn. steps0 (Suc 0, Bk # Bk↑(lna) @ <ab # rev list @ [aa]> @ Bk # Bk # ires,
   Bk # Oc↑(Suc (2*rs + 2)) @ Bk↑(rna)) t_wcode_main stp
   = (0, Bk # ires, Bk # Oc # Bk↑(ln) @ Bk #
   Bk # Oc↑(bl_bin (<aa # list @ [ab]> ) +  (2*rs + 2)* 2^(length (<aa # list @ [ab]>)
− Suc 0)) @ Bk↑(rn))
   apply(rule_tac ind2, simp_all)
   done
  from this obtain stpb lnb rnb where stp2:
   steps0 (Suc 0, Bk # Bk↑(lna) @  <ab # rev list @ [aa]> @ Bk # Bk # ires,
   Bk # Oc↑(Suc (2*rs + 2)) @ Bk↑(rna)) t_wcode_main stpb
   = (0, Bk # ires, Bk # Oc # Bk↑(lnb) @ Bk #
   Bk # Oc↑(bl_bin (<aa # list @ [ab]> ) +  (2*rs + 2)* 2^(length (<aa # list @ [ab]>)
− Suc 0)) @ Bk↑(rnb))
   by blast
  from stp1 and stp2 show
   ∃ stp ln rn.
    steps0 (Suc 0, Bk # Bk↑(m) @ Oc↑(Suc (Suc ab)) @ Bk # <rev list @ [aa]> @ Bk #
Bk # ires,
```

```
      Bk # Oc↑(Suc rs) @ Bk↑(n)) t_wcode_main stp =
      (0, Bk # ires, Bk # Oc # Bk↑(ln) @ Bk # Bk #
      Oc↑(bl_bin (Oc↑(Suc aa) @ Bk # <list @ [Suc ab]>) + rs * (2 * 2 ^ (aa + length (<list
@ [Suc ab]>))))
      @ Bk↑(rn))
      apply(rule_tac x = stpa + stpb in exI, rule_tac x = lnb in exI,
        rule_tac x = rnb in exI, simp add: steps_add tape_of_nl_cons_app1 replicate_Suc)
      done
    qed
  qed
 qed
qed


definition t_wcode_prepare :: instr list
 where
  t_wcode_prepare ≝
     [(W1, 2), (L, 1), (L, 3), (R, 2), (R, 4), (W0, 3),
      (R, 4), (R, 5), (R, 6), (R, 5), (R, 7), (R, 5),
      (W1, 7), (L, 0)]

fun wprepare_add_one :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_add_one m lm (l, r) =
   (∃ rn. l = [] ∧
        (r = <m # lm> @ Bk↑(rn) ∨
         r = Bk # <m # lm> @ Bk↑(rn)))

fun wprepare_goto_first_end :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_goto_first_end m lm (l, r) =
   (∃ ml mr rn. l = Oc↑(ml) ∧
            r = Oc↑(mr) @ Bk # <lm> @ Bk↑(rn) ∧
            ml + mr = Suc (Suc m))

fun wprepare_erase :: nat ⇒ nat list ⇒ tape ⇒  bool
 where
  wprepare_erase m lm (l, r) =
   (∃ rn. l = Oc↑(Suc m) ∧
        tl r = Bk # <lm> @ Bk↑(rn))

fun wprepare_goto_start_pos_B :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_goto_start_pos_B m lm (l, r) =
   (∃ rn. l = Bk # Oc↑(Suc m) ∧
        r = Bk # <lm> @ Bk↑(rn))

fun wprepare_goto_start_pos_O :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
```

```
  wprepare_goto_start_pos_O m lm (l, r) =
   (∃ rn. l = Bk # Bk # Oc↑(Suc m) ∧
         r = <lm> @ Bk↑(rn))

fun wprepare_goto_start_pos :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_goto_start_pos m lm (l, r) =
     (wprepare_goto_start_pos_B m lm (l, r) ∨
      wprepare_goto_start_pos_O m lm (l, r))

fun wprepare_loop_start_on_rightmost :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_loop_start_on_rightmost m lm (l, r) =
   (∃ rn mr. rev l @ r = Oc↑(Suc m) @ Bk # Bk # <lm> @ Bk↑(rn) ∧ l ≠ [] ∧
               r = Oc↑(mr) @ Bk↑(rn))

fun wprepare_loop_start_in_middle :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_loop_start_in_middle m lm (l, r) =
   (∃ rn (mr:: nat) (lm1::nat list).
 rev l @ r = Oc↑(Suc m) @ Bk # Bk # <lm> @ Bk↑(rn) ∧ l ≠ [] ∧
 r = Oc↑(mr) @ Bk # <lm1> @ Bk↑(rn) ∧ lm1 ≠ [])

fun wprepare_loop_start :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_loop_start m lm (l, r) = (wprepare_loop_start_on_rightmost m lm (l, r) ∨
                          wprepare_loop_start_in_middle m lm (l, r))

fun wprepare_loop_goon_on_rightmost :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_loop_goon_on_rightmost m lm (l, r) =
   (∃ rn. l = Bk # <rev lm> @ Bk # Bk # Oc↑(Suc m) ∧
         r = Bk↑(rn))

fun wprepare_loop_goon_in_middle :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_loop_goon_in_middle m lm (l, r) =
   (∃ rn (mr:: nat) (lm1::nat list).
 rev l @ r = Oc↑(Suc m) @ Bk # Bk # <lm> @ Bk↑(rn) ∧ l ≠ [] ∧
               (if lm1 = [] then r = Oc↑(mr) @ Bk↑(rn)
               else r = Oc↑(mr) @ Bk # <lm1> @ Bk↑(rn)) ∧ mr > 0)

fun wprepare_loop_goon :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_loop_goon m lm (l, r) =
          (wprepare_loop_goon_in_middle m lm (l, r) ∨
           wprepare_loop_goon_on_rightmost m lm (l, r))

fun wprepare_add_one2 :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
```

```
  wprepare_add_one2 m lm (l, r) =
      (∃ rn. l = Bk # Bk # <rev lm> @ Bk # Bk # Oc↑(Suc m) ∧
          (r = [] ∨ tl r = Bk↑(rn)))

fun wprepare_stop :: nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_stop m lm (l, r) =
     (∃ rn. l = Bk # <rev lm> @ Bk # Bk # Oc↑(Suc m) ∧
         r = Bk # Oc # Bk↑(rn))

fun wprepare_inv :: nat ⇒ nat ⇒ nat list ⇒ tape ⇒ bool
 where
  wprepare_inv st m lm (l, r) =
     (if st = 0 then wprepare_stop m lm (l, r)
      else if st = Suc 0 then wprepare_add_one m lm (l, r)
      else if st = Suc (Suc 0) then wprepare_goto_first_end m lm (l, r)
      else if st = Suc (Suc (Suc 0)) then wprepare_erase m lm (l, r)
      else if st = 4 then wprepare_goto_start_pos m lm (l, r)
      else if st = 5 then wprepare_loop_start m lm (l, r)
      else if st = 6 then wprepare_loop_goon m lm (l, r)
      else if st = 7 then wprepare_add_one2 m lm (l, r)
      else False)

fun wprepare_stage :: config ⇒ nat
 where
  wprepare_stage (st, l, r) =
   (if st ≥ 1 ∧ st ≤ 4 then 3
    else if st = 5 ∨ st = 6 then 2
    else 1)

fun wprepare_state :: config ⇒ nat
 where
  wprepare_state (st, l, r) =
    (if st = 1 then 4
     else if st = Suc (Suc 0) then 3
     else if st = Suc (Suc (Suc 0)) then 2
     else if st = 4 then 1
     else if st = 7 then 2
     else 0)

fun wprepare_step :: config ⇒ nat
 where
  wprepare_step (st, l, r) =
   (if st = 1 then (if hd r = Oc then Suc (length l)
                    else 0)
    else if st = Suc (Suc 0) then length r
    else if st = Suc (Suc (Suc 0)) then (if hd r = Oc then 1
                         else 0)
    else if st = 4 then length r
    else if st = 5 then Suc (length r)
```

else if st = *6 then* (*if r* = [] *then 0 else Suc* (*length r*))
else if st = *7 then* (*if* (*r* ≠ [] ∧ *hd r* = *Oc*) *then 0*
else 1)
else 0)

fun *wcode_prepare_measure* :: *config* ⇒ *nat* × *nat* × *nat*
where
wcode_prepare_measure (*st*, *l*, *r*) =
(*wprepare_stage* (*st*, *l*, *r*),
wprepare_state (*st*, *l*, *r*),
wprepare_step (*st*, *l*, *r*))

definition *wcode_prepare_le* :: (*config* × *config*) *set*
where *wcode_prepare_le* $\stackrel{def}{=}$ (*inv_image lex_triple wcode_prepare_measure*)

lemma *wf_wcode_prepare_le*[*intro*]: *wf wcode_prepare_le*
by(*auto intro*:*wf_inv_image simp*: *wcode_prepare_le_def*
lex_triple_def)

declare *wprepare_add_one*.*simps*[*simp del*] *wprepare_goto_first_end*.*simps*[*simp del*]
wprepare_erase.*simps*[*simp del*] *wprepare_goto_start_pos*.*simps*[*simp del*]
wprepare_loop_start.*simps*[*simp del*] *wprepare_loop_goon*.*simps*[*simp del*]
wprepare_add_one2.*simps*[*simp del*]

lemmas *wprepare_invs* = *wprepare_add_one*.*simps wprepare_goto_first_end*.*simps*
wprepare_erase.*simps wprepare_goto_start_pos*.*simps*
wprepare_loop_start.*simps wprepare_loop_goon*.*simps*
wprepare_add_one2.*simps*

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

lemma *fetch_t_wcode_prepare*[*simp*]:
fetch t_wcode_prepare (*Suc 0*) *Bk* = (*W1*, *2*)
fetch t_wcode_prepare (*Suc 0*) *Oc* = (*L*, *1*)
fetch t_wcode_prepare (*Suc* (*Suc 0*)) *Bk* = (*L*, *3*)
fetch t_wcode_prepare (*Suc* (*Suc 0*)) *Oc* = (*R*, *2*)
fetch t_wcode_prepare (*Suc* (*Suc* (*Suc 0*))) *Bk* = (*R*, *4*)
fetch t_wcode_prepare (*Suc* (*Suc* (*Suc 0*))) *Oc* = (*W0*, *3*)
fetch t_wcode_prepare 4 Bk = (*R*, *4*)
fetch t_wcode_prepare 4 Oc = (*R*, *5*)
fetch t_wcode_prepare 5 Oc = (*R*, *5*)
fetch t_wcode_prepare 5 Bk = (*R*, *6*)
fetch t_wcode_prepare 6 Oc = (*R*, *5*)
fetch t_wcode_prepare 6 Bk = (*R*, *7*)
fetch t_wcode_prepare 7 Oc = (*L*, *0*)
fetch t_wcode_prepare 7 Bk = (*W1*, *7*)
unfolding *fetch*.*simps t_wcode_prepare_def nth_of* .*simps*
numeral **by** *auto*

lemma *wprepare_add_one_nonempty_snd*[*simp*]: *lm* ≠ [] ⟹ *wprepare_add_one m lm* (*b*, []) = *False*
apply(*simp add*: *wprepare_invs*)
done

lemma *wprepare_goto_first_end_nonempty_snd*[*simp*]: *lm* ≠ [] ⟹ *wprepare_goto_first_end m lm* (*b*, []) = *False*
apply(*simp add*: *wprepare_invs*)
done

lemma *wprepare_erase_nonempty_snd*[*simp*]: *lm* ≠ [] ⟹ *wprepare_erase m lm* (*b*, []) = *False*
apply(*simp add*: *wprepare_invs*)
done

lemma *wprepare_goto_start_pos_nonempty_snd*[*simp*]: *lm* ≠ [] ⟹ *wprepare_goto_start_pos m lm* (*b*, []) = *False*
apply(*simp add*: *wprepare_invs*)
done

lemma *wprepare_loop_start_empty_nonempty_fst*[*simp*]: ⟦*lm* ≠ []; *wprepare_loop_start m lm* (*b*, [])⟧ ⟹ *b* ≠ []
apply(*simp add*: *wprepare_invs*)
done

lemma *rev_eq*: *rev xs* = *rev ys* ⟹ *xs* = *ys* **by** *auto*

lemma *wprepare_loop_goon_Bk_empty_snd*[*simp*]: ⟦*lm* ≠ []; *wprepare_loop_start m lm* (*b*, [])⟧ ⟹
wprepare_loop_goon m lm (*Bk* # *b*, [])
apply(*simp only*: *wprepare_invs*)
apply(*erule_tac disjE*)
apply(*rule_tac disjI2*)
apply(*simp add*: *wprepare_loop_start_on_rightmost.simps*
wprepare_loop_goon_on_rightmost.simps, *auto*)
apply(*rule_tac rev_eq*, *simp add*: *tape_of_nl_rev*)
done

lemma *wprepare_loop_goon_nonempty_fst*[*simp*]: ⟦*lm* ≠ []; *wprepare_loop_goon m lm* (*b*, [])⟧ ⟹ *b* ≠ []
apply(*simp only*: *wprepare_invs*, *auto*)
done

lemma *wprepare_add_one2_Bk_empty*[*simp*]:⟦*lm* ≠ []; *wprepare_loop_goon m lm* (*b*, [])⟧ ⟹
wprepare_add_one2 m lm (*Bk* # *b*, [])
apply(*simp only*: *wprepare_invs*, *auto split*: *if_splits*)
done

lemma *wprepare_add_one2_nonempty_fst*[*simp*]: *wprepare_add_one2 m lm* (*b*, []) ⟹ *b* ≠ []
apply(*simp only*: *wprepare_invs*, *auto*)
done

lemma *wprepare_add_one2_Oc*[*simp*]: *wprepare_add_one2 m lm* (*b*, []) $\Longrightarrow$ *wprepare_add_one2 m lm* (*b*, [*Oc*])
 apply(*simp only*: *wprepare_invs*, *auto*)
 done

lemma *Bk_not_tape_start*[*simp*]: (*Bk* # *list* = <(*m*::*nat*) # *lm*> @ *ys*) = *False*
 apply(*case_tac lm*, *auto simp*: *tape_of_nl_cons replicate_Suc*)
 done

lemma *wprepare_goto_first_end_cases*[*simp*]:
 ⟦*lm* ≠ []; *wprepare_add_one m lm* (*b*, *Bk* # *list*)⟧
 $\Longrightarrow$ (*b* = [] ⟶ *wprepare_goto_first_end m lm* ([], *Oc* # *list*)) ∧
 (*b* ≠ [] ⟶ *wprepare_goto_first_end m lm* (*b*, *Oc* # *list*))
 apply(*simp only*: *wprepare_invs*)
 apply(*auto simp*: *tape_of_nl_cons split*: *if_splits*)
 apply(*cases lm*, *auto simp add*:*tape_of_list_def replicate_Suc*)
 done

lemma *wprepare_goto_first_end_Bk_nonempty_fst*[*simp*]:
 wprepare_goto_first_end m lm (*b*, *Bk* # *list*) $\Longrightarrow$ *b* ≠ []
 apply(*simp only*: *wprepare_invs* , *auto simp*: *replicate_Suc*)
 done

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*)
 by(*simp add*: *wprepare_invs*)

lemma *wprepare_erase_Bk_nonempty_fst*[*simp*]: *wprepare_erase m lm* (*b*, *Bk* # *list*) $\Longrightarrow$ *b* ≠ []
 by(*simp add*: *wprepare_invs*)

lemma *wprepare_goto_start_pos_Bk*[*simp*]: *wprepare_erase m lm* (*b*, *Bk* # *list*) $\Longrightarrow$
 wprepare_goto_start_pos m lm (*Bk* # *b*, *list*)
 apply(*simp only*: *wprepare_invs*, *auto*)
 done

lemma *wprepare_add_one_Bk_nonempty_snd*[*simp*]: ⟦*wprepare_add_one m lm* (*b*, *Bk* # *list*)⟧ $\Longrightarrow$ *list* ≠ []
 apply(*simp only*: *wprepare_invs*)
 apply(*case_tac lm*, *simp_all add*: *tape_of_list_def tape_of_nat_def* , *auto*)
 done

lemma *wprepare_goto_first_end_nonempty_snd_tl*[*simp*]:
 ⟦*lm* ≠ []; *wprepare_goto_first_end m lm* (*b*, *Bk* # *list*)⟧ $\Longrightarrow$ *list* ≠ []
 by(*simp only*: *wprepare_invs*, *auto*)

lemma *wprepare_erase_Bk_nonempty_list*[*simp*]: ⟦*lm* ≠ []; *wprepare_erase m lm* (*b*, *Bk* # *list*)⟧ $\Longrightarrow$ *list* ≠ []

apply(*simp only*: *wprepare_invs*, *auto*)
done

lemma *wprepare_goto_start_pos_Bk_nonempty*[*simp*]: $[\![lm \neq []; \ \ wprepare\_goto\_start\_pos\ m\ lm$ $(b, Bk \# list)]\!] \Longrightarrow list \neq []$
by(*cases lm*;*cases list*;*simp only*: *wprepare_invs*, *auto*)

lemma *wprepare_goto_start_pos_Bk_nonempty_fst*[*simp*]: $[\![lm \neq []; \ \ wprepare\_goto\_start\_pos\ m$ $lm\ (b, Bk \# list)]\!] \Longrightarrow b \neq []$
apply(*simp only*: *wprepare_invs*)
apply(*auto*)
done

lemma *wprepare_loop_goon_Bk_nonempty*[*simp*]: $[\![lm \neq []; wprepare\_loop\_goon\ m\ lm\ (b, Bk \#$ $list)]\!] \Longrightarrow b \neq []$
apply(*simp only*: *wprepare_invs*, *auto*)
done

lemma *wprepare_loop_goon_wprepare_add_one2_cases*[*simp*]: $[\![lm \neq []; wprepare\_loop\_goon\ m$ $lm\ (b, Bk \# list)]\!] \Longrightarrow$
$(list = [] \longrightarrow wprepare\_add\_one2\ m\ lm\ (Bk \# b, [])) \wedge$
$(list \neq [] \longrightarrow wprepare\_add\_one2\ m\ lm\ (Bk \# b, list))$
unfolding *wprepare_invs*
apply(*cases list*;*auto split*:*nat.split if_splits*)
by (*metis list.sel*(*3*) *tl_replicate*)

lemma *wprepare_add_one2_nonempty*[*simp*]: $wprepare\_add\_one2\ m\ lm\ (b, Bk \# list) \Longrightarrow b \neq$ $[]$
apply(*simp only*: *wprepare_invs*, *simp*)
done

lemma *wprepare_add_one2_cases*[*simp*]: $wprepare\_add\_one2\ m\ lm\ (b, Bk \# list) \Longrightarrow$
$(list = [] \longrightarrow wprepare\_add\_one2\ m\ lm\ (b, [Oc])) \wedge$
$(list \neq [] \longrightarrow wprepare\_add\_one2\ m\ lm\ (b, Oc \# list))$
apply(*simp only*: *wprepare_invs*, *auto*)
done

lemma *wprepare_goto_first_end_cases_Oc*[*simp*]: $wprepare\_goto\_first\_end\ m\ lm\ (b, Oc \# list)$
$\Longrightarrow (b = [] \longrightarrow wprepare\_goto\_first\_end\ m\ lm\ ([Oc], list)) \wedge$
$(b \neq [] \longrightarrow wprepare\_goto\_first\_end\ m\ lm\ (Oc \# b, list))$
apply(*simp only*: *wprepare_invs*, *auto*)
apply(*rule_tac* $x = 1$ **in** *exI*, *auto*) **apply**(*rename_tac ml mr rn*)
apply(*case_tac mr*, *simp_all add*:)
apply(*case_tac ml*, *simp_all add*:)
apply(*rule_tac* $x = Suc\ ml$ **in** *exI*, *simp_all add*:)
apply(*rule_tac* $x = mr - 1$ **in** *exI*, *simp*)
done

lemma *wprepare_erase_nonempty*[*simp*]: $wprepare\_erase\ m\ lm\ (b, Oc \# list) \Longrightarrow b \neq []$

apply(*simp only*: *wprepare_invs*, *auto simp*:)
done

lemma *wprepare_erase_Bk*[*simp*]: *wprepare_erase m lm* (*b*, *Oc* # *list*)
$\Longrightarrow$ *wprepare_erase m lm* (*b*, *Bk* # *list*)
apply(*simp only*:*wprepare_invs*, *auto simp*:)
done

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*)
apply(*simp only*:*wprepare_invs*, *auto*)
apply(*case_tac* [!] *lm*, *simp*, *simp_all*)
done

lemma *wprepare_loop_start_b_nonempty*[*simp*]: *wprepare_loop_start m lm* (*b*, *aa*) $\Longrightarrow$ *b* $\neq$ []
apply(*simp only*:*wprepare_invs*, *auto*)
done
lemma *exists_exp_of_Bk*[*elim*]: *Bk* # *list* = *Oc*↑(*mr*) @ *Bk*↑(*rn*) $\Longrightarrow$ $\exists$ *rn*. *list* = *Bk*↑(*rn*)
apply(*case_tac mr*, *simp_all*)
apply(*case_tac rn*, *simp_all*)
done

lemma *wprepare_loop_start_in_middle_Bk_False*[*simp*]: *wprepare_loop_start_in_middle m lm* (*b*, [*Bk*]) = *False*
by(*auto*)

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*
apply(*simp add*: *wprepare_loop_goon_in_middle.simps*, *auto*)
done

lemma *wprepare_loop_goon_Bk*[*simp*]: $[\![$*lm* $\neq$ []; *wprepare_loop_start m lm* (*b*, [*Bk*])$]\!]$ $\Longrightarrow$
wprepare_loop_goon m lm (*Bk* # *b*, [])
unfolding *wprepare_invs*
apply(*auto simp add*: *wprepare_loop_goon_on_rightmost.simps*
wprepare_loop_start_on_rightmost.simps)
apply(*rule_tac rev_eq*)
apply(*simp add*: *tape_of_nl_rev*)
apply(*simp add*: *exp_ind replicate_Suc*[*THEN sym*] *del*: *replicate_Suc*)
done

lemma *wprepare_loop_goon_in_middle_Bk_False2*[*simp*]: *wprepare_loop_start_on_rightmost m lm* (*b*, *Bk* # *a* # *lista*)

