utm

By xujian

December 27, 2012

Contents

theory *turing-basic*
imports *Main*
begin

1 Basic definitions of Turing machine

Actions of Turing machine (Abbreviated TM in the following*).

datatype *taction* =
— Write zero
W0 |
— Write one
W1 |
— Move left
L |
— Move right
R |
— Do nothing
Nop

Tape contents in every block.

datatype *block* =
— Blank
Bk |
— Occupied
Oc

Tape is represented as a pair of lists (L_{left}, L_{right}), where L_left, named *left list*, is used to represent the tape to the left of RW-head and L_{right}, named *right list*, is used to represent the tape under and to the right of RW-head.

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

The state of turing machine.

type-synonym *tstate* = *nat*

Turing machine instruction is represented as a pair (*action*, *next-state*), where *action* is the action to take at the current state and *next-state* is the next state the machine is getting into after the action.

type-synonym *tinst* = *taction* × *tstate*

Program of Turing machine is represented as a list of Turing instructions and the execution of the program starts from the head of the list.

type-synonym *tprog* = *tinst list*

Turing machine configuration, which consists of the current state and the tape.

type-synonym *t-conf* = *tstate* × *tape*

```
fun nth-of :: 'a list ⇒ nat ⇒ 'a option
  where
  nth-of xs n = (if n < length xs then Some (xs!n)
                 else None)
```

The function used to fetech instruction out of Turing program.

```
fun fetch :: tprog ⇒ tstate ⇒ block ⇒ tinst
  where
  fetch p s b = (if s = 0 then (Nop, 0) else
                 case b of
                   Bk ⇒ case nth-of p (2 ∗ (s − 1)) of
                        Some i ⇒ i
                      | None ⇒ (Nop, 0)
                 | Oc ⇒ case nth-of p (2 ∗ (s − 1) +1) of
                        Some i ⇒ i
                      | None ⇒ (Nop, 0))
```

```
fun new-tape :: taction ⇒ tape ⇒ tape
where
   new-tape action (leftn, rightn) = (case action of
                                       W0 ⇒ (leftn, Bk#(tl rightn)) |
                                       W1 ⇒ (leftn, Oc#(tl rightn)) |
                                       L  ⇒  (if leftn = [] then (tl leftn, Bk#rightn)
                                              else (tl leftn, (hd leftn) # rightn)) |
                                       R  ⇒ if rightn = [] then (Bk#leftn,tl rightn)
                                              else ((hd rightn)#leftn, tl rightn) |
                                       Nop ⇒ (leftn, rightn)
                                     )
```

The one step function used to transfer Turing machine configuration.

```
fun tstep :: t-conf ⇒ tprog ⇒ t-conf
  where
  tstep c p = (let (s, l, r) = c in
                 let (ac, ns) = (fetch p s (case r of [] ⇒ Bk |
                                                    x # xs ⇒ x)) in
                   (ns, new-tape ac (l, r)))
```

The many-step function.

```
fun steps :: t-conf ⇒ tprog ⇒ nat ⇒ t-conf
  where
  steps c p 0 = c |
  steps c p (Suc n) = steps (tstep c p) p n
```

lemma *tstep-red*: *steps c p* (*Suc n*) = *tstep* (*steps c p n*) *p*
proof(*induct n arbitrary*: *c*)
fix *c*
show *steps c p* (*Suc 0*) = *tstep* (*steps c p 0*) *p* **by**(*simp add*: *steps.simps*)
next
fix *n c*
assume *ind*: $\bigwedge$ *c*. *steps c p* (*Suc n*) = *tstep* (*steps c p n*) *p*
have *steps* (*tstep c p*) *p* (*Suc n*) = *tstep* (*steps* (*tstep c p*) *p n*) *p*
by(*rule ind*)
thus *steps c p* (*Suc* (*Suc n*)) = *tstep* (*steps c p* (*Suc n*)) *p* **by**(*simp add*: *steps.simps*)
qed

declare *Let-def*[*simp*] *option.split*[*split*]

definition
iseven n ≡ ∃ *x*. *n* = *2* ∗ *x*

The following *t-correct* function is used to specify the wellformedness of Turing machine.

fun *t-correct* :: *tprog* ⇒ *bool*
where
t-correct p = (*length p* ≥ *2* ∧ *iseven* (*length p*) ∧
list-all (λ (*acn*, *s*). *s* ≤ *length p div 2*) *p*)

declare *t-correct.simps*[*simp del*]

lemma *allimp*: ⟦∀ *x*. *P x* ⟶ *Q x*; ∀ *x*. *P x*⟧ ⟹ ∀ *x*. *Q x*
by(*auto elim*: *allE*)

lemma *halt-lemma*: ⟦*wf LE*; ∀ *n*. (¬ *P* (*f n*) ⟶ (*f* (*Suc n*), (*f n*)) ∈ *LE*)⟧ ⟹ ∃ *n*. *P* (*f n*)
apply(*rule exCI*, *drule allimp*, *auto*)
apply(*drule-tac f* = *f* **in** *wf-inv-image*, *simp add*: *inv-image-def*)
apply(*erule wf-induct*, *auto*)
done

lemma *steps-add*: *steps c t* (*x* + *y*) = *steps* (*steps c t x*) *t y*
by(*induct x arbitrary*: *c*, *auto simp*: *steps.simps tstep-red*)

lemma *listall-set*: *list-all p t* ⟹ ∀ *a* ∈ *set t*. *p a*
by(*induct t*, *auto*)

lemma *fetch-ex*: ∃ *b a*. *fetch T aa ab* = (*b*, *a*)
by(*simp add*: *fetch.simps*)
definition *exponent* :: *'a* ⇒ *nat* ⇒ *'a list* (-⁻ [*0*, *0*]*100*)
where *exponent x n* = *replicate n x*

tinres l1 l2 means left list *l1* is congruent with *l2* with respect to the execution of Turing machine. Appending Blank to the right of eigther one does not affect the outcome of excution.

definition *tinres* :: *block list* ⇒ *block list* ⇒ *bool*
  where
  tinres bx by = (∃ *n*. *bx* = *by*@Bk^n ∨ *by* = *bx* @ Bk^n)

lemma *exp-zero*: a^0 = []
by(*simp add*: *exponent-def*)
lemma *exp-ind-def*: $a^{Suc\ x}$ = *a* # a^x
by(*simp add*: *exponent-def*)

The following lemma shows the meaning of *tinres* with respect to one step execution.

lemma *tinres-step*:
  ⟦*tinres l l'*; *tstep* (*ss*, *l*, *r*) *t* = (*sa*, *la*, *ra*); *tstep* (*ss*, *l'*, *r*) *t* = (*sb*, *lb*, *rb*)⟧
    ⟹ *tinres la lb* ∧ *ra* = *rb* ∧ *sa* = *sb*
apply(*auto simp*: *tstep.simps fetch.simps new-tape.simps*
        split: *if-splits taction.splits list.splits*
              block.splits)
apply(*case-tac* [!] *t* ! (*2* ∗ (*ss* − *Suc 0*)),
     auto simp: *exponent-def tinres-def split*: *if-splits taction.splits list.splits*
              block.splits)
apply(*case-tac* [!] *t* ! (*2* ∗ (*ss* − *Suc 0*) + *Suc 0*),
     auto simp: *exponent-def tinres-def split*: *if-splits taction.splits list.splits*
              block.splits)
done

declare *tstep.simps*[*simp del*] *steps.simps*[*simp del*]

The following lemma shows the meaning of *tinres* with respect to many step execution.

lemma *tinres-steps*:
  ⟦*tinres l l'*; *steps* (*ss*, *l*, *r*) *t stp* = (*sa*, *la*, *ra*); *steps* (*ss*, *l'*, *r*) *t stp* = (*sb*, *lb*, *rb*)⟧
    ⟹ *tinres la lb* ∧ *ra* = *rb* ∧ *sa* = *sb*
apply(*induct stp arbitrary*: *sa la ra sb lb rb*, *simp add*: *steps.simps*)
apply(*simp add*: *tstep-red*)
apply(*case-tac* (*steps* (*ss*, *l*, *r*) *t stp*))
apply(*case-tac* (*steps* (*ss*, *l'*, *r*) *t 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 stp* = (*sa*, *la*, *ra*);
          steps (*ss*, *l'*, *r*) *t stp* = (*sb*, *lb*, *rb*)⟧ ⟹ *tinres la lb* ∧ *ra* = *rb* ∧ *sa* = *sb*
  and *h*: *tinres l l' tstep* (*steps* (*ss*, *l*, *r*) *t stp*) *t* = (*sa*, *la*, *ra*)
         tstep (*steps* (*ss*, *l'*, *r*) *t stp*) *t* = (*sb*, *lb*, *rb*) *steps* (*ss*, *l*, *r*) *t stp* = (*a*, *b*, *c*)
         steps (*ss*, *l'*, *r*) *t stp* = (*aa*, *ba*, *ca*)
  have *tinres b ba* ∧ *c* = *ca* ∧ *a* = *aa*

apply(*rule-tac ind*, *simp-all add*: *h*)
done
thus *tinres la lb* ∧ *ra* = *rb* ∧ *sa* = *sb*
apply(*rule-tac l* = *b* **and** *l'* = *ba* **and** *r* = *c* **and** *ss* = *a*
and *t* = *t* **in** *tinres-step*)
using *h*
apply(*simp*, *simp*, *simp*)
done
qed

The following function *tshift tp n* is used to shift Turing programs *tp* by *n* when it is going to be combined with others.

fun *tshift* :: *tprog* ⇒ *nat* ⇒ *tprog*
where
tshift tp off = (*map* (λ (*action*, *state*). (*action*, (*if state* = *0 then 0*
else state + *off*))) *tp*)

When two Turing programs are combined, the end state (state *0*) of the one at the prefix position needs to be connected to the start state of the one at postfix position. If *tp* is the Turing program to be at the prefix, *change-termi-state tp* is the transformed Turing program.

fun *change-termi-state* :: *tprog* ⇒ *tprog*
where
change-termi-state t =
(*map* (λ (*acn*, *ns*). *if ns* = *0 then* (*acn*, *Suc* ((*length t*) *div 2*)) *else* (*acn*, *ns*)) *t*)

t-add tp1 tp2 is the combined Truing program.

fun *t-add* :: *tprog* ⇒ *tprog* ⇒ *tprog* (- |+| - [*0*, *0*] *100*)
where
t-add t1 t2 = ((*change-termi-state t1*) @ (*tshift t2* ((*length t1*) *div 2*)))

Tests whether the current configuration is at state *0*.

definition *isS0* :: *t-conf* ⇒ *bool*
where
isS0 c = (*let* (*s*, *l*, *r*) = *c in s* = *0*)

declare *tstep.simps*[*simp del*] *steps.simps*[*simp del*]
t-add.simps[*simp del*] *fetch.simps*[*simp del*]
new-tape.simps[*simp del*]

Single step execution starting from state *0* will not make any progress.

lemma *tstep-0*: *tstep* (*0*, *tp*) *p* = (*0*, *tp*)
apply(*simp add*: *tstep.simps fetch.simps new-tape.simps*)
done

Many step executions starting from state *0* will not make any progress.

lemma *steps-0*: *steps* (*0*, *tp*) *p stp* = (*0*, *tp*)

apply(*induct stp*)
apply(*simp add*: *steps.simps*)
apply(*simp add*: *tstep-red tstep-0*)
done

lemma *s-keep-step*: ⟦$a \leq$ *length A div 2*; *tstep* (*a*, *b*, *c*) *A* = (*s*, *l*, *r*); *t-correct A*⟧
$\Longrightarrow$ $s \leq$ *length A div 2*
apply(*simp add*: *tstep.simps fetch.simps t-correct.simps iseven-def*
split: *if-splits block.splits list.splits*)
apply(*case-tac* [!] *a*, *auto simp*: *list-all-length*)
apply(*erule-tac x* = *2* ∗ *nat* **in** *allE*, *auto*)
apply(*erule-tac x* = *2* ∗ *nat* **in** *allE*, *auto*)
apply(*erule-tac x* = *Suc* (*2* ∗ *nat*) **in** *allE*, *auto*)
done

lemma *s-keep*: ⟦*steps* (*Suc 0*, *tp*) *A stp* = (*s*, *l*, *r*); *t-correct A*⟧ $\Longrightarrow$ $s \leq$ *length A div 2*
proof(*induct stp arbitrary*: *s l r*)
case *0* **thus** *?case* **by**(*auto simp*: *t-correct.simps steps.simps*)
next
fix *stp s l r*
assume *ind*: ⋀*s l r*. ⟦*steps* (*Suc 0*, *tp*) *A stp* = (*s*, *l*, *r*); *t-correct A*⟧ $\Longrightarrow$ $s \leq$ *length A div 2*
and *h1*: *steps* (*Suc 0*, *tp*) *A* (*Suc stp*) = (*s*, *l*, *r*)
and *h2*: *t-correct A*
from *h1 h2* **show** $s \leq$ *length A div 2*
proof(*simp add*: *tstep-red*, *cases* (*steps* (*Suc 0*, *tp*) *A stp*), *simp*)
fix *a b c*
assume *h3*: *tstep* (*a*, *b*, *c*) *A* = (*s*, *l*, *r*)
and *h4*: *steps* (*Suc 0*, *tp*) *A stp* = (*a*, *b*, *c*)
have $a \leq$ *length A div 2*
using *h2 h4*
by(*rule-tac l* = *b* **and** *r* = *c* **in** *ind*, *auto*)
thus *?thesis*
using *h3 h2*
by(*simp add*: *s-keep-step*)
qed
qed

lemma *t-merge-fetch-pre*:
⟦*fetch A s b* = (*ac*, *ns*); $s \leq$ *length A div 2*; *t-correct A*; $s \neq 0$⟧ $\Longrightarrow$
fetch (*A* |+| *B*) *s b* = (*ac*, *if ns* = *0 then Suc* (*length A div 2*) *else ns*)
apply(*subgoal-tac 2* ∗ (*s* − *Suc 0*) < *length A* ∧ *Suc* (*2* ∗ (*s* − *Suc 0*)) < *length A*)
apply(*auto simp*: *fetch.simps t-add.simps split*: *if-splits block.splits*)
apply(*simp-all add*: *nth-append change-termi-state.simps*)
done

lemma [*simp*]: ⟦¬ $a \leq$ *length A div 2*; *t-correct A*⟧ $\Longrightarrow$ *fetch A a b* = (*Nop*, *0*)

apply(*auto simp*: *fetch.simps del*: *nth-of.simps split*: *block.splits*)
apply(*case-tac* [!] *a*, *auto simp*: *t-correct.simps iseven-def*)
done

lemma [*elim*]: ⟦*t-correct A*; ¬ *isS0* (*tstep* (*a*, *b*, *c*) *A*)⟧ ⟹ *a* ≤ *length A div 2*
apply(*rule-tac classical*, *auto simp*: *tstep.simps new-tape.simps isS0-def*)
done

lemma [*elim*]: ⟦*t-correct A*; ¬ *isS0* (*tstep* (*a*, *b*, *c*) *A*)⟧ ⟹ *0* < *a*
apply(*rule-tac classical*, *simp add*: *tstep-0 isS0-def*)
done

lemma *t-merge-pre-eq-step*: ⟦*tstep* (*a*, *b*, *c*) *A* = *cf*; *t-correct A*; ¬ *isS0 cf*⟧
⟹ *tstep* (*a*, *b*, *c*) (*A* |+| *B*) = *cf*
apply(*subgoal-tac a* ≤ *length A div 2* ∧ *a* ≠ *0*)
apply(*simp add*: *tstep.simps*)
apply(*case-tac fetch A a* (*case c of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*), *simp*)
apply(*drule-tac B* = *B* **in** *t-merge-fetch-pre*, *simp*, *simp*, *simp*, *simp add*: *isS0-def*, *auto*)
done

lemma *t-merge-pre-eq*: ⟦*steps* (*Suc 0*, *tp*) *A stp* = *cf*; ¬ *isS0 cf*; *t-correct A*⟧
⟹ *steps* (*Suc 0*, *tp*) (*A* |+| *B*) *stp* = *cf*
proof(*induct stp arbitrary*: *cf*)
  case *0* **thus** *?case* **by**(*simp add*: *steps.simps*)
next
  fix *stp cf*
  assume *ind*: ⋀*cf*. ⟦*steps* (*Suc 0*, *tp*) *A stp* = *cf*; ¬ *isS0 cf*; *t-correct A*⟧
    ⟹ *steps* (*Suc 0*, *tp*) (*A* |+| *B*) *stp* = *cf*
  and *h1*: *steps* (*Suc 0*, *tp*) *A* (*Suc stp*) = *cf*
  and *h2*: ¬ *isS0 cf*
  and *h3*: *t-correct A*
  from *h1 h2 h3* **show** *steps* (*Suc 0*, *tp*) (*A* |+| *B*) (*Suc stp*) = *cf*
  proof(*simp add*: *tstep-red*, *cases steps* (*Suc 0*, *tp*) (*A*) *stp*, *simp*)
    fix *a b c*
    assume *h4*: *tstep* (*a*, *b*, *c*) *A* = *cf*
    and *h5*: *steps* (*Suc 0*, *tp*) *A stp* = (*a*, *b*, *c*)
    have *steps* (*Suc 0*, *tp*) (*A* |+| *B*) *stp* = (*a*, *b*, *c*)
    proof(*cases a*)
      case *0* **thus** *?thesis*
        using *h4 h2*
        apply(*simp add*: *tstep-0*, *cases cf*, *simp add*: *isS0-def*)
        done
    next
      case (*Suc n*) **thus** *?thesis*
        using *h5 h3*
        apply(*rule-tac ind*, *auto simp*: *isS0-def*)
        done

```
    qed
    thus tstep (steps (Suc 0, tp) (A |+| B) stp) (A |+| B) = cf
      using h4 h5 h3 h2
      apply(simp)
      apply(rule t-merge-pre-eq-step, auto)
      done
  qed
qed

declare nth.simps[simp del] tshift.simps[simp del] change-termi-state.simps[simp
del]

lemma [simp]: length (change-termi-state A) = length A
by(simp add: change-termi-state.simps)

lemma first-halt-point: steps (Suc 0, tp) A stp = (0, tp')
 ⟹ ∃ stp. ¬ isS0 (steps (Suc 0, tp) A stp) ∧ steps (Suc 0, tp) A (Suc stp) = (0,
tp')
proof(induct stp)
  case 0 thus ?case by(simp add: steps.simps)
next
  case (Suc n)
  fix stp
  assume ind: steps (Suc 0, tp) A stp = (0, tp') ⟹
      ∃ stp. ¬ isS0 (steps (Suc 0, tp) A stp) ∧ steps (Suc 0, tp) A (Suc stp) = (0,
tp')
    and h: steps (Suc 0, tp) A (Suc stp) = (0, tp')
  from h show ∃ stp. ¬ isS0 (steps (Suc 0, tp) A stp) ∧ steps (Suc 0, tp) A (Suc
stp) = (0, tp')
  proof(simp add: tstep-red, cases steps (Suc 0, tp) A stp, simp, case-tac a)
    fix a b c
    assume g1: a = (0::nat)
    and g2: tstep (a, b, c) A = (0, tp')
    and g3: steps (Suc 0, tp) A stp = (a, b, c)
    have steps (Suc 0, tp) A stp = (0, tp')
      using g2 g1 g3
      by(simp add: tstep-0)
    hence ∃ stp. ¬ isS0 (steps (Suc 0, tp) A stp) ∧ steps (Suc 0, tp) A (Suc stp)
= (0, tp')
      by(rule ind)
    thus ∃ stp. ¬ isS0 (steps (Suc 0, tp) A stp) ∧ tstep (steps (Suc 0, tp) A stp)
A = (0, tp')
      apply(simp add: tstep-red)
      done
  next
    fix a b c nat
    assume g1: steps (Suc 0, tp) A stp = (a, b, c)
    and g2: steps (Suc 0, tp) A (Suc stp) = (0, tp') a= Suc nat
    thus ∃ stp. ¬ isS0 (steps (Suc 0, tp) A stp) ∧ tstep (steps (Suc 0, tp) A stp)
```

$A = (0, tp')$
    apply(*rule-tac x = stp* **in** *exI*)
    apply(*simp add*: *isS0-def tstep-red*)
    done
  qed
qed

lemma *t-merge-pre-halt-same′*:
  $[\![$¬ *isS0* (*steps* (*Suc 0*, *tp*) *A stp*) ; *steps* (*Suc 0*, *tp*) *A* (*Suc stp*) = (*0*, *tp′*); *t-correct A*$]\!]$
  ⟹ *steps* (*Suc 0*, *tp*) (*A* |+| *B*) (*Suc stp*) = (*Suc* (*length A div 2*), *tp′*)
proof(*simp add*: *tstep-red*, *cases steps* (*Suc 0*, *tp*) *A stp*, *simp*)
  fix *a b c*
  assume *h1*: ¬ *isS0* (*a*, *b*, *c*)
  and *h2*: *tstep* (*a*, *b*, *c*) *A* = (*0*, *tp′*)
  and *h3*: *t-correct A*
  and *h4*: *steps* (*Suc 0*, *tp*) *A stp* = (*a*, *b*, *c*)
  have *steps* (*Suc 0*, *tp*) (*A* |+| *B*) *stp* = (*a*, *b*, *c*)
    using *h1 h4 h3*
    apply(*rule-tac t-merge-pre-eq*, *auto*)
    done
  moreover have *tstep* (*a*, *b*, *c*) (*A* |+| *B*) = (*Suc* (*length A div 2*), *tp′*)
    using *h2 h3 h1 h4*
    apply(*simp add*: *tstep.simps*)
    apply(*case-tac fetch A a* (*case c of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*), *simp*)
    apply(*drule-tac B = B* **in** *t-merge-fetch-pre*, *auto simp*: *isS0-def intro*: *s-keep*)
    done
  ultimately show *tstep* (*steps* (*Suc 0*, *tp*) (*A* |+| *B*) *stp*) (*A* |+| *B*) = (*Suc* (*length A div 2*), *tp′*)
    by(*simp*)
qed

When Turing machine A and B are combined and the execution of A can termination within *stp* steps, the combined machine A |+| B will eventually get into the starting state of machine B.

lemma *t-merge-pre-halt-same*:
  $[\![$*steps* (*Suc 0*, *tp*) *A stp* = (*0*, *tp′*); *t-correct A*; *t-correct B*$]\!]$
    ⟹ ∃ *stp*. *steps* (*Suc 0*, *tp*) (*A* |+| *B*) *stp* = (*Suc* (*length A div 2*), *tp′*)
proof −
  assume *a-wf*: *t-correct A*
  and *b-wf*: *t-correct B*
  and *a-ht*: *steps* (*Suc 0*, *tp*) *A stp* = (*0*, *tp′*)
  have *halt-point*: ∃ *stp*. ¬ *isS0* (*steps* (*Suc 0*, *tp*) *A stp*) ∧ *steps* (*Suc 0*, *tp*) *A* (*Suc stp*) = (*0*, *tp′*)
    using *a-ht*
    by(*erule-tac first-halt-point*)
  then obtain *stp′* **where** ¬ *isS0* (*steps* (*Suc 0*, *tp*) *A stp′*) ∧ *steps* (*Suc 0*, *tp*) *A* (*Suc stp′*) = (*0*, *tp′*)**..**
  hence *steps* (*Suc 0*, *tp*) (*A* |+| *B*) (*Suc stp′*) = (*Suc* (*length A div 2*), *tp′*)

```
    using a-wf
    apply(rule-tac t-merge-pre-halt-same', auto)
    done
  thus ?thesis ..
qed

lemma fetch-0: fetch p 0 b = (Nop, 0)
by(simp add: fetch.simps)

lemma [simp]: length (tshift B x) = length B
by(simp add: tshift.simps)

lemma [simp]: t-correct A ⟹ 2 * (length A div 2) = length A
apply(simp add: t-correct.simps iseven-def, auto)
done

lemma t-merge-fetch-snd:
  ⟦fetch B a b = (ac, ns); t-correct A; t-correct B; a > 0 ⟧
  ⟹ fetch (A |+| B) (a + length A div 2) b
  = (ac, if ns = 0 then 0 else ns + length A div 2)
apply(auto simp: fetch.simps t-add.simps split: if-splits block.splits)
apply(case-tac [!] a, simp-all)
apply(simp-all add: nth-append change-termi-state.simps tshift.simps)
done

lemma t-merge-snd-eq-step:
  ⟦tstep (s, l, r) B = (s', l', r'); t-correct A; t-correct B; s > 0⟧
    ⟹ tstep (s + length A div 2, l, r) (A |+| B) =
       (if s' = 0 then 0 else s' + length A div 2, l' ,r')
apply(simp add: tstep.simps)
apply(cases fetch B s (case r of [] ⇒ Bk | x # xs ⇒ x))
apply(auto simp: t-merge-fetch-snd)
apply(frule-tac [!] t-merge-fetch-snd, auto)
done
```

Relates the executions of TM B, one is when B is executed alone, the other is the execution when B is in the combined TM.

```
lemma t-merge-snd-eq-steps:
  ⟦t-correct A; t-correct B; steps (s, l, r) B stp = (s', l', r'); s > 0⟧
  ⟹ steps (s + length A div 2, l, r) (A |+| B) stp =
      (if s' = 0 then 0 else s' + length A div 2, l', r')
proof(induct stp arbitrary: s' l' r')
  case 0 thus ?case
    by(simp add: steps.simps)
next
  fix stp s' l' r'
  assume ind: ⋀s' l' r'. ⟦t-correct A; t-correct B; steps (s, l, r) B stp = (s', l',
r'); 0 < s⟧
                 ⟹ steps (s + length A div 2, l, r) (A |+| B) stp =
```