```
 ⟹ wprepare_loop_goon_in_middle m lm (Bk # b, a # lista) = False
  apply(auto simp: wprepare_loop_start_on_rightmost.simps
    wprepare_loop_goon_in_middle.simps)
  done

lemma wprepare_loop_goon_on_rightbmost_Bk_False[simp]: ⟦lm ≠ []; wprepare_loop_start_on_rightmost
m lm (b, Bk # a # lista)⟧
   ⟹ wprepare_loop_goon_on_rightmost m lm (Bk # b, a # lista)
  apply(simp only: wprepare_loop_start_on_rightmost.simps
    wprepare_loop_goon_on_rightmost.simps, auto simp: tape_of_nl_rev)
  apply(simp add: replicate_Suc[THEN sym] exp_ind tape_of_nl_rev del: replicate_Suc)
  by (meson Cons_replicate_eq)


lemma wprepare_loop_goon_in_middle_Bk_False3[simp]:
  assumes lm ≠ [] wprepare_loop_start_in_middle m lm (b, Bk # a # lista)
  shows wprepare_loop_goon_in_middle m lm (Bk # b, a # lista) (is ?t1)
   and wprepare_loop_goon_on_rightmost m lm (Bk # b, a # lista) = False (is ?t2)
proof -
  from assms obtain rn mr lm1 where ∗:rev b @ Oc ↑ mr @ Bk # <lm1> = Oc # Oc ↑ m @
Bk # Bk # <lm>
      b ≠ [] Bk # a # lista = Oc ↑ mr @ Bk # <lm1::nat list> @ Bk ↑ rn lm1 ≠ []
   by(auto simp add: wprepare_loop_start_in_middle.simps)
  thus ?t1 apply(simp add: wprepare_loop_start_in_middle.simps
      wprepare_loop_goon_in_middle.simps, auto)
   apply(rule_tac x = rn in exI, simp)
   apply(case_tac mr, simp_all add: )
   apply(case_tac lm1, simp)
   apply(rule_tac x = Suc (hd lm1) in exI, simp)
   apply(rule_tac x = tl lm1 in exI)
   apply(case_tac tl lm1, simp_all add: tape_of_list_def  tape_of_nat_def)
   done
  from ∗ show ?t2
   apply(simp add: wprepare_loop_start_in_middle.simps
    wprepare_loop_goon_on_rightmost.simps del:split_head_repeat, auto simp del:split_head_repeat)
    apply(case_tac mr)
    apply(case_tac  lm1::nat list, simp_all, case_tac tl lm1, simp_all)
     apply(auto simp add: tape_of_list_def )
     apply(case_tac [!] rna, simp_all add: )
   apply(case_tac mr, simp_all add: )
   apply(case_tac lm1, simp, case_tac list, simp)
   apply(simp_all add: tape_of_nat_def)
   by (metis Bk_not_tape_start tape_of_list_def tape_of_nat_list.elims)
qed

lemma wprepare_loop_goon_Bk2[simp]: ⟦lm ≠ []; wprepare_loop_start m lm (b, Bk # a # lista)⟧
⟹
 wprepare_loop_goon m lm (Bk # b, a # lista)
  apply(simp add: wprepare_loop_start.simps
    wprepare_loop_goon.simps)
```

```
  apply(erule_tac disjE, simp, auto)
  done

lemma start_2_goon:
  ⟦lm ≠ []; wprepare_loop_start m lm (b, Bk # list)⟧ ⟹
   (list = [] ⟶ wprepare_loop_goon m lm (Bk # b, [])) ∧
  (list ≠ [] ⟶ wprepare_loop_goon m lm (Bk # b, list))
  apply(case_tac list, auto)
  done

lemma add_one_2_add_one: wprepare_add_one m lm (b, Oc # list)
  ⟹ (hd b = Oc ⟶ (b = [] ⟶ wprepare_add_one m lm ([], Bk # Oc # list)) ∧
                (b ≠ [] ⟶ wprepare_add_one m lm (tl b, Oc # Oc # list))) ∧
  (hd b ≠ Oc ⟶ (b = [] ⟶ wprepare_add_one m lm ([], Bk # Oc # list)) ∧
              (b ≠ [] ⟶ wprepare_add_one m lm (tl b, hd b # Oc # list)))
  unfolding wprepare_add_one.simps by auto

lemma wprepare_loop_start_on_rightmost_Oc[simp]: wprepare_loop_start_on_rightmost m lm (b,
Oc # list) ⟹
  wprepare_loop_start_on_rightmost m lm (Oc # b, list)
  apply(simp add: wprepare_loop_start_on_rightmost.simps)
 by (metis Cons_replicate_eq cell.distinct(1) list.sel(3) self_append_conv2 tl_append2 tl_replicate)

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 −
  from assms obtain mr lm1 rn
    where rev b @ Oc ↑ mr @ Bk # <lm1::nat list> = Oc # Oc ↑ m @ Bk # Bk # <lm>
      Oc # list = Oc ↑ mr @ Bk # <lm1> @ Bk ↑ rn lm1 ≠ []
    by(auto simp add: wprepare_loop_start_in_middle.simps)
  thus ?thesis
    apply(auto simp add: wprepare_loop_start_in_middle.simps)
    apply(rule_tac x = rn in exI, auto)
    apply(case_tac mr, simp, simp add: )
    apply(rule_tac x = mr − 1 in exI, simp)
    apply(rule_tac x = lm1 in exI, simp)
    done
qed

lemma start_2_start: wprepare_loop_start m lm (b, Oc # list) ⟹
      wprepare_loop_start m lm (Oc # b, list)
  apply(simp add: wprepare_loop_start.simps)
  apply(erule_tac disjE, simp_all )
  done

lemma wprepare_loop_goon_Oc_nonempty[simp]: wprepare_loop_goon m lm (b, Oc # list) ⟹
b ≠ []
  apply(simp add: wprepare_loop_goon.simps
     wprepare_loop_goon_in_middle.simps
```

```
   wprepare_loop_goon_on_rightmost.simps)
 apply(auto)
 done

lemma wprepare_goto_start_pos_Oc_nonempty[simp]: wprepare_goto_start_pos m lm (b, Oc #
list) ⟹ b ≠ []
 apply(simp add: wprepare_goto_start_pos.simps)
 done

lemma wprepare_loop_goon_on_rightmost_Oc_False[simp]: wprepare_loop_goon_on_rightmost m
lm (b, Oc # list) = False
 apply(simp add: wprepare_loop_goon_on_rightmost.simps)
 done

lemma wprepare_loop1: ⟦rev b @ Oc↑(mr) =  Oc↑(Suc m) @ Bk # Bk # <lm>;
      b ≠ []; 0 < mr; Oc # list = Oc↑(mr) @ Bk↑(rn)⟧
     ⟹ wprepare_loop_start_on_rightmost m lm (Oc # b, list)
 apply(simp add: wprepare_loop_start_on_rightmost.simps)
 apply(rule_tac x = rn in exI, simp)
 apply(case_tac mr, simp, simp)
 done

lemma wprepare_loop2: ⟦rev b @ Oc↑(mr) @ Bk # <a # lista> = Oc↑(Suc m) @ Bk # Bk #
<lm>;
           b ≠ []; Oc # list = Oc↑(mr) @ Bk # <(a::nat) # lista> @ Bk↑(rn)⟧
     ⟹  wprepare_loop_start_in_middle m lm (Oc # b, list)
 apply(simp add: wprepare_loop_start_in_middle.simps)
 apply(rule_tac x = rn in exI, simp)
 apply(case_tac mr, simp_all add: ) apply(rename_tac nat)
 apply(rule_tac x = nat in exI, simp)
 apply(rule_tac x = a#lista in exI, simp)
 done

lemma wprepare_loop_goon_in_middle_cases[simp]: wprepare_loop_goon_in_middle m lm (b, Oc
# list) ⟹
           wprepare_loop_start_on_rightmost m lm (Oc # b, list) ∨
           wprepare_loop_start_in_middle m lm (Oc # b, list)
 apply(simp add: wprepare_loop_goon_in_middle.simps split: if_splits) apply(rename_tac lm1)
 apply(case_tac lm1, simp_all add: wprepare_loop1 wprepare_loop2)
 done

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)
  apply(auto simp add: wprepare_add_one.simps wprepare_loop_goon.simps
   wprepare_loop_start.simps)
 done

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)
  apply(auto simp: wprepare_goto_start_pos.simps
    wprepare_loop_start_on_rightmost.simps) apply(rename_tac rn)
  apply(rule_tac x = rn in exI, simp)
  apply(simp add: replicate_Suc[THEN sym] exp_ind del: replicate_Suc)
  done

lemma wprepare_loop_start_in_middle_2_Oc[simp]:  wprepare_goto_start_pos m (a # aa # lis-
taa) (b, Oc # list)
      ⟹wprepare_loop_start_in_middle m (a # aa # listaa) (Oc # b, list)
  apply(auto simp: wprepare_goto_start_pos.simps
    wprepare_loop_start_in_middle.simps) apply(rename_tac rn)
  apply(rule_tac x = rn in exI, simp)
  apply(simp add: exp_ind[THEN sym])
  apply(rule_tac x = a in exI, rule_tac x = aa#listaa in exI, simp)
  apply(simp add: tape_of_nl_cons)
  done

lemma wprepare_loop_start_Oc2[simp]: ⟦lm ≠ []; wprepare_goto_start_pos m lm (b, Oc # list)⟧
      ⟹ wprepare_loop_start m lm (Oc # b, list)
  by(cases lm;cases tl lm, auto simp add: wprepare_loop_start.simps)

lemma wprepare_add_one2_Oc_nonempty[simp]: wprepare_add_one2 m lm (b, Oc # list) ⟹ b
≠ []
  apply(auto simp: wprepare_add_one2.simps)
  done

lemma add_one_2_stop:
  wprepare_add_one2 m lm (b, Oc # list)
  ⟹  wprepare_stop m lm (tl b, hd b # Oc # list)
  apply(simp add: wprepare_add_one2.simps)
  done

declare wprepare_stop.simps[simp del]

lemma wprepare_correctness:
  assumes h: lm ≠ []
  shows let P = (λ (st, l, r). st = 0) in
  let Q = (λ (st, l, r). wprepare_inv st m lm (l, r)) in
  let f = (λ stp. steps0 (Suc 0, [], (<m # lm>)) t_wcode_prepare stp) in
    ∃ n .P (f n) ∧ Q (f n)
proof −
  let ?P = (λ (st, l, r). st = 0)
  let ?Q = (λ (st, l, r). wprepare_inv st m lm (l, r))
  let ?f = (λ stp. steps0 (Suc 0, [], (<m # lm>)) t_wcode_prepare stp)
  have ∃ n. ?P (?f n) ∧ ?Q (?f n)
  proof(rule_tac halt_lemma2)
    show ∀ n. ¬ ?P (?f n) ∧ ?Q (?f n) ⟶
            ?Q (?f (Suc n)) ∧ (?f (Suc n), ?f n) ∈ wcode_prepare_le
```

```
    using h
    apply(rule_tac allI, rule_tac impI) apply(rename_tac n)
    apply(case_tac ?f n, simp add: step.simps) apply(rename_tac c)
    apply(case_tac c, simp, case_tac [2] aa)
     apply(simp_all add: wprepare_inv.simps wcode_prepare_le_def lex_triple_def lex_pair_def
       split: if_splits)
      apply(simp_all add: start_2_goon  start_2_start
       add_one_2_add_one add_one_2_stop)
    apply(auto simp: wprepare_add_one2.simps)
    done
  qed (auto simp add: steps.simps wprepare_inv.simps wprepare_invs)
  thus ?thesis
   apply(auto)
   done
qed

lemma tm_wf_t_wcode_prepare[intro]: tm_wf (t_wcode_prepare, 0)
  apply(simp add:tm_wf.simps t_wcode_prepare_def)
  done

lemma is_28_even[intro]: (28 + (length t_twice_compile + length t_fourtimes_compile)) mod 2
= 0
  by(auto simp: t_twice_compile_def t_fourtimes_compile_def)

lemma b_le_28[elim]: (a, b) ∈ set t_wcode_main_first_part ⟹
  b ≤ (28 + (length t_twice_compile + length t_fourtimes_compile)) div 2
  apply(auto simp: t_wcode_main_first_part_def t_twice_def)
  done



lemma tm_wf_change_termi:
  assumes tm_wf (tp, 0)
  shows list_all (λ(acn, st). (st ≤ Suc (length tp div 2))) (adjust0 tp)
proof −
  { fix acn st n
    assume tp ! n = (acn, st) n < length tp 0 < st
    hence (acn, st)∈set tp by (metis nth_mem)
    with assms tm_wf.simps have st ≤ length tp div 2 + 0 by auto
    hence st ≤ Suc (length tp div 2) by auto
  }
  thus ?thesis
    by(auto simp: tm_wf.simps List.list_all_length adjust.simps split: if_splits prod.split)
qed

lemma tm_wf_shift:
  assumes list_all (λ(acn, st). (st ≤ y)) tp
  shows list_all (λ(acn, st). (st ≤ y + off)) (shift tp off)
proof −
  have [dest!]:⋀ P Q n. ∀ n. Q n ⟶ P n ⟹ Q n ⟹ P n by metis
```

```
 from assms show ?thesis by(auto simp: tm_wf.simps List.list_all_length shift.simps)
qed

declare length_tp'[simp del]

lemma length_mopup_1[simp]: length (mopup (Suc 0)) = 16
 apply(auto simp: mopup.simps)
 done

lemma twice_plus_28_elim[elim]: (a, b) ∈ set (shift (adjust0 t_twice_compile) 12) ⟹
 b ≤ (28 + (length t_twice_compile + length t_fourtimes_compile)) div 2
 apply(simp add: t_twice_compile_def t_fourtimes_compile_def)
proof -
 assume g: (a, b)
  ∈ set (shift
      (adjust
       (tm_of abc_twice @
        shift (mopup (Suc 0)) (length (tm_of abc_twice) div 2))
       (Suc ((length (tm_of abc_twice) + 16) div 2)))
      12)
 moreover have length (tm_of abc_twice) mod 2 = 0 by auto
 moreover have length (tm_of abc_fourtimes) mod 2 = 0 by auto
 ultimately have list_all (λ(acn, st). (st ≤ (60 + (length (tm_of abc_twice) + length (tm_of
abc_fourtimes))) div 2))
  (shift (adjust0 t_twice_compile) 12)
 proof(auto simp add: mod_ex1 del: adjust.simps)
  assume even (length (tm_of abc_twice))
  then obtain q where q:length (tm_of abc_twice) = 2 * q by auto
  assume even (length (tm_of abc_fourtimes))
  then obtain qa where qa:length (tm_of abc_fourtimes) = 2 * qa by auto
  note h = q qa
  hence list_all (λ(acn, st). st ≤ (18 + (q + qa)) + 12) (shift (adjust0 t_twice_compile) 12)
  proof(rule_tac tm_wf_shift t_twice_compile_def)
   have list_all (λ(acn, st). st ≤ Suc (length t_twice_compile div 2)) (adjust0 t_twice_compile)
    by(rule_tac tm_wf_change_termi, auto)
   thus list_all (λ(acn, st). st ≤ 18 + (q + qa)) (adjust0 t_twice_compile)
    using h
    apply(simp add: t_twice_compile_def, auto simp: List.list_all_length)
    done
  qed
  thus list_all (λ(acn, st). st ≤ 30 + (length (tm_of abc_twice) div 2 + length (tm_of abc_fourtimes)
div 2))
   (shift (adjust0 t_twice_compile) 12) using h
    by simp
 qed
 thus b ≤ (60 + (length (tm_of abc_twice) + length (tm_of abc_fourtimes))) div 2
  using g
  apply(auto simp:t_twice_compile_def)
  apply(simp add: Ball_set[THEN sym])
  apply(erule_tac x = (a, b) in ballE, simp, simp)
```

```
  done
qed

lemma length_plus_28_elim2[elim]: (a, b) ∈ set (shift (adjust0 t_fourtimes_compile) (t_twice_len
+ 13))
  ⟹ b ≤ (28 + (length t_twice_compile + length t_fourtimes_compile)) div 2
 apply(simp add: t_twice_compile_def t_fourtimes_compile_def t_twice_len_def)
proof -
 assume g: (a, b)
  ∈ set (shift
        (adjust (tm_of abc_fourtimes @ shift (mopup (Suc 0)) (length (tm_of abc_fourtimes) div
2))
          (Suc ((length (tm_of abc_fourtimes) + 16) div 2)))
        (length t_twice div 2 + 13))
 moreover have length (tm_of abc_twice) mod 2 = 0 by auto
 moreover have length (tm_of abc_fourtimes) mod 2 = 0 by auto
 ultimately have list_all (λ(acn, st). (st ≤ (60 + (length (tm_of abc_twice) + length (tm_of
abc_fourtimes))) div 2))
  (shift (adjust0 (tm_of abc_fourtimes @ shift (mopup (Suc 0))
  (length (tm_of abc_fourtimes) div 2))) (length t_twice div 2 + 13))
 proof(auto simp: mod_ex1 t_twice_def t_twice_compile_def)
  assume even (length (tm_of abc_twice))
  then obtain q where q:length (tm_of abc_twice) = 2 * q by auto
  assume even (length (tm_of abc_fourtimes))
  then obtain qa where qa:length (tm_of abc_fourtimes) = 2 * qa by auto
  note h = q qa
  hence list_all (λ(acn, st). st ≤ (9 + qa + (21 + q)))
   (shift (adjust0 (tm_of abc_fourtimes @ shift (mopup (Suc 0)) qa)) (21 + q))
  proof(rule_tac tm_wf_shift t_twice_compile_def)
   have list_all (λ(acn, st). st ≤ Suc (length (tm_of abc_fourtimes @ shift
    (mopup (Suc 0)) qa) div 2)) (adjust0 (tm_of abc_fourtimes @ shift (mopup (Suc 0)) qa))
    apply(rule_tac tm_wf_change_termi)
    using wf_fourtimes h
    apply(simp add: t_fourtimes_compile_def)
    done
   thus list_all (λ(acn, st). st ≤ 9 + qa)
    (adjust (tm_of abc_fourtimes @ shift (mopup (Suc 0)) qa)
     (Suc (length (tm_of abc_fourtimes @ shift (mopup (Suc 0)) qa) div
        2)))
    using h
    apply(simp)
    done
  qed
  thus list_all
   (λ(acn, st). st ≤ 30 + (length (tm_of abc_twice) div 2 + length (tm_of abc_fourtimes) div 2))
   (shift
    (adjust (tm_of abc_fourtimes @ shift (mopup (Suc 0)) (length (tm_of abc_fourtimes) div 2))
     (9 + length (tm_of abc_fourtimes) div 2))
    (21 + length (tm_of abc_twice) div 2))
   apply(subgoal_tac qa + q = q + qa)
```

```
   apply(simp add: h)
  apply(simp)
  done
 qed
 thus b ≤ (60 + (length (tm_of abc_twice) + length (tm_of abc_fourtimes))) div 2
  using g
  apply(simp add: Ball_set[THEN sym])
  apply(erule_tac x = (a, b) in ballE, simp, simp)
  done
qed

lemma tm_wf_t_wcode_main[intro]: tm_wf (t_wcode_main, 0)
 by(auto simp: t_wcode_main_def tm_wf.simps
   t_twice_def t_fourtimes_def del: List.list_all_iff)

declare tm_comp.simps[simp del]

lemma prepare_mainpart_lemma:
 args ≠ [] ⟹
 ∃ stp ln rn. steps0 (Suc 0, [], <m # args>) (t_wcode_prepare |+| t_wcode_main) stp
          = (0,  Bk # Oc↑(Suc m), Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(bl_bin (<args>))
@ Bk↑(rn))
proof −
 let ?P1 = (λ (l, r). (l::cell list) = [] ∧ r = <m # args>)
 let ?Q1 = (λ (l, r). wprepare_stop m args (l, r))
 let ?P2 = ?Q1
 let ?Q2 = (λ (l, r). (∃ ln rn. l = Bk # Oc↑(Suc m) ∧
                 r =  Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(bl_bin (<args>)) @ Bk↑(rn)))
 let ?P3 = λ tp. False
 assume h: args ≠ []
 have {?P1} t_wcode_prepare |+| t_wcode_main {?Q2}
 proof(rule_tac Hoare_plus_halt)
  show {?P1} t_wcode_prepare {?Q1}
  proof(rule_tac Hoare_haltI, auto)
   show ∃ n. is_final (steps0 (Suc 0, [], <m # args>) t_wcode_prepare n) ∧
    wprepare_stop m args holds_for steps0 (Suc 0, [], <m # args>) t_wcode_prepare n
    using wprepare_correctness[of args m,OF h]
    apply(auto simp add: wprepare_inv.simps)
    by (metis holds_for.simps is_finalI)
  qed
 next
  show {?P2} t_wcode_main {?Q2}
  proof(rule_tac Hoare_haltI, auto)
   fix l r
   assume wprepare_stop m args (l, r)
   thus ∃ n. is_final (steps0 (Suc 0, l, r) t_wcode_main n) ∧
      (λ(l, r). l = Bk # Oc # Oc ↑ m ∧ (∃ ln rn. r = Bk # Oc # Bk ↑ ln @
    Bk # Bk # Oc ↑ bl_bin (<args>) @ Bk ↑ rn)) holds_for steps0 (Suc 0, l, r) t_wcode_main
n
   proof(auto simp: wprepare_stop.simps)
```

```
      fix rn
      show ∃n. is_final (steps0 (Suc 0, Bk # <rev args> @ Bk # Bk # Oc # Oc ↑ m, Bk # Oc
# Bk ↑ rn) t_wcode_main n) ∧
        (λ(l, r). l = Bk # Oc # Oc ↑ m ∧
        (∃ln rn. r = Bk # Oc # Bk ↑ ln @
        Bk # Bk # Oc ↑ bl_bin (<args>) @
         Bk ↑ rn)) holds_for steps0 (Suc 0, Bk # <rev args> @ Bk # Bk # Oc # Oc ↑ m, Bk #
Oc # Bk ↑ rn) t_wcode_main n
        using t_wcode_main_lemma_pre[of args <args> 0 Oc↑(Suc m) 0 rn,OF h refl]
        apply(auto simp: tape_of_nl_rev)
        apply(rename_tac stp ln rna)
        apply(rule_tac x = stp in exI, auto)
        done
    qed
  qed
 next
  show tm_wf0 t_wcode_prepare
    by auto
 qed
 then obtain n
  where ⋀tp. (case tp of (l, r) ⇒ l = [] ∧ r = <m # args>) ⟶
    (is_final (steps0 (1, tp) (t_wcode_prepare |+| t_wcode_main) n) ∧
        (λ(l, r).
          ∃ln rn.
            l = Bk # Oc ↑ Suc m ∧
            r = Bk # Oc # Bk ↑ ln @ Bk # Bk # Oc ↑ bl_bin (<args>) @ Bk ↑ rn) holds_for
steps0 (1, tp) (t_wcode_prepare |+| t_wcode_main) n)
  unfolding Hoare_halt_def by auto
 thus ?thesis
  apply(rule_tac x = n in exI)
  apply(case_tac (steps0 (Suc 0, [], <m # args>)
   (adjust0 t_wcode_prepare @ shift t_wcode_main (length t_wcode_prepare div 2)) n))
  apply(auto simp: tm_comp.simps)
  done
qed

definition tinres :: cell list ⇒ cell list ⇒ bool
 where
  tinres xs ys = (∃n. xs = ys @ Bk ↑ n ∨ ys = xs @ Bk ↑ n)

lemma tinres_fetch_congr[simp]: tinres r r′ ⟹
 fetch t ss (read r) =
 fetch t ss (read r′)
 apply(simp add: fetch.simps, auto split: if_splits simp: tinres_def)
 using hd_replicate apply fastforce
 using hd_replicate apply fastforce
 done

lemma nonempty_hd_tinres[simp]: ⟦tinres r r′; r ≠ []; r′ ≠ []⟧ ⟹ hd r = hd r′
 apply(auto simp: tinres_def)
```

```
  done

lemma tinres_nonempty[simp]:
  ⟦tinres r []; r ≠ []⟧ ⟹ hd r = Bk
  ⟦tinres [] r'; r' ≠ []⟧ ⟹ hd r' = Bk
  ⟦tinres r [];  r ≠ []⟧ ⟹ tinres (tl r) []
  tinres r r' ⟹ tinres (b # r) (b # r')
  by(auto simp: tinres_def)

lemma ex_move_tl[intro]: ∃ na. tl r = tl (r @ Bk↑(n)) @ Bk↑(na) ∨ tl (r @ Bk↑(n)) = tl r @
Bk↑(na)
  apply(case_tac r, simp)
  by(case_tac n, auto)

lemma tinres_tails[simp]: tinres r r' ⟹ tinres (tl r) (tl r')
  apply(auto simp: tinres_def)
  by(case_tac r', auto)

lemma tinres_empty[simp]:
  ⟦tinres [] r'⟧ ⟹ tinres [] (tl r')
  tinres r [] ⟹ tinres (Bk # tl r) [Bk]
  tinres r [] ⟹ tinres (Oc # tl r) [Oc]
  by(auto simp: tinres_def)

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 (cases fetch t ss (read r'))
  case (Pair a b)
  have sa:sa = sb using assms Pair by(force simp: step.simps)
  have la = lb ∧ tinres ra rb using assms Pair
    by(cases a, auto simp: step.simps split: if_splits)
  thus ?thesis using sa by auto
qed