(*if* $s' = 0$ *then* 0 *else* s' + *length* A *div* 2, l', r')
and *h1*: *steps* (s, l, r) B (*Suc stp*) = (s', l', r')
and *h2*: *t-correct* A
and *h3*: *t-correct* B
and *h4*: 0 < s
from *h1* **show** *steps* (s + *length* A *div* 2, l, r) (A |+| B) (*Suc stp*)
= (*if* $s' = 0$ *then* 0 *else* s' + *length* A *div* 2, l', r')
proof(*simp only*: *tstep-red*, *cases steps* (s, l, r) B *stp*)
fix a b c
assume *h5*: *steps* (s, l, r) B *stp* = (a, b, c) *tstep* (*steps* (s, l, r) B *stp*) B = (s', l', r')
hence *h6*: (*steps* (s + *length* A *div* 2, l, r) (A |+| B) *stp*) =
((*if* a = 0 *then* 0 *else* a + *length* A *div* 2, b, c))
using *h2 h3 h4*
by(*rule-tac ind*, *auto*)
thus *tstep* (*steps* (s + *length* A *div* 2, l, r) (A |+| B) *stp*) (A |+| B) =
(*if* $s' = 0$ *then* 0 *else* s'+ *length* A *div* 2, l', r')
using *h5*
proof(*auto*)
assume *tstep* (0, b, c) B = (0, l', r') **thus** *tstep* (0, b, c) (A |+| B) = (0, l', r')
by(*simp add*: *tstep-0*)
next
assume *tstep* (0, b, c) B = (s', l', r') 0 < s'
thus *tstep* (0, b, c) (A |+| B) = (s' + *length* A *div* 2, l', r')
by(*simp add*: *tstep-0*)
next
assume *tstep* (a, b, c) B = (0, l', r') 0 < a
thus *tstep* (a + *length* A *div* 2, b, c) (A |+| B) = (0, l', r')
using *h2 h3*
by(*drule-tac t-merge-snd-eq-step*, *auto*)
next
assume *tstep* (a, b, c) B = (s', l', r') 0 < a 0 < s'
thus *tstep* (a + *length* A *div* 2, b, c) (A |+| B) = (s' + *length* A *div* 2, l', r')
using *h2 h3*
by(*drule-tac t-merge-snd-eq-step*, *auto*)
qed
qed
qed

lemma *t-merge-snd-halt-eq*:
⟦*steps* (*Suc* 0, *tp*) B *stp* = (0, tp'); *t-correct* A; *t-correct* B⟧
⟹ ∃ *stp*. *steps* (*Suc* (*length* A *div* 2), *tp*) (A |+| B) *stp* = (0, tp')
apply(*case-tac tp*, *cases* tp', *simp*)
apply(*drule-tac* s = *Suc* 0 **in** *t-merge-snd-eq-steps*, *auto*)
done

lemma *t-inj*: ⟦*steps* (*Suc* 0, *tp*) A *stpa* = (0, *tp1*); *steps* (*Suc* 0, *tp*) A *stpb* = (0, *tp2*)⟧

$\Longrightarrow$ *tp1 = tp2*
proof −
assume *h1*: *steps (Suc 0, tp) A stpa = (0, tp1)*
and *h2*: *steps (Suc 0, tp) A stpb = (0, tp2)*
thus *?thesis*
proof(*cases stpa < stpb*)
case *True* **thus** *?thesis*
using *h1 h2*
apply(*drule-tac less-imp-Suc-add, auto*)
apply(*simp del: add-Suc-right add-Suc add: add-Suc-right[THEN sym] steps-add steps-0*)
done
next
case *False* **thus** *?thesis*
using *h1 h2*
apply(*drule-tac leI*)
apply(*case-tac stpb = stpa, auto*)
apply(*subgoal-tac stpb < stpa*)
apply(*drule-tac less-imp-Suc-add, auto*)
apply(*simp del: add-Suc-right add-Suc add: add-Suc-right[THEN sym] steps-add steps-0*)
done
qed
qed

type-synonym *t-assert = tape ⇒ bool*

definition *t-imply* :: *t-assert ⇒ t-assert ⇒ bool* (- ⊢−> - [*0*, *0*] *100*)
where
t-imply a1 a2 = (∀ *tp. a1 tp* ⟶ *a2 tp*)

locale *turing-merge* =
fixes *A* :: *tprog* **and** *B* :: *tprog* **and** *P1* :: *t-assert*
and *P2* :: *t-assert*
and *P3* :: *t-assert*
and *P4* :: *t-assert*
and *Q1*:: *t-assert*
and *Q2* :: *t-assert*
assumes
A-wf : *t-correct A*
and *B-wf* : *t-correct B*
and *A-halt* : *P1 tp* $\Longrightarrow$ ∃ *stp. let (s, tp′) = steps (Suc 0, tp) A stp in s = 0* ∧ *Q1 tp′*
and *B-halt* : *P2 tp* $\Longrightarrow$ ∃ *stp. let (s, tp′) = steps (Suc 0, tp) B stp in s = 0* ∧ *Q2 tp′*
and *A-uhalt* : *P3 tp* $\Longrightarrow$ ¬ (∃ *stp. isS0 (steps (Suc 0, tp) A stp)*)
and *B-uhalt*: *P4 tp* $\Longrightarrow$ ¬ (∃ *stp. isS0 (steps (Suc 0, tp) B stp)*)
begin

The following lemma tries to derive the Hoare logic rule for sequentially combined TMs. It deals with the situtation when both A and B are terminated.

```
lemma t-merge-halt:
  assumes aimpb: Q1 ⊢−> P2
  shows P1 ⊢−>  λ tp. (∃ stp tp'. steps (Suc 0, tp) (A |+| B)  stp = (0, tp') ∧
Q2 tp')
proof(simp add: t-imply-def, auto)
  fix a b
  assume h: P1 (a, b)
  hence ∃ stp. let (s, tp') = steps (Suc 0, a, b) A stp in s = 0 ∧ Q1 tp'
    using A-halt by simp
  from this obtain stp1 where let (s, tp') = steps (Suc 0, a, b) A stp1 in s = 0
∧ Q1 tp' ..
  thus ∃stp aa ba. steps (Suc 0, a, b) (A |+| B) stp = (0, aa, ba) ∧ Q2 (aa, ba)
  proof(case-tac steps (Suc 0, a, b) A stp1, simp, erule-tac conjE)
    fix aa ba c
    assume g1: Q1 (ba, c)
      and g2: steps (Suc 0, a, b) A stp1 = (0, ba, c)
    hence P2 (ba, c)
      using aimpb apply(simp add: t-imply-def)
      done
    hence ∃ stp. let (s, tp') = steps (Suc 0, ba, c) B stp in s = 0 ∧ Q2 tp'
      using B-halt by simp
    from this obtain stp2 where let (s, tp') = steps (Suc 0, ba, c) B stp2 in s =
0 ∧ Q2 tp' ..
    thus ?thesis
    proof(case-tac steps (Suc 0, ba, c) B stp2, simp, erule-tac conjE)
      fix aa bb ca
      assume g3:  Q2 (bb, ca) steps (Suc 0, ba, c) B stp2 = (0, bb, ca)
      have ∃ stp. steps (Suc 0, a, b) (A |+| B) stp = (Suc (length A div 2), ba ,
c)
        using g2 A-wf B-wf
        by(rule-tac t-merge-pre-halt-same, auto)
      moreover have ∃ stp. steps (Suc (length A div 2), ba, c) (A |+| B) stp =
(0, bb, ca)
        using g3 A-wf B-wf
        apply(rule-tac t-merge-snd-halt-eq, auto)
        done
      ultimately show ∃stp aa ba. steps (Suc 0, a, b) (A |+| B) stp = (0, aa,
ba) ∧ Q2 (aa, ba)
        apply(erule-tac exE, erule-tac exE)
        apply(rule-tac x = stp + stpa in exI, simp add: steps-add)
        using g3 by simp
    qed
  qed
qed

lemma  t-merge-uhalt-tmp:
```

```
  assumes B-uh: ∀stp. ¬ isS0 (steps (Suc 0, b, c) B stp)
  and merge-ah: steps (Suc 0, tp) (A |+| B) stpa = (Suc (length A div 2), b, c)
  shows ∀ stp. ¬ isS0 (steps (Suc 0, tp) (A |+| B) stp)
  using B-uh merge-ah
apply(rule-tac allI)
apply(case-tac stp > stpa)
apply(erule-tac x = stp − stpa in allE)
apply(case-tac (steps (Suc 0, b, c) B (stp − stpa)), simp)
proof −
  fix stp a ba ca
  assume h1: ¬ isS0 (a, ba, ca) stpa < stp
  and h2: steps (Suc 0, b, c) B (stp − stpa) = (a, ba, ca)
  have steps (Suc 0 + length A div 2, b, c) (A |+| B) (stp − stpa) =
      (if a = 0 then 0 else a + length A div 2, ba, ca)
    using A-wf B-wf h2
    by(rule-tac t-merge-snd-eq-steps, auto)
  moreover have a > 0 using h1 by(simp add: isS0-def)
  moreover have ∃ stpb. stp = stpa + stpb
    using h1 by(rule-tac x = stp − stpa in exI, simp)
  ultimately show ¬ isS0 (steps (Suc 0, tp) (A |+| B) stp)
    using merge-ah
    by(auto simp: steps-add isS0-def)
next
  fix stp
  assume h: steps (Suc 0, tp) (A |+| B) stpa = (Suc (length A div 2), b, c) ¬
stpa < stp
  hence ∃ stpb. stpa = stp + stpb apply(rule-tac x = stpa − stp in exI, auto)
done
  thus ¬ isS0 (steps (Suc 0, tp) (A |+| B) stp)
    using h
    apply(auto)
     apply(cases steps (Suc 0, tp) (A |+| B) stp, simp add: steps-add isS0-def
steps-0)
    done
qed
```

The following lemma deals with the situation when TM B can not terminate.

```
lemma t-merge-uhalt:
  assumes aimpb: Q1 ⊢−> P4
  shows P1 ⊢−> λ tp. ¬ (∃ stp. isS0 (steps (Suc 0, tp) (A |+| B) stp))
proof(simp only: t-imply-def, rule-tac allI, rule-tac impI)
  fix tp
  assume init-asst: P1 tp
  show ¬ (∃stp. isS0 (steps (Suc 0, tp) (A |+| B) stp))
  proof −
    have ∃ stp. let (s, tp′) = steps (Suc 0, tp) A stp in s = 0 ∧ Q1 tp′
      using A-halt[of tp] init-asst
      by(simp)
    from this obtain stpx where let (s, tp′) = steps (Suc 0, tp) A stpx in s = 0
```

```
∧ Q1 tp' ..
    thus ?thesis
    proof(cases steps (Suc 0, tp) A stpx, simp, erule-tac conjE)
      fix a b c
      assume Q1 (b, c)
        and h3: steps (Suc 0, tp) A stpx = (0, b, c)
      hence h2: P4 (b, c)  using aimpb
        by(simp add: t-imply-def)
      have ∃ stp. steps (Suc 0, tp) (A |+| B) stp = (Suc (length A div 2), b, c)
        using h3 A-wf B-wf
        apply(rule-tac stp = stpx in t-merge-pre-halt-same, auto)
        done
       from this obtain stpa where h4:steps (Suc 0, tp) (A |+| B) stpa = (Suc
(length A div 2), b, c) ..
      have  ¬ (∃ stp. isS0 (steps (Suc 0, b, c) B stp))
        using B-uhalt [of (b, c)] h2 apply simp
        done
      from this and h4 show ∀ stp. ¬ isS0 (steps (Suc 0, tp) (A |+| B) stp)
        by(rule-tac t-merge-uhalt-tmp, auto)
    qed
  qed
qed
end

end
```

2 Undeciablity of the *Halting problem*

```
theory uncomputable
imports Main turing-basic
begin
```

The *Copying* TM, which duplicates its input.

```
definition tcopy :: tprog
where
tcopy ≡ [(W0, 0), (R, 2), (R, 3), (R, 2),
          (W1, 3), (L, 4), (L, 4), (L, 5), (R, 11), (R, 6),
          (R, 7), (W0, 6), (R, 7), (R, 8), (W1, 9), (R, 8),
          (L, 10), (L, 9), (L, 10), (L, 5), (R, 12), (R, 12),
          (W1, 13), (L, 14), (R, 12), (R, 12), (L, 15), (W0, 14),
          (R, 0), (L, 15)]
```

wipeLastBs tp removes all blanks at the end of tape *tp*.

```
fun wipeLastBs :: block list ⇒ block list
  where
  wipeLastBs bl = rev (dropWhile (λa. a = Bk) (rev bl))
```

fun *isBk* :: *block* ⇒ *bool*
where
isBk b = (*b* = *Bk*)

The following functions are used to expression invariants of *Copying* TM.

```
fun tcopy-F0 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F0 x tp = (let (ln, rn) = tp in
          list-all isBk ln & rn = replicate x Oc
                                  @ [Bk] @ replicate x Oc)

fun tcopy-F1 :: nat ⇒ tape ⇒ bool
  where
   tcopy-F1 x (ln, rn) = (ln = [] & rn = replicate x Oc)

fun tcopy-F2 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F2 0 tp = False |
  tcopy-F2 (Suc x) (ln, rn) = (length ln > 0 &
            ln @ rn = replicate (Suc x) Oc)

fun tcopy-F3 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F3 0 tp = False |
  tcopy-F3 (Suc x) (ln, rn) =
          (ln = Bk # replicate (Suc x) Oc & length rn <= 1)

fun tcopy-F4 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F4 0 tp = False |
  tcopy-F4 (Suc x) (ln, rn) =
          ((ln = replicate x Oc & rn = [Oc, Bk, Oc])
          | (ln = replicate (Suc x) Oc & rn = [Bk, Oc]))

fun tcopy-F5 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F5 0 tp = False |
  tcopy-F5 (Suc x) (ln, rn) =
        (if rn = [] then False
         else if hd rn = Bk then (ln = [] &
                  rn = Bk # (Oc # replicate (Suc x) Bk
                        @ replicate (Suc x) Oc))
         else if hd rn = Oc then
          (∃ n. ln = replicate (x - n) Oc
             & rn = Oc # (Oc # replicate n Bk @ replicate n Oc)
             & n > 0 & n <= x)
          else False)
```

```
fun tcopy-F6 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F6 0 tp = False |
  tcopy-F6 (Suc x) (ln, rn) =
          (∃ n. ln = replicate (Suc x −n) Oc
                & tl rn = replicate n Bk @ replicate n Oc
           & n > 0 & n <= x)

fun tcopy-F7 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F7 0 tp = False |
  tcopy-F7 (Suc x) (ln, rn) =
          (let lrn = (rev ln) @ rn in
          (∃ n. lrn = replicate ((Suc x) − n) Oc @
                    replicate (Suc n) Bk @ replicate n Oc
           & n > 0 & n <= x &
             length rn >= n & length rn <= 2 ∗ n ))

fun tcopy-F8 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F8 0 tp = False |
  tcopy-F8 (Suc x) (ln, rn) =
          (let lrn = (rev ln) @ rn in
          (∃ n. lrn = replicate ((Suc x) − n) Oc @
                   replicate (Suc n) Bk @ replicate n Oc
             & n > 0 & n <= x & length rn < n))

fun tcopy-F9 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F9 0 tp = False |
  tcopy-F9 (Suc x) (ln, rn) =
          (let lrn = (rev ln) @ rn in
          (∃ n. lrn = replicate (Suc (Suc x) − n) Oc
                      @ replicate n Bk @ replicate n Oc
           & n > Suc 0 & n <= Suc x & length rn > 0
                & length rn <= Suc n))

fun tcopy-F10 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F10 0 tp = False |
  tcopy-F10 (Suc x) (ln, rn) =
         (let lrn = (rev ln) @ rn in
          (∃ n. lrn = replicate (Suc (Suc x) − n) Oc
               @ replicate n Bk @ replicate n Oc & n > Suc 0
             & n <= Suc x & length rn > Suc n &
               length rn <= 2∗n + 1 ))

fun tcopy-F11 :: nat ⇒ tape ⇒ bool
```

```
  where
  tcopy-F11 0 tp = False |
  tcopy-F11 (Suc x) (ln, rn) =
          (ln = [Bk] & rn = Oc # replicate (Suc x) Bk
                       @ replicate (Suc x) Oc)

fun tcopy-F12 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F12 0 tp = False |
  tcopy-F12 (Suc x) (ln, rn) =
         (let lrn = ((rev ln) @ rn) in
         (∃ n. n > 0 & n <= Suc (Suc x)
       & lrn = Bk # replicate n Oc @ replicate (Suc (Suc x) − n) Bk
            @ replicate (Suc x) Oc
       & length ln = Suc n))

fun tcopy-F13 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F13 0 tp = False |
  tcopy-F13 (Suc x) (ln, rn) =
         (let lrn = ((rev ln) @ rn) in
         (∃ n. n > Suc 0 & n <= Suc (Suc x)
       & lrn = Bk # replicate n Oc @ replicate (Suc (Suc x) − n) Bk
            @ replicate (Suc x) Oc
       & length ln = n))

fun tcopy-F14 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F14 0 tp = False |
  tcopy-F14 (Suc x) (ln, rn) =
           (ln = replicate (Suc x) Oc @ [Bk] &
            tl rn = replicate (Suc x) Oc)

fun tcopy-F15 :: nat ⇒ tape ⇒ bool
  where
  tcopy-F15 0 tp = False |
  tcopy-F15 (Suc x) (ln, rn) =
          (let lrn = ((rev ln) @ rn) in
           lrn = Bk # replicate (Suc x) Oc @ [Bk] @
                replicate (Suc x) Oc & length ln <= (Suc x))
```

The following *inv-tcopy* is the invariant of the *Copying* TM.

```
fun inv-tcopy :: nat ⇒ t-conf ⇒ bool
  where
  inv-tcopy x c = (let (state, tp) = c in
                  if state = 0 then tcopy-F0 x tp
                  else if state = 1 then tcopy-F1 x tp
                  else if state = 2 then tcopy-F2 x tp
                  else if state = 3 then tcopy-F3 x tp
```

```
                 else if state = 4 then tcopy-F4 x tp
                 else if state = 5 then tcopy-F5 x tp
                 else if state = 6 then tcopy-F6 x tp
                 else if state = 7 then tcopy-F7 x tp
                 else if state = 8 then tcopy-F8 x tp
                 else if state = 9 then tcopy-F9 x tp
                 else if state = 10 then tcopy-F10 x tp
                 else if state = 11 then tcopy-F11 x tp
                 else if state = 12 then tcopy-F12 x tp
                 else if state = 13 then tcopy-F13 x tp
                 else if state = 14 then tcopy-F14 x tp
                 else if state = 15 then tcopy-F15 x tp
                 else False)
declare tcopy-F0.simps [simp del]
    tcopy-F1.simps [simp del]
    tcopy-F2.simps [simp del]
    tcopy-F3.simps [simp del]
    tcopy-F4.simps [simp del]
    tcopy-F5.simps [simp del]
    tcopy-F6.simps [simp del]
    tcopy-F7.simps [simp del]
    tcopy-F8.simps [simp del]
    tcopy-F9.simps [simp del]
    tcopy-F10.simps [simp del]
    tcopy-F11.simps [simp del]
    tcopy-F12.simps [simp del]
    tcopy-F13.simps [simp del]
    tcopy-F14.simps [simp del]
    tcopy-F15.simps [simp del]

lemma list-replicate-Bk[dest]: list-all isBk list ⟹
                    list = replicate (length list) Bk
apply(induct list)
apply(simp+)
done

lemma [simp]: dropWhile (λ a. a = b) (replicate x b @ ys) =
               dropWhile (λ a. a = b) ys
apply(induct x)
apply(simp)
apply(simp)
done

lemma [elim]: ⟦tstep (0, a, b) tcopy = (s, l, r); s ≠ 0⟧ ⟹ RR
apply(simp add: tstep.simps tcopy-def fetch.simps)
done

lemma [elim]: ⟦tstep (Suc 0, a, b) tcopy = (s, l, r); s ≠ 0; s ≠ 2⟧
             ⟹ RR
```

apply(*simp add*: *tstep.simps tcopy-def fetch.simps*)
apply(*simp split*: *block.splits list.splits*)
done

lemma [*elim*]: $[\![tstep\ (2,\ a,\ b)\ tcopy = (s,\ l,\ r);\ s \neq 2;\ s \neq 3]\!]$
$\Longrightarrow RR$
apply(*simp add*: *tstep.simps tcopy-def fetch.simps*)
apply(*simp split*: *block.splits list.splits*)
done

lemma [*elim*]: $[\![tstep\ (3,\ a,\ b)\ tcopy = (s,\ l,\ r);\ s \neq 3;\ s \neq 4]\!]$
$\Longrightarrow RR$
by(*simp add*: *tstep.simps tcopy-def fetch.simps*
split: *block.splits list.splits*)

lemma [*elim*]: $[\![tstep\ (4,\ a,\ b)\ tcopy = (s,\ l,\ r);\ s \neq 4;\ s \neq 5]\!]$
$\Longrightarrow RR$
by(*simp add*: *tstep.simps tcopy-def fetch.simps*
split: *block.splits list.splits*)

lemma [*elim*]: $[\![tstep\ (5,\ a,\ b)\ tcopy = (s,\ l,\ r);\ s \neq 6;\ s \neq 11]\!]$
$\Longrightarrow RR$
by(*simp add*: *tstep.simps tcopy-def fetch.simps*
split: *block.splits list.splits*)

lemma [*elim*]: $[\![tstep\ (6,\ a,\ b)\ tcopy = (s,\ l,\ r);\ s \neq 6;\ s \neq 7]\!]$
$\Longrightarrow RR$
by(*simp add*: *tstep.simps tcopy-def fetch.simps*
split: *block.splits list.splits*)

lemma [*elim*]: $[\![tstep\ (7,\ a,\ b)\ tcopy = (s,\ l,\ r);\ s \neq 7;\ s \neq 8]\!]$
$\Longrightarrow RR$
by(*simp add*: *tstep.simps tcopy-def fetch.simps*
split: *block.splits list.splits*)

lemma [*elim*]: $[\![tstep\ (8,\ a,\ b)\ tcopy = (s,\ l,\ r);\ s \neq 8;\ s \neq 9]\!]$
$\Longrightarrow RR$
by(*simp add*: *tstep.simps tcopy-def fetch.simps*
split: *block.splits list.splits*)

lemma [*elim*]: $[\![tstep\ (9,\ a,\ b)\ tcopy = (s,\ l,\ r);\ s \neq 9;\ s \neq 10]\!]$
$\Longrightarrow RR$
by(*simp add*: *tstep.simps tcopy-def fetch.simps*
split: *block.splits list.splits*)

lemma [*elim*]: $[\![tstep\ (10,\ a,\ b)\ tcopy = (s,\ l,\ r);\ s \neq 10;\ s \neq 5]\!]$
$\Longrightarrow RR$
by(*simp add*: *tstep.simps tcopy-def fetch.simps*
split: *block.splits list.splits*)

lemma [*elim*]: ⟦*tstep* (*11*, *a*, *b*) *tcopy* = (*s*, *l*, *r*); *s* ≠ *12*⟧ ⟹ *RR*
by(*simp add*: *tstep.simps tcopy-def fetch.simps*
split: *block.splits list.splits*)

lemma [*elim*]: ⟦*tstep* (*12*, *a*, *b*) *tcopy* = (*s*, *l*, *r*); *s* ≠ *13*; *s* ≠ *14*⟧
⟹ *RR*
by(*simp add*: *tstep.simps tcopy-def fetch.simps*
split: *block.splits list.splits*)

lemma [*elim*]: ⟦*tstep* (*13*, *a*, *b*) *tcopy* = (*s*, *l*, *r*); *s* ≠ *12*⟧ ⟹ *RR*
by(*simp add*: *tstep.simps tcopy-def fetch.simps*
split: *block.splits list.splits*)

lemma [*elim*]: ⟦*tstep* (*14*, *a*, *b*) *tcopy* = (*s*, *l*, *r*); *s* ≠ *14*; *s* ≠ *15*⟧
⟹ *RR*
by(*simp add*: *tstep.simps tcopy-def fetch.simps*
split: *block.splits list.splits*)

lemma [*elim*]: ⟦*tstep* (*15*, *a*, *b*) *tcopy* = (*s*, *l*, *r*); *s* ≠ *0*; *s* ≠ *15*⟧
⟹ *RR*
by(*simp add*: *tstep.simps tcopy-def fetch.simps*
split: *block.splits list.splits*)

lemma *min-Suc4*: *min* (*Suc* (*Suc x*)) *x* = *x*
by *auto*

lemma *takeWhile2replicate*:
∃ *n*. *takeWhile* (λ*a*. *a* = *b*) *list* = *replicate n b*
apply(*induct list*)
apply(*rule-tac x* = *0* **in** *exI*, *simp*)
apply(*auto*)
apply(*rule-tac x* = *Suc n* **in** *exI*, *simp*)
done

lemma *rev-replicate-same*: *rev* (*replicate x b*) = *replicate x b*
by(*simp*)

lemma *rev-equal*: *a* = *b* ⟹ *rev a* = *rev b*
by *simp*

lemma *rev-equal-rev*: *rev a* = *rev b* ⟹ *a* = *b*
by *simp*

lemma *rep-suc-rev*[*simp*]:*replicate n b* @ [*b*] = *replicate* (*Suc n*) *b*
apply(*rule rev-equal-rev*)
apply(*simp only*: *rev-append rev-replicate-same*)
apply(*auto*)
done

lemma *replicate-Cons-simp*: *b* # *replicate n b* @ *xs* =
replicate n b @ *b* # *xs*
apply(*simp*)
done

lemma [*elim*]: ⟦*tstep* (*14*, *b*, *c*) *tcopy* = (*15*, *ab*, *ba*);
tcopy-F14 x (*b*, *c*)⟧ ⟹ *tcopy-F15 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*auto simp*: *tstep.simps tcopy-def*
tcopy-F14.simps tcopy-F15.simps fetch.simps new-tape.simps
split: *if-splits list.splits block.splits*)
done

lemma *dropWhile-drophd*: ¬ *p a* ⟹
(*dropWhile p xs* @ (*a* # *as*)) = (*dropWhile p* (*xs* @ [*a*]) @ *as*)
apply(*induct xs*)
apply(*auto*)
done

lemma *dropWhile-append3*: ⟦¬ *p a*;
listall ((*dropWhile p xs*) @ [*a*]) *isBk*⟧ ⟹
listall (*dropWhile p* (*xs* @ [*a*])) *isBk*
apply(*drule-tac p* = *p* **and** *xs* = *xs* **and** *a* = *a* **in** *dropWhile-drophd*, *simp*)
done

lemma *takeWhile-append3*: ⟦¬*p a*; (*takeWhile p xs*) = *b*⟧
⟹ *takeWhile p* (*xs* @ (*a* # *as*)) = *b*
apply(*drule-tac P* = *p* **and** *xs* = *xs* **and** *x* = *a* **and** *l* = *as* **in**
takeWhile-tail)
apply(*simp*)
done

lemma *listall-append*: *list-all p* (*xs* @ *ys*) =
(*list-all p xs* ∧ *list-all p ys*)
apply(*induct xs*)
apply(*simp*+)
done

lemma [*elim*]: ⟦*tstep* (*15*, *b*, *c*) *tcopy* = (*15*, *ab*, *ba*);
tcopy-F15 x (*b*, *c*)⟧ ⟹ *tcopy-F15 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*auto simp*: *tstep.simps tcopy-F15.simps*
tcopy-def fetch.simps new-tape.simps
split: *if-splits list.splits block.splits*)
apply(*case-tac b*, *simp*+)
done

lemma [*elim*]: ⟦*tstep* (*14*, *b*, *c*) *tcopy* = (*14*, *ab*, *ba*);
tcopy-F14 *x* (*b*, *c*)⟧ ⟹ *tcopy-F14* *x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*auto simp*: *tcopy-F14.simps tcopy-def tstep.simps*
tcopy-F14.simps fetch.simps new-tape.simps
split: *if-splits list.splits block.splits*)
done

lemma [*intro*]: *list-all isBk* (*replicate x Bk*)
apply(*induct x*, *simp+*)
done

lemma [*elim*]: *list-all isBk* (*dropWhile* (λ*a*. *a* = *Oc*) *b*) ⟹
list-all isBk (*dropWhile* (λ*a*. *a* = *Oc*) (*tl b*))
apply(*case-tac b*, *auto split*: *if-splits*)
apply(*drule list-replicate-Bk*)
apply(*case-tac length list*, *auto*)
done

lemma [*elim*]: *list-all* (λ *a*. *a* = *Oc*) *list* ⟹
list = *replicate* (*length list*) *Oc*
apply(*induct list*)
apply(*simp+*)
done

lemma *append-length*: ⟦*as* @ *bs* = *cs* @ *ds*; *length bs* = *length ds*⟧
⟹ *as* = *cs* & *bs* = *ds*
apply(*auto*)
done

lemma *Suc-elim*: *Suc* (*Suc m*) − *n* = *Suc na* ⟹ *Suc m* − *n* = *na*
apply(*simp*)
done

lemma [*elim*]: ⟦*0* < *n*; *n* ≤ *Suc* (*Suc na*);
rev b @ *Oc* # *list* =
Bk # *replicate n Oc* @ *replicate* (*Suc* (*Suc na*) − *n*) *Bk* @
Oc # *replicate na Oc*;
length b = *Suc n*; *b* ≠ []⟧
⟹ *list-all isBk* (*dropWhile* (λ*a*. *a* = *Oc*) (*tl b*))
apply(*case-tac rev b*, *auto*)
done

lemma *b-cons-same*: *b*#*bs* = *replicate x a* @ *as* ⟹ *a* ≠ *b* ⟶ *x* = *0*
apply(*case-tac x*, *simp+*)
done

lemma *tcopy-tmp*[*elim*]:
⟦*0* < *n*; *n* ≤ *Suc* (*Suc na*);

rev b @ *Oc* # *list* =
Bk # *replicate n Oc* @ *replicate* (*Suc* (*Suc na*) − *n*) *Bk*
@ *Oc* # *replicate na Oc*; *length b* = *Suc n*; *b* ≠ []⟧
⟹ *list* = *replicate na Oc*
apply(*case-tac rev b*, *simp*+)
apply(*auto*)
apply(*frule b-cons-same*, *auto*)
done

lemma [*elim*]: ⟦*tstep* (*12*, *b*, *c*) *tcopy* = (*14*, *ab*, *ba*);
tcopy-F12 x (*b*, *c*)⟧ ⟹ *tcopy-F14 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*auto simp*:*tcopy-F12.simps tcopy-F14.simps*
tcopy-def tstep.simps fetch.simps new-tape.simps
split: *if-splits list.splits block.splits*)
apply(*frule tcopy-tmp*, *simp*+)
apply(*case-tac n*, *simp*+)
apply(*case-tac nata*, *simp*+)
done

lemma *replicate-app-Cons*: *replicate a b* @ *b* # *replicate c b*
= *replicate* (*Suc* (*a* + *c*)) *b*
apply(*simp*)
apply(*simp add*: *replicate-app-Cons-same*)
apply(*simp only*: *replicate-add*[*THEN sym*])
done

lemma *replicate-same-exE-pref*: ∃ *x*. *bs* @ (*b* # *cs*) = *replicate x y*
⟹ (∃ *n*. *bs* = *replicate n y*)
apply(*induct bs*)
apply(*rule-tac x* = *0* **in** *exI*, *simp*)
apply(*drule impI*)
apply(*erule impE*)
apply(*erule exE*, *simp*+)
apply(*case-tac x*, *auto*)
apply(*case-tac x*, *auto*)
apply(*rule-tac x* = *Suc n* **in** *exI*, *simp*+)
done

lemma *replicate-same-exE-inf*: ∃ *x*. *bs* @ (*b* # *cs*) = *replicate x y* ⟹ *b* = *y*
apply(*induct bs*, *auto*)
apply(*case-tac x*, *auto*)
apply(*drule impI*)
apply(*erule impE*)
apply(*case-tac x*, *simp*+)
done

lemma *replicate-same-exE-suf*:
∃ *x*. *bs* @ (*b* # *cs*) = *replicate x y* ⟹ ∃ *n*. *cs* = *replicate n y*

apply(*induct bs*, *auto*)
apply(*case-tac x*, *simp+*)
apply(*drule impI*, *erule impE*)
apply(*case-tac x*, *simp+*)
done

lemma *replicate-same-exE*: ∃ *x*. *bs* @ (*b* # *cs*) = *replicate x y*
⟹ (∃ *n*. *bs* = *replicate n y*) & (*b* = *y*) & (∃ *m*. *cs* = *replicate m y*)
apply(*rule conjI*)
apply(*drule replicate-same-exE-pref*, *simp*)
apply(*rule conjI*)
apply(*drule replicate-same-exE-inf*, *simp*)
apply(*drule replicate-same-exE-suf*, *simp*)
done

lemma *replicate-same*: *bs* @ (*b* # *cs*) = *replicate x y*
⟹ (∃ *n*. *bs* = *replicate n y*) & (*b* = *y*) & (∃ *m*. *cs* = *replicate m y*)
apply(*rule-tac replicate-same-exE*)
apply(*rule-tac x* = *x* **in** *exI*)
apply(*assumption*)
done

lemma [*elim*]: ⟦ *0* < *n*; *n* ≤ *Suc* (*Suc na*);
(*rev ab* @ *Bk* # *list*) = *Bk* # *replicate n Oc*
@ *replicate* (*Suc* (*Suc na*) − *n*) *Bk* @ *Oc* # *replicate na Oc*; *ab* ≠ []⟧
⟹ *n* ≤ *Suc na*
apply(*rule contrapos-pp*, *simp+*)
apply(*case-tac rev ab*, *simp+*)
apply(*auto*)
apply(*simp only*: *replicate-app-Cons*)
apply(*drule replicate-same*)
apply(*auto*)
done

lemma [*elim*]: ⟦*0* < *n*; *n* ≤ *Suc* (*Suc na*);
rev ab @ *Bk* # *list* = *Bk* # *replicate n Oc* @
replicate (*Suc* (*Suc na*) − *n*) *Bk* @ *Oc* # *replicate na Oc*;
length ab = *Suc n*; *ab* ≠ []⟧
⟹ *rev ab* @ *Oc* # *list* = *Bk* # *Oc* # *replicate n Oc* @
replicate (*Suc na* − *n*) *Bk* @ *Oc* # *replicate na Oc*
apply(*case-tac rev ab*, *simp+*)
apply(*auto*)
apply(*simp only*: *replicate-Cons-simp*)
apply(*simp*)
apply(*case-tac Suc* (*Suc na*) − *n*, *simp+*)
done

lemma [*elim*]: ⟦*tstep* (*12*, *b*, *c*) *tcopy* = (*13*, *ab*, *ba*);

tcopy-F12 x (*b*, *c*)⟧ ⟹ *tcopy-F13 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*simp-all add*:*tcopy-F12.simps tcopy-F13.simps tcopy-def tstep.simps fetch.simps new-tape.simps*)
apply(*simp split*: *if-splits list.splits block.splits*)
apply(*auto*)
done

lemma [*elim*]: ⟦*tstep* (*11*, *b*, *c*) *tcopy* = (*12*, *ab*, *ba*);
tcopy-F11 x (*b*, *c*)⟧ ⟹ *tcopy-F12 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*simp-all add*:*tcopy-F12.simps tcopy-F11.simps tcopy-def tstep.simps fetch.simps new-tape.simps*)
apply(*auto*)
done

lemma *equal-length*: *a* = *b* ⟹ *length a* = *length b*
by(*simp*)

lemma [*elim*]: ⟦*tstep* (*13*, *b*, *c*) *tcopy* = (*12*, *ab*, *ba*);
tcopy-F13 x (*b*, *c*)⟧ ⟹ *tcopy-F12 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*simp-all add*:*tcopy-F12.simps tcopy-F13.simps tcopy-def tstep.simps fetch.simps new-tape.simps*)
apply(*simp split*: *if-splits list.splits block.splits*)
apply(*auto*)
apply(*drule equal-length*, *simp*)
done

lemma [*elim*]: ⟦*tstep* (*5*, *b*, *c*) *tcopy* = (*11*, *ab*, *ba*);
tcopy-F5 x (*b*, *c*)⟧ ⟹ *tcopy-F11 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*simp-all add*:*tcopy-F11.simps tcopy-F5.simps tcopy-def tstep.simps fetch.simps new-tape.simps*)
apply(*simp split*: *if-splits list.splits block.splits*)
done

lemma *less-equal*: ⟦*length xs* <= *b*; ¬ *Suc* (*length xs*) <= *b*⟧ ⟹
length xs = *b*
apply(*simp*)
done

lemma *length-cons-same*: ⟦*xs* @ *b* # *ys* = *as* @ *bs*;
length ys = *length bs*⟧ ⟹ *xs* @ [*b*] = *as* & *ys* = *bs*
apply(*drule rev-equal*)
apply(*simp*)
apply(*auto*)
apply(*drule rev-equal*, *simp*)

```
done

lemma replicate-set-equal: ⟦ xs @ [a] = replicate n b; a ≠ b⟧ ⟹ RR
apply(drule rev-equal, simp)
apply(case-tac n, simp+)
done

lemma [elim]: ⟦tstep (10, b, c) tcopy = (10, ab, ba);
             tcopy-F10 x (b, c)⟧ ⟹ tcopy-F10 x (ab, ba)
apply(case-tac x)
apply(auto simp:tcopy-F10.simps tcopy-def tstep.simps fetch.simps
             new-tape.simps
         split: if-splits list.splits block.splits)
apply(rule-tac x = n in exI, auto)
apply(case-tac b, simp+)
apply(rule contrapos-pp, simp+)
apply(frule less-equal, simp+)
apply(drule length-cons-same, auto)
apply(drule replicate-set-equal, simp+)
done

lemma less-equal2: ¬ (n::nat) ≤ m ⟹ ∃ x. n = x + m & x > 0
apply(rule-tac x = n − m in exI)
apply(auto)
done

lemma replicate-tail-length[dest]:
   ⟦rev b @ Bk # list = xs @ replicate n Bk @ replicate n Oc⟧
 ⟹ length list >= n
apply(rule contrapos-pp, simp+)
apply(drule less-equal2, auto)
apply(drule rev-equal)
apply(simp add: replicate-add)
apply(auto)
apply(case-tac x, simp+)
done


lemma [elim]: ⟦tstep (9, b, c) tcopy = (10, ab, ba);
             tcopy-F9 x (b, c)⟧ ⟹ tcopy-F10 x (ab, ba)
apply(case-tac x)
apply(auto simp:tcopy-F10.simps tcopy-F9.simps tcopy-def
             tstep.simps fetch.simps new-tape.simps
         split: if-splits list.splits block.splits)
apply(rule-tac x = n in exI, auto)
apply(case-tac b, simp+)
done

lemma [elim]: ⟦tstep (9, b, c) tcopy = (9, ab, ba);
```

```
              tcopy-F9 x (b, c)⟧ ⟹ tcopy-F9 x (ab, ba)
apply(case-tac x)
apply(simp-all add: tcopy-F9.simps tcopy-def
                  tstep.simps fetch.simps new-tape.simps
  split: if-splits list.splits block.splits)
apply(rule-tac x = n in exI, auto)
apply(case-tac b, simp+)
apply(rule contrapos-pp, simp+)
apply(drule less-equal, simp+)
apply(drule rev-equal, auto)
apply(case-tac length list, simp+)
done

lemma app-cons-app-simp: xs @ a # bs @ ys = (xs @ [a]) @ bs @ ys
apply(simp)
done

lemma [elim]: ⟦tstep (8, b, c) tcopy = (9, ab, ba);
              tcopy-F8 x (b, c)⟧ ⟹ tcopy-F9 x (ab, ba)
apply(case-tac x)
apply(auto simp:tcopy-F8.simps tcopy-F9.simps tcopy-def
              tstep.simps fetch.simps new-tape.simps
  split: if-splits list.splits block.splits)
apply(rule-tac x = Suc n in exI, auto)
apply(rule-tac x = n in exI, auto)
apply(simp only: app-cons-app-simp)
apply(frule replicate-tail-length, simp)
done

lemma [elim]: ⟦tstep (8, b, c) tcopy = (8, ab, ba);
              tcopy-F8 x (b, c)⟧ ⟹ tcopy-F8 x (ab, ba)
apply(case-tac x)
apply(simp-all add:tcopy-F8.simps tcopy-def tstep.simps
                 fetch.simps new-tape.simps)
apply(simp split: if-splits list.splits block.splits)
apply(rule-tac x = n in exI, auto)
done

lemma ex-less-more: ⟦(x::nat) >= m ; x <= n⟧ ⟹
                    ∃ y. x = m + y & y <= n - m
by(rule-tac x = x -m in exI, auto)

lemma replicate-split: x <= n ⟹
                   (∃ y. replicate n b = replicate (y + x) b)
apply(rule-tac x = n - x in exI)
apply(simp)
done

lemma app-app-app-app-simp: as @ bs @ cs @ ds =
```

(*as* @ *bs*) @ (*cs* @ *ds*)
by *simp*

lemma *lengthtailsame-append-elim*:
⟦*as* @ *bs* = *cs* @ *ds*; *length bs* = *length ds*⟧ ⟹ *bs* = *ds*
apply(*simp*)
done

lemma *rep-suc*: *replicate* (*Suc n*) *x* = *replicate n x* @ [*x*]
by(*induct n*, *auto*)

lemma *length-append-diff-cons*:
⟦*b* @ *x* # *ba* = *xs* @ *replicate m y* @ *replicate n x*; *x* ≠ *y*;
Suc (*length ba*) ≤ *m* + *n*⟧
⟹ *length ba* < *n*
apply(*induct n arbitrary*: *ba*, *simp*)
apply(*drule-tac b* = *y* **in** *replicate-split*,
simp add: *replicate-add*, *erule exE*, *simp del*: *replicate.simps*)
proof −
fix *ba ya*
assume *h1*:
b @ *x* # *ba* = *xs* @ *y* # *replicate ya y* @ *replicate* (*length ba*) *y*
and *h2*: *x* ≠ *y*
thus *False*
using *append-eq-append-conv*[*of b* @ [*x*]
xs @ *y* # *replicate ya y ba replicate* (*length ba*) *y*]
apply(*auto*)
apply(*case-tac ya*, *simp*,
simp add: *rep-suc del*: *rep-suc-rev replicate.simps*)
done
next
fix *n ba*
assume *ind*: ⋀*ba*. ⟦*b* @ *x* # *ba* = *xs* @ *replicate m y* @ *replicate n x*;
x ≠ *y*; *Suc* (*length ba*) ≤ *m* + *n*⟧
⟹ *length ba* < *n*
and *h1*: *b* @ *x* # *ba* = *xs* @ *replicate m y* @ *replicate* (*Suc n*) *x*
and *h2*: *x* ≠ *y* **and** *h3*: *Suc* (*length ba*) ≤ *m* + *Suc n*
show *length ba* < *Suc n*
proof(*cases length ba*)
case *0* **thus** *?thesis* **by** *simp*
next
fix *nat*
assume *length ba* = *Suc nat*
hence ∃ *ys a*. *ba* = *ys* @ [*a*]
apply(*rule-tac x* = *butlast ba* **in** *exI*)
apply(*rule-tac x* = *last ba* **in** *exI*)
using *append-butlast-last-id*[*of ba*]
apply(*case-tac ba*, *auto*)
done

```
    from this obtain ys where ∃ a. ba = ys @ [a] ..
    from this obtain a where ba = ys @ [a] ..
    thus ?thesis
      using ind[of ys] h1 h2 h3
      apply(simp del: rep-suc-rev replicate.simps add: rep-suc)
      done
  qed
qed

lemma [elim]:
  ⟦b @ Oc # ba = xs @ Bk # replicate n Bk @ replicate n Oc;
   Suc (length ba) ≤ 2 ∗ n⟧
  ⟹ length ba < n
  apply(rule-tac length-append-diff-cons[of b Oc ba xs Suc n Bk n])
  apply(simp, simp, simp)
  done

lemma [elim]: ⟦tstep (7, b, c) tcopy = (8, ab, ba);
                tcopy-F7 x (b, c)⟧ ⟹ tcopy-F8 x (ab, ba)
apply(case-tac x)
apply(simp-all add:tcopy-F8.simps tcopy-F7.simps
                   tcopy-def tstep.simps fetch.simps new-tape.simps)
apply(simp split: if-splits list.splits block.splits)
apply(rule-tac x = n in exI, auto)
done

lemma [elim]: ⟦tstep (7, b, c) tcopy = (7, ab, ba);
                tcopy-F7 x (b, c)⟧ ⟹ tcopy-F7 x (ab, ba)
apply(case-tac x)
apply(auto simp:tcopy-F7.simps tcopy-def tstep.simps
                fetch.simps new-tape.simps
  split: if-splits list.splits block.splits)
apply(rule-tac x = n in exI, auto)
apply(simp only: app-cons-app-simp)
apply(frule replicate-tail-length, simp)
done

lemma Suc-more: n <= m ⟹ Suc m − n = Suc (m − n)
by simp

lemma [elim]: ⟦tstep (6, b, c) tcopy = (7, ab, ba);
                tcopy-F6 x (b, c)⟧ ⟹ tcopy-F7 x (ab, ba)
apply(case-tac x)
apply(auto simp:tcopy-F7.simps tcopy-F6.simps
                tcopy-def tstep.simps fetch.simps new-tape.simps
  split: if-splits list.splits block.splits)
done

lemma [elim]: ⟦tstep (6, b, c) tcopy = (6, ab, ba);
```

```
      tcopy-F6 x (b, c)⟧ ⟹ tcopy-F6 x (ab, ba)
apply(case-tac x)
apply(auto simp:tcopy-F6.simps tcopy-def tstep.simps
      new-tape.simps fetch.simps
  split: if-splits list.splits block.splits)
done

lemma [elim]: ⟦tstep (5, b, c) tcopy = (6, ab, ba);
      tcopy-F5 x (b, c)⟧ ⟹ tcopy-F6 x (ab, ba)
apply(case-tac x)
apply(auto simp:tcopy-F5.simps tcopy-F6.simps tcopy-def
      tstep.simps fetch.simps new-tape.simps
  split: if-splits list.splits block.splits)
apply(rule-tac x = n in exI, simp)
apply(rule-tac x = n in exI, simp)
apply(drule Suc-more, simp)
done

lemma ex-less-more2: ⟦(n::nat) < x ; x <= 2 * n⟧ ⟹
                     ∃ y. (x = n + y & y <= n)
apply(rule-tac x = x − n in exI)
apply(auto)
done

lemma app-app-app-simp: xs @ ys @ za = (xs @ ys) @ za
apply(simp)
done

lemma [elim]: rev xs = replicate n b ⟹ xs = replicate n b
using rev-replicate[of n b]
thm rev-equal
by(drule-tac rev-equal, simp)

lemma app-cons-tail-same[dest]:
  ⟦rev b @ Oc # list =
    replicate (Suc (Suc na) − n) Oc @ replicate n Bk @ replicate n Oc;
  Suc 0 < n; n ≤ Suc na; n < length list; length list ≤ 2 * n; b ≠ []⟧
  ⟹ list = replicate n Bk @ replicate n Oc
        & b = replicate (Suc na − n) Oc
using length-append-diff-cons[of rev b Oc list
      replicate (Suc (Suc na) − n) Oc n Bk n]
apply(case-tac length list = 2*n, simp)
using append-eq-append-conv[of rev b @ [Oc] replicate
    (Suc (Suc na) − n) Oc list replicate n Bk @ replicate n Oc]
apply(case-tac n, simp, simp add: Suc-more rep-suc
                  del: rep-suc-rev replicate.simps, auto)
done

lemma hd-replicate-false1: ⟦replicate x Oc ≠ [];
```

hd ($replicate$ x Oc) = Bk⟧ ⟹ RR
apply(*case-tac x*, *auto*)
done

lemma *hd-replicate-false2*: ⟦*replicate x Oc* ≠ []; *hd* (*replicate x Oc*) ≠ *Oc*⟧ ⟹ *RR*
apply(*case-tac x*, *auto*)
done

lemma *Suc-more-less*: ⟦$n \le Suc\ m$; $n >= m$⟧ ⟹ $n = m \mid n = Suc\ m$
apply(*auto*)
done

lemma *replicate-not-Nil*: *replicate x a* ≠ [] ⟹ $x > 0$
apply(*case-tac x*, *simp+*)
done

lemma [*elim*]: ⟦*tstep* (*10*, *b*, *c*) *tcopy* = (*5*, *ab*, *ba*);
tcopy-F10 x (*b*, *c*)⟧ ⟹ *tcopy-F5 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*auto simp*:*tcopy-F5.simps tcopy-F10.simps tcopy-def*
tstep.simps fetch.simps new-tape.simps
split: *if-splits list.splits block.splits*)
apply(*frule app-cons-tail-same*, *simp+*)
apply(*rule-tac x* = *n* **in** *exI*, *auto*)
done

lemma [*elim*]: ⟦*tstep* (*4*, *b*, *c*) *tcopy* = (*5*, *ab*, *ba*);
tcopy-F4 x (*b*, *c*)⟧ ⟹ *tcopy-F5 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*auto simp*:*tcopy-F5.simps tcopy-F4.simps tcopy-def*
tstep.simps fetch.simps new-tape.simps
split: *if-splits list.splits block.splits*)
done

lemma [*elim*]: ⟦*tstep* (*3*, *b*, *c*) *tcopy* = (*4*, *ab*, *ba*);
tcopy-F3 x (*b*, *c*)⟧ ⟹ *tcopy-F4 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*auto simp*:*tcopy-F3.simps tcopy-F4.simps*
tcopy-def tstep.simps fetch.simps new-tape.simps
split: *if-splits list.splits block.splits*)
done

lemma [*elim*]: ⟦*tstep* (*4*, *b*, *c*) *tcopy* = (*4*, *ab*, *ba*);
tcopy-F4 x (*b*, *c*)⟧ ⟹ *tcopy-F4 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*auto simp*:*tcopy-F3.simps tcopy-F4.simps*
tcopy-def tstep.simps fetch.simps new-tape.simps
split: *if-splits list.splits block.splits*)

done

lemma [*elim*]: ⟦*tstep* (*3*, *b*, *c*) *tcopy* = (*3*, *ab*, *ba*);
tcopy-F3 x (*b*, *c*)⟧ ⟹ *tcopy-F3 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*auto simp*:*tcopy-F3.simps tcopy-F4.simps*
tcopy-def tstep.simps fetch.simps new-tape.simps
split: *if-splits list.splits block.splits*)
done

lemma *replicate-cons-back*: *y* # *replicate x y* = *replicate* (*Suc x*) *y*
apply(*simp*)
done

lemma *replicate-cons-same*: *bs* @ (*b* # *cs*) = *y* # *replicate x y* ⟹
(∃ *n*. *bs* = *replicate n y*) & (*b* = *y*) & (∃ *m*. *cs* = *replicate m y*)
apply(*simp only*: *replicate-cons-back*)
apply(*drule-tac replicate-same*)
apply(*simp*)
done

lemma [*elim*]: ⟦*tstep* (*2*, *b*, *c*) *tcopy* = (*3*, *ab*, *ba*);
tcopy-F2 x (*b*, *c*)⟧ ⟹ *tcopy-F3 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*auto simp*:*tcopy-F3.simps tcopy-F2.simps*
tcopy-def tstep.simps fetch.simps new-tape.simps
split: *if-splits list.splits block.splits*)
apply(*drule replicate-cons-same*, *auto*)+
done

lemma [*elim*]: ⟦*tstep* (*2*, *b*, *c*) *tcopy* = (*2*, *ab*, *ba*);
tcopy-F2 x (*b*, *c*)⟧ ⟹ *tcopy-F2 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*auto simp*:*tcopy-F3.simps tcopy-F2.simps*
tcopy-def tstep.simps fetch.simps new-tape.simps
split: *if-splits list.splits block.splits*)
apply(*frule replicate-cons-same*, *auto*)
apply(*simp add*: *replicate-app-Cons-same*)
done

lemma [*elim*]: ⟦*tstep* (*Suc 0*, *b*, *c*) *tcopy* = (*2*, *ab*, *ba*);
tcopy-F1 x (*b*, *c*)⟧ ⟹ *tcopy-F2 x* (*ab*, *ba*)
apply(*case-tac x*)
apply(*simp-all add*:*tcopy-F2.simps tcopy-F1.simps*
tcopy-def tstep.simps fetch.simps new-tape.simps)
apply(*auto*)
done

lemma [*elim*]: ⟦*tstep* (*Suc 0*, *b*, *c*) *tcopy* = (*0*, *ab*, *ba*);

```
              tcopy-F1 x (b, c)⟧ ⟹ tcopy-F0 x (ab, ba)
apply(case-tac x)
apply(simp-all add:tcopy-F0.simps tcopy-F1.simps
                tcopy-def tstep.simps fetch.simps new-tape.simps)
done

lemma ex-less: Suc x <= y ⟹ ∃ z. y = x + z & z > 0
apply(rule-tac x = y − x in exI, auto)
done

lemma [elim]: ⟦xs @ Bk # ba =
  Bk # Oc # replicate n Oc @ Bk # Oc # replicate n Oc;
  length xs ≤ Suc n; xs ≠ []⟧ ⟹ RR
apply(case-tac xs, auto)
apply(case-tac list, auto)
apply(drule ex-less, auto)
apply(simp add: replicate-add)
apply(auto)
apply(case-tac z, simp+)
done

lemma [elim]: ⟦tstep (15, b, c) tcopy = (0, ab, ba);
              tcopy-F15 x (b, c)⟧ ⟹ tcopy-F0 x (ab, ba)
apply(case-tac x)
apply(auto simp: tcopy-F15.simps tcopy-F0.simps
               tcopy-def tstep.simps new-tape.simps fetch.simps
          split: if-splits list.splits block.splits)
done

lemma [elim]: ⟦tstep (0, b, c) tcopy = (0, ab, ba);
              tcopy-F0 x (b, c)⟧ ⟹ tcopy-F0 x (ab, ba)
apply(case-tac x)
apply(simp-all add: tcopy-F0.simps tcopy-def
                 tstep.simps new-tape.simps fetch.simps)
done

declare tstep.simps[simp del]
```