lemma tinres_steps2:
  ⟦tinres r r'; steps0 (ss, l, r) t stp = (sa, la, ra); steps0 (ss, l, r') t stp = (sb, lb, rb)⟧
    ⟹ la = lb ∧ tinres ra rb ∧ sa = sb
proof(induct stp arbitrary: sa la ra sb lb rb)
  case (Suc stp sa la ra sb lb rb)
  then show ?case
    apply(simp)
    apply(case_tac (steps0 (ss, l, r) t stp))
    apply(case_tac (steps0 (ss, l, r') t stp))
  proof -
    fix stp a b c aa ba ca
    assume ind: ⋀sa la ra sb lb rb. ⟦steps0 (ss, l, r) t stp = (sa, la, ra);
    steps0 (ss, l, r') t stp = (sb, lb, rb)⟧ ⟹ la = lb ∧ tinres ra rb ∧ sa = sb
      and h:  tinres r r' step0 (steps0 (ss, l, r) t stp) t = (sa, la, ra)
      step0 (steps0 (ss, l, r') t stp) t = (sb, lb, rb) steps0 (ss, l, r) t stp = (a, b, c)
```

steps0 (*ss*, *l*, *r'*) *t stp* = (*aa*, *ba*, *ca*)
have $b = ba \wedge$ *tinres c ca* $\wedge a = aa$
apply(*rule_tac ind*, *simp_all add*: *h*)
done
thus $la = lb \wedge$ *tinres ra rb* $\wedge sa = sb$
apply(*rule_tac* $l = b$ **and** $r = c$ **and** $ss = a$ **and** $r' = ca$
and $t = t$ **in** *tinres_step2*)
using *h*
apply(*simp*, *simp*, *simp*)
done
qed
qed (*simp add*: *steps.simps*)

definition *t_wcode_adjust* :: *instr list*
where
t_wcode_adjust = [(*W1*, *1*), (*R*, *2*), (*Nop*, *2*), (*R*, *3*), (*R*, *3*), (*R*, *4*),
(*L*, *8*), (*L*, *5*), (*L*, *6*), (*W0*, *5*), (*L*, *6*), (*R*, *7*),
(*W1*, *2*), (*Nop*, *7*), (*L*, *9*), (*W0*, *8*), (*L*, *9*), (*L*, *10*),
(*L*, *11*), (*L*, *10*), (*R*, *0*), (*L*, *11*)]

lemma *fetch_t_wcode_adjust*[*simp*]:
fetch t_wcode_adjust (*Suc 0*) *Bk* = (*W1*, *1*)
fetch t_wcode_adjust (*Suc 0*) *Oc* = (*R*, *2*)
fetch t_wcode_adjust (*Suc* (*Suc 0*)) *Oc* = (*R*, *3*)
fetch t_wcode_adjust (*Suc* (*Suc* (*Suc 0*))) *Oc* = (*R*, *4*)
fetch t_wcode_adjust (*Suc* (*Suc* (*Suc 0*))) *Bk* = (*R*, *3*)
fetch t_wcode_adjust 4 Bk = (*L*, *8*)
fetch t_wcode_adjust 4 Oc = (*L*, *5*)
fetch t_wcode_adjust 5 Oc = (*W0*, *5*)
fetch t_wcode_adjust 5 Bk = (*L*, *6*)
fetch t_wcode_adjust 6 Oc = (*R*, *7*)
fetch t_wcode_adjust 6 Bk = (*L*, *6*)
fetch t_wcode_adjust 7 Bk = (*W1*, *2*)
fetch t_wcode_adjust 8 Bk = (*L*, *9*)
fetch t_wcode_adjust 8 Oc = (*W0*, *8*)
fetch t_wcode_adjust 9 Oc = (*L*, *10*)
fetch t_wcode_adjust 9 Bk = (*L*, *9*)
fetch t_wcode_adjust 10 Bk = (*L*, *11*)
fetch t_wcode_adjust 10 Oc = (*L*, *10*)
fetch t_wcode_adjust 11 Oc = (*L*, *11*)
fetch t_wcode_adjust 11 Bk = (*R*, *0*)
by(*auto simp*: *fetch.simps t_wcode_adjust_def nth_of.simps numeral*)

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 $(l, r) =$
$(\exists\ ln\ rn\ ml\ mr.\ l = Oc\uparrow(ml) @ Bk \# Oc\uparrow(Suc\ m) \ \wedge$
$r = Oc \# Bk\uparrow(ln) @ Bk \# Oc\uparrow(mr) @ Bk\uparrow(rn) \wedge$
$ml + mr = Suc\ (Suc\ rs) \wedge mr > 0)$

fun *wadjust_loop_right_move* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust_loop_right_move m rs $(l, r) =$
$(\exists\ ml\ mr\ nl\ nr\ rn.\ l = Bk\uparrow(nl) @ Oc \# Oc\uparrow(ml) @ Bk \# Oc\uparrow(Suc\ m) \wedge$
$r = Bk\uparrow(nr) @ Oc\uparrow(mr) @ Bk\uparrow(rn) \wedge$
$ml + mr = Suc\ (Suc\ rs) \wedge mr > 0 \wedge$
$nl + nr > 0)$

fun *wadjust_loop_check* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust_loop_check m rs $(l, r) =$
$(\exists\ ml\ mr\ ln\ rn.\ l = Oc \# Bk\uparrow(ln) @ Bk \# Oc \# Oc\uparrow(ml) @ Bk \# Oc\uparrow(Suc\ m) \wedge$
$r = Oc\uparrow(mr) @ Bk\uparrow(rn) \wedge ml + mr = (Suc\ rs))$

fun *wadjust_loop_erase* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust_loop_erase m rs $(l, r) =$
$(\exists\ ml\ mr\ ln\ rn.\ l = Bk\uparrow(ln) @ Bk \# Oc \# Oc\uparrow(ml) @ Bk \# Oc\uparrow(Suc\ m) \wedge$
$tl\ r = Oc\uparrow(mr) @ Bk\uparrow(rn) \wedge ml + mr = (Suc\ rs) \wedge mr > 0)$

fun *wadjust_loop_on_left_moving_O* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust_loop_on_left_moving_O m rs $(l, r) =$
$(\exists\ ml\ mr\ ln\ rn.\ l = Oc\uparrow(ml) @ Bk \# Oc\uparrow(Suc\ m\ ) \wedge$
$r = Oc \# Bk\uparrow(ln) @ Bk \# Bk \# Oc\uparrow(mr) @ Bk\uparrow(rn) \wedge$
$ml + mr = Suc\ rs \wedge mr > 0)$

fun *wadjust_loop_on_left_moving_B* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust_loop_on_left_moving_B m rs $(l, r) =$
$(\exists\ ml\ mr\ nl\ nr\ rn.\ l = Bk\uparrow(nl) @ Oc \# Oc\uparrow(ml) @ Bk \# Oc\uparrow(Suc\ m) \wedge$
$r = Bk\uparrow(nr) @ Bk \# Bk \# Oc\uparrow(mr) @ Bk\uparrow(rn) \wedge$
$ml + mr = Suc\ rs \wedge mr > 0)$

fun *wadjust_loop_on_left_moving* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust_loop_on_left_moving m rs $(l, r) =$
(*wadjust_loop_on_left_moving_O m rs* $(l, r) \vee$
wadjust_loop_on_left_moving_B m rs $(l, r))$

fun *wadjust_loop_right_move2* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where

$wadjust\_loop\_right\_move2\ m\ rs\ (l, r) =$
$(\exists\ ml\ mr\ ln\ rn.\ l = Oc \# Oc\uparrow(ml) @ Bk \# Oc\uparrow(Suc\ m) \wedge$
$r = Bk\uparrow(ln) @ Bk \# Bk \# Oc\uparrow(mr) @ Bk\uparrow(rn) \wedge$
$ml + mr = Suc\ rs \wedge mr > 0)$

fun $wadjust\_erase2 :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
where
$wadjust\_erase2\ m\ rs\ (l, r) =$
$(\exists\ ln\ rn.\ l = Bk\uparrow(ln) @ Bk \# Oc \# Oc\uparrow(Suc\ rs) @ Bk \# Oc\uparrow(Suc\ m) \wedge$
$tl\ r = Bk\uparrow(rn))$

fun $wadjust\_on\_left\_moving\_O :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
where
$wadjust\_on\_left\_moving\_O\ m\ rs\ (l, r) =$
$(\exists\ rn.\ l = Oc\uparrow(Suc\ rs) @ Bk \# Oc\uparrow(Suc\ m) \wedge$
$r = Oc \# Bk\uparrow(rn))$

fun $wadjust\_on\_left\_moving\_B :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
where
$wadjust\_on\_left\_moving\_B\ m\ rs\ (l, r) =$
$(\exists\ ln\ rn.\ l = Bk\uparrow(ln) @ Oc \# Oc\uparrow(Suc\ rs) @ Bk \# Oc\uparrow(Suc\ m) \wedge$
$r = Bk\uparrow(rn))$

fun $wadjust\_on\_left\_moving :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
where
$wadjust\_on\_left\_moving\ m\ rs\ (l, r) =$
$(wadjust\_on\_left\_moving\_O\ m\ rs\ (l, r) \vee$
$wadjust\_on\_left\_moving\_B\ m\ rs\ (l, r))$

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 $wadjust\_goon\_left\_moving\_O :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
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 $wadjust\_goon\_left\_moving :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
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 $wadjust\_backto\_standard\_pos\_B :: nat \Rightarrow nat \Rightarrow tape \Rightarrow bool$
where

```
  wadjust_backto_standard_pos_B m rs (l, r) =
     (∃ rn. l = [] ∧
          r = Bk # Oc↑(Suc m )@ Bk # Oc↑(Suc (Suc rs)) @ Bk↑(rn))

fun wadjust_backto_standard_pos_O :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_backto_standard_pos_O m rs (l, r) =
   (∃ ml mr rn. l = Oc↑(ml) ∧
                r = Oc↑(mr) @ Bk # Oc↑(Suc (Suc rs)) @ Bk↑(rn) ∧
                ml + mr = Suc m ∧ mr > 0)

fun wadjust_backto_standard_pos :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_backto_standard_pos m rs (l, r) =
     (wadjust_backto_standard_pos_B m rs (l, r) ∨
     wadjust_backto_standard_pos_O m rs (l, r))

fun wadjust_stop :: nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_stop m rs (l, r) =
     (∃ rn. l = [Bk] ∧
          r = Oc↑(Suc m )@ Bk # Oc↑(Suc (Suc rs)) @ Bk↑(rn))

declare wadjust_start.simps[simp del]  wadjust_loop_start.simps[simp del]
 wadjust_loop_right_move.simps[simp del]  wadjust_loop_check.simps[simp del]
 wadjust_loop_erase.simps[simp del] wadjust_loop_on_left_moving.simps[simp del]
 wadjust_loop_right_move2.simps[simp del] wadjust_erase2.simps[simp del]
 wadjust_on_left_moving_O.simps[simp del] wadjust_on_left_moving_B.simps[simp del]
 wadjust_on_left_moving.simps[simp del] wadjust_goon_left_moving_B.simps[simp del]
 wadjust_goon_left_moving_O.simps[simp del] wadjust_goon_left_moving.simps[simp del]
 wadjust_backto_standard_pos.simps[simp del] wadjust_backto_standard_pos_B.simps[simp del]
 wadjust_backto_standard_pos_O.simps[simp del] wadjust_stop.simps[simp del]

fun wadjust_inv :: nat ⇒ nat ⇒ nat ⇒ tape ⇒ bool
 where
  wadjust_inv st m rs (l, r) =
    (if st = Suc 0 then wadjust_start m rs (l, r)
     else if st = Suc (Suc 0) then wadjust_loop_start m rs (l, r)
     else if st = Suc (Suc (Suc 0)) then wadjust_loop_right_move m rs (l, r)
     else if st = 4 then wadjust_loop_check m rs (l, r)
     else if st = 5 then wadjust_loop_erase m rs (l, r)
     else if st = 6 then wadjust_loop_on_left_moving m rs (l, r)
     else if st = 7 then wadjust_loop_right_move2 m rs (l, r)
     else if st = 8 then wadjust_erase2 m rs (l, r)
     else if st = 9 then wadjust_on_left_moving m rs (l, r)
     else if st = 10 then wadjust_goon_left_moving m rs (l, r)
     else if st = 11 then wadjust_backto_standard_pos m rs (l, r)
     else if st = 0 then wadjust_stop m rs (l, r)
     else False
)
```

```
declare wadjust_inv.simps[simp del]

fun wadjust_phase :: nat ⇒ config ⇒ nat
  where
    wadjust_phase rs (st, l, r) =
        (if st = 1 then 3
         else if st ≥ 2 ∧ st ≤ 7 then 2
         else if st ≥ 8 ∧ st ≤ 11 then 1
         else 0)

fun wadjust_stage :: nat ⇒ config ⇒ nat
  where
    wadjust_stage rs (st, l, r) =
        (if st ≥ 2 ∧ st ≤ 7 then
             rs − length (takeWhile (λ a. a = Oc)
                   (tl (dropWhile (λ a. a = Oc) (rev l @ r))))
         else 0)

fun wadjust_state :: nat ⇒ config ⇒ nat
  where
    wadjust_state rs (st, l, r) =
      (if st ≥ 2 ∧ st ≤ 7 then 8 − st
       else if st ≥ 8 ∧ st ≤ 11 then 12 − st
       else 0)

fun wadjust_step :: nat ⇒ config ⇒ nat
  where
    wadjust_step rs (st, l, r) =
      (if st = 1 then (if hd r = Bk then 1
                   else 0)
       else if st = 3 then length r
       else if st = 5 then (if hd r = Oc then 1
                     else 0)
       else if st = 6 then length l
       else if st = 8 then (if hd r = Oc then 1
                     else 0)
       else if st = 9 then length l
       else if st = 10 then length l
       else if st = 11 then (if hd r = Bk then 0
                      else Suc (length l))
       else 0)

fun wadjust_measure :: (nat × 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*
 by(*auto intro*:*wf_lex_prod simp*: *Abacus*.*lex_pair_def lex_square_def*
 Abacus.*lex_triple_def*)

lemma *wf_wadjust_le*[*intro*]: *wf wadjust_le*
 by(*auto intro*:*wf_inv_image simp*: *wadjust_le_def*
 Abacus.*lex_triple_def Abacus*.*lex_pair_def*)

lemma *wadjust_start_snd_nonempty*[*simp*]: *wadjust_start m rs* (*c*, []) = *False*
 apply(*auto simp*: *wadjust_start*.*simps*)
 done

lemma *wadjust_loop_right_move_fst_nonempty*[*simp*]: *wadjust_loop_right_move m rs* (*c*, []) ⟹ *c* ≠ []
 apply(*auto simp*: *wadjust_loop_right_move*.*simps*)
 done

lemma *wadjust_loop_check_fst_nonempty*[*simp*]: *wadjust_loop_check m rs* (*c*, []) ⟹ *c* ≠ []
 apply(*simp only*: *wadjust_loop_check*.*simps*, *auto*)
 done

lemma *wadjust_loop_start_snd_nonempty*[*simp*]: *wadjust_loop_start m rs* (*c*, []) = *False*
 apply(*simp add*: *wadjust_loop_start*.*simps*)
 done

lemma *wadjust_erase2_singleton*[*simp*]: *wadjust_loop_check m rs* (*c*, []) ⟹ *wadjust_erase2 m rs* (*tl c*, [*hd c*])
 apply(*simp only*: *wadjust_loop_check*.*simps wadjust_erase2*.*simps*, *auto*)
 done

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*
 by(*auto simp*: *wadjust_loop_on_left_moving*.*simps*
 wadjust_loop_right_move2.*simps*
 wadjust_erase2.*simps*)

lemma *wadjust_on_left_moving_B_Bk1*[*simp*]: *wadjust_on_left_moving_B m rs*
 (*Oc* # *Oc* # *Oc*↑(*rs*) @ *Bk* # *Oc* # *Oc*↑(*m*), [*Bk*])
 apply(*simp add*: *wadjust_on_left_moving_B*.*simps*, *auto*)
 done

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*])

```
apply(simp add: wadjust_on_left_moving_B.simps , auto)
apply(rule_tac x = Suc n in exI, simp add: exp_ind del: replicate_Suc)
done

lemma wadjust_on_left_moving_singleton[simp]: ⟦wadjust_erase2 m rs (c, []); c ≠ []⟧ ⟹
      wadjust_on_left_moving m rs (tl c, [hd c]) unfolding wadjust_erase2.simps
 apply(auto simp add: wadjust_on_left_moving.simps)
 apply (metis (no_types, lifting) empty_replicate hd_append hd_replicate list.sel(1) list.sel(3)
   self_append_conv2 tl_append2 tl_replicate
   wadjust_on_left_moving_B_Bk1 wadjust_on_left_moving_B_Bk2)+
done

lemma wadjust_erase2_cases[simp]: wadjust_erase2 m rs (c, [])
  ⟹ (c = [] ⟶ wadjust_on_left_moving m rs ([], [Bk])) ∧
   (c ≠ [] ⟶ wadjust_on_left_moving m rs (tl c, [hd c]))
 apply(auto)
 done

lemma wadjust_on_left_moving_nonempty[simp]:
 wadjust_on_left_moving m rs ([], []) = False
 wadjust_on_left_moving_O m rs (c, []) = False
 apply(auto simp: wadjust_on_left_moving.simps
   wadjust_on_left_moving_O.simps wadjust_on_left_moving_B.simps)
 done

lemma wadjust_on_left_moving_B_singleton_Bk[simp]:
 ⟦wadjust_on_left_moving_B m rs (c, []); c ≠ []; hd c = Bk⟧ ⟹
                wadjust_on_left_moving_B m rs (tl c, [Bk])
 apply(auto simp add: wadjust_on_left_moving_B.simps hd_append)
 by (metis cell.distinct(1) empty_replicate list.sel(1) tl_append2 tl_replicate)

lemma wadjust_on_left_moving_B_singleton_Oc[simp]:
 ⟦wadjust_on_left_moving_B m rs (c, []); c ≠ []; hd c = Oc⟧ ⟹
               wadjust_on_left_moving_O m rs (tl c, [Oc])
 apply(auto simp add: wadjust_on_left_moving_B.simps wadjust_on_left_moving_O.simps hd_append)
 apply (metis cell.distinct(1) empty_replicate hd_replicate list.sel(3) self_append_conv2)+
 done

lemma wadjust_on_left_moving_singleton2[simp]:
 ⟦wadjust_on_left_moving m rs (c, []); c ≠ []⟧ ⟹
 wadjust_on_left_moving m rs (tl c, [hd c])
 apply(simp add: wadjust_on_left_moving.simps)
 apply(case_tac hd c, simp_all)
 done

lemma wadjust_nonempty[simp]: wadjust_goon_left_moving m rs (c, []) = False
 wadjust_backto_standard_pos m rs (c, []) = False
 by(auto simp: wadjust_goon_left_moving.simps wadjust_goon_left_moving_B.simps
   wadjust_goon_left_moving_O.simps wadjust_backto_standard_pos.simps
   wadjust_backto_standard_pos_B.simps wadjust_backto_standard_pos_O.simps)
```

lemma *wadjust_loop_start_no_Bk*[*simp*]: *wadjust_loop_start m rs* (*c*, *Bk* # *list*) = *False*
apply(*auto simp*: *wadjust_loop_start.simps*)
done

lemma *wadjust_loop_check_nonempty*[*simp*]: *wadjust_loop_check m rs* (*c*, *b*) $\Longrightarrow$ *c* $\neq$ []
apply(*simp only*: *wadjust_loop_check.simps*, *auto*)
done

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*)
by (*auto simp*: *wadjust_loop_check.simps wadjust_erase2.simps*)

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*)
unfolding *wadjust_loop_erase.simps wadjust_loop_on_left_moving_B.simps*
apply(*erule_tac exE*)+
apply(*rename_tac ml mr ln rn*)
apply(*rule_tac x* = *ml* **in** *exI*, *rule_tac x* = *mr* **in** *exI*,
rule_tac x = *ln* **in** *exI*, *rule_tac x* = *0* **in** *exI*)
apply(*case_tac ln*, *auto*)
apply(*simp add*: *exp_ind* [*THEN sym*])
done