Finally establishes the invariant of Copying TM, which is used to dervie the parital correctness of Copying TM.

```
lemma inv-tcopy-step:inv-tcopy x c ⟹ inv-tcopy x (tstep c tcopy)
apply(induct c)
apply(auto split: if-splits block.splits list.splits taction.splits)
apply(auto simp: tstep.simps tcopy-def fetch.simps new-tape.simps
  split: if-splits list.splits block.splits taction.splits)
done

declare inv-tcopy.simps[simp del]
```

Invariant under mult-step execution.

lemma *inv-tcopy-steps*:
  inv-tcopy x (*steps* (*Suc 0*, [], *replicate x Oc*) *tcopy stp*)
apply(*induct stp*)
apply(*simp add*: *tstep.simps tcopy-def steps.simps*
        tcopy-F1.simps inv-tcopy.simps)
apply(*drule-tac inv-tcopy-step*, *simp add*: *tstep-red*)
done

The followng lemmas gives the parital correctness of Copying TM.

theorem *inv-tcopy-rs*:
  steps (*Suc 0*, [], *replicate x Oc*) *tcopy stp* = (*0*, *l*, *r*)
  $\Longrightarrow \exists$ *n*. *l* = *replicate n Bk* $\wedge$
      r = *replicate x Oc* @ *Bk* # *replicate x Oc*
apply(*insert inv-tcopy-steps*[*of x stp*])
apply(*auto simp*: *inv-tcopy.simps tcopy-F0.simps isBk.simps*)
done

3 The following definitions are used to construct the measure function used to show the termnation of Copying TM.

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 *lex-square* ::
  ((*nat* × *nat* × *nat* × *nat*) × (*nat* × *nat* × *nat* × *nat*)) *set*
  where
lex-square ≡ *less-than* <∗*lex*∗> *lex-triple*

lemma *wf-lex-triple*: *wf lex-triple*
  by (*auto intro*:*wf-lex-prod simp*:*lex-triple-def lex-pair-def*)

lemma *wf-lex-square*: *wf lex-square*
  by (*auto intro*:*wf-lex-prod*
      simp:*lex-triple-def lex-square-def lex-pair-def*)

A measurement functions used to show the termination of copying machine:

fun *tcopy-phase* :: *t-conf* ⇒ *nat*
  where
  tcopy-phase c = (*let* (*state*, *tp*) = *c in*

if state > 0 & state <= 4 then 5
else if state >=5 & state <= 10 then 4
else if state = 11 then 3
else if state = 12 | state = 13 then 2
else if state = 14 | state = 15 then 1
else 0)

fun *tcopy-phase4-stage* :: *tape* ⇒ *nat*
where
tcopy-phase4-stage (*ln*, *rn*) =
(*let lrn* = (*rev ln*) @ *rn*
in length (*takeWhile* (λ*a*. *a* = *Oc*) *lrn*))

fun *tcopy-stage* :: *t-conf* ⇒ *nat*
where
tcopy-stage c = (*let* (*state*, *ln*, *rn*) = *c in*
if tcopy-phase c = 5 then 0
else if tcopy-phase c = 4 then
tcopy-phase4-stage (*ln*, *rn*)
else if tcopy-phase c = 3 then 0
else if tcopy-phase c = 2 then length rn
else if tcopy-phase c = 1 then 0
else 0)

fun *tcopy-phase4-state* :: *t-conf* ⇒ *nat*
where
tcopy-phase4-state c = (*let* (*state*, *ln*, *rn*) = *c in*
if state = 6 & hd rn = Oc then 0
else if state = 5 then 1
else 12 − state)

fun *tcopy-state* :: *t-conf* ⇒ *nat*
where
tcopy-state c = (*let* (*state*, *ln*, *rn*) = *c in*
if tcopy-phase c = 5 then 4 − state
else if tcopy-phase c = 4 then
tcopy-phase4-state c
else if tcopy-phase c = 3 then 0
else if tcopy-phase c = 2 then 13 − state
else if tcopy-phase c = 1 then 15 − state
else 0)

fun *tcopy-step2* :: *t-conf* ⇒ *nat*
where
tcopy-step2 (*s*, *l*, *r*) = *length r*

fun *tcopy-step3* :: *t-conf* ⇒ *nat*
where
tcopy-step3 (*s*, *l*, *r*) = (*if r* = [] | *r* = [*Bk*] *then Suc 0 else 0*)

```
fun tcopy-step4 :: t-conf ⇒ nat
  where
  tcopy-step4 (s, l, r) = length l

fun tcopy-step7 :: t-conf ⇒ nat
  where
  tcopy-step7 (s, l, r) = length r

fun tcopy-step8 :: t-conf ⇒ nat
  where
  tcopy-step8 (s, l, r) = length r

fun tcopy-step9 :: t-conf ⇒ nat
  where
  tcopy-step9 (s, l, r) = length l

fun tcopy-step10 :: t-conf ⇒ nat
  where
  tcopy-step10 (s, l, r) = length l

fun tcopy-step14 :: t-conf ⇒ nat
  where
  tcopy-step14 (s, l, r) = (case hd r of
                             Oc ⇒ 1 |
                             Bk   ⇒ 0)

fun tcopy-step15 :: t-conf ⇒ nat
  where
  tcopy-step15 (s, l, r) = length l

fun tcopy-step :: t-conf ⇒ nat
  where
  tcopy-step c = (let (state, ln, rn) = c in
            if state = 0 | state = 1 | state = 11 |
               state = 5 | state = 6 | state = 12 | state = 13 then 0
                  else if state = 2 then tcopy-step2 c
                  else if state = 3 then tcopy-step3 c
                  else if state = 4 then tcopy-step4 c
                  else if state = 7 then tcopy-step7 c
                  else if state = 8 then tcopy-step8 c
                  else if state = 9 then tcopy-step9 c
                  else if state = 10 then tcopy-step10 c
                  else if state = 14 then tcopy-step14 c
                  else if state = 15 then tcopy-step15 c
                  else 0)
```

The measure function used to show the termination of Copying TM.

```
fun tcopy-measure :: t-conf ⇒ (nat * nat * nat * nat)
```

where
tcopy-measure c =
(*tcopy-phase c*, *tcopy-stage c*, *tcopy-state c*, *tcopy-step c*)

definition *tcopy-LE* :: ((*nat* × *block list* × *block list*) ×
(*nat* × *block list* × *block list*)) *set*
where
tcopy-LE ≡ (*inv-image lex-square tcopy-measure*)

lemma *wf-tcopy-le*: *wf tcopy-LE*
by(*auto intro*:*wf-inv-image wf-lex-square simp*:*tcopy-LE-def*)

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

declare *tcopy-phase.simps*[*simp del*] *tcopy-stage.simps*[*simp del*]
tcopy-state.simps[*simp del*] *tcopy-step.simps*[*simp del*]
inv-tcopy.simps[*simp del*]
declare *tcopy-F0.simps* [*simp*]
tcopy-F1.simps [*simp*]
tcopy-F2.simps [*simp*]
tcopy-F3.simps [*simp*]
tcopy-F4.simps [*simp*]
tcopy-F5.simps [*simp*]
tcopy-F6.simps [*simp*]
tcopy-F7.simps [*simp*]
tcopy-F8.simps [*simp*]
tcopy-F9.simps [*simp*]
tcopy-F10.simps [*simp*]
tcopy-F11.simps [*simp*]
tcopy-F12.simps [*simp*]
tcopy-F13.simps [*simp*]
tcopy-F14.simps [*simp*]
tcopy-F15.simps [*simp*]
fetch.simps[*simp*]
new-tape.simps[*simp*]
lemma [*elim*]: *tcopy-F1 x* (*b*, *c*) ⟹
(*tstep* (*Suc 0*, *b*, *c*) *tcopy*, *Suc 0*, *b*, *c*) ∈ *tcopy-LE*
apply(*simp add*: *tcopy-F1.simps tstep.simps tcopy-def tcopy-LE-def*
lex-square-def lex-triple-def lex-pair-def tcopy-phase.simps
tcopy-stage.simps tcopy-state.simps tcopy-step.simps)
apply(*simp split*: *if-splits list.splits block.splits taction.splits*)
done

lemma [*elim*]: *tcopy-F2 x* (*b*, *c*) ⟹
(*tstep* (*2*, *b*, *c*) *tcopy*, *2*, *b*, *c*) ∈ *tcopy-LE*
apply(*simp add*:*tstep.simps tcopy-def tcopy-LE-def lex-square-def*
lex-triple-def lex-pair-def tcopy-phase.simps tcopy-stage.simps
tcopy-state.simps tcopy-step.simps)

apply(*simp split*: *if-splits list.splits block.splits taction.splits*)
done

lemma [*elim*]: *tcopy-F3 x* (*b*, *c*) $\Longrightarrow$
(*tstep* (*3*, *b*, *c*) *tcopy*, *3*, *b*, *c*) $\in$ *tcopy-LE*
apply(*simp add*: *tstep.simps tcopy-def tcopy-LE-def lex-square-def lex-triple-def lex-pair-def tcopy-phase.simps tcopy-stage.simps tcopy-state.simps tcopy-step.simps*)
apply(*simp split*: *if-splits list.splits block.splits taction.splits*)
apply(*auto*)
apply(*case-tac x*, *simp+*)
done

lemma [*elim*]: *tcopy-F4 x* (*b*, *c*) $\Longrightarrow$
(*tstep* (*4*, *b*, *c*) *tcopy*, *4*, *b*, *c*) $\in$ *tcopy-LE*
apply(*case-tac x*, *simp*)
apply(*simp add*: *tcopy-F4.simps tstep.simps tcopy-def tcopy-LE-def lex-square-def lex-triple-def lex-pair-def tcopy-phase.simps tcopy-stage.simps tcopy-state.simps tcopy-step.simps*)
apply(*simp split*: *if-splits list.splits block.splits taction.splits*)
apply(*auto*)
done

lemma[*simp*]: *takeWhile* ($\lambda a.\ a = b$) (*replicate x b* @ *ys*) =
replicate x b @ (*takeWhile* ($\lambda a.\ a = b$) *ys*)
apply(*induct x*)
apply(*simp+*)
done

lemma [*elim*]: *tcopy-F5 x* (*b*, *c*) $\Longrightarrow$
(*tstep* (*5*, *b*, *c*) *tcopy*, *5*, *b*, *c*) $\in$ *tcopy-LE*
apply(*case-tac x*, *simp*)
apply(*simp add*: *tstep.simps tcopy-def tcopy-LE-def lex-square-def lex-triple-def lex-pair-def tcopy-phase.simps*)
apply(*simp split*: *if-splits list.splits block.splits taction.splits*)
apply(*auto*)
apply(*simp-all add*: *tcopy-phase.simps tcopy-stage.simps tcopy-state.simps*)
done

lemma [*elim*]: $[\![$*replicate n x* = []; $n > 0]\!] \Longrightarrow RR$
apply(*case-tac n*, *simp+*)
done

lemma [*elim*]: *tcopy-F6 x* (*b*, *c*) $\Longrightarrow$
(*tstep* (*6*, *b*, *c*) *tcopy*, *6*, *b*, *c*) $\in$ *tcopy-LE*
apply(*case-tac x*, *simp*)
apply(*simp add*: *tstep.simps tcopy-def tcopy-LE-def lex-square-def lex-triple-def lex-pair-def*

tcopy-phase.simps)
apply(*simp split*: *if-splits list.splits block.splits taction.splits*)
apply(*auto*)
apply(*simp-all add*: *tcopy-phase.simps tcopy-stage.simps*
tcopy-state.simps tcopy-step.simps)
done

lemma [*elim*]: *tcopy-F7 x* (*b*, *c*) $\Longrightarrow$
(*tstep* (*7*, *b*, *c*) *tcopy*, *7*, *b*, *c*) $\in$ *tcopy-LE*
apply(*case-tac x*, *simp*)
apply(*simp add*: *tstep.simps tcopy-def tcopy-LE-def lex-square-def*
lex-triple-def lex-pair-def tcopy-phase.simps)
apply(*simp split*: *if-splits list.splits block.splits taction.splits*)
apply(*auto*)
apply(*simp-all add*: *tcopy-phase.simps tcopy-stage.simps*
tcopy-state.simps tcopy-step.simps)
done

lemma [*elim*]: *tcopy-F8 x* (*b*, *c*) $\Longrightarrow$
(*tstep* (*8*, *b*, *c*) *tcopy*, *8*, *b*, *c*) $\in$ *tcopy-LE*
apply(*case-tac x*, *simp*)
apply(*simp add*: *tstep.simps tcopy-def tcopy-LE-def lex-square-def*
lex-triple-def lex-pair-def tcopy-phase.simps)
apply(*simp split*: *if-splits list.splits block.splits taction.splits*)
apply(*auto*)
apply(*simp-all add*: *tcopy-phase.simps tcopy-stage.simps*
tcopy-state.simps tcopy-step.simps)
apply(*simp only*: *app-cons-app-simp*, *frule replicate-tail-length*, *simp*)
done

lemma *app-app-app-equal*: *xs* @ *ys* @ *zs* = (*xs* @ *ys*) @ *zs*
by *simp*

lemma *append-cons-assoc*: *as* @ *b* # *bs* = (*as* @ [*b*]) @ *bs*
apply(*rule rev-equal-rev*)
apply(*simp*)
done

lemma *rev-tl-hd-merge*: *bs* $\neq$ [] $\Longrightarrow$
rev (*tl bs*) @ *hd bs* # *as* = *rev bs* @ *as*
apply(*rule rev-equal-rev*)
apply(*simp*)
done

lemma [*elim*]: *tcopy-F9 x* (*b*, *c*) $\Longrightarrow$
(*tstep* (*9*, *b*, *c*) *tcopy*, *9*, *b*, *c*) $\in$ *tcopy-LE*
apply(*case-tac x*, *simp*)
apply(*simp add*: *tstep.simps tcopy-def tcopy-LE-def lex-square-def*
lex-triple-def lex-pair-def tcopy-phase.simps)

```
apply(simp split: if-splits list.splits block.splits taction.splits)
apply(auto)
apply(simp-all add: tcopy-phase.simps tcopy-stage.simps
          tcopy-state.simps tcopy-step.simps)
apply(drule-tac bs = b and as = Bk # list in rev-tl-hd-merge)
apply(simp)
apply(drule-tac bs = b and as = Oc # list in rev-tl-hd-merge)
apply(simp)
done

lemma [elim]: tcopy-F10 x (b, c) ⟹
      (tstep (10, b, c) tcopy, 10, b, c) ∈ tcopy-LE
apply(case-tac x, simp)
apply(simp add: tstep.simps tcopy-def tcopy-LE-def lex-square-def
        lex-triple-def lex-pair-def tcopy-phase.simps)
apply(simp split: if-splits list.splits block.splits taction.splits)
apply(auto)
apply(simp-all add: tcopy-phase.simps tcopy-stage.simps
          tcopy-state.simps tcopy-step.simps)
apply(drule-tac bs = b and as = Bk # list in rev-tl-hd-merge)
apply(simp)
apply(drule-tac bs = b and as = Oc # list in rev-tl-hd-merge)
apply(simp)
done

lemma [elim]: tcopy-F11 x (b, c) ⟹
      (tstep (11, b, c) tcopy, 11, b, c) ∈ tcopy-LE
apply(case-tac x, simp)
apply(simp add: tstep.simps tcopy-def tcopy-LE-def
        lex-square-def lex-triple-def lex-pair-def
        tcopy-phase.simps)
done

lemma [elim]: tcopy-F12 x (b, c) ⟹
      (tstep (12, b, c) tcopy, 12, b, c) ∈ tcopy-LE
apply(case-tac x, simp)
apply(simp add: tstep.simps tcopy-def tcopy-LE-def lex-square-def
        lex-triple-def lex-pair-def tcopy-phase.simps)
apply(simp split: if-splits list.splits block.splits taction.splits)
apply(auto)
apply(simp-all add: tcopy-phase.simps tcopy-stage.simps
          tcopy-state.simps tcopy-step.simps)
done

lemma [elim]: tcopy-F13 x (b, c) ⟹
      (tstep (13, b, c) tcopy, 13, b, c) ∈ tcopy-LE
apply(case-tac x, simp)
apply(simp add: tstep.simps tcopy-def tcopy-LE-def lex-square-def
        lex-triple-def lex-pair-def tcopy-phase.simps)
```

```
apply(simp split: if-splits list.splits block.splits taction.splits)
apply(auto)
apply(simp-all add: tcopy-phase.simps tcopy-stage.simps
                 tcopy-state.simps tcopy-step.simps)
apply(drule equal-length, simp)+
done

lemma [elim]: tcopy-F14 x (b, c) ⟹
         (tstep (14, b, c) tcopy, 14, b, c) ∈ tcopy-LE
apply(case-tac x, simp)
apply(simp add: tstep.simps tcopy-def tcopy-LE-def lex-square-def
              lex-triple-def lex-pair-def tcopy-phase.simps)
apply(simp split: if-splits list.splits block.splits taction.splits)
apply(auto)
apply(simp-all add: tcopy-phase.simps tcopy-stage.simps
                 tcopy-state.simps tcopy-step.simps)
done

lemma [elim]: tcopy-F15 x (b, c) ⟹
        (tstep (15, b, c) tcopy, 15, b, c) ∈ tcopy-LE
apply(case-tac x, simp)
apply(simp add: tstep.simps tcopy-def tcopy-LE-def lex-square-def
              lex-triple-def lex-pair-def tcopy-phase.simps )
apply(simp split: if-splits list.splits block.splits taction.splits)
apply(auto)
apply(simp-all add: tcopy-phase.simps tcopy-stage.simps
                 tcopy-state.simps tcopy-step.simps)
done

lemma tcopy-wf-step:⟦a > 0; inv-tcopy x (a, b, c)⟧ ⟹
                 (tstep (a, b, c) tcopy, (a, b, c)) ∈ tcopy-LE
apply(simp add:inv-tcopy.simps split: if-splits, auto)
apply(auto simp: tstep.simps tcopy-def  tcopy-LE-def lex-square-def
              lex-triple-def lex-pair-def tcopy-phase.simps
              tcopy-stage.simps tcopy-state.simps tcopy-step.simps
          split: if-splits list.splits block.splits taction.splits)
done

lemma tcopy-wf:
∀ n. let nc = steps (Suc 0, [], replicate x Oc) tcopy n in
      let Sucnc = steps (Suc 0, [], replicate x Oc) tcopy (Suc n) in
  ¬ isS0 nc ⟶ ((Sucnc, nc) ∈ tcopy-LE)
proof(rule allI, case-tac
   steps (Suc 0, [], replicate x Oc) tcopy n, auto simp: tstep-red)
  fix n a b c
  assume h: ¬ isS0 (a, b, c)
       steps (Suc 0, [], replicate x Oc) tcopy n = (a, b, c)
  hence  inv-tcopy x (a, b, c)
    using inv-tcopy-steps[of x n] by(simp)
```

thus (*tstep* (*a*, *b*, *c*) *tcopy*, *a*, *b*, *c*) ∈ *tcopy-LE*
using *h*
by(*rule-tac tcopy-wf-step*, *auto simp*: *isS0-def*)
qed

The termination of Copying TM:

lemma *tcopy-halt*:
∃ *n*. *isS0* (*steps* (*Suc 0*, [], *replicate x Oc*) *tcopy n*)
apply(*insert halt-lemma*
[*of tcopy-LE isS0 steps* (*Suc 0*, [], *replicate x Oc*) *tcopy*])
apply(*insert tcopy-wf* [*of x*])
apply(*simp only*: *Let-def*)
apply(*insert wf-tcopy-le*)
apply(*simp*)
done

The total correntess of Copying TM:

theorem *tcopy-halt-rs*: ∃ *stp m*.
steps (*Suc 0*, [], *replicate x Oc*) *tcopy stp* =
(*0*, *replicate m Bk*, *replicate x Oc* @ *Bk* # *replicate x Oc*)
using *tcopy-halt*[*of x*]
proof(*erule-tac exE*)
fix *n*
assume *h*: *isS0* (*steps* (*Suc 0*, [], *replicate x Oc*) *tcopy n*)
have *inv-tcopy x* (*steps* (*Suc 0*, [], *replicate x Oc*) *tcopy n*)
using *inv-tcopy-steps*[*of x n*] **by** *simp*
thus *?thesis*
using *h*
apply(*cases* (*steps* (*Suc 0*, [], *replicate x Oc*) *tcopy n*),
auto simp: *isS0-def inv-tcopy.simps*)
apply(*rule-tac x* = *n* **in** *exI*, *auto*)
done
qed

4 The *Dithering* Turing Machine

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

definition *dither* :: *tprog*
where
dither ≡ [(*W0*, *1*), (*R*, *2*), (*L*, *1*), (*L*, *0*)]

lemma *dither-halt-rs*:
∃ *stp*. *steps* (*Suc 0*, Bk^m, [*Oc*, *Oc*]) *dither stp* =
(*0*, Bk^m, [*Oc*, *Oc*])
apply(*rule-tac x* = *Suc* (*Suc* (*Suc 0*)) **in** *exI*)
apply(*simp add*: *dither-def steps.simps*

tstep.simps fetch.simps new-tape.simps)
done

lemma *dither-unhalt-state*:
 (*steps* (*Suc 0*, Bk^m, [*Oc*]) *dither stp* =
 (*Suc 0*, Bk^m, [*Oc*])) ∨
 (*steps* (*Suc 0*, Bk^m, [*Oc*]) *dither stp* = (*2*, *Oc* # Bk^m, []))
 apply(*induct stp*, *simp add*: *steps.simps*)
 apply(*simp add*: *tstep-red*, *auto*)
 apply(*auto simp*: *tstep.simps fetch.simps dither-def new-tape.simps*)
 done

lemma *dither-unhalt-rs*:
 ¬ (∃ *stp*. *isS0* (*steps* (*Suc 0*, Bk^m, [*Oc*]) *dither stp*))
proof(*auto*)
 fix *stp*
 assume *h1*: *isS0* (*steps* (*Suc 0*, Bk^m, [*Oc*]) *dither stp*)
 have ¬ *isS0* ((*steps* (*Suc 0*, Bk^m, [*Oc*]) *dither stp*))
 using *dither-unhalt-state*[*of m stp*]
 by(*auto simp*: *isS0-def*)
 from *h1* **and** *this* **show** *False* **by** (*auto*)
qed

5 The final diagnal arguments to show the undecidability of Halting problem.

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

definition *haltP* :: *tprog* ⇒ *nat* ⇒ *bool*
 where
 haltP t x = (∃ *n a b c*. *steps* (*Suc 0*, [], Oc^x) *t n* = (*0*, Bk^a, Oc^b @ Bk^c))

lemma [*simp*]: *length* (*A* |+| *B*) = *length A* + *length B*
by(*auto simp*: *t-add.simps tshift.simps*)

lemma [*intro*]: ⟦*iseven* (*x*::*nat*); *iseven y*⟧ ⟹ *iseven* (*x* + *y*)
apply(*auto simp*: *iseven-def*)
apply(*rule-tac x* = *x* + *xa* **in** *exI*, *simp*)
done

lemma *t-correct-add*[*intro*]:
 ⟦*t-correct A*; *t-correct B*⟧ ⟹ *t-correct* (*A* |+| *B*)
apply(*auto simp*: *t-correct.simps tshift.simps t-add.simps*
 change-termi-state.simps list-all-iff)
apply(*erule-tac x* = (*a*, *b*) **in** *ballE*, *auto*)
apply(*case-tac ba* = *0*, *auto*)
done

lemma [*intro*]: *t-correct tcopy*
apply(*simp add*: *t-correct.simps tcopy-def iseven-def*)
apply(*rule-tac x* = *15* **in** *exI*, *simp*)
done

lemma [*intro*]: *t-correct dither*
apply(*simp add*: *t-correct.simps dither-def iseven-def*)
apply(*rule-tac x* = *2* **in** *exI*, *simp*)
done

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* =
— The coding function of TM, interestingly, the detailed definition of this funciton *code* does not affect the final result.
fixes *code* :: *tprog* ⇒ *nat*
— The TM *H* is the one which is assummed being able to solve the Halting problem.
and *H* :: *tprog*
assumes *h-wf*[*intro*]: *t-correct H*
— The following two assumptions specifies that *H* does solve the Halting problem.

and *h-case*:
⋀ *M n*. ⟦(*haltP M n*)⟧ ⟹
∃ *na nb*. (*steps* (*Suc 0*, Bk^x, $Oc^{code\ M}$ @ *Bk* # Oc^n) *H na* = (*0*, Bk^{nb}, [*Oc*]))
and *nh-case*:
⋀ *M n*. ⟦(¬ *haltP M n*)⟧ ⟹
∃ *na nb*. (*steps* (*Suc 0*, Bk^x, $Oc^{code\ M}$ @ *Bk* # Oc^n) *H na* = (*0*, Bk^{nb}, [*Oc*, *Oc*]))
begin

term *t-correct*
declare *haltP-def*[*simp del*]
definition *tcontra* :: *tprog* ⇒ *tprog*
where
tcontra h ≡ ((*tcopy* |+| *h*) |+| *dither*)

lemma [*simp*]: a^0 = []
by(*simp add*: *exponent-def*)
lemma *haltP-weaking*:
haltP (*tcontra H*) (*code* (*tcontra H*)) ⟹
∃ *stp*. *isS0* (*steps* (*Suc 0*, [], $Oc^{code\ (tcontra\ H)}$)
((*tcopy* |+| *H*) |+| *dither*) *stp*)
apply(*simp add*: *haltP-def*, *auto*)
apply(*rule-tac x* = *n* **in** *exI*, *simp add*: *isS0-def tcontra-def*)
done

lemma *h-uh*: *haltP* (*tcontra H*) (*code* (*tcontra H*))
    $\Longrightarrow$ ¬ *haltP* (*tcontra H*) (*code* (*tcontra H*))
proof −
  let *?cn* = *code* (*tcontra H*)
  let *?P1* = λ *tp*. *let* (*l*, *r*) = *tp in* (*l* = [] ∧
    (*r*::*block list*) = $Oc^{(?cn)}$)
  let *?Q1* = λ (*l*, *r*).(∃ *nb*. *l* = Bk^{nb} ∧
    r = $Oc^{(?cn)}$ @ *Bk* # $Oc^{(?cn)}$)
  let *?P2* = *?Q1*
  let *?Q2* = λ (*l*, *r*). (∃ *nd*. *l* = Bk^{nd} ∧ *r* = [*Oc*])
  let *?P3* = λ *tp*. *False*
  assume *h*: *haltP* (*tcontra H*) (*code* (*tcontra H*))
  hence *h1*: ⋀ *x*. ∃ *na nb*. *steps* (*Suc 0*, Bk^{x}, $Oc^{code\ (tcontra\ H)}$ @ *Bk* #
      $Oc^{code\ (tcontra\ H)}$) *H na* = (*0*, Bk^{nb}, [*Oc*])
    by(*drule-tac x* = *x* **in** *h-case*, *simp*)
  have *?P1* ⊢−> λ *tp*. (∃ *stp tp′*. *steps* (*Suc 0*, *tp*) (*tcopy* |+| *H*) *stp* = (*0*, *tp′*)
∧ *?Q2 tp′*)
  proof(*rule-tac turing-merge.t-merge-halt*[*of tcopy H ?P1 ?P2 ?P3*
      ?P3 ?Q1 ?Q2], *auto simp*: *turing-merge-def*)
    show ∃ *stp*. *case steps* (*Suc 0*, [], $Oc^{?cn}$) *tcopy stp of* (*s*, *tp′*) ⇒
        s = *0* ∧ (*case tp′ of* (*l*, *r*) ⇒ (∃ *nb*. *l* = Bk^{nb}) ∧ *r* = $Oc^{?cn}$ @ *Bk*
$Oc^{?cn}$)
      using *tcopy-halt-rs*[*of ?cn*]
      apply(*auto*)
      apply(*rule-tac x* = *stp* **in** *exI*, *auto simp*: *exponent-def*)
      done
  next
    fix *nb*
    show ∃ *stp*. *case steps* (*Suc 0*, Bk^{nb}, $Oc^{code\ (tcontra\ H)}$ @ *Bk* # $Oc^{code\ (tcontra\ H)}$)
H stp of
        (*s*, *tp′*) ⇒ *s* = *0* ∧ (*case tp′ of* (*l*, *r*) ⇒ (∃ *nd*. *l* = Bk^{nd}) ∧ *r* =
[*Oc*])
      using *h1*[*of nb*]
      apply(*auto*)
      apply(*rule-tac x* = *na* **in** *exI*, *auto*)
      done
  next
    show λ(*l*, *r*). ((∃ *nb*. *l* = Bk^{nb}) ∧ *r* = $Oc^{code\ (tcontra\ H)}$ @ *Bk* # $Oc^{code\ (tcontra\ H)}$)
⊢−>
      λ(*l*, *r*). ((∃ *nb*. *l* = Bk^{nb}) ∧ *r* = $Oc^{code\ (tcontra\ H)}$ @ *Bk* # $Oc^{code\ (tcontra\ H)}$)
      apply(*simp add*: *t-imply-def*)
      done
  qed
  hence ∃ *stp tp′*. *steps* (*Suc 0*, [], $Oc^{?cn}$) (*tcopy* |+| *H*) *stp* = (*0*, *tp′*) ∧
      (*case tp′ of* (*l*, *r*) ⇒ ∃ *nd*. *l* = Bk^{nd} ∧ *r* = [*Oc*])
    apply(*simp add*: *t-imply-def*)
    done

hence *?P1* ⊢−> λ *tp*. ¬ (∃ *stp*. *isS0* (*steps* (*Suc 0*, *tp*) ((*tcopy* |+| *H*) |+| *dither*) *stp*))
proof(*rule-tac turing-merge.t-merge-uhalt*[*of tcopy* |+| *H dither ?P1 ?P3 ?P3 ?Q2 ?Q2 ?Q2*], *simp add*: *turing-merge-def*, *auto*)
fix *stp nd*
assume *steps* (*Suc 0*, [], $Oc^{code\ (tcontra\ H)}$) (*tcopy* |+| *H*) *stp* = (*0*, Bk^{nd}, [*Oc*])
thus ∃ *stp*. *case steps* (*Suc 0*, [], $Oc^{code\ (tcontra\ H)}$) (*tcopy* |+| *H*) *stp of* (*s*, *tp′*)
⇒ *s* = *0* ∧ (*case tp′ of* (*l*, *r*) ⇒ (∃ *nd*. *l* = Bk^{nd}) ∧ *r* = [*Oc*])
apply(*rule-tac x* = *stp* **in** *exI*, *auto*)
done
next
fix *stp nd nda stpa*
assume *isS0* (*steps* (*Suc 0*, Bk^{nda}, [*Oc*]) *dither stpa*)
thus *False*
using *dither-unhalt-rs*[*of nda*]
apply *auto*
done
next
fix *stp nd*
show λ(*l*, *r*). ((∃ *nd*. *l* = Bk^{nd}) ∧ *r* = [*Oc*]) ⊢−>
λ(*l*, *r*). ((∃ *nd*. *l* = Bk^{nd}) ∧ *r* = [*Oc*])
by (*simp add*: *t-imply-def*)
qed
thus ¬ *haltP* (*tcontra H*) (*code* (*tcontra H*))
apply(*simp add*: *t-imply-def haltP-def tcontra-def*, *auto*)
apply(*erule-tac x* = *n* **in** *allE*, *simp add*: *isS0-def*)
done
qed

lemma *uh-h*:
assumes *uh*: ¬ *haltP* (*tcontra H*) (*code* (*tcontra H*))
shows *haltP* (*tcontra H*) (*code* (*tcontra H*))
proof −
let *?cn* = *code* (*tcontra H*)
have *h1*: ⋀ *x*. ∃ *na nb*. *steps* (*Suc 0*, Bk^{x}, $Oc^{?cn}$ @ *Bk* # $Oc^{?cn}$)
H na = (*0*, Bk^{nb}, [*Oc*, *Oc*])
using *uh*
by(*drule-tac x* = *x* **in** *nh-case*, *simp*)
let *?P1* = λ *tp*. *let* (*l*, *r*) = *tp in* (*l* = [] ∧
(*r*::*block list*) = $Oc^{(?cn)}$)
let *?Q1* = λ (*l*, *r*).(∃ *na*. *l* = Bk^{na} ∧ *r* = [*Oc*, *Oc*])
let *?P2* = *?Q1*
let *?Q2* = *?Q1*
let *?P3* = λ *tp*. *False*
have *?P1* ⊢−> λ *tp*. (∃ *stp tp′*. *steps* (*Suc 0*, *tp*) ((*tcopy* |+| *H*) |+| *dither*)
stp = (*0*, *tp′*) ∧ *?Q2 tp′*)

proof(*rule-tac turing-merge.t-merge-halt*[*of tcopy* |+| *H dither ?P1 ?P2 ?P3 ?P3*
?Q1 ?Q2], *auto simp*: *turing-merge-def*)
show $\exists$ *stp*. *case steps* (*Suc 0*, [], $Oc^{code\ (tcontra\ H)}$) (*tcopy* |+| *H*) *stp of* (*s*, *tp'*) $\Rightarrow$
$s = 0 \wedge (case\ tp'\ of\ (l, r) \Rightarrow (\exists na.\ l = Bk^{na}) \wedge r = [Oc, Oc])$
proof −
let $?Q1 = \lambda\ (l, r).(\exists\ nb.\ l = Bk^{nb} \wedge\ r = Oc^{(?cn)}$ @ Bk # $Oc^{(?cn)})$
let $?P2 = ?Q1$
let $?Q2 = \lambda\ (l, r).(\exists\ na.\ l = Bk^{na} \wedge r = [Oc, Oc])$
have *?P1* $\vdash$−> λ *tp*. ($\exists$ *stp tp'*. *steps* (*Suc 0*, *tp*) (*tcopy* |+| *H*)
stp = (*0*, *tp'*) $\wedge$ *?Q2 tp'*)
proof(*rule-tac turing-merge.t-merge-halt*[*of tcopy H ?P1 ?P2 ?P3 ?P3*
?Q1 ?Q2], *auto simp*: *turing-merge-def*)
show $\exists$ *stp*. *case steps* (*Suc 0*, [], $Oc^{code\ (tcontra\ H)}$) *tcopy stp of* (*s*, *tp'*) $\Rightarrow s = 0$
$\wedge$ (*case tp' of* (*l*, *r*) $\Rightarrow$ ($\exists$ *nb*. $l = Bk^{nb}$) $\wedge$ $r = Oc^{code\ (tcontra\ H)}$ @ *Bk* # $Oc^{code\ (tcontra\ H)}$)
using *tcopy-halt-rs*[*of ?cn*]
apply(*auto*)
apply(*rule-tac x* = *stp* **in** *exI*, *simp add*: *exponent-def*)
done
next
fix *nb*
show $\exists$ *stp*. *case steps* (*Suc 0*, Bk^{nb}, $Oc^{code\ (tcontra\ H)}$ @ *Bk* # $Oc^{code\ (tcontra\ H)}$) *H stp of*
(*s*, *tp'*) $\Rightarrow s = 0 \wedge$ (*case tp' of* (*l*, *r*) $\Rightarrow$ ($\exists$ *na*. $l = Bk^{na}$) $\wedge$ *r* = [*Oc*, *Oc*])
using *h1*[*of nb*]
apply(*auto*)
apply(*rule-tac x* = *na* **in** *exI*, *auto*)
done
next
show λ(*l*, *r*). (($\exists$ *nb*. $l = Bk^{nb}$) $\wedge$ $r = Oc^{code\ (tcontra\ H)}$ @ *Bk* # $Oc^{code\ (tcontra\ H)}$) $\vdash$−>
λ(*l*, *r*). (($\exists$ *nb*. $l = Bk^{nb}$) $\wedge$ $r = Oc^{code\ (tcontra\ H)}$ @ *Bk* # $Oc^{code\ (tcontra\ H)}$)
by(*simp add*: *t-imply-def*)
qed
hence ($\exists$ *stp tp'*. *steps* (*Suc 0*, [], $Oc^{?cn}$) (*tcopy* |+| *H*) *stp* = (*0*, *tp'*) $\wedge$ *?Q2 tp'*)
apply(*simp add*: *t-imply-def*)
done
thus *?thesis*
apply(*auto*)
apply(*rule-tac x* = *stp* **in** *exI*, *auto*)
done

```
  qed
 next
  fix na
  show ∃stp. case steps (Suc 0, Bk^na, [Oc, Oc]) dither stp of (s, tp')
          ⇒ s = 0 ∧ (case tp' of (l, r) ⇒ (∃na. l = Bk^na) ∧ r = [Oc, Oc])
   using dither-halt-rs[of na]
   apply(auto)
   apply(rule-tac x = stp in exI, auto)
   done
 next
  show λ(l, r). ((∃na. l = Bk^na) ∧ r = [Oc, Oc]) ⊢->
                    (λ(l, r). (∃na. l = Bk^na) ∧ r = [Oc, Oc])
   by (simp add: t-imply-def)
 qed
 hence ∃ stp tp'. steps (Suc 0, [], Oc^?cn) ((tcopy |+| H ) |+| dither)
            stp = (0, tp') ∧ ?Q2 tp'
  apply(simp add: t-imply-def)
  done
 thus haltP (tcontra H) (code (tcontra H))
  apply(auto simp: haltP-def tcontra-def)
  apply(rule-tac x = stp in exI,
     rule-tac x = na in exI,
     rule-tac x = Suc (Suc 0) in exI,
     rule-tac x = 0 in exI, simp add: exp-ind-def)
  done
qed
```

False is finally derived.

```
lemma False
using uh-h h-uh
by auto
end

end
```

6 *abacus* a kind of register machine

```
theory abacus
imports Main turing-basic
begin
```

Abacus instructions:

```
datatype abc-inst =
 — Inc n increments the memory cell (or register) with address n by one.
   Inc nat
 — Dec n label decrements the memory cell with address n by one. If cell n is
```

already zero, no decrements happens and the executio jumps to the instruction labeled by *label*.

| *Dec nat nat*

— *Goto label* unconditionally jumps to the instruction labeled by *label*.

| *Goto nat*

Abacus programs are defined as lists of Abacus instructions.

type-synonym *abc-prog* = *abc-inst list*

7 Sample Abacus programs

Abacus for addition and clearance.

fun *plus-clear* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *abc-prog*
where
plus-clear m n e = [*Dec m e*, *Inc n*, *Goto 0*]

Abacus for clearing memory untis.

fun *clear* :: *nat* ⇒ *nat* ⇒ *abc-prog*
where
clear n e = [*Dec n e*, *Goto 0*]

fun *plus*:: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *abc-prog*
where
plus m n p e = [*Dec m 4*, *Inc n*, *Inc p*,
Goto 0, *Dec p e*, *Inc m*, *Goto 4*]

fun *mult* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *abc-prog*
where
mult m1 m2 n p e = [*Dec m1 e*]@ *plus m1 m2 p 1*

fun *expo* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *abc-prog*
where
expo n m1 m2 p e = [*Inc n*, *Dec m1 e*] @ *mult m2 n n p 2*

The state of Abacus machine.

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-l* = *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-l* ⇒ *abc-inst option* ⇒ *abc-conf-l*
where
abc-step-l (*s*, *lm*) *a* = (*case a of*
None ⇒ (*s*, *lm*) |
Some (*Inc n*) ⇒ (*let nv* = *abc-lm-v lm n in*
(*s* + *1*, *abc-lm-s lm n* (*nv* + *1*))) |
Some (*Dec n e*) ⇒ (*let nv* = *abc-lm-v lm n in*
if (*nv* = *0*) *then* (*e*, *abc-lm-s lm n 0*)
else (*s* + *1*, *abc-lm-s lm n* (*nv* − *1*))) |
Some (*Goto n*) ⇒ (*n*, *lm*)
)

Multi-step execution of Abacus machine.

fun *abc-steps-l* :: *abc-conf-l* ⇒ *abc-prog* ⇒ *nat* ⇒ *abc-conf-l*
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*

8 Compiling Abacus machines into Truing machines

8.1 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* ⇒ *tprog*
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* :: *tprog*
where
tinc-b ≡ [(*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*)]

tshift tm off shifts *tm* by offset *off*, leaving instructions concerning state *0* unchanged, because state *0* is the end state, which needs not be changed with shift operation.

fun *tshift* :: *tprog* ⇒ *nat* ⇒ *tprog*
where
tshift tp off = (*map* (λ (*action*, *state*).
(*action*, (*if state* = *0 then 0*
else state + *off*))) *tp*)

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* ⇒ *tprog*
where
tinc ss n = *tshift* (*findnth n* @ *tshift 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* :: *tprog*
where
tdec-b ≡ [(*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*)]

sete tp e attaches the termination edges (edges leading to state *0*) of TM *tp* to the intruction labelled by *e*.

fun *sete* :: *tprog* ⇒ *nat* ⇒ *tprog*
where
sete tp e = *map* (λ (*action*, *state*). (*action*, *if state* = *0 then e else state*)) *tp*

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* ⇒ *tprog*
where
tdec ss n e = *sete* (*tshift* (*findnth n* @ *tshift 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* ⇒ *tprog*
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 0 = *Suc 0* |
start-of ly (*Suc as*) =
(*if as* < *length ly then start-of ly as* + (*ly* ! *as*)
else start-of ly as)

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* ⇒ *tprog*
where
ci ly ss i = (*case i of*

```
          Inc n   ⇒ tinc ss n |
          Dec n e ⇒ tdec ss n (start-of ly e) |
          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 ⇒ tprog 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 ⇒ tprog
  where tm-of ap = concat (tms-of ap)
```

The following two functions specify the well-formedness of complied TM.

```
fun t-ncorrect :: tprog ⇒ bool
  where
  t-ncorrect tp = (length tp mod 2 = 0)

fun abc2t-correct :: abc-prog ⇒ bool
  where
  abc2t-correct ap = list-all (λ (n, tm).
           t-ncorrect (ci (layout-of ap) n tm)) (tpairs-of ap)

lemma findnth-length: length (findnth n) div 2 = 2 ∗ n
apply(induct n, simp, simp)
done

lemma ci-length : length (ci ns n ai) div 2 = length-of ai
apply(auto simp: ci.simps tinc-b-def tdec-b-def findnth-length
              split: abc-inst.splits)
done
```

8.2 Representation of Abacus memory by TM tape

```
consts tape-of :: 'a ⇒ block list (<-> 100)
```

tape-of-nat-list am returns the TM tape representing Abacus memory *am*.

```
fun tape-of-nat-list :: nat list ⇒ block list
  where
  tape-of-nat-list [] = [] |
  tape-of-nat-list [n] = Oc^(n+1) |
```

tape-of-nat-list $(n\#ns) = (Oc^{n+1})$ @ $[Bk]$ @ $($*tape-of-nat-list ns*$)$

defs (**overloaded**)
tape-of-nl-abv: <*am*> ≡ *tape-of-nat-list am*
tape-of-nat-abv : <(*n*::*nat*)> ≡ Oc^{n+1}

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

```
fun crsp-l :: layout ⇒ abc-conf-l ⇒ t-conf ⇒ block list ⇒ bool
  where
  crsp-l ly (as, lm) (ts, (l, r)) inres =
          (ts = start-of ly as ∧ (∃ rn. r = <lm> @ Bk^rn)
           ∧ l = Bk # Bk # inres)

declare crsp-l.simps[simp del]
```

8.3 A more general definition of TM execution.

t-step tcf (*tp*, *ss*) returns the result of one step exection of TM *tp* assuming *tp* starts from instial state *ss*.

```
fun t-step :: t-conf ⇒ (tprog × nat) ⇒ t-conf
  where
  t-step c (p, off) =
          (let (state, leftn, rightn) = c in
           let (action, next-state) = fetch p (state−off)
                          (case rightn of
                             [] ⇒ Bk |
                             Bk # xs ⇒ Bk |
                             Oc # xs ⇒ Oc
                          )
            in
           (next-state, new-tape action (leftn, rightn)))
```

t-steps tcf (*tp*, *ss*) *n* returns the result of *n*-step exection of TM *tp* assuming *tp* starts from instial state *ss*.

```
fun t-steps :: t-conf ⇒ (tprog × nat) ⇒ nat ⇒ t-conf
  where
  t-steps c (p, off) 0 = c |
  t-steps c (p, off) (Suc n) = t-steps
                  (t-step c (p, off)) (p, off) n

lemma stepn: t-steps c (p, off) (Suc n) =
            t-step (t-steps c (p, off) n) (p, off)
apply(induct n arbitrary: c, simp add: t-steps.simps)
apply(simp add: t-steps.simps)
done
```

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

```
type-synonym inc-inv-t = abc-conf-l ⇒ t-conf ⇒ block list ⇒ bool

declare tms-of.simps[simp del] tm-of.simps[simp del]
        layout-of.simps[simp del] abc-fetch.simps [simp del]
        t-step.simps[simp del] t-steps.simps[simp del]
        tpairs-of.simps[simp del] start-of.simps[simp del]
        fetch.simps [simp del] t-ncorrect.simps[simp del]
        new-tape.simps [simp del] ci.simps [simp del] length-of.simps[simp del]
        layout-of.simps[simp del] crsp-l.simps[simp del]
        abc2t-correct.simps[simp del]

lemma tct-div2: t-ncorrect tp ⟹ (length tp) mod 2 = 0
apply(simp add: t-ncorrect.simps)
done

lemma t-shift-fetch:
    ⟦t-ncorrect tp1; t-ncorrect tp;
      length tp1 div 2 < a ∧ a ≤ length tp1 div 2 + length tp div 2⟧
    ⟹ fetch tp (a − length tp1 div 2) b =
         fetch (tp1 @ tp @ tp2) a b
apply(subgoal-tac ∃ x. a = length tp1 div 2 + x, erule exE, simp)
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(case-tac 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(simp add: t-ncorrect.simps, auto)
apply(subgoal-tac 2 ∗ (length tp1 div 2) = length tp1, simp)
apply(subgoal-tac 2 ∗ nat < length tp, simp add: nth-append, auto)
apply(simp add: t-ncorrect.simps, auto)
apply(rule-tac x = a − length tp1 div 2 in exI, simp)
done

lemma t-shift-in-step:
      ⟦t-step (a, aa, ba) (tp, length tp1 div 2) = (s, l, r);
        t-ncorrect tp1; t-ncorrect tp;
        length tp1 div 2 < a ∧ a ≤ length tp1 div 2 + length tp div 2⟧
       ⟹ t-step (a, aa, ba) (tp1 @ tp @ tp2, 0) = (s, l, r)
apply(simp add: t-step.simps)
apply(subgoal-tac fetch tp (a − length tp1 div 2) (case ba of [] ⇒
                 Bk | x # xs ⇒ x)
             = fetch (tp1 @ tp @ tp2) a (case ba of [] ⇒ Bk | x # xs
                 ⇒ x))
apply(case-tac fetch tp (a − length tp1 div 2) (case ba of [] ⇒ Bk
               | x # xs ⇒ x))
apply(auto intro: t-shift-fetch)
apply(case-tac ba, simp, simp)
```

apply(*case-tac aaa*, *simp*, *simp*)
done

declare *add-Suc-right*[*simp del*]
lemma *t-step-add*: *t-steps c* (*p*, *off*) (*m* + *n*) =
t-steps (*t-steps c* (*p*, *off*) *m*) (*p*, *off*) *n*
apply(*induct m arbitrary*: *n*, *simp add*: *t-steps.simps*, *simp*)
apply(*subgoal-tac t-steps c* (*p*, *off*) (*Suc* (*m* + *n*)) =
t-steps c (*p*, *off*) (*m* + *Suc n*), *simp*)
apply(*subgoal-tac t-steps* (*t-steps c* (*p*, *off*) *m*) (*p*, *off*) (*Suc n*) =
t-steps (*t-step* (*t-steps c* (*p*, *off*) *m*) (*p*, *off*))
(*p*, *off*) *n*)
apply(*simp*, *simp add*: *stepn*)
apply(*simp only*: *t-steps.simps*)
apply(*simp only*: *add-Suc-right*)
done
declare *add-Suc-right*[*simp*]

lemma *s-out-fetch*: ⟦*t-ncorrect tp*;
¬ (*length tp1 div 2* < *a* ∧ *a* ≤ *length tp1 div 2* +
length tp div 2)⟧
⟹ *fetch tp* (*a* − *length tp1 div 2*) *b* = (*Nop*, *0*)
apply(*auto*)
apply(*simp add*: *fetch.simps*)
apply(*subgoal-tac* ∃ *x*. *a* − *length tp1 div 2* = *length tp div 2* + *x*)
apply(*erule exE*, *simp*)
apply(*case-tac x*, *simp*)
apply(*auto simp add*: *fetch.simps*)
apply(*subgoal-tac 2* ∗ (*length tp div 2*) = *length tp*)
apply(*auto simp*: *t-ncorrect.simps split*: *block.splits*)
apply(*rule-tac x* = *a* − *length tp1 div 2* − *length tp div 2* **in** *exI*
, *simp*)
done

lemma *conf-keep-step*:
⟦*t-ncorrect tp*;
¬ (*length tp1 div 2* < *a* ∧ *a* ≤ *length tp1 div 2* +
length tp div 2)⟧
⟹ *t-step* (*a*, *aa*, *ba*) (*tp*, *length tp1 div 2*) = (*0*, *aa*, *ba*)
apply(*simp add*: *t-step.simps*)
apply(*subgoal-tac fetch tp* (*a* − *length tp1 div 2*) (*case ba of* [] ⇒
Bk | *Bk* # *xs* ⇒ *Bk* | *Oc* # *xs* ⇒ *Oc*) = (*Nop*, *0*))
apply(*simp add*: *new-tape.simps*)
apply(*rule s-out-fetch*, *simp*, *simp*)
done

lemma *conf-keep*:
⟦*t-ncorrect tp*;
¬ (*length tp1 div 2* < *a* ∧

```
      a ≤ length tp1 div 2 + length tp div 2); n > 0⟧
    ⟹ t-steps (a, aa, ba) (tp, length tp1 div 2) n = (0, aa, ba)
apply(induct n, simp)
apply(case-tac n, simp add: t-steps.simps)
apply(rule-tac conf-keep-step, simp+)
apply(subgoal-tac  t-steps (a, aa, ba)
            (tp, length tp1 div 2) (Suc (Suc nat))
      = t-step (t-steps (a, aa, ba)
          (tp, length tp1 div 2) (Suc nat)) (tp, length tp1 div 2))
apply(simp)
apply(rule-tac conf-keep-step, simp, simp)
apply(rule stepn)
done

lemma state-bef-inside:
  ⟦t-ncorrect tp1; t-ncorrect tp;
    t-steps (s0, l0, r0) (tp, length tp1 div 2) stp = (s, l, r);
    length tp1 div 2 < s0 ∧
      s0 ≤ length tp1 div 2 + length tp div 2;
    length tp1 div 2 < s ∧ s ≤ length tp1 div 2 + length tp div 2;
    n < stp; t-steps (s0, l0, r0) (tp, length tp1 div 2) n =
    (a, aa, ba)⟧
    ⟹  length tp1 div 2 < a ∧
      a ≤ length tp1 div 2 + length tp div 2
apply(subgoal-tac ∃ x. stp = n + x, erule exE)
apply(simp only: t-step-add)
apply(rule classical)
apply(subgoal-tac t-steps (a, aa, ba)
        (tp, length tp1 div 2) x = (0, aa, ba))
apply(simp)
apply(rule conf-keep, simp, simp, simp)
apply(rule-tac x = stp − n in exI, simp)
done

lemma turing-shift-inside:
    ⟦t-steps (s0, l0, r0) (tp, length tp1 div 2) stp = (s, l, r);
      length tp1 div 2 < s0 ∧
      s0 ≤ length tp1 div 2 + length tp div 2;
      t-ncorrect tp1; t-ncorrect tp;
      length tp1 div 2 < s ∧
      s ≤ length tp1 div 2 + length tp div 2⟧
   ⟹ t-steps (s0, l0, r0) (tp1 @ tp @ tp2, 0) stp = (s, l, r)
apply(induct stp arbitrary: s l r)
apply(simp add: t-steps.simps)
apply(subgoal-tac  t-steps (s0, l0, r0)
      (tp, length tp1 div 2) (Suc stp)
              = t-step (t-steps (s0, l0, r0)
      (tp, length tp1 div 2) stp) (tp, length tp1 div 2))
apply(case-tac t-steps (s0, l0, r0) (tp, length tp1 div 2) stp)
```

apply(*subgoal-tac length tp1 div 2 < a* ∧
a ≤ *length tp1 div 2* + *length tp div 2*)
apply(*subgoal-tac t-steps* (*s0*, *l0*, *r0*)
(*tp1* @ *tp* @ *tp2*, *0*) *stp* = (*a*, *b*, *c*))
apply(*simp only*: *stepn*, *simp*)
apply(*rule-tac t-shift-in-step*, *simp*+)
defer
apply(*rule stepn*)
apply(*rule-tac n* = *stp* **and** *stp* = *Suc stp* **and** *a* = *a*
and *aa* = *b* **and** *ba* = *c* **in** *state-bef-inside*, *simp*+)
done

lemma *take-Suc-last*[*elim*]: *Suc as* ≤ *length xs* ⟹
take (*Suc as*) *xs* = *take as xs* @ [*xs* ! *as*]
apply(*induct xs arbitrary*: *as*, *simp*, *simp*)
apply(*case-tac as*, *simp*, *simp*)
done