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*)
apply(*auto simp*: *wadjust_loop_erase.simps wadjust_loop_on_left_moving_O.simps*)
by (*metis cell.distinct*(*1*) *empty_replicate hd_append hd_replicate list.sel*(*1*))

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*)
apply(*case_tac hd c*, *simp_all add*:*wadjust_loop_on_left_moving.simps*)
done

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*)
apply(*simp only*: *wadjust_loop_on_left_moving_B.simps*)
apply(*erule_tac exE*)+
by (*metis* (*no_types*, *lifting*) *cell.distinct*(*1*) *list.sel*(*1*)
replicate_Suc_iff_anywhere self_append_conv2 tl_append2 tl_replicate)

lemma *wadjust_loop_on_left_moving_O_Oc_move*[*simp*]:

$\llbracket$*wadjust_loop_on_left_moving_B m rs* (*c*, *Bk* # *list*); *hd c* = *Oc*$\rrbracket$
$\Longrightarrow$ *wadjust_loop_on_left_moving_O m rs* (*tl c*, *Oc* # *Bk* # *list*)
apply(*simp only*: *wadjust_loop_on_left_moving_O.simps*
wadjust_loop_on_left_moving_B.simps)
by (*metis cell.distinct*(*1*) *empty_replicate hd_append hd_replicate list.sel*(*3*) *self_append_conv2*)

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*
by(*auto simp*: *wadjust_loop_erase.simps wadjust_loop_on_left_moving.simps*
wadjust_loop_on_left_moving_O.simps wadjust_loop_on_left_moving_B.simps
wadjust_loop_right_move2.simps wadjust_erase2.simps
wadjust_on_left_moving.simps
wadjust_on_left_moving_O.simps
wadjust_on_left_moving_B.simps wadjust_goon_left_moving.simps
wadjust_goon_left_moving_B.simps
wadjust_goon_left_moving_O.simps)

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*)
apply(*simp add*: *wadjust_loop_on_left_moving.simps*)
apply(*case_tac hd c*, *simp_all*)
done

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*)
apply(*auto simp*: *wadjust_loop_right_move2.simps wadjust_loop_start.simps replicate_app_Cons_same*)
by (*metis add_Suc replicate_Suc*)

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*)
apply(*auto simp*: *wadjust_erase2.simps wadjust_on_left_moving.simps replicate_app_Cons_same*
wadjust_on_left_moving_O.simps wadjust_on_left_moving_B.simps)
apply (*metis exp_ind replicate_append_same*)+
done

lemma *wadjust_on_left_moving_B_Bk_drop_one*: $\llbracket$*wadjust_on_left_moving_B m rs* (*c*, *Bk* # *list*);
hd c = *Bk*$\rrbracket$
$\Longrightarrow$ *wadjust_on_left_moving_B m rs* (*tl c*, *Bk* # *Bk* # *list*)
apply(*auto simp*: *wadjust_on_left_moving_B.simps*)
by (*metis cell.distinct*(*1*) *hd_append list.sel*(*1*) *tl_append2 tl_replicate*)

```
lemma wadjust_on_left_moving_B_Bk_drop_Oc: ⟦wadjust_on_left_moving_B m rs (c, Bk # list);
hd c = Oc⟧
  ⟹ wadjust_on_left_moving_O m rs (tl c, Oc # Bk # list)
 apply(auto simp: wadjust_on_left_moving_O.simps wadjust_on_left_moving_B.simps)
 by (metis cell.distinct(1) empty_replicate hd_append hd_replicate list.sel(3) self_append_conv2)

lemma wadjust_on_left_moving_B_drop[simp]: wadjust_on_left_moving  m rs (c, Bk # list) ⟹
          wadjust_on_left_moving m rs (tl c, hd c # Bk # list)
 by(cases hd c, auto simp:wadjust_on_left_moving.simps wadjust_on_left_moving_B_Bk_drop_one
    wadjust_on_left_moving_B_Bk_drop_Oc)

lemma wadjust_goon_left_moving_O_no_Bk[simp]: wadjust_goon_left_moving_O m rs (c, Bk #
list) = False
 by (auto simp add: wadjust_goon_left_moving_O.simps)

lemma wadjust_backto_standard_pos_via_left_Bk[simp]:
 wadjust_goon_left_moving m rs (c, Bk # list) ⟹
 wadjust_backto_standard_pos m rs (tl c, hd c # Bk # list)
 by(case_tac hd c, simp_all add: wadjust_backto_standard_pos.simps wadjust_goon_left_moving.simps
    wadjust_goon_left_moving_B.simps wadjust_backto_standard_pos_O.simps)

lemma wadjust_loop_right_move_Oc[simp]:
 wadjust_loop_start m rs (c, Oc # list) ⟹ wadjust_loop_right_move m rs (Oc # c, list)
 apply(auto simp add: wadjust_loop_start.simps wadjust_loop_right_move.simps
    simp del:split_head_repeat)
 apply(rename_tac ln rn ml mr)
 apply(rule_tac x = ml in exI, rule_tac x = mr in exI,
    rule_tac x = 0 in exI, simp)
 apply(rule_tac x = Suc ln in exI, simp add: exp_ind del: replicate_Suc)
 done

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 −
 from assms obtain ml mr nl nr rn
  where c = Bk ↑ nl @ Oc # Oc ↑ ml @ Bk # Oc ↑ m @ [Oc]
     Oc # list = Bk ↑ nr @ Oc ↑ mr @ Bk ↑ rn
     ml + mr = Suc (Suc rs) 0 < mr 0 < nl + nr
  unfolding wadjust_loop_right_move.simps exp_ind
   wadjust_loop_check.simps by auto
 hence ∃ ln. Oc # c = Oc # Bk ↑ ln @ Bk # Oc # Oc ↑ ml @ Bk # Oc ↑ Suc m
    ∃ rn. list = Oc ↑ (mr − 1) @ Bk ↑ rn ml + (mr − 1) = Suc rs
  by(cases nl;cases nr;cases mr;force simp add: wadjust_loop_right_move.simps exp_ind
   wadjust_loop_check.simps replicate_append_same)+
 thus ?thesis unfolding wadjust_loop_check.simps by auto
qed

lemma wadjust_loop_erase_move_Oc[simp]: wadjust_loop_check m rs (c, Oc # list) ⟹
         wadjust_loop_erase m rs (tl c, hd c # Oc # list)
```

apply(*simp only*: *wadjust_loop_check.simps wadjust_loop_erase.simps*)
apply(*erule_tac exE*)+
using *Cons_replicate_eq* **by** *fastforce*

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*)
by(*auto simp*: *wadjust_loop_on_left_moving_B.simps wadjust_loop_on_left_moving_O.simps*
wadjust_loop_right_move2.simps replicate_app_Cons_same wadjust_loop_on_left_moving.simps
wadjust_on_left_moving_B.simps wadjust_loop_erase.simps)

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*)
apply(*auto simp*: *wadjust_on_left_moving_O.simps*
wadjust_goon_left_moving_B.simps)
done

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*)
apply(*auto simp*: *wadjust_on_left_moving_O.simps*
wadjust_goon_left_moving_O.simps)
apply(*auto simp*: *numeral_2_eq_2*)
done

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*)
by(*cases hd c*; *force simp*: *wadjust_on_left_moving.simps wadjust_goon_left_moving.simps*
wadjust_goon_left_moving_B_Bk_Oc wadjust_goon_left_moving_O_Oc_Oc)+

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*)
apply(*auto simp*: *wadjust_goon_left_moving_O.simps wadjust_goon_left_moving_B.simps hd_append*
dest!: *gr0_implies_Suc*)
apply (*metis cell.distinct*(*1*) *empty_replicate hd_replicate list.sel*(*3*) *self_append_conv2*)
by (*metis add_cancel_right_left cell.distinct*(*1*) *hd_replicate replicate_Suc_iff_anywhere*)

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*)
apply(*auto simp*: *wadjust_goon_left_moving_O.simps wadjust_goon_left_moving_B.simps*)
apply(*rename_tac mlx mrx rnx*)
apply(*rule_tac x* = *mlx* − *1* **in** *exI*, *simp*)

apply(*case_tac mlx*, *simp_all add*:)
apply(*rule_tac x* = *Suc mrx* **in** *exI*, *auto simp*:)
done

lemma *wadjust_goon_left_moving_B_no_Oc*[*simp*]:
wadjust_goon_left_moving_B m rs (*c*, *Oc* # *list*) = *False*
apply(*auto simp*: *wadjust_goon_left_moving_B*.*simps*)
done

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*)
by(*cases hd c*,*auto simp*: *wadjust_goon_left_moving*.*simps*)

lemma *wadjust_backto_standard_pos_B_no_Oc*[*simp*]:
wadjust_backto_standard_pos_B m rs (*c*, *Oc* # *list*) = *False*
apply(*simp add*: *wadjust_backto_standard_pos_B*.*simps*)
done

lemma *wadjust_backto_standard_pos_O_no_Bk*[*simp*]:
wadjust_backto_standard_pos_O m rs (*c*, *Bk* # *xs*) = *False*
by(*simp add*: *wadjust_backto_standard_pos_O*.*simps*)

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*)
apply(*auto simp*: *wadjust_backto_standard_pos_O*.*simps*
wadjust_backto_standard_pos_B.*simps*)
done

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*)
apply(*simp add*:*wadjust_backto_standard_pos_O*.*simps*
wadjust_backto_standard_pos_B.*simps*, *auto*)
done

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*)
apply(*simp add*: *wadjust_backto_standard_pos_O*.*simps*, *auto*)
by *force*

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*))
apply(*auto simp*: *wadjust_backto_standard_pos*.*simps*)
apply(*case_tac hd c*, *simp_all*)
done

lemma *wadjust_loop_right_move_nonempty_snd*[*simp*]: *wadjust_loop_right_move m rs* (*c*, []) = *False*
proof −
{**fix** *nl ml mr rn nr*
have ($c = Bk \uparrow nl$ @ Oc # $Oc \uparrow ml$ @ Bk # $Oc \uparrow Suc\ m \wedge$
[] = $Bk \uparrow nr$ @ $Oc \uparrow mr$ @ $Bk \uparrow rn \wedge ml + mr = Suc\ (Suc\ rs) \wedge 0 < mr \wedge 0 < nl + nr) =$
False **by** *auto*
} **note** *t=this*
thus *?thesis* **unfolding** *wadjust_loop_right_move.simps t* **by** *blast*
qed

lemma *wadjust_loop_erase_nonempty_snd*[*simp*]: *wadjust_loop_erase m rs* (*c*, []) = *False*
apply(*simp only*: *wadjust_loop_erase.simps*, *auto*)
done

lemma *wadjust_loop_erase_cases2*[*simp*]: ⟦*Suc* (*Suc rs*) = *a*; *wadjust_loop_erase m rs* (*c*, *Bk* # *list*)⟧
⟹ *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev* (*tl c*) @ *hd c* # *Bk* # *list*))))
< *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Bk* # *list*)))) ∨
a − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev* (*tl c*) @ *hd c* # *Bk* # *list*)))) =
a − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Bk* # *list*))))
apply(*simp only*: *wadjust_loop_erase.simps*)
apply(*rule_tac disjI2*)
apply(*case_tac c*, *simp*, *simp*)
done

lemma *dropWhile_exp1*: *dropWhile* (λ*a*. *a* = *Oc*) (*Oc*↑(*n*) @ *xs*) = *dropWhile* (λ*a*. *a* = *Oc*) *xs*
apply(*induct n*, *simp_all add*:)
done
lemma *takeWhile_exp1*: *takeWhile* (λ*a*. *a* = *Oc*) (*Oc*↑(*n*) @ *xs*) = *Oc*↑(*n*) @ *takeWhile* (λ*a*. *a* = *Oc*) *xs*
apply(*induct n*, *simp_all add*:)
done

lemma *wadjust_correctness_helper_1*:
assumes *Suc* (*Suc rs*) = *a* *wadjust_loop_right_move2 m rs* (*c*, *Bk* # *list*)
shows *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Oc* # *list*))))
< *a* − *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl* (*dropWhile* (λ*a*. *a* = *Oc*) (*rev c* @ *Bk* # *list*))))
proof −
have $ml + mr = Suc\ rs \Longrightarrow 0 < mr \Longrightarrow$
rs − (*ml* + *length* (*takeWhile* (λ*a*. *a* = *Oc*) *list*))
< *Suc rs* −
(*ml* +
length
(*takeWhile* (λ*a*. *a* = *Oc*)
($Bk \uparrow ln$ @ Bk # Bk # $Oc \uparrow mr$ @ $Bk \uparrow rn$)))

for *ml mr ln rn*
by(*cases ln*, *auto*)
thus *?thesis* **using** *assms*
by (*auto simp*: *wadjust_loop_right_move2.simps dropWhile_exp1 takeWhile_exp1*)
qed

lemma *wadjust_correctness_helper_2*:
$[\![Suc\ (Suc\ rs) = a;\ \ wadjust\_loop\_on\_left\_moving\ m\ rs\ (c, Bk\ \#\ list)]\!]$
$\Longrightarrow a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ (tl\ c)\ @\ hd\ c\ \#\ Bk\ \#\ list))))$
$< a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Bk\ \#\ list)))) \vee$
$a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ (tl\ c)\ @\ hd\ c\ \#\ Bk\ \#\ list)))) =$
$a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Bk\ \#\ list))))$
apply(*subgoal_tac* $c \neq []$)
apply(*case_tac c*, *simp_all*)
done

lemma *wadjust_loop_check_empty_false*[*simp*]: $wadjust\_loop\_check\ m\ rs\ ([], b) = False$
apply(*simp add*: *wadjust_loop_check.simps*)
done

lemma *wadjust_loop_check_cases*: $[\![Suc\ (Suc\ rs) = a;\ \ wadjust\_loop\_check\ m\ rs\ (c, Oc\ \#\ list)]\!]$
$\Longrightarrow a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ (tl\ c)\ @\ hd\ c\ \#\ Oc\ \#\ list))))$
$< a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Oc\ \#\ list)))) \vee$
$a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ (tl\ c)\ @\ hd\ c\ \#\ Oc\ \#\ list)))) =$
$a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Oc\ \#\ list))))$
apply(*case_tac c*, *simp_all*)
done

lemma *wadjust_loop_erase_cases_or*:
$[\![Suc\ (Suc\ rs) = a;\ \ wadjust\_loop\_erase\ m\ rs\ (c, Oc\ \#\ list)]\!]$
$\Longrightarrow a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Bk\ \#\ list))))$
$< a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Oc\ \#\ list)))) \vee$
$a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Bk\ \#\ list)))) =$
$a - length\ (takeWhile\ (\lambda a.\ a = Oc)\ (tl\ (dropWhile\ (\lambda a.\ a = Oc)\ (rev\ c\ @\ Oc\ \#\ list))))$
apply(*simp add*: *wadjust_loop_erase.simps*)
apply(*rule_tac disjI2*)
apply(*auto*)
apply(*simp add*: *dropWhile_exp1 takeWhile_exp1*)
done

lemmas *wadjust_correctness_helpers* = *wadjust_correctness_helper_2 wadjust_correctness_helper_1 wadjust_loop_erase_cases_or wadjust_loop_check_cases*

declare *numeral_2_eq_2*[*simp del*]