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-take-suc-iff*: *Suc n* ≤ *length tps* ⟹
concat (*take n tps*) @ (*tps* ! *n*) = *concat* (*take* (*Suc n*) *tps*)
apply(*drule-tac concat-suc*, *simp*)
done

lemma *concat-drop-suc-iff*:
Suc n < *length tps* ⟹ *concat* (*drop* (*Suc n*) *tps*) =
tps ! *Suc n* @ *concat* (*drop* (*Suc* (*Suc n*)) *tps*)
apply(*induct tps arbitrary*: *n*, *simp*, *simp*)
apply(*case-tac tps*, *simp*, *simp*)
apply(*case-tac n*, *simp*, *simp*)
done

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*)
apply(*induct n*, *simp*)
apply(*case-tac tps*, *simp*, *simp*, *simp*)
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), simp)
apply(*rule-tac concat-drop-suc-iff, simp*)
apply(*rule-tac concat-take-suc-iff, simp*)
done

declare *append-assoc*[*simp*]

lemma *map-of*: *n < length xs ⟹ (map f xs) ! n = f (xs ! n)*
by(*auto*)

lemma [*simp*]: *length (tms-of aprog) = length aprog*
apply(*auto simp*: *tms-of.simps tpairs-of.simps*)
done

lemma *ci-nth*: ⟦*ly = layout-of aprog*; *as < length 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 map-of del: *map-append*)
done

lemma *t-split*:⟦
ly = layout-of aprog;
as < length 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*, *simp*, *simp*)
apply(*rule-tac ci-nth*, *auto*)
done

lemma *math-sub*: ⟦*x >= Suc 0*; *x − 1 = z*⟧ ⟹ *x + y − Suc 0 = z + y*
by *auto*

lemma *start-more-one*: *as ≠ 0 ⟹ start-of ly as >= Suc 0*
apply(*induct as*, *simp add*: *start-of.simps*)
apply(*case-tac as*, *auto simp*: *start-of.simps*)
done

lemma *tm-ct*: ⟦*abc2t-correct aprog*; *tp ∈ set (tms-of aprog)*⟧ ⟹
t-ncorrect tp
apply(*simp add*: *abc2t-correct.simps tms-of.simps*)
apply(*auto*)
apply(*simp add*:*list-all-iff*, *auto*)

done

lemma div-apart: ⟦x mod (2::nat) = 0; y mod 2 = 0⟧
⟹ (x + y) div 2 = x div 2 + y div 2
apply(drule mod-eqD)+
apply(auto)
done

lemma div-apart-iff: ⟦x mod (2::nat) = 0; y mod 2 = 0⟧ ⟹
(x + y) mod 2 = 0
apply(auto)
done

lemma tms-ct: ⟦abc2t-correct aprog; n < length aprog⟧ ⟹
t-ncorrect (concat (take n (tms-of aprog)))
apply(induct n, simp add: t-ncorrect.simps, simp)
apply(subgoal-tac concat (take (Suc n) (tms-of aprog)) =
concat (take n (tms-of aprog)) @ (tms-of aprog ! n), simp)
apply(simp add: t-ncorrect.simps)
apply(rule-tac div-apart-iff, simp)
apply(subgoal-tac t-ncorrect (tms-of aprog ! n),
simp add: t-ncorrect.simps)
apply(rule-tac tm-ct, simp)
apply(rule-tac nth-mem, simp add: tms-of.simps tpairs-of.simps)
apply(rule-tac concat-suc, simp add: tms-of.simps tpairs-of.simps)
done

lemma tcorrect-div2: ⟦abc2t-correct aprog; Suc as < length aprog⟧
⟹ (length (concat (take as (tms-of aprog))) + length (tms-of aprog
! as)) div 2 = length (concat (take as (tms-of aprog))) div 2 +
length (tms-of aprog ! as) div 2
apply(subgoal-tac t-ncorrect (tms-of aprog ! as))
apply(subgoal-tac t-ncorrect (concat (take as (tms-of aprog))))
apply(rule-tac div-apart)
apply(rule tct-div2, simp)+
apply(erule-tac tms-ct, simp)
apply(rule-tac tm-ct, simp)
apply(rule-tac nth-mem)
apply(simp add: tms-of.simps tpairs-of.simps)
done

lemma [simp]: length (layout-of aprog) = length aprog
apply(auto simp: layout-of.simps)
done

lemma start-of-ind: ⟦as < length aprog; ly = layout-of aprog⟧ ⟹
start-of ly (Suc as) = start-of ly as +
length ((tms-of aprog) ! as) div 2
apply(simp only: start-of.simps, simp)

apply(*auto simp*: *start-of.simps tms-of.simps layout-of.simps*
tpairs-of.simps)
apply(*simp add*: *ci-length*)
done

lemma *concat-take-suc*: *Suc n* ≤ *length xs* ⟹
concat (*take* (*Suc n*) *xs*) = *concat* (*take n xs*) @ (*xs* ! *n*)
apply(*subgoal-tac take* (*Suc n*) *xs* =
take n xs @ [*xs* ! *n*])
apply(*auto*)
done

lemma *ci-length-not0*: *Suc 0* <= *length* (*ci ly as i*) *div 2*
apply(*subgoal-tac length* (*ci ly as i*) *div 2* = *length-of i*)
apply(*simp add*: *length-of.simps split*: *abc-inst.splits*)
apply(*rule ci-length*)
done

lemma *findnth-length2*: *length* (*findnth n*) = *4* ∗ *n*
apply(*induct n*, *simp*)
apply(*simp*)
done

lemma *ci-length2*: *length* (*ci ly as i*) = *2* ∗ (*length-of i*)
apply(*simp add*: *ci.simps length-of.simps tinc-b-def tdec-b-def*
split: *abc-inst.splits*, *auto*)
apply(*simp add*: *findnth-length2*)+
done

lemma *tm-mod2*: *as* < *length aprog* ⟹
length (*tms-of aprog* ! *as*) *mod 2* = *0*
apply(*simp add*: *tms-of.simps*)
apply(*subgoal-tac map* (λ(*x*, *y*). *ci* (*layout-of aprog*) *x y*)
(*tpairs-of aprog*) ! *as*
= (λ(*x*, *y*). *ci* (*layout-of aprog*) *x y*)
((*tpairs-of aprog*) ! *as*), *simp*)
apply(*case-tac* (*tpairs-of aprog* ! *as*), *simp*)
apply(*subgoal-tac length* (*ci* (*layout-of aprog*) *a b*) =
2 ∗ (*length-of b*), *simp*)
apply(*rule ci-length2*)
apply(*rule map-of*, *simp add*: *tms-of.simps tpairs-of.simps*)
done

lemma *tms-mod2*: *as* ≤ *length aprog* ⟹
length (*concat* (*take as* (*tms-of aprog*))) *mod 2* = *0*
apply(*induct as*, *simp*, *simp*)
apply(*subgoal-tac concat* (*take* (*Suc as*) (*tms-of aprog*))
= *concat* (*take as* (*tms-of aprog*)) @
(*tms-of aprog* ! *as*), *auto*)

apply(*rule div-apart-iff*, *simp*, *rule tm-mod2*, *simp*)
apply(*rule concat-take-suc*, *simp add*: *tms-of.simps tpairs-of.simps*)
done

lemma [*simp*]: ⟦*as* < *length aprog*; (*abc-fetch as aprog*) = *Some ins*⟧
⟹ *ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*ins*) ∈ *set* (*tms-of aprog*)
apply(*insert ci-nth*[*of layout-of aprog aprog as*], *simp*)
done

lemma *startof-not0*: *start-of ly as* > *0*
apply(*induct as*, *simp add*: *start-of.simps*)
apply(*case-tac as*, *auto simp*: *start-of.simps*)
done

declare *abc-step-l.simps*[*simp del*]
lemma *pre-lheq*: ⟦*tp* = *concat* (*take as* (*tms-of aprog*));
abc2t-correct aprog; *as* ≤ *length aprog*⟧ ⟹
start-of (*layout-of aprog*) *as* − *Suc 0* = *length tp div 2*
apply(*induct as arbitrary*: *tp*, *simp add*: *start-of.simps*, *simp*)
proof −
fix *as tp*
assume *h1*: ⋀*tp*. *tp* = *concat* (*take as* (*tms-of aprog*)) ⟹
start-of (*layout-of aprog*) *as* − *Suc 0* =
length (*concat* (*take as* (*tms-of aprog*))) *div 2*
and *h2*: *abc2t-correct aprog Suc as* ≤ *length aprog*
from *h2* **show** *start-of* (*layout-of aprog*) (*Suc as*) − *Suc 0* =
length (*concat* (*take* (*Suc as*) (*tms-of aprog*))) *div 2*
apply(*insert h1*[*of concat* (*take as* (*tms-of aprog*))], *simp*)
apply(*insert start-of-ind*[*of as aprog layout-of aprog*], *simp*)
apply(*subgoal-tac* (*take* (*Suc as*) (*tms-of aprog*)) =
take as (*tms-of aprog*) @ [(*tms-of aprog*) ! *as*], *simp*)
apply(*subgoal-tac* (*length* (*concat* (*take as* (*tms-of aprog*))) +
length (*tms-of aprog* ! *as*)) *div 2*
= *length* (*concat* (*take as* (*tms-of aprog*))) *div 2* +
length (*tms-of aprog* ! *as*) *div 2*, *simp*)
apply(*subgoal-tac start-of* (*layout-of aprog*) *as* =
length (*concat* (*take as* (*tms-of aprog*))) *div 2* + *Suc 0*, *simp*)
apply(*subgoal-tac start-of* (*layout-of aprog*) *as* > *0*, *simp*,
rule-tac startof-not0)
apply(*insert tm-mod2*[*of as aprog*], *simp*)
apply(*insert tms-mod2*[*of as aprog*], *simp*, *arith*)
apply(*rule take-Suc-last*, *simp*)
done
qed

lemma *crsp2stateq*:
⟦*as* < *length aprog*; *abc2t-correct aprog*;
crsp-l (*layout-of aprog*) (*as*, *am*) (*a*, *aa*, *ba*) *inres*⟧ ⟹

```
     a = length (concat (take as (tms-of aprog))) div 2 + 1
apply(simp add: crsp-l.simps)
apply(insert pre-lheq[of (concat (take as (tms-of aprog))) as aprog]
, simp)
apply(subgoal-tac start-of (layout-of aprog) as > 0,
    auto intro: startof-not0)
done

lemma turing-shift-outside:
   ⟦t-steps (s0, l0, r0) (tp, length tp1 div 2) stp = (s, l, r);
     s ≠ 0; stp > 0;
     length tp1 div 2 < s0 ∧
     s0 ≤ length tp1 div 2 + length tp div 2;
     t-ncorrect tp1; t-ncorrect tp;
     ¬ (length tp1 div 2 < s ∧
    s ≤ length tp1 div 2 + length tp div 2)⟧
  ⟹ ∃ stp' > 0. t-steps (s0, l0, r0) (tp1 @ tp @ tp2, 0) stp'
          = (s, l, r)
apply(rule-tac x = stp in exI)
apply(case-tac stp, simp add: t-steps.simps)
apply(simp only: stepn)
apply(case-tac t-steps (s0, l0, r0) (tp, length tp1 div 2) nat)
apply(subgoal-tac length tp1 div 2 < a ∧
           a ≤ length tp1 div 2 + length tp div 2)
apply(subgoal-tac t-steps (s0, l0, r0) (tp1 @ tp @ tp2, 0) nat
           = (a, b, c), simp)
apply(rule-tac t-shift-in-step, simp+)
apply(rule-tac turing-shift-inside, simp+)
apply(rule classical)
apply(subgoal-tac t-step (a,b,c)
      (tp, length tp1 div 2) = (0, b, c), simp)
apply(rule-tac conf-keep-step, simp+)
done

lemma turing-shift:
 ⟦t-steps (s0, (l0, r0)) (tp, (length tp1 div 2)) stp
  = (s, (l, r)); s ≠ 0; stp > 0;
 (length tp1 div 2 < s0 ∧ s0 <= length tp1 div 2 + length tp div 2);
 t-ncorrect tp1; t-ncorrect tp⟧ ⟹
     ∃ stp' > 0. t-steps (s0, (l0, r0)) (tp1 @ tp @ tp2, 0) stp' =
           (s, (l, r))
apply(case-tac s > length tp1 div 2 ∧
         s <= length tp1 div 2 + length tp div 2)
apply(subgoal-tac  t-steps (s0, l0, r0) (tp1 @ tp @ tp2, 0) stp =
           (s, l, r))
apply(rule-tac x = stp in exI, simp)
apply(rule-tac turing-shift-inside, simp+)
apply(rule-tac turing-shift-outside, simp+)
done
```

lemma *inc-startof-not0*: *start-of ly as* $\geq$ *Suc 0*
apply(*induct as*, *simp add*: *start-of.simps*)
apply(*simp add*: *start-of.simps*)
done

lemma *s-crsp*:
  ⟦*as* < *length aprog*; *abc-fetch as aprog* = *Some ins*;
  abc2t-correct aprog;
  crsp-l (*layout-of aprog*) (*as*, *am*) (*a*, *aa*, *ba*) *inres*⟧ $\Longrightarrow$
  length (*concat* (*take as* (*tms-of aprog*))) *div 2* < *a*
      $\wedge$ *a* $\leq$ *length* (*concat* (*take as* (*tms-of aprog*))) *div 2* +
        length (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
        ins) *div 2*
apply(*subgoal-tac a* = *length* (*concat* (*take as* (*tms-of aprog*))) *div*
                2 + *1*, *simp*)
apply(*rule-tac ci-length-not0*)
apply(*rule crsp2stateq*, *simp*+)
done

lemma *tms-out-ex*:
  ⟦*ly* = *layout-of aprog*; *tprog* = *tm-of aprog*;
  abc2t-correct aprog;
  crsp-l ly (*as*, *am*) *tc inres*; *as* < *length aprog*;
  abc-fetch as aprog = *Some ins*;
  t-steps tc (*ci ly* (*start-of ly as*) *ins*,
  (*start-of ly as*) − *1*) *n* = (*s*, *l*, *r*);
  n > *0*;
  abc-step-l (*as*, *am*) (*abc-fetch as aprog*) = (*as′*, *am′*);
  s = *start-of ly as′*
  ⟧
  $\Longrightarrow$ ∃ *stp* > *0*. (*t-steps tc* (*tprog*, *0*) *stp* = (*s*, (*l*, *r*)))
apply(*simp only*: *tm-of.simps*)
apply(*subgoal-tac* ∃ *tp1 tp2*. *concat* (*tms-of aprog*) =
      tp1 @ (*ci ly* (*start-of ly as*) *ins*) @ *tp2*
    $\wedge$ *tp1* = *concat* (*take as* (*tms-of aprog*)) $\wedge$
      tp2 = *concat* (*drop* (*Suc as*) (*tms-of aprog*)))
apply(*erule exE*, *erule exE*, *erule conjE*, *erule conjE*,
      case-tac tc, *simp*)
apply(*rule turing-shift*)
apply(*subgoal-tac start-of* (*layout-of aprog*) *as* − *Suc 0*
              = *length tp1 div 2*, *simp*)
apply(*rule-tac pre-lheq*, *simp*, *simp*, *simp*)
apply(*simp add*: *startof-not0*, *simp*)
apply(*rule-tac s-crsp*, *simp*, *simp*, *simp*, *simp*)
apply(*rule tms-ct*, *simp*, *simp*)
apply(*rule tm-ct*, *simp*)
apply(*subgoal-tac ci* (*layout-of aprog*)
              (*start-of* (*layout-of aprog*) *as*) *ins*

= (*tms-of aprog* ! *as*), *simp*)
apply(*simp add*: *tms-of.simps tpairs-of.simps*)
apply(*simp add*: *tms-of.simps tpairs-of.simps abc-fetch.simps*)
apply(*erule-tac t-split*, *auto simp*: *tm-of.simps*)
done

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

fun *at-begin-fst-bwtn* :: *inc-inv-t*
where
at-begin-fst-bwtn (*as*, *lm*) (*s*, *l*, *r*) *ires* =
(∃ *lm1 tn rn*. *lm1* = (*lm* @ (0^{tn})) ∧ *length lm1* = *s* ∧
(*if lm1* = [] *then l* = *Bk* # *Bk* # *ires*
else l = [*Bk*]@<*rev lm1*>@*Bk*#*Bk*#*ires*) ∧ *r* = (Bk^{rn}))

fun *at-begin-fst-awtn* :: *inc-inv-t*
where
at-begin-fst-awtn (*as*, *lm*) (*s*, *l*, *r*) *ires* =
(∃ *lm1 tn rn*. *lm1* = (*lm* @ (0^{tn})) ∧ *length lm1* = *s* ∧
(*if lm1* = [] *then l* = *Bk* # *Bk* # *ires*
else l = [*Bk*]@<*rev lm1*>@*Bk*#*Bk*#*ires*) ∧ *r* = [*Oc*]@Bk^{rn}
)

fun *at-begin-norm* :: *inc-inv-t*
where
at-begin-norm (*as*, *lm*) (*s*, *l*, *r*) *ires*=
(∃ *lm1 lm2 rn*. *lm* = *lm1* @ *lm2* ∧ *length lm1* = *s* ∧
(*if lm1* = [] *then l* = *Bk* # *Bk* # *ires*
else l = *Bk* # <*rev lm1*> @ *Bk*# *Bk* # *ires*) ∧ *r* = <*lm2*> @ (Bk^{rn}))

fun *in-middle* :: *inc-inv-t*
where
in-middle (*as*, *lm*) (*s*, *l*, *r*) *ires* =
(∃ *lm1 lm2 tn m ml mr rn*. *lm* @ 0^{tn} = *lm1* @ [*m*] @ *lm2*
∧ *length lm1* = *s* ∧ *m* + *1* = *ml* + *mr* ∧
ml ≠ *0* ∧ *tn* = *s* + *1* − *length lm* ∧
(*if lm1* = [] *then l* = Oc^{ml} @ *Bk* # *Bk* # *ires*
else l = (Oc^{ml})@[*Bk*]@<*rev lm1*>@
Bk # *Bk* # *ires*) ∧ (*r* = (Oc^{mr}) @ [*Bk*] @ <*lm2*>@ (Bk^{rn}) ∨
(*lm2* = [] ∧ *r* = (Oc^{mr})))
)

fun *inv-locate-a* :: *inc-inv-t*
where *inv-locate-a* (*as*, *lm*) (*s*, *l*, *r*) *ires* =
(*at-begin-norm* (*as*, *lm*) (*s*, *l*, *r*) *ires* ∨
at-begin-fst-bwtn (*as*, *lm*) (*s*, *l*, *r*) *ires* ∨
at-begin-fst-awtn (*as*, *lm*) (*s*, *l*, *r*) *ires*
)

fun *inv-locate-b* :: *inc-inv-t*
where *inv-locate-b* (*as*, *lm*) (*s*, *l*, *r*) *ires* =
(*in-middle* (*as*, *lm*) (*s*, *l*, *r*)) *ires*

fun *inv-after-write* :: *inc-inv-t*
where *inv-after-write* (*as*, *lm*) (*s*, *l*, *r*) *ires* =
(∃ *rn m lm1 lm2*. *lm* = *lm1* @ *m* # *lm2* ∧
(*if lm1* = [] *then l* = Oc^{m} @ *Bk* # *Bk* # *ires*
else Oc # *l* = $Oc^{Suc\ m}$ @ *Bk* # <*rev lm1*> @
Bk # *Bk* # *ires*) ∧ *r* = [*Oc*] @ <*lm2*> @ (Bk^{rn}))

fun *inv-after-move* :: *inc-inv-t*
where *inv-after-move* (*as*, *lm*) (*s*, *l*, *r*) *ires* =
(∃ *rn m lm1 lm2*. *lm* = *lm1* @ *m* # *lm2* ∧
(*if lm1* = [] *then l* = $Oc^{Suc\ m}$ @ *Bk* # *Bk* # *ires*
else l = $Oc^{Suc\ m}$@ *Bk* # <*rev lm1*> @ *Bk* # *Bk* # *ires*) ∧
r = <*lm2*> @ (Bk^{rn}))

fun *inv-after-clear* :: *inc-inv-t*
where *inv-after-clear* (*as*, *lm*) (*s*, *l*, *r*) *ires* =
(∃ *rn m lm1 lm2 r′*. *lm* = *lm1* @ *m* # *lm2* ∧
(*if lm1* = [] *then l* = $Oc^{Suc\ m}$ @ *Bk* # *Bk* # *ires*
else l = $Oc^{Suc\ m}$@ *Bk* # <*rev lm1*> @ *Bk* # *Bk* # *ires*) ∧
r = *Bk* # *r′* ∧ *Oc* # *r′* = <*lm2*> @ (Bk^{rn}))

fun *inv-on-right-moving* :: *inc-inv-t*
where *inv-on-right-moving* (*as*, *lm*) (*s*, *l*, *r*) *ires* =
(∃ *lm1 lm2 m ml mr rn*. *lm* = *lm1* @ [*m*] @ *lm2* ∧
ml + *mr* = *m* ∧
(*if lm1* = [] *then l* = Oc^{ml} @ *Bk* # *Bk* # *ires*
else l = (Oc^{ml}) @ [*Bk*] @ <*rev lm1*> @ *Bk* # *Bk* # *ires*) ∧
((*r* = (Oc^{mr}) @ [*Bk*] @ <*lm2*> @ (Bk^{rn})) ∨
(*r* = (Oc^{mr}) ∧ *lm2* = [])))

fun *inv-on-left-moving-norm* :: *inc-inv-t*
where *inv-on-left-moving-norm* (*as*, *lm*) (*s*, *l*, *r*) *ires* =
(∃ *lm1 lm2 m ml mr rn*. *lm* = *lm1* @ [*m*] @ *lm2* ∧
ml + *mr* = *Suc m* ∧ *mr* > *0* ∧ (*if lm1* = [] *then l* = Oc^{ml} @ *Bk* # *Bk* # *ires*
else l = (Oc^{ml}) @ *Bk* # <*rev lm1*> @ *Bk* # *Bk* # *ires*)
∧ (*r* = (Oc^{mr}) @ *Bk* # <*lm2*> @ (Bk^{rn}) ∨
(*lm2* = [] ∧ *r* = Oc^{mr})))

fun *inv-on-left-moving-in-middle-B*:: *inc-inv-t*
where *inv-on-left-moving-in-middle-B* (*as*, *lm*) (*s*, *l*, *r*) *ires* =
(∃ *lm1 lm2 rn*. *lm* = *lm1* @ *lm2* ∧

```
                    (if lm1 = [] then l = Bk # ires
                     else l = <rev lm1> @ Bk # Bk # ires) ∧
                     r = Bk # <lm2> @ (Bk^rn))

fun inv-on-left-moving :: inc-inv-t
  where inv-on-left-moving (as, lm) (s, l, r) ires =
       (inv-on-left-moving-norm  (as, lm) (s, l, r) ires ∨
        inv-on-left-moving-in-middle-B (as, lm) (s, l, r) ires)


fun inv-check-left-moving-on-leftmost :: inc-inv-t
  where inv-check-left-moving-on-leftmost (as, lm) (s, l, r) ires =
                (∃ rn. l = ires ∧ r = [Bk, Bk] @ <lm> @ (Bk^rn))

fun inv-check-left-moving-in-middle :: inc-inv-t
  where inv-check-left-moving-in-middle (as, lm) (s, l, r) ires =

              (∃ lm1 lm2 r' rn. lm = lm1 @ lm2 ∧
                (Oc # l = <rev lm1> @ Bk # Bk # ires) ∧ r = Oc # Bk # r' ∧
                         r' = <lm2> @ (Bk^rn))

fun inv-check-left-moving :: inc-inv-t
  where inv-check-left-moving (as, lm) (s, l, r) ires =
              (inv-check-left-moving-on-leftmost (as, lm) (s, l, r) ires ∨
              inv-check-left-moving-in-middle (as, lm) (s, l, r) ires)

fun inv-after-left-moving :: inc-inv-t
  where inv-after-left-moving (as, lm) (s, l, r) ires=
              (∃ rn. l = Bk # ires ∧ r = Bk # <lm> @ (Bk^rn))

fun inv-stop :: inc-inv-t
  where inv-stop (as, lm) (s, l, r) ires=
              (∃ rn. l = Bk # Bk # ires ∧ r = <lm> @ (Bk^rn))


fun inc-inv :: layout ⇒ nat ⇒ inc-inv-t
  where
  inc-inv ly n (as, lm) (s, l, r) ires =
              (let ss = start-of ly as in
               let lm' = abc-lm-s lm n ((abc-lm-v lm n)+1) in
                if s = 0 then False
                else if s < ss then False
                else if s < ss + 2 * n then
                   if (s − ss) mod 2 = 0 then
                       inv-locate-a (as, lm) ((s − ss) div 2, l, r) ires
                   else inv-locate-b (as, lm) ((s − ss) div 2, l, r) ires
                else if s = ss + 2 * n then
                        inv-locate-a (as, lm) (n, l, r) ires
                else if s = ss + 2 * n + 1 then
```

inv-locate-b (*as*, *lm*) (*n*, *l*, *r*) *ires*
else if $s = ss + 2 * n + 2$ *then*
inv-after-write (*as*, *lm'*) ($s - ss$, *l*, *r*) *ires*
else if $s = ss + 2 * n + 3$ *then*
inv-after-move (*as*, *lm'*) ($s - ss$, *l*, *r*) *ires*
else if $s = ss + 2 * n + 4$ *then*
inv-after-clear (*as*, *lm'*) ($s - ss$, *l*, *r*) *ires*
else if $s = ss + 2 * n + 5$ *then*
inv-on-right-moving (*as*, *lm'*) ($s - ss$, *l*, *r*) *ires*
else if $s = ss + 2 * n + 6$ *then*
inv-on-left-moving (*as*, *lm'*) ($s - ss$, *l*, *r*) *ires*
else if $s = ss + 2 * n + 7$ *then*
inv-check-left-moving (*as*, *lm'*) ($s - ss$, *l*, *r*) *ires*
else if $s = ss + 2 * n + 8$ *then*
inv-after-left-moving (*as*, *lm'*) ($s - ss$, *l*, *r*) *ires*
else if $s = ss + 2 * n + 9$ *then*
inv-stop (*as*, *lm'*) ($s - ss$, *l*, *r*) *ires*
else False)

lemma *fetch-intro*:
⟦⋀*xs*.⟦*ba* = *Oc* # *xs*⟧ ⟹ *P* (*fetch prog i Oc*);
⋀*xs*.⟦*ba* = *Bk* # *xs*⟧ ⟹ *P* (*fetch prog i Bk*);
ba = [] ⟹ *P* (*fetch prog i Bk*)
⟧ ⟹ *P* (*fetch prog i*
(*case ba of* [] ⇒ *Bk* | *Bk* # *xs* ⇒ *Bk* | *Oc* # *xs* ⇒ *Oc*))
by (*auto split*:*list.splits block.splits*)

lemma *length-findnth*[*simp*]: *length* (*findnth n*) = $4 * n$
apply(*induct n*, *simp*)
apply(*simp*)
done

declare *tshift.simps*[*simp del*]
declare *findnth.simps*[*simp del*]

lemma *findnth-nth*:
⟦$n > q$; $x < 4$⟧ ⟹
(*findnth n*) ! ($4 * q + x$) = (*findnth* (*Suc q*) ! ($4 * q + x$))
apply(*induct n*, *simp*)
apply(*case-tac* $q < n$, *simp add*: *findnth.simps*, *auto*)
apply(*simp add*: *nth-append*)
apply(*subgoal-tac* $q = n$, *simp*)
apply(*arith*)
done

lemma *Suc-pre*[*simp*]: ¬ *a* < *start-of ly as* ⟹
(*Suc a* − *start-of ly as*) = *Suc* (*a* − *start-of ly as*)
apply(*arith*)
done

```
lemma fetch-locate-a-o:
⋀a  q xs.
   ⟦¬ a < start-of (layout-of aprog) as;
    a < start-of (layout-of aprog) as + 2 ∗ n;
    a − start-of (layout-of aprog) as = 2 ∗ q;
    start-of (layout-of aprog) as > 0⟧
   ⟹ (fetch (ci (layout-of aprog) (start-of (layout-of aprog) as)
       (Inc n)) (Suc (2 ∗ q)) Oc) = (R, a+1)
apply(auto simp: ci.simps findnth.simps fetch.simps
                nth-of.simps tshift.simps nth-append Suc-pre)
apply(subgoal-tac (findnth n ! Suc (4 ∗ q)) =
               findnth (Suc q) ! (4 ∗ q + 1))
apply(simp add: findnth.simps nth-append)
apply(subgoal-tac  findnth n !(4 ∗ q + 1) =
               findnth (Suc q) ! (4 ∗ q + 1), simp)
apply(rule-tac findnth-nth, auto)
done

lemma fetch-locate-a-b:
⋀a  q xs.
   ⟦abc-fetch as aprog = Some (Inc n);
    ¬ a < start-of (layout-of aprog) as;
    a < start-of (layout-of aprog) as + 2 ∗ n;
    a − start-of (layout-of aprog) as = 2 ∗ q;
    start-of (layout-of aprog) as > 0⟧
   ⟹ (fetch (ci (layout-of aprog)
     (start-of (layout-of aprog) as) (Inc n)) (Suc (2 ∗ q)) Bk)
      = (W1, a)
apply(auto simp: ci.simps findnth.simps fetch.simps
               tshift.simps nth-append)
apply(subgoal-tac (findnth n ! (4 ∗ q)) =
                        findnth (Suc q) ! (4 ∗ q ))
apply(simp add: findnth.simps nth-append)
apply(subgoal-tac  findnth n !(4 ∗ q + 0) =
                         findnth (Suc q) ! (4 ∗ q + 0), simp)
apply(rule-tac findnth-nth, auto)
done

lemma [intro]: x mod 2 = Suc 0 ⟹ ∃ q. x = Suc (2 ∗ q)
apply(drule mod-eqD, auto)
done

lemma  add3-Suc: x + 3 = Suc (Suc (Suc x))
apply(arith)
done

declare start-of.simps[simp]
```

```
lemma [simp]:
 ⟦¬ a < start-of (layout-of aprog) as;
   a − start-of (layout-of aprog) as = Suc (2 ∗ q);
   abc-fetch as aprog = Some (Inc n);
   start-of (layout-of aprog) as > 0⟧
    =⇒ Suc (Suc (2 ∗ q + start-of (layout-of aprog) as − Suc 0)) = a
apply(subgoal-tac
Suc (Suc (2 ∗ q + start-of (layout-of aprog) as − Suc 0))
              = 2 + 2 ∗ q + start-of (layout-of aprog) as − Suc 0,
  simp, simp add: inc-startof-not0)
done

lemma fetch-locate-b-o:
⋀a xs.
    ⟦0 < a; ¬ a < start-of (layout-of aprog) as;
  a < start-of (layout-of aprog) as + 2 ∗ n;
 (a − start-of (layout-of aprog) as) mod 2 = Suc 0;
 start-of (layout-of aprog) as > 0⟧
    =⇒ (fetch (ci (layout-of aprog) (start-of (layout-of aprog) as)
      (Inc n)) (Suc (a − start-of (layout-of aprog) as)) Oc) = (R, a)
apply(auto simp: ci.simps findnth.simps fetch.simps
                  nth-of.simps tshift.simps nth-append)
apply(subgoal-tac ∃ q. (a − start-of (layout-of aprog) as) =
                         2 ∗ q + 1, auto)
apply(subgoal-tac (findnth n ! Suc (Suc (Suc (4 ∗ q))))
                   = findnth (Suc q) ! (4 ∗ q + 3))
apply(simp add: findnth.simps nth-append)
apply(subgoal-tac  findnth n ! (4 ∗ q + 3) =
                  findnth (Suc q) ! (4 ∗ q + 3), simp add: add3-Suc)
apply(rule-tac findnth-nth, auto)
done

lemma fetch-locate-b-b:
⋀a xs.
    ⟦0 < a;  ¬ a < start-of (layout-of aprog) as;
     a < start-of (layout-of aprog) as + 2 ∗ n;
     (a − start-of (layout-of aprog) as) mod 2 = Suc 0;
     start-of (layout-of aprog) as > 0⟧
    =⇒ (fetch (ci (layout-of aprog) (start-of (layout-of aprog) as)
       (Inc n)) (Suc (a − start-of (layout-of aprog) as)) Bk)
       = (R, a + 1)
apply(auto simp: ci.simps findnth.simps fetch.simps
                  nth-of.simps tshift.simps nth-append)
apply(subgoal-tac ∃ q. (a − start-of (layout-of aprog) as) =
                  2 ∗ q + 1, auto)
apply(subgoal-tac (findnth n ! Suc ((Suc (4 ∗ q)))) =
                    findnth (Suc q) ! (4 ∗ q + 2))
apply(simp add: findnth.simps nth-append)
apply(subgoal-tac  findnth n ! (4 ∗ q + 2) =
```

findnth (Suc q) ! (4 ∗ q + 2), simp)
apply(*rule-tac findnth-nth, auto*)
done

lemma *fetch-locate-n-a-o*:
start-of (layout-of aprog) as > 0
⟹ (fetch (ci (layout-of aprog)
(start-of (layout-of aprog) as) (Inc n)) (Suc (2 ∗ n)) Oc) =
(R, start-of (layout-of aprog) as + 2 ∗ n + 1)
apply(*auto simp: ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tinc-b-def)
done

lemma *fetch-locate-n-a-b*:
start-of (layout-of aprog) as > 0
⟹ (fetch (ci (layout-of aprog)
(start-of (layout-of aprog) as) (Inc n)) (Suc (2 ∗ n)) Bk)
= (W1, start-of (layout-of aprog) as + 2 ∗ n)
apply(*auto simp: ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tinc-b-def)
done

lemma *fetch-locate-n-b-o*:
start-of (layout-of aprog) as > 0
⟹ (fetch (ci (layout-of aprog) (start-of (layout-of aprog) as)
(Inc n)) (Suc (Suc (2 ∗ n))) Oc) =
(R, start-of (layout-of aprog) as + 2 ∗ n + 1)
apply(*auto simp: ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tinc-b-def)
done

lemma *fetch-locate-n-b-b*:
start-of (layout-of aprog) as > 0
⟹ (fetch (ci (layout-of aprog) (start-of (layout-of aprog) as)
(Inc n)) (Suc (Suc (2 ∗ n))) Bk) =
(W1, start-of (layout-of aprog) as + 2 ∗ n + 2)
apply(*auto simp: ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tinc-b-def)
done

lemma *fetch-after-write-o*:
start-of (layout-of aprog) as > 0
⟹ (fetch (ci (layout-of aprog) (start-of (layout-of aprog) as)
(Inc n)) (Suc (Suc (Suc (2 ∗ n)))) Oc) =
(R, start-of (layout-of aprog) as + 2∗n + 3)
apply(*auto simp: ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tinc-b-def)
done

lemma *fetch-after-move-o*:
start-of (*layout-of aprog*) *as* > *0*
⟹ (*fetch* (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Inc n*)) (*4* + *2* ∗ *n*) *Oc*)
= (*W0*, *start-of* (*layout-of aprog*) *as* + *2* ∗ *n* + *4*)
apply(*auto simp*: *ci.simps findnth.simps tshift.simps*
tinc-b-def add3-Suc)
apply(*subgoal-tac 4* + *2*∗*n* = *Suc* (*2*∗*n* + *3*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *fetch-after-move-b*:
start-of (*layout-of aprog*) *as* > *0*
⟹(*fetch* (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Inc n*)) (*4* + *2* ∗ *n*) *Bk*)
= (*L*, *start-of* (*layout-of aprog*) *as* + *2* ∗ *n* + *6*)
apply(*auto simp*: *ci.simps findnth.simps tshift.simps*
tinc-b-def add3-Suc)
apply(*subgoal-tac 4* + *2*∗*n* = *Suc* (*2*∗*n* + *3*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *fetch-clear-b*:
start-of (*layout-of aprog*) *as* > *0*
⟹ (*fetch* (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Inc n*)) (*5* + *2* ∗ *n*) *Bk*)
= (*R*, *start-of* (*layout-of aprog*) *as* + *2* ∗ *n* + *5*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tinc-b-def add3-Suc)
apply(*subgoal-tac 5* + *2*∗*n* = *Suc* (*2*∗*n* + *4*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *fetch-right-move-o*:
start-of (*layout-of aprog*) *as* > *0*
⟹ (*fetch* (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Inc n*)) (*6* + *2*∗*n*) *Oc*)
= (*R*, *start-of* (*layout-of aprog*) *as* + *2* ∗ *n* + *5*)
apply(*auto simp*: *ci.simps findnth.simps tshift.simps*
tinc-b-def add3-Suc)
apply(*subgoal-tac 6* + *2*∗*n* = *Suc* (*2*∗*n* + *5*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *fetch-right-move-b*:
start-of (*layout-of aprog*) *as* > *0*
⟹ (*fetch* (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Inc n*)) (*6* + *2*∗*n*) *Bk*)
= (*W1*, *start-of* (*layout-of aprog*) *as* + *2* ∗ *n* + *2*)

```
apply(auto simp: ci.simps findnth.simps
              tshift.simps tinc-b-def add3-Suc)
apply(subgoal-tac 6 + 2*n = Suc (2*n + 5), simp only: fetch.simps)
apply(auto simp: nth-of.simps nth-append)
done

lemma fetch-left-move-o:
    start-of (layout-of aprog) as > 0
    ⟹ (fetch (ci (layout-of aprog)
          (start-of (layout-of aprog) as) (Inc n)) (7 + 2*n) Oc)
    = (L, start-of (layout-of aprog) as + 2 * n + 6)
apply(auto simp: ci.simps findnth.simps tshift.simps
              tinc-b-def add3-Suc)
apply(subgoal-tac 7 + 2*n = Suc (2*n + 6), simp only: fetch.simps)
apply(auto simp: nth-of.simps nth-append)
done

lemma fetch-left-move-b:
    start-of (layout-of aprog) as > 0
    ⟹ (fetch (ci (layout-of aprog)
          (start-of (layout-of aprog) as) (Inc n)) (7 + 2*n) Bk)
    = (L, start-of (layout-of aprog) as + 2 * n + 7)
apply(auto simp: ci.simps findnth.simps
              tshift.simps tinc-b-def add3-Suc)
apply(subgoal-tac 7 + 2*n = Suc (2*n + 6), simp only: fetch.simps)
apply(auto simp: nth-of.simps nth-append)
done

lemma fetch-check-left-move-o:
    start-of (layout-of aprog) as > 0
    ⟹ (fetch (ci (layout-of aprog)
          (start-of (layout-of aprog) as) (Inc n)) (8 + 2*n) Oc)
    = (L, start-of (layout-of aprog) as + 2 * n + 6)
apply(auto simp: ci.simps findnth.simps tshift.simps tinc-b-def)
apply(subgoal-tac 8 + 2 * n = Suc (2 * n + 7),
                              simp only: fetch.simps)
apply(auto simp: nth-of.simps nth-append)
done

lemma fetch-check-left-move-b:
    start-of (layout-of aprog) as > 0
    ⟹ (fetch (ci (layout-of aprog)
         (start-of (layout-of aprog) as) (Inc n)) (8 + 2*n) Bk)
    = (R, start-of (layout-of aprog) as + 2 * n + 8)
apply(auto simp: ci.simps findnth.simps
              tshift.simps tinc-b-def add3-Suc)
apply(subgoal-tac 8 + 2*n= Suc (2*n + 7), simp only: fetch.simps)
apply(auto simp: nth-of.simps nth-append)
done
```

lemma *fetch-after-left-move*:
start-of (*layout-of aprog*) *as* > *0*
⟹ (*fetch* (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Inc n*)) (*9* + *2*∗*n*) *Bk*)
= (*R*, *start-of* (*layout-of aprog*) *as* + *2* ∗ *n* + *9*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tinc-b-def)
done

lemma *fetch-stop*:
start-of (*layout-of aprog*) *as* > *0*
⟹ (*fetch* (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Inc n*)) (*10* + *2* ∗*n*) *b*)
= (*Nop*, *0*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tinc-b-def
split: *block.splits*)
done

lemma *fetch-state0*:
(*fetch* (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Inc n*)) *0 b*)
= (*Nop*, *0*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tinc-b-def)
done

lemmas *fetch-simps* =
fetch-locate-a-o fetch-locate-a-b fetch-locate-b-o fetch-locate-b-b
fetch-locate-n-a-b fetch-locate-n-a-o fetch-locate-n-b-o
fetch-locate-n-b-b fetch-after-write-o fetch-after-move-o
fetch-after-move-b fetch-clear-b fetch-right-move-o
fetch-right-move-b fetch-left-move-o fetch-left-move-b
fetch-after-left-move fetch-check-left-move-o fetch-stop
fetch-state0 fetch-check-left-move-b

declare *exponent-def*[*simp del*] *tape-of-nat-list.simps*[*simp del*]
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*] *t-step.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*]
declare *tms-of.simps*[*simp del*] *tm-of.simps*[*simp del*]
layout-of.simps[*simp del*] *abc-fetch.simps* [*simp del*]
t-step.simps[*simp del*] *t-steps.simps*[*simp del*]
tpairs-of.simps[*simp del*] *start-of.simps*[*simp del*]
fetch.simps [*simp del*] *new-tape.simps* [*simp del*]
nth-of.simps [*simp del*] *ci.simps* [*simp del*]
length-of.simps[*simp del*]

lemma [*simp*]: $Suc\ (2 * q)\ mod\ 2 = Suc\ 0$
by *arith*

lemma [*simp*]: $Suc\ (2 * q)\ div\ 2 = q$
by *arith*

lemma [*simp*]: $\llbracket \neg\ a < start\text{-}of\ ly\ as;$
$a < start\text{-}of\ ly\ as + 2 * n;\ a - start\text{-}of\ ly\ as = 2 * q\rrbracket$
$\Longrightarrow Suc\ a < start\text{-}of\ ly\ as + 2 * n$
apply(*arith*)
done

lemma [*simp*]: $x\ mod\ 2 = Suc\ 0 \Longrightarrow (Suc\ x)\ mod\ 2 = 0$
by *arith*

lemma [*simp*]: $x\ mod\ 2 = Suc\ 0 \Longrightarrow (Suc\ x)\ div\ 2 = Suc\ (x\ div\ 2)$
by *arith*
lemma *exp-def*[*simp*]: $a^{Suc\ n} = a \# a^n$
by(*simp add*: *exponent-def*)
lemma [*intro*]: $Bk \# r = Oc^{mr} @ r' \Longrightarrow mr = 0$
by(*case-tac mr*, *auto simp*: *exponent-def*)

lemma [*intro*]: $Bk \# r = replicate\ mr\ Oc \Longrightarrow mr = 0$
by(*case-tac mr*, *auto*)
lemma *tape-of-nl-abv-cons*[*simp*]: $xs \neq [] \Longrightarrow$
$<x \# xs> = Oc^{Suc\ x} @ Bk \# <xs>$
apply(*simp add*: *tape-of-nl-abv tape-of-nat-list.simps*)
apply(*case-tac xs*, *simp*, *simp add*: *tape-of-nat-list.simps*)
done

lemma [*simp*]: $<[]::nat\ list> = []$
by(*auto simp*: *tape-of-nl-abv tape-of-nat-list.simps*)
lemma [*simp*]: $Oc \# r = <(lm::nat\ list)> @ Bk^{rn} \Longrightarrow lm \neq []$
apply(*auto simp*: *tape-of-nl-abv tape-of-nat-list.simps*)
apply(*case-tac rn*, *auto simp*: *exponent-def*)
done

lemma *BkCons-nil*: *Bk* # *xs* = <*lm*::*nat list*> @ Bk^{rn}⟹ *lm* = []
apply(*case-tac lm*, *simp*)
apply(*case-tac list*, *auto simp*: *tape-of-nl-abv tape-of-nat-list.simps*)
done
lemma *BkCons-nil'*: *Bk* # *xs* = <*lm*::*nat list*> @ Bk^{ln}⟹ *lm* = []
by(*auto intro*: *BkCons-nil*)

lemma *hd-tl-tape-of-nat-list*:
tl (*lm*::*nat list*) ≠ [] ⟹ <*lm*> = <*hd lm*> @ *Bk* # <*tl lm*>
apply(*frule tape-of-nl-abv-cons*[*of tl lm hd lm*])
apply(*simp add*: *tape-of-nat-abv Bk-def del*: *tape-of-nl-abv-cons*)
apply(*subgoal-tac lm* = *hd lm* # *tl lm*, *auto*)
apply(*case-tac lm*, *auto*)
done
lemma [*simp*]: *Oc* # *xs* = Oc^{mr} @ *Bk* # <*lm2*> @ Bk^{rn}⟹ *mr* > *0*
apply(*case-tac mr*, *auto simp*: *exponent-def*)
done

lemma *tape-of-nat-list-cons*: *xs* ≠ [] ⟹ *tape-of-nat-list* (*x* # *xs*) =
replicate (*Suc x*) *Oc* @ *Bk* # *tape-of-nat-list xs*
apply(*drule tape-of-nl-abv-cons*[*of xs x*])
apply(*auto simp*: *tape-of-nl-abv tape-of-nat-abv Oc-def Bk-def exponent-def*)
done

lemma *rev-eq*: *rev xs* = *rev ys* ⟹ *xs* = *ys*
by *simp*

lemma *tape-of-nat-list-eq*: *xs* = *ys* ⟹
tape-of-nat-list xs = *tape-of-nat-list ys*
by *simp*

lemma *tape-of-nl-nil-eq*: <(*lm*::*nat list*)> = [] = (*lm* = [])
apply(*simp add*: *tape-of-nl-abv tape-of-nat-list.simps*)
apply(*case-tac lm*, *simp add*: *tape-of-nat-list.simps*)
apply(*case-tac list*)
apply(*auto simp*: *tape-of-nat-list.simps*)
done

lemma *rep-ind*: *replicate* (*Suc n*) *a* = *replicate n a* @ [*a*]
apply(*induct n*, *simp*, *simp*)
done

lemma [*simp*]: *Oc* # *r* = <*lm*::*nat list*> @ *replicate rn Bk* ⟹ *Suc 0* ≤ *length lm*
apply(*rule-tac classical*, *auto*)
apply(*case-tac lm*, *simp*, *case-tac rn*, *auto*)
done
lemma *Oc-Bk-Cons*: *Oc* # *Bk* # *list* = <*lm*::*nat list*> @ Bk^{ln} ⟹
lm ≠ [] ∧ *hd lm* = *0*

```
apply(case-tac lm, simp, case-tac ln, simp add: exponent-def, simp add: exponent-def,
simp)
apply(case-tac lista, auto simp: tape-of-nl-abv tape-of-nat-list.simps)
done

lemma Oc-nil-zero[simp]: [Oc] = <lm::nat list> @ Bk^ln
          ⟹ lm = [0] ∧ ln = 0
apply(case-tac lm, simp)
apply(case-tac ln, auto simp: exponent-def)
apply(case-tac [!] list,
     auto simp: tape-of-nl-abv tape-of-nat-list.simps)
done

lemma  [simp]: Oc # r = <lm2> @ replicate rn Bk ⟹
    (∃ rn. r = replicate (hd lm2) Oc @ Bk # <tl lm2> @
                 replicate rn Bk) ∨
      tl lm2 = [] ∧ r = replicate (hd lm2) Oc
apply(rule-tac disjCI, simp)
apply(case-tac tl lm2 = [], simp)
apply(case-tac lm2, simp add: tape-of-nl-abv tape-of-nat-list.simps)
apply(case-tac rn, simp, simp, simp)
apply(simp add: tape-of-nl-abv tape-of-nat-list.simps exponent-def)
apply(case-tac rn, simp, simp)
apply(rule-tac x = rn in exI)
apply(simp add: hd-tl-tape-of-nat-list)
apply(simp add: tape-of-nat-abv Oc-def exponent-def)
done


lemma [simp]:
   inv-locate-a (as, lm) (q, l, Oc # r) ires
    ⟹ inv-locate-b (as, lm) (q, Oc # l, r) ires
apply(simp only: inv-locate-a.simps inv-locate-b.simps in-middle.simps
      at-begin-norm.simps at-begin-fst-bwtn.simps
      at-begin-fst-awtn.simps)
apply(erule disjE, erule exE, erule exE, erule exE)
apply(rule-tac x = lm1 in exI, rule-tac x = tl lm2 in exI, simp)
apply(rule-tac x = 0 in exI, rule-tac x = hd lm2 in exI,
           auto simp: exponent-def)
apply(rule-tac x = Suc 0 in exI, simp add:exponent-def)
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)
apply(simp only: rep-ind, simp)
apply(rule-tac x = Suc 0 in exI, auto)
apply(case-tac [1−3] rn, simp-all )
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*, *simp add*: *rep-ind del*: *replicate-Suc split*:*if-splits*)
apply(*rule-tac x = Suc 0* **in** *exI*, *auto*)
apply(*case-tac rn*, *simp*, *simp*)
apply(*rule-tac* [!] *x = Suc 0* **in** *exI*, *auto*)
apply(*case-tac* [!] *rn*, *simp-all*)
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*)
apply(*auto simp*: *Oc-def*)
by(*case-tac* [!] *rn*::*nat*, *auto simp*: *exponent-def*)
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* = []
proof(*rule-tac xs* = *xs* **and** *rn* = *rn* **in** *BkCons-nil*, *simp*)
qed
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: Oc-def exponent-def
                 tape-of-nl-abv tape-of-nat-list.simps)
      by(case-tac rn::nat, simp, simp)
    thus ?thesis by blast
  qed
qed

lemma [intro]: ∃ rn. [a] = a^rn
by(rule-tac x = Suc 0 in exI, simp add: exponent-def)

lemma [intro]: ∃ tn. [] = a^tn
apply(rule-tac x = 0 in exI, simp add: exponent-def)
done

lemma [intro]:  at-begin-norm (as, am) (q, aaa, []) ires
           ⟹ at-begin-norm (as, am) (q, aaa, [Bk]) ires
apply(simp add: at-begin-norm.simps, erule-tac exE, erule-tac exE)
apply(rule-tac x = lm1 in exI, simp, auto)
done

lemma [intro]: at-begin-fst-bwtn (as, am) (q, aaa, []) ires
          ⟹ at-begin-fst-bwtn (as, am) (q, aaa, [Bk]) ires
apply(simp only: at-begin-fst-bwtn.simps, erule-tac exE, erule-tac exE, erule-tac
exE)
apply(rule-tac x = am @ 0^tn in exI, auto)
done

lemma [intro]: at-begin-fst-awtn (as, am) (q, aaa, []) ires
          ⟹ at-begin-fst-awtn (as, am) (q, aaa, [Bk]) ires
apply(auto simp: at-begin-fst-awtn.simps)
done

lemma [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 [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 [simp]: inv-locate-b (as, am) (q, aaa, Oc # xs) ires
```