lemma *wadjust_start_Oc*[*simp*]: $wadjust\_start\ m\ rs\ (c, Bk\ \#\ list)$

```
    ⟹ wadjust_start m rs (c, Oc # list)
  apply(auto simp: wadjust_start.simps)
  done

lemma wadjust_stop_Bk[simp]: wadjust_backto_standard_pos m rs (c, Bk # list)
    ⟹ wadjust_stop m rs (Bk # c, list)
  apply(auto simp: wadjust_backto_standard_pos.simps
    wadjust_stop.simps wadjust_backto_standard_pos_B.simps)
  done

lemma wadjust_loop_start_Oc[simp]:
  assumes wadjust_start m rs (c, Oc # list)
  shows wadjust_loop_start m rs (Oc # c, list)
proof -
  from assms[unfolded wadjust_start.simps] obtain ln rn where
   c = Bk # Oc # Oc ↑ m list = Oc # Bk ↑ ln @ Bk # Oc # Oc ↑ rs @ Bk ↑ rn
   by(auto)
  hence Oc # c = Oc ↑ 1 @ Bk # Oc ↑ Suc m ∧
      list = Oc # Bk ↑ ln @ Bk # Oc ↑Suc rs @ Bk ↑ rn ∧ 1 + (Suc rs) = Suc (Suc rs) ∧ 0 <
Suc rs
   by auto
  thus ?thesis unfolding wadjust_loop_start.simps by blast
qed

lemma erase2_Bk_if_Oc[simp]: wadjust_erase2 m rs (c, Oc # list)
    ⟹ wadjust_erase2 m rs (c, Bk # list)
  apply(auto simp: wadjust_erase2.simps)
  done

lemma wadjust_loop_right_move_Bk[simp]: wadjust_loop_right_move m rs (c, Bk # list)
  ⟹ wadjust_loop_right_move m rs (Bk # c, list)
  apply(simp only: wadjust_loop_right_move.simps)
  apply(erule_tac exE)+
  apply auto
   apply (metis cell.distinct(1) empty_replicate hd_append hd_replicate less_SucI
    list.sel(1) list.sel(3) neq0_conv replicate_Suc_iff_anywhere tl_append2 tl_replicate)+
  done

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)) t_wcode_adjust stp)) in
   ∃ n .P (f n) ∧ Q (f n)
proof -
  let ?P = (λ (len, st, l, r). st = 0)
  let ?Q = λ (len, st, l, r). wadjust_inv st m rs (l, r)
  let ?f = λ stp. (Suc (Suc rs),  steps0 (Suc 0, Bk # Oc↑(Suc m),
            Bk # Oc # Bk↑(ln) @ Bk # Oc↑(Suc rs) @ Bk↑(rn)) t_wcode_adjust stp)
  have ∃ n. ?P (?f n) ∧ ?Q (?f n)
```

```
proof(rule_tac halt_lemma2)
  show wf wadjust_le by auto
next
  { fix n assume a:¬ ?P (?f n) ∧ ?Q (?f n)
    have ?Q (?f (Suc n)) ∧ (?f (Suc n), ?f n) ∈ wadjust_le
    proof(cases ?f n)
      case (fields a b c d)
      then show ?thesis proof(cases d)
        case Nil
        then show ?thesis using a fields apply(simp add: step.simps)
          apply(simp_all only: wadjust_inv.simps split: if_splits)
                  apply(simp_all add: wadjust_inv.simps wadjust_le_def
            wadjust_correctness_helpers
            Abacus.lex_triple_def Abacus.lex_pair_def lex_square_def  split: if_splits).
      next
        case (Cons aa list)
        then show ?thesis using a fields Nil Cons
          apply((case_tac aa); simp add: step.simps)
           apply(simp_all only: wadjust_inv.simps split: if_splits)
                         apply(simp_all)
                        apply(simp_all add: wadjust_inv.simps wadjust_le_def
            wadjust_correctness_helpers
            Abacus.lex_triple_def Abacus.lex_pair_def lex_square_def  split: if_splits).
      qed
    qed
  }
  thus ∀ n. ¬ ?P (?f n) ∧ ?Q (?f n) ⟶
             ?Q (?f (Suc n)) ∧ (?f (Suc n), ?f n) ∈ wadjust_le by auto
next
  show ?Q (?f 0) by(auto simp add: steps.simps wadjust_inv.simps wadjust_start.simps)
next
  show ¬ ?P (?f 0) by (simp add: steps.simps)
qed
thus?thesis by simp
qed

lemma tm_wf_t_wcode_adjust[intro]: tm_wf (t_wcode_adjust, 0)
  by(auto simp: t_wcode_adjust_def tm_wf.simps)

lemma bl_bin_nonzero[simp]: args ≠ [] ⟹ bl_bin (<args::nat list>) > 0
  by(cases args)
    (auto simp: tape_of_nl_cons bl_bin.simps)

lemma wcode_lemma_pre':
  args ≠ [] ⟹
  ∃ stp rn. steps0 (Suc 0, [], <m # args>)
            ((t_wcode_prepare |+| t_wcode_main) |+| t_wcode_adjust) stp
  = (0,  [Bk],  Oc↑(Suc m) @ Bk # Oc↑(Suc (bl_bin (<args>))) @ Bk↑(rn))
proof -
  let ?P1 = λ (l, r). l = [] ∧ r = <m # args>
```

```
let ?Q1 = λ(l, r). l = Bk # Oc↑(Suc m) ∧
  (∃ ln rn. r = Bk # Oc # Bk↑(ln) @ Bk # Bk # Oc↑(bl_bin (<args>)) @ Bk↑(rn))
let ?P2 = ?Q1
let ?Q2 = λ (l, r). (wadjust_stop m (bl_bin (<args>) − 1) (l, r))
let ?P3 = λ tp. False
assume h: args ≠ []
hence a: bl_bin (<args>) > 0
  using h by simp
hence {?P1} (t_wcode_prepare |+| t_wcode_main) |+| t_wcode_adjust {?Q2}
proof(rule_tac Hoare_plus_halt)
next
  show tm_wf (t_wcode_prepare |+| t_wcode_main, 0)
    by(rule_tac tm_comp_wf, auto)
next
  show {?P1} t_wcode_prepare |+| t_wcode_main {?Q1}
  proof(rule_tac Hoare_haltI, auto)
    show
      ∃ n. is_final (steps0 (Suc 0, [], <m # args>) (t_wcode_prepare |+| t_wcode_main) n) ∧
      (λ(l, r). l = Bk # Oc # Oc ↑ m ∧
      (∃ ln rn. r = Bk # Oc # Bk ↑ ln @ Bk # Bk # Oc ↑ bl_bin (<args>) @ Bk ↑ rn))
      holds_for steps0 (Suc 0, [], <m # args>) (t_wcode_prepare |+| t_wcode_main) n
      using h prepare_mainpart_lemma[of args m]
      apply(auto) apply(rename_tac stp ln rn)
      apply(rule_tac x = stp in exI, simp)
      apply(rule_tac x = ln in exI, auto)
      done
  qed
next
  show {?P2} t_wcode_adjust {?Q2}
  proof(rule_tac Hoare_haltI, auto del: replicate_Suc)
    fix ln rn
    obtain n a b where steps0
      (Suc 0, Bk # Oc ↑ m @ [Oc],
       Bk # Oc # Bk ↑ ln @ Bk # Bk # Oc ↑ (bl_bin (<args>) − Suc 0) @ Oc # Bk ↑ rn)
      t_wcode_adjust n = (0, a, b)
      wadjust_inv 0 m (bl_bin (<args>) − Suc 0) (a, b)
      using wadjust_correctness[of m bl_bin (<args>) − 1 Suc ln rn,unfolded Let_def]
      by(simp del: replicate_Suc add: replicate_Suc[THEN sym] exp_ind, auto)
    thus ∃ n. is_final (steps0 (Suc 0, Bk # Oc # Oc ↑ m,
      Bk # Oc # Bk ↑ ln @ Bk # Bk # Oc ↑ bl_bin (<args>) @ Bk ↑ rn) t_wcode_adjust n) ∧
      wadjust_stop m (bl_bin (<args>) − Suc 0) holds_for steps0
       (Suc 0, Bk # Oc # Oc ↑ m, Bk # Oc # Bk ↑ ln @ Bk # Bk # Oc ↑ bl_bin (<args>) @
Bk ↑ rn) t_wcode_adjust n
      apply(rule_tac x = n in exI)
      using a
    apply(case_tac bl_bin (<args>), simp, simp del: replicate_Suc add: exp_ind wadjust_inv.simps)
      by (simp add: replicate_append_same)
  qed
qed
thus ?thesis
```

apply(*simp add*: *Hoare_halt_def*, *auto*)
apply(*rename_tac n*)
apply(*case_tac* (*steps0* (*Suc 0*, [], <(*m*::*nat*) # *args*>)
((*t_wcode_prepare* |+| *t_wcode_main*) |+| *t_wcode_adjust*) *n*))
apply(*rule_tac x* = *n* **in** *exI*, *auto simp*: *wadjust_stop.simps*)
using *a*
apply(*case_tac bl_bin* (<*args*>), *simp_all*)
done
qed

The initialization TM *t_wcode*.

definition *t_wcode* :: *instr list*
where
t_wcode = (*t_wcode_prepare* |+| *t_wcode_main*) |+| *t_wcode_adjust*

The correctness of *t_wcode*.

lemma *wcode_lemma_1*:
args ≠ [] ⟹
∃ *stp ln rn*. *steps0* (*Suc 0*, [], <*m* # *args*>) (*t_wcode*) *stp* =
(*0*, [*Bk*], *Oc*↑(*Suc m*) @ *Bk* # *Oc*↑(*Suc* (*bl_bin* (<*args*>))) @ *Bk*↑(*rn*))
apply(*simp add*: *wcode_lemma_pre′ t_wcode_def del*: *replicate_Suc*)
done

lemma *wcode_lemma*:
args ≠ [] ⟹
∃ *stp ln rn*. *steps0* (*Suc 0*, [], <*m* # *args*>) (*t_wcode*) *stp* =
(*0*, [*Bk*], <[*m* ,*bl_bin* (<*args*>)]> @ *Bk*↑(*rn*))
using *wcode_lemma_1*[*of args m*]
apply(*simp add*: *t_wcode_def tape_of_list_def tape_of_nat_def*)
done

28 The universal TM

This section gives the explicit construction of *Universal Turing Machine*, defined as *UTM* and proves its correctness. It is pretty easy by composing the partial results we have got so far.

definition *UTM* :: *instr list*
where
UTM = (*let* (*aprog*, *rs_pos*, *a_md*) = *rec_ci rec_F in*
let abc_F = *aprog* [+] *dummy_abc* (*Suc* (*Suc 0*)) *in*
(*t_wcode* |+| (*tm_of abc_F* @ *shift* (*mopup* (*Suc* (*Suc 0*))) (*length* (*tm_of abc_F*) *div 2*))))

definition *F_aprog* :: *abc_prog*
where
F_aprog $\stackrel{def}{=}$ (*let* (*aprog*, *rs_pos*, *a_md*) = *rec_ci rec_F in*
aprog [+] *dummy_abc* (*Suc* (*Suc 0*)))

definition *F_tprog* :: *instr list*

where
  F_tprog = *tm_of* (*F_aprog*)

definition *t_utm* :: *instr list*
  where
  t_utm $\stackrel{def}{=}$
   F_tprog @ *shift* (*mopup* (*Suc* (*Suc 0*))) (*length F_tprog div 2*)

definition *UTM_pre* :: *instr list*
  where
  UTM_pre = *t_wcode* |+| *t_utm*

lemma *tinres_step1*:
  assumes *tinres l l′ step* (*ss*, *l*, *r*) (*t*, *0*) = (*sa*, *la*, *ra*)
  step (*ss*, *l′*, *r*) (*t*, *0*) = (*sb*, *lb*, *rb*)
  shows *tinres la lb* ∧ *ra* = *rb* ∧ *sa* = *sb*
proof(*cases r*)
  case *Nil*
  then show *?thesis* **using** *assms*
    by (*cases* (*fetch t ss Bk*);*cases fst* (*fetch t ss Bk*);*auto simp*:*step*.*simps split*:*if_splits*)
next
  case (*Cons a list*)
  then show *?thesis* **using** *assms*
    by (*cases* (*fetch t ss a*);*cases fst* (*fetch t ss a*);*auto simp*:*step*.*simps split*:*if_splits*)
qed

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 (*induct stp arbitrary*: *sa la ra sb lb rb*)
  case (*Suc stp*)
  then show *?case* **apply** *simp*
    apply(*case_tac* (*steps* (*ss*, *l*, *r*) (*t*, *0*) *stp*))
    apply(*case_tac* (*steps* (*ss*, *l′*, *r*) (*t*, *0*) *stp*))
  proof −
    fix *stp sa la ra sb lb rb a b c aa ba ca*
    assume *ind*: ⋀*sa la ra sb lb rb*. ⟦*steps* (*ss*, *l*, *r*) (*t*, *0*) *stp* = (*sa*, (*la*::*cell list*), *ra*);
        steps (*ss*, *l′*, *r*) (*t*, *0*) *stp* = (*sb*, *lb*, *rb*)⟧ ⟹ *tinres la lb* ∧ *ra* = *rb* ∧ *sa* = *sb*
      and *h*: *tinres l l′ step* (*steps* (*ss*, *l*, *r*) (*t*, *0*) *stp*) (*t*, *0*) = (*sa*, *la*, *ra*)
      step (*steps* (*ss*, *l′*, *r*) (*t*, *0*) *stp*) (*t*, *0*) = (*sb*, *lb*, *rb*) *steps* (*ss*, *l*, *r*) (*t*, *0*) *stp* = (*a*, *b*, *c*)
      steps (*ss*, *l′*, *r*) (*t*, *0*) *stp* = (*aa*, *ba*, *ca*)
    have *tinres b ba* ∧ *c* = *ca* ∧ *a* = *aa*
      using *ind h* **by** *metis*
    thus *tinres la lb* ∧ *ra* = *rb* ∧ *sa* = *sb*
      using *tinres_step1 h* **by** *metis*
  qed
qed (*simp add*: *steps*.*simps*)

```
lemma tinres_some_exp[simp]:
  tinres (Bk ↑ m @ [Bk, Bk]) la ⟹ ∃m. la = Bk ↑ m unfolding tinres_def
proof −
  let ?c1 = λ n. Bk ↑ m @ [Bk, Bk] = la @ Bk ↑ n
  let ?c2 = λ n. la = (Bk ↑ m @ [Bk, Bk]) @ Bk ↑ n
  assume ∃n. ?c1 n ∨ ?c2 n
  then obtain n where ?c1 n ∨ ?c2 n by auto
  then consider ?c1 n | ?c2 n by blast
  thus ?thesis proof(cases)
    case 1
    hence Bk ↑ Suc (Suc m) = la @ Bk ↑ n
      by (metis exp_ind append_Cons append_eq_append_conv2 self_append_conv2)
    hence la = Bk ↑ (Suc (Suc m) − n)
     by (metis replicate_add append_eq_append_conv diff_add_inverse2 length_append length_replicate)
    then show ?thesis by auto
  next
    case 2
    hence la = Bk ↑ (m + Suc (Suc n))
     by (metis append_Cons append_eq_append_conv2 replicate_Suc replicate_add self_append_conv2)
    then show ?thesis by blast
  qed
qed

lemma t_utm_halt_eq:
  assumes tm_wf: tm_wf (tp, 0)
    and exec: steps0 (Suc 0, Bk↑(l), <lm::nat list>) tp stp = (0, Bk↑(m), Oc↑(rs)@Bk↑(n))
    and resutl: 0 < rs
  shows ∃stp m n. steps0 (Suc 0, [Bk], <[code tp, bl2wc (<lm>)]> @ Bk↑(i)) t_utm stp =
                                     (0, Bk↑(m), Oc↑(rs) @ Bk↑(n))
proof −
  obtain ap arity fp where a: rec_ci rec_F = (ap, arity, fp)
    by (metis prod_cases3)
  moreover have b: rec_exec rec_F [code tp, (bl2wc (<lm>))] = (rs − Suc 0)
    using assms
    apply(rule_tac F_correct, simp_all)
    done
  have ∃ stp m l. steps0 (Suc 0, Bk # Bk # [], <[code tp, bl2wc (<lm>)]> @ Bk↑i)
    (F_tprog @ shift (mopup (length [code tp, bl2wc (<lm>)])) (length F_tprog div 2)) stp
    = (0, Bk↑m @ Bk # Bk # [], Oc↑Suc (rec_exec rec_F [code tp, (bl2wc (<lm>))]) @ Bk↑l)
  proof(rule_tac recursive_compile_to_tm_correct1)
    show rec_ci rec_F = (ap, arity, fp) using a by simp
  next
    show terminate rec_F [code tp, bl2wc (<lm>)]
      using assms
      by(rule_tac terminate_F, simp_all)
  next
    show F_tprog = tm_of (ap [+] dummy_abc (length [code tp, bl2wc (<lm>)]))
      using a
      apply(simp add: F_tprog_def F_aprog_def numeral_2_eq_2)
      done
```

```
 qed
 then obtain stp m l where
  steps0 (Suc 0, Bk # Bk # [], <[code tp, bl2wc (<lm>)]> @ Bk↑i)
  (F_tprog @ shift (mopup (length [code tp, (bl2wc (<lm>))])) (length F_tprog div 2)) stp
  = (0, Bk↑m @ Bk # Bk # [], Oc↑Suc (rec_exec rec_F [code tp, (bl2wc (<lm>))]) @ Bk↑l)
by blast
 hence ∃ m. steps0 (Suc 0, [Bk], <[code tp, bl2wc (<lm>)]> @ Bk↑i)
  (F_tprog @ shift (mopup 2) (length F_tprog div 2)) stp
  = (0, Bk↑m, Oc↑Suc (rs − 1) @ Bk↑l)
 proof −
  assume g: steps0 (Suc 0, [Bk, Bk], <[code tp, bl2wc (<lm>)]> @ Bk ↑ i)
   (F_tprog @ shift (mopup (length [code tp, bl2wc (<lm>)])) (length F_tprog div 2)) stp =
   (0, Bk ↑ m @ [Bk, Bk], Oc ↑ Suc ((rec_exec rec_F [code tp, bl2wc (<lm>)])) @ Bk ↑ l)
  moreover have tinres [Bk, Bk] [Bk]
   apply(auto simp: tinres_def)
   done
  moreover obtain sa la ra where steps0 (Suc 0, [Bk], <[code tp, bl2wc (<lm>)]> @ Bk↑i)
  (F_tprog @ shift (mopup 2) (length F_tprog div 2)) stp = (sa, la, ra)
   apply(case_tac steps0 (Suc 0, [Bk], <[code tp, bl2wc (<lm>)]> @ Bk↑i)
  (F_tprog @ shift (mopup 2) (length F_tprog div 2)) stp, auto)
   done
  ultimately show ?thesis
   using b
   apply(drule_tac la = Bk↑m @ [Bk, Bk] in tinres_steps1, auto simp: numeral_2_eq_2)
   done
 qed
 thus ?thesis
  apply(auto)
  apply(rule_tac x = stp in exI, simp add: t_utm_def)
  using assms
  apply(case_tac rs, simp_all add: numeral_2_eq_2)
  done
qed

lemma tm_wf_t_wcode[intro]: tm_wf (t_wcode, 0)
 apply(simp add: t_wcode_def)
 apply(rule_tac tm_comp_wf)
  apply(rule_tac tm_comp_wf, auto)
 done

lemma UTM_halt_lemma_pre:
 assumes wf_tm: tm_wf (tp, 0)
  and result: 0 < rs
  and args: args ≠ []
  and exec: steps0 (Suc 0, Bk↑(i), <args::nat list>) tp stp = (0, Bk↑(m), Oc↑(rs)@Bk↑(k))
 shows ∃ stp m n. steps0 (Suc 0, [], <code tp # args>) UTM_pre stp =
                          (0, Bk↑(m), Oc↑(rs) @ Bk↑(n))
proof −
 let ?Q2 = λ (l, r). (∃ ln rn. l = Bk↑(ln) ∧ r = Oc↑(rs) @ Bk↑(rn))
 let ?P1 = λ (l, r). l = [] ∧ r = <code tp # args>
```

```
 let ?Q1 = λ (l, r). (l = [Bk] ∧
   (∃ rn. r = Oc↑(Suc (code tp)) @ Bk # Oc↑(Suc (bl_bin (<args>))) @ Bk↑(rn)))
 let ?P2 = ?Q1
 let ?P3 = λ (l, r). False
 have {?P1} (t_wcode |+| t_utm) {?Q2}
 proof(rule_tac Hoare_plus_halt)
  show tm_wf (t_wcode, 0) by auto
 next
  show {?P1} t_wcode {?Q1}
   apply(rule_tac Hoare_haltI, auto)
   using wcode_lemma_1[of args code tp] args
   apply(auto)
   by (metis (mono_tags, lifting) holds_for.simps is_finalI old.prod.case)
 next
  show {?P2} t_utm {?Q2}
  proof(rule_tac Hoare_haltI, auto)
   fix rn
    show ∃n. is_final (steps0 (Suc 0, [Bk], Oc # Oc ↑ code tp @ Bk # Oc # Oc ↑ bl_bin
(<args>) @ Bk ↑ rn) t_utm n) ∧
    (λ(l, r). (∃ln. l = Bk ↑ ln) ∧
    (∃rn. r = Oc ↑ rs @ Bk ↑ rn)) holds_for steps0 (Suc 0, [Bk],
    Oc # Oc ↑ code tp @ Bk # Oc # Oc ↑ bl_bin (<args>) @ Bk ↑ rn) t_utm n
    using t_utm_halt_eq[of tp i args stp m rs k rn] assms
    apply(auto simp: bin_wc_eq tape_of_list_def tape_of_nat_def)
    apply(rename_tac stpa) apply(rule_tac x = stpa in exI, simp)
    done
  qed
 qed
 thus ?thesis
  apply(auto simp: Hoare_halt_def UTM_pre_def)
  apply(case_tac steps0 (Suc 0, [], <code tp # args>) (t_wcode |+| t_utm) n,simp)
  by auto
qed
```

The correctness of *UTM*, the halt case.

```
lemma UTM_halt_lemma':
 assumes tm_wf: tm_wf (tp, 0)
  and result: 0 < rs
  and args: args ≠ []
  and exec: steps0 (Suc 0, Bk↑(i), <args::nat list>) tp stp = (0, Bk↑(m), Oc↑(rs)@Bk↑(k))
 shows ∃stp m n. steps0 (Suc 0, [], <code tp # args>) UTM stp =
                             (0, Bk↑(m), Oc↑(rs) @ Bk↑(n))
 using UTM_halt_lemma_pre[of tp rs args i stp m k] assms
 apply(simp add: UTM_pre_def t_utm_def UTM_def F_aprog_def F_tprog_def)
 apply(case_tac rec_ci rec_F, simp)
 done

definition TSTD:: config ⇒ bool
 where
  TSTD c = (let (st, l, r) = c in
```

```
    st = 0 ∧ (∃ m. l = Bk↑(m)) ∧ (∃ rs n. r = Oc↑(Suc rs) @ Bk↑(n)))

lemma nstd_case1: 0 < a ⟹ NSTD (trpl_code (a, b, c))
 by(simp add: NSTD.simps trpl_code.simps)

lemma nonzero_bl2wc[simp]: ∀ m. b ≠ Bk↑(m) ⟹ 0 < bl2wc b
proof −
 have ∀ m. b ≠ Bk ↑ m ⟹ bl2wc b = 0 ⟹ False proof(induct b)
  case (Cons a b)
  then show ?case
   apply(simp add: bl2wc.simps, case_tac a, simp_all
     add: bl2nat.simps bl2nat_double)
   apply(case_tac ∃ m. b = Bk↑(m), erule exE)
    apply(metis append_Nil2 replicate_Suc_iff_anywhere)
   by simp
 qed auto
 thus ∀ m. b ≠ Bk↑(m) ⟹ 0 < bl2wc b by auto
qed

lemma nstd_case2: ∀ m. b ≠ Bk↑(m) ⟹ NSTD (trpl_code (a, b, c))
 apply(simp add: NSTD.simps trpl_code.simps)
 done

lemma even_not_odd[elim]: Suc (2 ∗ x) = 2 ∗ y ⟹ RR
proof(induct x arbitrary: y)
 case (Suc x) thus ?case by(cases y;auto)
qed auto

declare replicate_Suc[simp del]

lemma bl2nat_zero_eq[simp]: (bl2nat c 0 = 0) = (∃ n. c = Bk↑(n))
proof(induct c)
 case (Cons a c)
 then show ?case by (cases a;auto simp: bl2nat.simps bl2nat_double Cons_replicate_eq)
qed (auto simp: bl2nat.simps)

lemma bl2wc_exp_ex:
 ⟦Suc (bl2wc c) = 2 ^ m⟧ ⟹ ∃ rs n. c = Oc↑(rs) @ Bk↑(n)
proof(induct c arbitrary: m)
 case (Cons a c m)
 { fix n
  have Bk # Bk ↑ n = Oc ↑ 0 @ Bk ↑ Suc n by (auto simp:replicate_Suc)
  hence ∃ rs na. Bk # Bk ↑ n = Oc ↑ rs @ Bk ↑ na by blast
 }
 with Cons show ?case apply(cases a, auto)
   apply(case_tac m, simp_all add: bl2wc.simps, auto)
  apply(simp add: bl2wc.simps bl2nat.simps bl2nat_double Cons)
  apply(case_tac m, simp, simp add: bin_wc_eq bl2wc.simps twice_power )
  by (metis Cons.hyps Suc_pred bl2wc.simps neq0_conv power_not_zero
    replicate_Suc_iff_anywhere zero_neq_numeral)
```

```
qed (simp add: bl2wc.simps bl2nat.simps)

lemma lg_bin:
 assumes ∀ rs n. c ≠ Oc↑(Suc rs) @ Bk↑(n)
  bl2wc c = 2 ^ lg (Suc (bl2wc c)) 2 − Suc 0
 shows bl2wc c = 0
proof −
 from assms obtain rs nat n where ∗:2 ^ rs − Suc 0 = nat
  c = Oc ↑ rs @ Bk ↑ n
  using bl2wc_exp_ex[of c lg (Suc (bl2wc c)) 2]
  by(case_tac (2::nat) ^ lg (Suc (bl2wc c)) 2,
    simp, simp, erule_tac exE, erule_tac exE, simp)
 have r:bl2wc (Oc ↑ rs) = nat
  by (metis ∗(1) bl2nat_exp_zero bl2wc.elims)
 hence Suc (bl2wc c) = 2^rs using ∗
  by(case_tac (2::nat)^rs, auto)
 thus ?thesis using ∗ assms(1)
  apply(drule_tac bl2wc_exp_ex, simp, erule_tac exE, erule_tac exE)
  by(case_tac rs, simp, simp)
qed

lemma nstd_case3:
 ∀ rs n. c ≠ Oc↑(Suc rs) @ Bk↑(n) ⟹ NSTD (trpl_code (a, b, c))
 apply(simp add: NSTD.simps trpl_code.simps)
 apply(auto)
 apply(drule_tac lg_bin, simp_all)
 done

lemma NSTD_1: ¬ TSTD (a, b, c)
   ⟹ rec_exec rec_NSTD [trpl_code (a, b, c)] = Suc 0
 using NSTD_lemma1[of trpl_code (a, b, c)]
  NSTD_lemma2[of trpl_code (a, b, c)]
 apply(simp add: TSTD_def)
 apply(erule_tac disjE, erule_tac nstd_case1)
 apply(erule_tac disjE, erule_tac nstd_case2)
 apply(erule_tac nstd_case3)
 done

lemma nonstop_t_uhalt_eq:
 ⟦tm_wf (tp, 0);
 steps0 (Suc 0, Bk↑(l), <lm>) tp stp = (a, b, c);
 ¬ TSTD (a, b, c)⟧
 ⟹ rec_exec rec_nonstop [code tp, bl2wc (<lm>), stp] = Suc 0
 apply(simp add: rec_nonstop_def rec_exec.simps)
 apply(subgoal_tac
    rec_exec rec_conf [code tp, bl2wc (<lm>), stp] =
 trpl_code (a, b, c), simp)
  apply(erule_tac NSTD_1)
 using rec_t_eq_steps[of tp l lm stp]
 apply(simp)
```

```
 done

lemma nonstop_true:
 ⟦tm_wf (tp, 0);
 ∀ stp. (¬ TSTD (steps0 (Suc 0, Bk↑(l), <lm>) tp stp))⟧
 ⟹ ∀ y. rec_exec rec_nonstop ([code tp, bl2wc (<lm>), y]) = (Suc 0)
proof fix y
 assume a:tm_wf0 tp ∀ stp. ¬ TSTD (steps0 (Suc 0, Bk ↑ l, <lm>) tp stp)
 hence ¬ TSTD (steps0 (Suc 0, Bk ↑ l, <lm>) tp y) by auto
 thus rec_exec rec_nonstop [code tp, bl2wc (<lm>), y] = Suc 0
  by (cases steps0 (Suc 0, Bk↑(l), <lm>) tp y)
   (auto intro: nonstop_t_uhalt_eq[OF a(1)])
qed

lemma cn_arity:  rec_ci (Cn n f gs) = (a, b, c) ⟹ b = n
 by(case_tac rec_ci f, simp add: rec_ci.simps)

lemma mn_arity: rec_ci (Mn n f) = (a, b, c) ⟹ b = n
 by(case_tac rec_ci f, simp add: rec_ci.simps)

lemma F_aprog_uhalt:
 assumes wf_tm: tm_wf (tp,0)
  and unhalt:  ∀ stp. (¬ TSTD (steps0 (Suc 0, Bk↑(l), <lm>) tp stp))
  and compile: rec_ci rec_F = (F_ap, rs_pos, a_md)
 shows {λ nl. nl = [code tp, bl2wc (<lm>)] @ 0↑(a_md − rs_pos ) @ suflm} (F_ap) ↑
 using compile
proof(simp only: rec_F_def)
 assume h: rec_ci (Cn (Suc (Suc 0)) rec_valu [Cn (Suc (Suc 0)) rec_right [Cn (Suc (Suc 0))
  rec_conf [recf.id (Suc (Suc 0)) 0, recf.id (Suc (Suc 0)) (Suc 0), rec_halt]]]) =
  (F_ap, rs_pos, a_md)
 moreover hence rs_pos = Suc (Suc 0)
  using cn_arity
  by simp
 moreover obtain ap1 ar1 ft1 where a: rec_ci
  (Cn (Suc (Suc 0)) rec_right
  [Cn (Suc (Suc 0)) rec_conf [recf.id (Suc (Suc 0)) 0, recf.id (Suc (Suc 0)) (Suc 0), rec_halt]])
= (ap1, ar1, ft1)
  by(case_tac rec_ci (Cn (Suc (Suc 0)) rec_right [Cn (Suc (Suc 0))
   rec_conf [recf.id (Suc (Suc 0)) 0, recf.id (Suc (Suc 0)) (Suc 0), rec_halt]]), auto)
 moreover hence b: ar1 = Suc (Suc 0)
  using cn_arity by simp
 ultimately show ?thesis
 proof(rule_tac i = 0 in cn_unhalt_case, auto)
  fix anything
  obtain ap2 ar2 ft2 where c:
    rec_ci (Cn (Suc (Suc 0)) rec_conf [recf.id (Suc (Suc 0)) 0, recf.id (Suc (Suc 0)) (Suc 0),
rec_halt])
    = (ap2, ar2, ft2)
   by(case_tac rec_ci (Cn (Suc (Suc 0)) rec_conf
    [recf.id (Suc (Suc 0)) 0, recf.id (Suc (Suc 0)) (Suc 0), rec_halt]), auto)
```

```
  moreover hence d:ar2 = Suc (Suc 0)
    using cn_arity by simp
  ultimately have {λnl. nl = [code tp, bl2wc (<lm>)] @ 0 ↑ (ft1 − Suc (Suc 0)) @ anything}
ap1 ↑
    using a b c d
  proof(rule_tac i = 0 in cn_unhalt_case, auto)
    fix anything
    obtain ap3 ar3 ft3 where e: rec_ci rec_halt = (ap3, ar3, ft3)
      by(case_tac rec_ci rec_halt, auto)
    hence f: ar3 = Suc (Suc 0)
      using mn_arity
      by(simp add: rec_halt_def)
    have {λnl. nl = [code tp, bl2wc (<lm>)] @ 0 ↑ (ft2 − Suc (Suc 0)) @ anything} ap2 ↑
      using c d e f
    proof(rule_tac i = 2 in cn_unhalt_case, auto simp: rec_halt_def)
      fix anything
      have {λnl. nl = [code tp, bl2wc (<lm>)] @ 0 ↑ (ft3 − Suc (Suc 0)) @ anything} ap3 ↑
        using e f
      proof(rule_tac mn_unhalt_case, auto simp: rec_halt_def)
        fix i
        show terminate rec_nonstop [code tp, bl2wc (<lm>), i]
          by(rule_tac primerec_terminate, auto)
      next
        fix i
        show 0 < rec_exec rec_nonstop [code tp, bl2wc (<lm>), i]
          using assms
          by(drule_tac nonstop_true, auto)
      qed
      thus {λnl. nl = code tp # bl2wc (<lm>) # 0 ↑ (ft3 − Suc (Suc 0)) @ anything} ap3 ↑ by
simp
    next
      fix apj arj ftj j  anything
      assume j<2 rec_ci ([recf.id (Suc (Suc 0)) 0, recf.id (Suc (Suc 0)) (Suc 0), Mn (Suc (Suc
0)) rec_nonstop] ! j) = (apj, arj, ftj)
      hence {λnl. nl = [code tp, bl2wc (<lm>)] @ 0 ↑ (ftj − arj) @ anything} apj
        {λnl. nl = [code tp, bl2wc (<lm>)] @
          rec_exec ([recf.id (Suc (Suc 0)) 0, recf.id (Suc (Suc 0)) (Suc 0), Mn (Suc (Suc 0))
rec_nonstop] ! j) [code tp, bl2wc (<lm>)] #
          0 ↑ (ftj − Suc arj) @ anything}
        apply(rule_tac recursive_compile_correct)
         apply(case_tac j, auto)
         apply(rule_tac [!] primerec_terminate)
        by(auto)
      thus {λnl. nl = code tp # bl2wc (<lm>) # 0 ↑ (ftj − arj) @ anything} apj
        {λnl. nl = code tp # bl2wc (<lm>) # rec_exec ([recf.id (Suc (Suc 0)) 0, recf.id (Suc
(Suc 0))
       (Suc 0), Mn (Suc (Suc 0)) rec_nonstop] ! j) [code tp, bl2wc (<lm>)] # 0 ↑ (ftj − Suc arj)
@ anything}
        by simp
    next
```

```
      fix j
      assume (j::nat) < 2
      thus terminate ([recf.id (Suc (Suc 0)) 0, recf.id (Suc (Suc 0)) (Suc 0), Mn (Suc (Suc 0))
rec_nonstop] ! j)
        [code tp, bl2wc (<lm>)]
        by(case_tac j, auto intro!: primerec_terminate)
    qed
    thus {λnl. nl = code tp # bl2wc (<lm>) # 0↑(ft2 − Suc (Suc 0)) @ anything} ap2↑
      by simp
  qed
  thus {λnl. nl = code tp # bl2wc (<lm>) # 0↑(ft1 − Suc (Suc 0)) @ anything} ap1↑ by
simp
 qed
qed

lemma uabc_uhalt':
 ⟦tm_wf (tp, 0);
 ∀ stp. (¬ TSTD (steps0 (Suc 0, Bk↑(l), <lm>) tp stp));
 rec_ci rec_F = (ap, pos, md)⟧
 ⟹ {λ nl. nl = [code tp, bl2wc (<lm>)]} ap↑
proof(frule_tac F_ap = ap and rs_pos = pos and a_md = md
  and suflm = [] in F_aprog_uhalt, auto simp: abc_Hoare_unhalt_def,
  case_tac abc_steps_l (0, [code tp, bl2wc (<lm>)]) ap n, simp)
 fix n a b
 assume h:
  ∀n. abc_notfinal (abc_steps_l (0, code tp # bl2wc (<lm>) # 0↑(md − pos)) ap n) ap
  abc_steps_l (0, [code tp, bl2wc (<lm>)]) ap n = (a, b)
  tm_wf (tp, 0)
  rec_ci rec_F = (ap, pos, md)
 moreover have a: ap ≠ []
  using h rec_ci_not_null[of rec_F pos md] by auto
 ultimately show a < length ap
 proof(erule_tac x = n in allE)
   assume g: abc_notfinal (abc_steps_l (0, code tp # bl2wc (<lm>) # 0↑(md − pos)) ap n)
ap
   obtain ss nl where b : abc_steps_l (0, code tp # bl2wc (<lm>) # 0↑(md − pos)) ap n =
(ss, nl)
     by (metis prod.exhaust)
   then have c: ss < length ap
     using g by simp
   thus ?thesis
     using a b c
     using abc_list_crsp_steps[of [code tp, bl2wc (<lm>)]
        md − pos ap n ss nl] h
     by(simp)
 qed
qed

lemma uabc_uhalt:
 ⟦tm_wf (tp, 0);
```

```
 ∀ stp. (¬ TSTD (steps0 (Suc 0, Bk↑(l), <lm>) tp stp))⟧
 ⟹ {λ nl. nl = [code tp, bl2wc (<lm>)]} F_aprog ↑
proof −
 obtain a b c where abc:rec_ci rec_F = (a,b,c) by (cases rec_ci rec_F) force
 assume a:tm_wf (tp, 0) ∀ stp. (¬ TSTD (steps0 (Suc 0, Bk↑(l), <lm>) tp stp))
 from uabc_uhalt'[OF a abc] abc_Hoare_plus_unhalt1
 show {λ nl. nl = [code tp, bl2wc (<lm>)]} F_aprog ↑
   by(simp add: F_aprog_def abc)
qed

lemma tutm_uhalt':
 assumes tm_wf: tm_wf (tp,0)
   and unhalt: ∀ stp. (¬ TSTD (steps0 (1, Bk↑(l), <lm>) tp stp))
 shows ∀ stp. ¬ is_final (steps0 (1, [Bk, Bk], <[code tp, bl2wc (<lm>)]>) t_utm stp)
 unfolding t_utm_def
proof(rule_tac compile_correct_unhalt, auto)
 show F_tprog = tm_of F_aprog
   by(simp add: F_tprog_def)
next
 show crsp (layout_of F_aprog) (0, [code tp, bl2wc (<lm>)]) (Suc 0, [Bk, Bk], <[code tp, bl2wc
(<lm>)]>) []
   by(auto simp: crsp.simps start_of.simps)
next
 fix stp a b
 show abc_steps_l (0, [code tp, bl2wc (<lm>)]) F_aprog stp = (a, b) ⟹ a < length F_aprog
   using assms
   apply(drule_tac uabc_uhalt, auto simp: abc_Hoare_unhalt_def)
   by(erule_tac x = stp in allE, erule_tac x = stp in allE, simp)
qed

lemma tinres_commute: tinres r r' ⟹ tinres r' r
 apply(auto simp: tinres_def)
 done

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(case_tac steps0 (st, l', r) tp stp)
 fix aa ba ca
 assume h: steps0 (st, l, r) tp stp = (a, b, c)
   steps0 (st, l', r') tp stp = (a', b', c')
   tinres l l' tinres r r'
   steps0 (st, l', r) tp stp = (aa, ba, ca)
 have tinres b ba ∧ c = ca ∧ a = aa
   using h
   apply(rule_tac tinres_steps1, auto)
   done
 moreover have b' = ba ∧ tinres c' ca ∧ a' = aa
   using h
```

```
  apply(rule_tac tinres_steps2, auto intro: tinres_commute)
  done
 ultimately show ?thesis
  apply(auto intro: tinres_commute)
  done
qed

lemma tape_normalize:
 assumes ∀ stp. ¬ is_final(steps0 (Suc 0, [Bk,Bk], <[code tp, bl2wc (<lm>)]>) t_utm stp)
 shows  ∀ stp. ¬ is_final (steps0 (Suc 0, Bk↑(m), <[code tp, bl2wc (<lm>)]> @ Bk↑(n)) t_utm
stp)
  (is ∀ stp. ?P stp)
proof
 fix stp
 from assms[rule_format,of stp] show ?P stp
   apply(case_tac steps0 (Suc 0, Bk↑(m), <[code tp, bl2wc (<lm>)]> @ Bk↑(n)) t_utm stp,
simp)
  apply(case_tac steps0 (Suc 0, [Bk, Bk], <[code tp, bl2wc (<lm>)]>) t_utm stp, simp)
  apply(drule_tac inres_tape, auto)
   apply(auto simp: tinres_def)
  apply(case_tac m > Suc (Suc 0))
   apply(rule_tac x = m − Suc (Suc 0) in exI)
   apply(case_tac m, simp_all)
   apply(metis Suc_lessD Suc_pred replicate_Suc)
  apply(rule_tac x = 2 − m in exI, simp add: replicate_add[THEN sym])
  apply(simp only: numeral_2_eq_2, simp add: replicate_Suc)
  done
qed

lemma tutm_uhalt:
 ⟦tm_wf (tp,0);
  ∀ stp. (¬ TSTD (steps0 (Suc 0, Bk↑(l), <args>) tp stp))⟧
 ⟹ ∀ stp. ¬ is_final (steps0 (Suc 0, Bk↑(m), <[code tp, bl2wc (<args>)]> @ Bk↑(n)) t_utm
stp)
 apply(rule_tac tape_normalize)
 apply(rule_tac tutm_uhalt'[simplified], simp_all)
 done

lemma UTM_uhalt_lemma_pre:
 assumes tm_wf: tm_wf (tp, 0)
  and exec: ∀ stp. (¬ TSTD (steps0 (Suc 0, Bk↑(l), <args>) tp stp))
  and args: args ≠ []
 shows ∀ stp. ¬ is_final (steps0 (Suc 0, [], <code tp # args>)  UTM_pre stp)
proof −
 let ?P1 = λ (l, r). l = [] ∧ r = <code tp # args>
 let ?Q1 = λ (l, r). (l = [Bk] ∧
        (∃ rn. r = Oc↑(Suc (code tp)) @ Bk # Oc↑(Suc (bl_bin (<args>))) @ Bk↑(rn)))
 let ?P2 = ?Q1
 have {?P1} (t_wcode |+| t_utm) ↑
 proof(rule_tac Hoare_plus_unhalt)
```

```
   show tm_wf (t_wcode, 0) by auto
  next
   show {?P1} t_wcode {?Q1}
    apply(rule_tac Hoare_haltI, auto)
    using wcode_lemma_1[of args code tp] args
    apply(auto)
    by (metis (mono_tags, lifting) holds_for.simps is_finalI old.prod.case)
  next
   show {?P2} t_utm ↑
   proof(rule_tac Hoare_unhaltI, auto)
    fix n rn
      assume h: is_final (steps0 (Suc 0, [Bk], Oc ↑ Suc (code tp) @ Bk # Oc ↑ Suc (bl_bin
(<args>)) @ Bk ↑ rn) t_utm n)
    have ∀ stp. ¬ is_final (steps0 (Suc 0, Bk↑(Suc 0), <[code tp, bl2wc (<args>)]> @ Bk↑(rn))
t_utm stp)
      using assms
      apply(rule_tac tutm_uhalt, simp_all)
      done
    thus False
      using h
      apply(erule_tac x = n in allE)
      apply(simp add: tape_of_list_def bin_wc_eq tape_of_nat_def)
      done
   qed
  qed
  thus ?thesis
   apply(simp add: Hoare_unhalt_def UTM_pre_def)
   done
qed
```

The correctness of *UTM*, the unhalt case.

```
lemma UTM_uhalt_lemma':
 assumes tm_wf: tm_wf (tp, 0)
  and unhalt: ∀ stp. (¬ TSTD (steps0 (Suc 0, Bk↑(l), <args>) tp stp))
  and args: args ≠ []
 shows ∀ stp. ¬ is_final (steps0 (Suc 0, [], <code tp # args>) UTM stp)
 using UTM_uhalt_lemma_pre[of tp l args] assms
 apply(simp add: UTM_pre_def t_utm_def UTM_def F_aprog_def F_tprog_def)
 apply(case_tac rec_ci rec_F, simp)
 done

lemma UTM_halt_lemma:
 assumes tm_wf: tm_wf (p, 0)
  and resut: rs > 0
  and args: (args::nat list) ≠ []
  and exec: {(λtp. tp = (Bk↑i, <args>))} p {(λtp. tp = (Bk↑m, Oc↑rs @ Bk↑k))}
 shows {(λtp. tp = ([], <code p # args>))} UTM {(λtp. (∃ m n. tp = (Bk↑m, Oc↑rs @
Bk↑n)))}
proof -
 let ?steps0 = steps0 (Suc 0, [], <code p # args>)
```

```
let ?stepsBk = steps0 (Suc 0, Bk↑i, <args>) p
from wcode_lemma_1[OF args,of code p] obtain stp ln rn where
  wcl1:?steps0 t_wcode stp =
   (0, [Bk], Oc ↑ Suc (code p) @ Bk # Oc ↑ Suc (bl_bin (<args>)) @ Bk ↑ rn) by fast
from exec Hoare_halt_def obtain n where
  n:{λtp. tp = (Bk ↑ i, <args>)} p {λtp. tp = (Bk ↑ m, Oc ↑ rs @ Bk ↑ k)}
    is_final (?stepsBk n)
    (λtp. tp = (Bk ↑ m, Oc ↑ rs @ Bk ↑ k)) holds_for steps0 (Suc 0, Bk ↑ i, <args>) p n
  by auto
obtain a where a:a = fst (rec_ci rec_F) by blast
have {(λ (l, r). l = [] ∧ r = <code p # args>)} (t_wcode |+| t_utm)
       {(λ (l, r). (∃ m. l = Bk↑m) ∧ (∃ n. r = Oc↑rs @ Bk↑n))}
proof(rule_tac Hoare_plus_halt)
  show {λ(l, r). l = [] ∧ r = <code p # args>} t_wcode {λ (l, r). (l = [Bk] ∧
  (∃ rn. r = Oc↑(Suc (code p)) @ Bk # Oc↑(Suc (bl_bin (<args>))) @ Bk↑(rn)))}
    using wcl1 by (auto intro!:Hoare_haltI exI[of _ stp])
next
  have ∃ stp. (?stepsBk stp = (0, Bk↑m, Oc↑rs @ Bk↑k))
    using n by (case_tac ?stepsBk n, auto)
  then obtain stp where k: steps0 (Suc 0, Bk↑i, <args>) p stp = (0, Bk↑m, Oc↑rs @ Bk↑k)
    ..
  thus {λ(l, r). l = [Bk] ∧ (∃rn. r = Oc ↑ Suc (code p) @ Bk # Oc ↑ Suc (bl_bin (<args>))
@ Bk ↑ rn)}
    t_utm {λ(l, r). (∃m. l = Bk ↑ m) ∧ (∃n. r = Oc ↑ rs @ Bk ↑ n)}
  proof(rule_tac Hoare_haltI, auto)
    fix rn
    from t_utm_halt_eq[OF assms(1) k assms(2),of rn] assms k
    have ∃ ma n stp. steps0 (Suc 0, [Bk], <[code p, bl2wc (<args>)]> @ Bk ↑ rn) t_utm stp =
     (0, Bk ↑ ma, Oc ↑ rs @ Bk ↑ n) by (auto simp add: bin_wc_eq)
    then obtain stpx m′ n′ where
      t:steps0 (Suc 0, [Bk], <[code p, bl2wc (<args>)]> @ Bk ↑ rn) t_utm stpx =
     (0, Bk ↑ m′, Oc ↑ rs @ Bk ↑ n′) by auto
   show ∃n. is_final (steps0 (Suc 0, [Bk], Oc ↑ Suc (code p) @ Bk # Oc ↑ Suc (bl_bin (<args>))
@ Bk ↑ rn) t_utm n) ∧
         (λ(l, r). (∃m. l = Bk ↑ m) ∧ (∃n. r = Oc ↑ rs @ Bk ↑ n)) holds_for steps0
      (Suc 0, [Bk], Oc ↑ Suc (code p) @ Bk # Oc ↑ Suc (bl_bin (<args>)) @ Bk ↑ rn) t_utm n

      using t
      by(auto simp: bin_wc_eq tape_of_list_def tape_of_nat_def intro:exI[of _ stpx])
  qed
next
  show tm_wf0 t_wcode by auto
qed
then obtain n where
  is_final (?steps0 (t_wcode |+| t_utm) n)
  (λ(l, r). (∃m. l = Bk ↑ m) ∧
       (∃n. r = Oc ↑ rs @ Bk ↑ n)) holds_for ?steps0 (t_wcode |+| t_utm) n
  by(auto simp add: Hoare_halt_def a)
thus ?thesis
  apply(case_tac rec_ci rec_F)
```

```
  apply(auto simp add: UTM_def Hoare_halt_def)
  apply(case_tac (?steps0 (t_wcode |+| t_utm) n))
  apply(rule_tac x=n in exI)
  apply(auto simp add:a t_utm_def F_aprog_def F_tprog_def)
  done
qed

lemma UTM_halt_lemma2:
 assumes tm_wf: tm_wf (p, 0)
  and args: (args::nat list) ≠ []
  and exec: {(λtp. tp = ([], <args>))} p {(λtp. tp = (Bk↑m, <(n::nat)> @ Bk↑k))}
 shows {(λtp. tp = ([], <code p # args>))} UTM {(λtp. (∃ m k. tp = (Bk↑m, <n> @ Bk↑k)))}
 using UTM_halt_lemma[OF assms(1) _ assms(2), where i=0]
 using assms(3)
 apply(simp add: tape_of_nat_def)
 done


lemma UTM_unhalt_lemma:
 assumes tm_wf: tm_wf (p, 0)
  and unhalt: {(λtp. tp = (Bk↑i, <args>))} p ↑
  and args: args ≠ []
 shows {(λtp. tp = ([], <code p # args>))} UTM ↑
proof −
 have (¬ TSTD (steps0 (Suc 0, Bk↑(i), <args>) p stp)) for stp
  using unhalt
  apply(auto simp: Hoare_unhalt_def)
  apply(case_tac steps0 (Suc 0, Bk ↑ i, <args>) p stp, simp)
  apply(erule_tac allE[of _ stp], simp add: TSTD_def)
  done
 then have ∀ stp. ¬ is_final (steps0 (Suc 0, [], <code p # args>)  UTM stp)
  using assms
  apply(rule_tac  UTM_uhalt_lemma', auto)
  done
 thus ?thesis
  apply(simp add: Hoare_unhalt_def)
  done
qed

lemma UTM_unhalt_lemma2:
 assumes tm_wf: tm_wf (p, 0)
  and unhalt: {(λtp. tp = ([], <args>))} p ↑
  and args: args ≠ []
 shows {(λtp. tp = ([], <code p # args>))} UTM ↑
 using UTM_unhalt_lemma[OF assms(1), where i=0]
 using assms(2−3)
 apply(simp add: tape_of_nat_def)
 done
```

end

References

[1] G. Boolos, J. P. Burgess, and R. C. Jeffrey. *Computability and Logic (5th ed.)*. Cambridge University Press, 2007.

[2] B. Felgenhauer. Minsky machines. *Archive of Formal Proofs*, Aug. 2018. `http://isa-afp.org/entries/Minsky_Machines.html`, Formal proof development.

[3] S. J. Joosten. Finding models through graph saturation. *Journal of Logical and Algebraic Methods in Programming*, 100:98 – 112, 2018.

[4] S. J. C. Joosten. Graph saturation. *Archive of Formal Proofs*, Nov. 2018. `http://isa-afp.org/entries/Graph_Saturation.html`, Formal proof development.

[5] M. Nedzelsky. Recursion theory i. *Archive of Formal Proofs*, Apr. 2008. `http://isa-afp.org/entries/Recursion-Theory-I.html`, Formal proof development.

[6] J. Xu, X. Zhang, and C. Urban. Mechanising turing machines and computability theory in isabelle/hol. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, pages 147–162, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.