$\Longrightarrow$ *inv-locate-b* (*as*, *am*) (*q*, *Oc* # *aaa*, *xs*) *ires*
apply(*simp only*: *inv-locate-b.simps in-middle.simps*)
apply(*erule exE*)+
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 add*: *exponent-def*, *auto*)
done
lemma *zero-and-nil*[*intro*]: (*Bk* # Bk^n = Oc^{mr} @ *Bk* # <*lm*::*nat list*> @
Bk^{rn}) ∨ (*lm2* = [] ∧ *Bk* # Bk^n = Oc^{mr})
$\Longrightarrow$ *mr* = *0* ∧ *lm* = []
apply(*rule context-conjI*)
apply(*case-tac mr*, *auto simp*:*exponent-def*)
apply(*insert BkCons-nil*[*of replicate* (*n* − *1*) *Bk lm rn*])
apply(*case-tac n*, *auto simp*: *exponent-def Bk-def tape-of-nl-nil-eq*)
done

lemma *tape-of-nat-def*: <[*m*::*nat*]> = *Oc* # Oc^m
apply(*simp add*: *tape-of-nl-abv tape-of-nat-list.simps*)
done
lemma [*simp*]: ⟦*inv-locate-b* (*as*, *am*) (*q*, *aaa*, *Bk* # *xs*) *ires*; ∃ *n*. *xs* = Bk^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 disjI2*, *rule-tac disjI1*)
apply(*simp only*: *in-middle.simps at-begin-fst-bwtn.simps*)
apply(*erule-tac exE*)+
apply(*rule-tac x* = *lm1* @ [*m*] **in** *exI*, *rule-tac x* = *tn* **in** *exI*, *simp*)
apply(*subgoal-tac mr* = *0* ∧ *lm2* = [])
defer
apply(*rule-tac n* = *n* **and** *mr* = *mr* **and** *lm* = *lm2*
and *rn* = *rn* **and** *n* = *n* **in** *zero-and-nil*)
apply(*auto simp*: *exponent-def*)
apply(*case-tac lm1* = [], *auto simp*: *tape-of-nat-def*)
done

lemma *length-equal*: *xs* = *ys* $\Longrightarrow$ *length xs* = *length ys*
by *auto*
lemma [*simp*]: a^0 = []
by(*simp add*: *exp-zero*)

lemma [*simp*]: *length* (a^b) = *b*
apply(*simp add*: *exponent-def*)
done

lemma [*simp*]: ⟦*inv-locate-b* (*as*, *am*) (*q*, *aaa*, *Bk* # *xs*) *ires*;
¬ (∃ *n*. *xs* = Bk^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(*rule-tac x* = *lm1* @ [*m*] **in** *exI*, *rule-tac x* = *lm2* **in** *exI*, *simp*)
apply(*subgoal-tac tn* = *0*, *simp add*: *exponent-def* , *auto split*: *if-splits*)
apply(*case-tac* [!] *mr*, *simp-all add*: *tape-of-nat-def*, *auto*)
apply(*case-tac lm2*, *simp*, *erule-tac x* = *rn* **in** *allE*, *simp*)
apply(*case-tac am*, *simp*, *simp*)
apply(*case-tac lm2*, *simp*, *erule-tac x* = *rn* **in** *allE*, *simp*)
apply(*drule-tac length-equal*, *simp*)
done

lemma *locate-b-2-a*[*intro*]:
  inv-locate-b (*as*, *am*) (*q*, *aaa*, *Bk* # *xs*) *ires*
 ⟹ *inv-locate-a* (*as*, *am*) (*Suc q*, *Bk* # *aaa*, *xs*) *ires*
apply(*case-tac* ∃ *n*. *xs* = Bk^n, *simp*, *simp*)
done

lemma *locate-b-2-locate-a*[*simp*]:
 ⟦¬ *a* < *start-of ly as*;
  a < *start-of ly as* + *2* ∗ *n*;
  (*a* − *start-of ly as*) *mod 2* = *Suc 0*;
  inv-locate-b (*as*, *am*) ((*a* − *start-of ly as*) *div 2*, *aaa*, *Bk* # *xs*) *ires*⟧
 ⟹ (*Suc a* < *start-of ly as* + *2* ∗ *n* ⟶ *inv-locate-a* (*as*, *am*)
  (*Suc* ((*a* − *start-of ly as*) *div 2*), *Bk* # *aaa*, *xs*) *ires*) ∧
  (¬ *Suc a* < *start-of ly as* + *2* ∗ *n* ⟶
   inv-locate-a (*as*, *am*) (*n*, *Bk* # *aaa*, *xs*) *ires*)
apply(*auto*)
apply(*subgoal-tac n* > *0*)
apply(*subgoal-tac* (*a* − *start-of ly as*) *div 2* = *n* − *1*)
apply(*insert* locate-b-2-a [*of as am n* − *1 aaa xs*], *simp*)
apply(*arith*)
apply(*case-tac n*, *simp*, *simp*)
done

lemma [*simp*]: *inv-locate-b* (*as*, *am*) (*q*, *l*, []) *ires*
  ⟹ *inv-locate-b* (*as*, *am*) (*q*, *l*, [*Bk*]) *ires*
apply(*simp only*: *inv-locate-b.simps in-middle.simps*)
apply(*erule exE*)+
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*,
  rule-tac x = *ml* **in** *exI*, *rule-tac x* = *mr* **in** *exI*)
apply(*auto*)
done

lemma *locate-b-2-locate-a-B*[*simp*]:
⟦¬ *a* < *start-of ly as*;
 a < *start-of ly as* + *2* ∗ *n*;
 (*a* − *start-of ly as*) *mod 2* = *Suc 0*;

inv-locate-b (*as*, *am*) ((*a* − *start-of ly as*) *div 2*, *aaa*, []) *ires*⟧
⟹ (*Suc a* < *start-of ly as* + *2* ∗ *n* ⟶
inv-locate-a (*as*, *am*)
(*Suc* ((*a* − *start-of ly as*) *div 2*), *Bk* # *aaa*, []) *ires*)
∧ (¬ *Suc a* < *start-of ly as* + *2* ∗ *n* ⟶
inv-locate-a (*as*, *am*) (*n*, *Bk* # *aaa*, []) *ires*)
apply(*insert locate-b-2-locate-a* [*of a ly as n am aaa* []], *simp*)
done

lemma *inv-locate-b-2-after-write*[*simp*]:
inv-locate-b (*as*, *am*) (*n*, *aaa*, *Bk* # *xs*) *ires*
⟹ *inv-after-write* (*as*, *abc-lm-s am n* (*Suc* (*abc-lm-v am n*)))
(*Suc* (*Suc* (*2* ∗ *n*)), *aaa*, *Oc* # *xs*) *ires*
apply(*auto simp*: *in-middle.simps inv-after-write.simps*
abc-lm-v.simps abc-lm-s.simps inv-locate-b.simps)
apply(*subgoal-tac* [!] *mr* = *0*, *auto simp*: *exponent-def split*: *if-splits*)
apply(*subgoal-tac lm2* = [], *simp*)
apply(*rule-tac x* = *rn* **in** *exI*, *rule-tac x* = *Suc m* **in** *exI*,
rule-tac x = *lm1* **in** *exI*, *simp*, *rule-tac x* = [] **in** *exI*, *simp*)
apply(*case-tac Suc* (*length lm1*) − *length am*, *simp*, *simp only*: *rep-ind*, *simp*)
apply(*subgoal-tac length lm1* − *length am* = *nat*, *simp*, *arith*)
apply(*drule-tac length-equal*, *simp*)
done

lemma [*simp*]: *inv-locate-b* (*as*, *am*) (*n*, *aaa*, []) *ires* ⟹
inv-after-write (*as*, *abc-lm-s am n* (*Suc* (*abc-lm-v am n*)))
(*Suc* (*Suc* (*2* ∗ *n*)), *aaa*, [*Oc*]) *ires*
apply(*insert inv-locate-b-2-after-write* [*of as am n aaa* []])
by(*simp*)

lemma [*simp*]: *inv-after-write* (*as*, *lm*) (*Suc* (*Suc* (*2* ∗ *n*)), *l*, *Oc* # *r*) *ires*
⟹ *inv-after-move* (*as*, *lm*) (*2* ∗ *n* + *3*, *Oc* # *l*, *r*) *ires*
apply(*auto simp*:*inv-after-move.simps inv-after-write.simps split*: *if-splits*)
done

lemma [*simp*]: *inv-after-write* (*as*, *abc-lm-s am n* (*Suc* (*abc-lm-v am n*)
)) (*Suc* (*Suc* (*2* ∗ *n*)), *aaa*, *Bk* # *xs*) *ires* = *False*
apply(*simp add*: *inv-after-write.simps*)
done

lemma [*simp*]:
inv-after-write (*as*, *abc-lm-s am n* (*Suc* (*abc-lm-v am n*)))
(*Suc* (*Suc* (*2* ∗ *n*)), *aaa*, []) *ires* = *False*
apply(*simp add*: *inv-after-write.simps*)
done

lemma [*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*]:
  inv-after-move (*as*, *lm*) (*s*, *l*, *Bk* # *r*) *ires*
  ⟹ (*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*)
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(*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*, *simp*, *case-tac rn*)
apply(*auto simp*: *exponent-def intro*: *BkCons-nil split*: *if-splits*)
done

lemma [*elim*]: [] = <*lm*::*nat list*> ⟹ *lm* = []
using *tape-of-nl-nil-eq*[*of lm*]
by *simp*

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(*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*, *simp*, *case-tac rn*)
apply(*auto simp*: *exponent-def tape-of-nl-nil-eq intro*: *BkCons-nil split*: *if-splits*)
done

lemma [*simp*]: *Oc* # *r* = *replicate rn Bk* = *False*
apply(*case-tac rn*, *simp*, *simp*)
done

lemma *inv-after-clear-2-inv-on-right-moving*[*simp*]:

inv-after-clear (*as*, *lm*) (*2* ∗ *n* + *4*, *l*, *Bk* # *r*) *ires*
⟹ *inv-on-right-moving* (*as*, *lm*) (*2* ∗ *n* + *5*, *Bk* # *l*, *r*) *ires*
apply(*auto simp*: *inv-after-clear.simps inv-on-right-moving.simps*)
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*)
apply(*rule-tac x* = *0* **in** *exI*, *rule-tac x* = *hd lm2* **in** *exI*)
apply(*simp add*: *exponent-def*, *rule conjI*)
apply(*case-tac* [!] *lm2*::*nat list*, *auto simp*: *exponent-def*)
apply(*case-tac rn*, *auto split*: *if-splits simp*: *tape-of-nat-def*)
apply(*case-tac list*,
simp add: *tape-of-nl-abv tape-of-nat-list.simps exponent-def*)
apply(*erule-tac x* = *rn* − *1* **in** *allE*,
case-tac rn, *auto simp*: *exponent-def*)
apply(*case-tac list*,
simp add: *tape-of-nl-abv tape-of-nat-list.simps exponent-def*)
apply(*erule-tac x* = *rn* − *1* **in** *allE*,
case-tac rn, *auto simp*: *exponent-def*)
done

lemma [*simp*]: *inv-after-clear* (*as*, *lm*) (*2* ∗ *n* + *4*, *l*, []) *ires*⟹
inv-after-clear (*as*, *lm*) (*2* ∗ *n* + *4*, *l*, [*Bk*]) *ires*
by(*auto simp*: *inv-after-clear.simps*)

lemma [*simp*]: *inv-after-clear* (*as*, *lm*) (*2* ∗ *n* + *4*, *l*, []) *ires*
⟹ *inv-on-right-moving* (*as*, *lm*) (*2* ∗ *n* + *5*, *Bk* # *l*, []) *ires*
by(*insert*
inv-after-clear-2-inv-on-right-moving[*of as lm n l* []], *simp*)

lemma [*simp*]: *inv-on-right-moving* (*as*, *lm*) (*2* ∗ *n* + *5*, *l*, *Oc* # *r*) *ires*
⟹ *inv-on-right-moving* (*as*, *lm*) (*2* ∗ *n* + *5*, *Oc* # *l*, *r*) *ires*
apply(*auto simp*: *inv-on-right-moving.simps*)
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(*case-tac mr*, *auto simp*: *exponent-def*)
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*, *simp*)
apply(*case-tac mr*, *auto split*: *if-splits simp*: *exponent-def*)
done

lemma *inv-on-right-moving-2-inv-on-right-moving*[*simp*]:
inv-on-right-moving (*as*, *lm*) (*2* ∗ *n* + *5*, *l*, *Bk* # *r*) *ires*
⟹ *inv-after-write* (*as*, *lm*) (*Suc* (*Suc* (*2* ∗ *n*)), *l*, *Oc* # *r*) *ires*

apply(*auto simp*: *inv-on-right-moving.simps inv-after-write.simps*)
apply(*case-tac mr*, *auto simp*: *exponent-def split*: *if-splits*)
apply(*case-tac* [!] *mr*, *simp-all*)
done

lemma [*simp*]: *inv-on-right-moving* (*as*, *lm*) (*2* ∗ *n* + *5*, *l*, []) *ires*⟹
inv-on-right-moving (*as*, *lm*) (*2* ∗ *n* + *5*, *l*, [*Bk*]) *ires*
apply(*auto simp*: *inv-on-right-moving.simps exponent-def*)
apply(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = [] **in** *exI*, *simp*)
apply (*rule-tac x* = *m* **in** *exI*, *auto split*: *if-splits simp*: *exponent-def*)
done

lemma [*simp*]: *inv-on-right-moving* (*as*, *lm*) (*2* ∗ *n* + *5*, *l*, []) *ires*
⟹ *inv-after-write* (*as*, *lm*) (*Suc* (*Suc* (*2* ∗ *n*)), *l*, [*Oc*]) *ires*
apply(*rule-tac inv-on-right-moving-2-inv-on-right-moving*, *simp*)
done

lemma [*simp*]: *inv-on-left-moving-in-middle-B* (*as*, *lm*)
(*s*, *l*, *Oc* # *r*) *ires* = *False*
apply(*auto simp*: *inv-on-left-moving-in-middle-B.simps*)
done

lemma [*simp*]: *inv-on-left-moving-norm* (*as*, *lm*) (*s*, *l*, *Bk* # *r*) *ires*
= *False*
apply(*auto simp*: *inv-on-left-moving-norm.simps*)
apply(*case-tac* [!] *mr*, *auto simp*:)
done

lemma [*intro*]: ∃ *rna*. *Oc* # Oc^{m} @ *Bk* # <*lm*> @ Bk^{rn} = <*m* # *lm*> @ Bk^{rna}
apply(*case-tac lm*, *simp add*: *tape-of-nl-abv tape-of-nat-list.simps*)
apply(*rule-tac x* = *Suc rn* **in** *exI*, *simp*)
apply(*case-tac list*, *simp-all add*: *tape-of-nl-abv tape-of-nat-list.simps*, *auto*)
done

lemma [*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(*case-tac l*, *simp*, *simp*)
apply(*simp only*: *inv-on-left-moving-norm.simps*
inv-on-left-moving-in-middle-B.simps)
apply(*erule-tac exE*)+
apply(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = *m* # *lm2* **in** *exI*, *auto*)
apply(*case-tac* [!] *ml*, *auto*)
apply(*rule-tac* [!] *x* = *0* **in** *exI*, *simp-all add*: *tape-of-nl-abv tape-of-nat-list.simps*)
done

```
lemma [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(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: exponent-def split: if-splits)
done

lemma [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
apply(auto simp: inv-on-left-moving-norm.simps
          inv-on-left-moving-in-middle-B.simps split: if-splits)
done

lemma [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(case-tac l ≠ [], rule conjI, simp, simp)
apply(case-tac hd l, simp, simp, simp)
done


lemma [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(auto simp: inv-on-left-moving-in-middle-B.simps
          inv-check-left-moving-on-leftmost.simps split: if-splits)
apply(case-tac [!] rev lm1, simp-all)
apply(case-tac [!] lista, simp-all add: tape-of-nl-abv tape-of-nat-list.simps)
done

lemma [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 [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 [*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 [*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*)
apply(*simp add*: *inv-on-left-moving.simps inv-check-left-moving.simps*)
apply(*case-tac l*, *simp*, *simp*)
apply(*case-tac a*, *simp*, *simp*)
done

lemma [*simp*]: *inv-on-left-moving-norm* (*as*, *lm*) (*s*, *l*, []) *ires* = *False*
apply(*auto simp*: *inv-on-left-moving-norm.simps*)
by(*case-tac* [!] *mr*, *auto*)

lemma [*simp*]: *inv-on-left-moving* (*as*, *lm*) (*s*, *l*, []) *ires*⟹
inv-on-left-moving (*as*, *lm*) (*6* + *2* ∗ *n*, *l*, [*Bk*]) *ires*
apply(*simp add*: *inv-on-left-moving.simps*)
apply(*auto simp*: *inv-on-left-moving-in-middle-B.simps*)
done

lemma [*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 [*simp*]: *inv-on-left-moving* (*as*, *lm*) (*s*, *l*, []) *ires*
⟹ (*l* = [] ⟶ *inv-check-left-moving* (*as*, *lm*) (*s*′, [], [*Bk*]) *ires*) ∧
(*l* ≠ [] ⟶ *inv-check-left-moving* (*as*, *lm*) (*s*′, *tl l*, [*hd l*]) *ires*)
by *simp*

lemma *Oc-Bk-Cons-ex*[*simp*]:
Oc # *Bk* # *list* = <*lm*::*nat list*> @ Bk^{ln} ⟹
∃ *ln*. *list* = <*tl* (*lm*)> @ Bk^{ln}
apply(*case-tac lm*, *simp*)
apply(*case-tac ln*, *simp-all add*: *exponent-def*)
apply(*case-tac lista*,
auto simp: *tape-of-nl-abv tape-of-nat-list.simps exponent-def*)

apply(*case-tac* [!] *a*, *auto simp*:)
apply(*case-tac ln*, *simp*, *rule-tac x* = *nat* **in** *exI*, *simp*)
done

lemma [*simp*]:
Oc # *Bk* # *list* = <*rev lm1*::*nat list*> @ Bk^{ln} ⟹
∃ *rna*. *Oc* # *Bk* # <*lm2*> @ Bk^{rn} = <*hd* (*rev lm1*) # *lm2*> @ Bk^{rna}
apply(*frule Oc-Bk-Cons*, *simp*)
apply(*case-tac lm2*,
auto simp: *tape-of-nl-abv tape-of-nat-list.simps exponent-def*)
apply(*rule-tac x* = *Suc rn* **in** *exI*, *simp*)
done

lemma [*intro*]: ∃ *rna*. *a* # a^{rn} = a^{rna}
apply(*rule-tac x* = *Suc rn* **in** *exI*, *simp*)
done

lemma
inv-check-left-moving-in-middle-2-on-left-moving-in-middle-B[*simp*]:
inv-check-left-moving-in-middle (*as*, *lm*) (*s*, *Bk* # *list*, *Oc* # *r*) *ires*
⟹ *inv-on-left-moving-in-middle-B* (*as*, *lm*) (*s'*, *list*, *Bk* # *Oc* # *r*) *ires*
apply(*simp only*: *inv-check-left-moving-in-middle.simps*
inv-on-left-moving-in-middle-B.simps)
apply(*erule-tac exE*)+
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*,*simp-all add*: *tape-of-nl-abv tape-of-nat-list.simps*)
apply(*case-tac* [!] *a*, *simp-all*)
apply(*case-tac* [*1*] *lm2*, *simp-all add*: *tape-of-nat-list.simps*, *auto*)
apply(*case-tac* [*3*] *lm2*, *simp-all add*: *tape-of-nat-list.simps*, *auto*)
apply(*case-tac* [!] *lista*, *simp-all add*: *tape-of-nat-list.simps*)
done

lemma [*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 [*simp*]:
inv-check-left-moving-in-middle (*as*, *lm*) (*s*, [], *Oc* # *r*) *ires*
⟹ *inv-on-left-moving-in-middle-B* (*as*, *lm*) (*s'*, [], *Bk* # *Oc* # *r*) *ires*
apply(*insert*
inv-check-left-moving-in-middle-2-on-left-moving-in-middle-B[*of*
as lm n [] *r*], *simp*)
done

lemma [*simp*]: a^{0} = []

apply(*simp add*: *exponent-def*)
done

lemma [*simp*]: *inv-check-left-moving-in-middle* (*as*, *lm*)
(*s*, *Oc* # *list*, *Oc* # *r*) *ires*
$\Longrightarrow$ *inv-on-left-moving-norm* (*as*, *lm*) (*s'*, *list*, *Oc* # *Oc* # *r*) *ires*
apply(*auto simp*: *inv-check-left-moving-in-middle.simps*
inv-on-left-moving-norm.simps)
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* [!] *a*, *simp-all add*: *tape-of-nl-abv tape-of-nat-list.simps*, *auto*)
done

lemma [*simp*]: *inv-check-left-moving* (*as*, *lm*) (*s*, *l*, *Oc* # *r*) *ires*
$\Longrightarrow$ (*l* = [] $\longrightarrow$ *inv-on-left-moving* (*as*, *lm*) (*s'*, [], *Bk* # *Oc* # *r*) *ires*) $\wedge$
(*l* $\neq$ [] $\longrightarrow$ *inv-on-left-moving* (*as*, *lm*) (*s'*, *tl l*, *hd l* # *Oc* # *r*) *ires*)
apply(*case-tac l*,
auto simp: *inv-check-left-moving.simps inv-on-left-moving.simps*)
apply(*case-tac a*, *simp*, *simp*)
done

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

lemma [*simp*]:
inv-after-left-moving (*as*, *lm*) (*s*, *aaa*, *Oc* # *xs*) *ires* = *False*
by(*auto simp*: *inv-after-left-moving.simps*)

lemma *start-of-not0*: *as* $\neq$ *0* $\Longrightarrow$ *start-of ly as* > *0*
apply(*rule startof-not0*)
done

The single step currectness of the TM complied from Abacus instruction *Inc n*. It shows every single step execution of this TM keeps the invariant.

lemma *inc-inv-step*:
assumes
— Invariant holds on the start
h11: *inc-inv ly n* (*as*, *am*) *tc ires*
— The layout of Abacus program *aprog* is *ly*
and *h12*: *ly* = *layout-of aprog*
— The instruction at position *as* is *Inc n*
and *h21*: *abc-fetch as aprog* = *Some* (*Inc n*)
— TM not yet reach the final state, where *start-of ly as* + *2*∗*n* + *9* is the state where the current TM stops and the next TM starts.
and *h22*: (λ (*s*, *l*, *r*). *s* $\neq$ *start-of ly as* + *2*∗*n* + *9*) *tc*
shows
— Single step execution of the TM keeps the invaraint, where the TM compiled from *Inc n* is (*ci ly* (*start-of ly as*) (*Inc n*)) *start-of ly as* − *Suc 0*) is the offset used to execute this *shifted* TM.
inc-inv ly n (*as*, *am*) (*t-step tc* (*ci ly* (*start-of ly as*) (*Inc n*), *start-of ly as* − *Suc 0*)) *ires*
proof −
from *h21 h22* **have** *h3* : *start-of* (*layout-of aprog*) *as* > *0*
apply(*case-tac as*, *simp add*: *start-of.simps abc-fetch.simps*)
apply(*insert start-of-not0*[*of as layout-of aprog*], *simp*)
done
from *h11 h12* **and** *h21 h22* **and** *this* **show** *?thesis*
apply(*case-tac tc*, *simp*)
apply(*case-tac a* = *0*,
auto split:*if-splits simp add*:*t-step.simps*,
tactic ⟪ *ALLGOALS* (*resolve-tac* [@{*thm fetch-intro*}]) ⟫)
apply (*simp-all add*:*fetch-simps new-tape.simps*)

```
  done
qed


lemma t-steps-ind: t-steps tc (p, off) (Suc n)
              = t-step (t-steps tc (p, off) n) (p, off)
apply(induct n arbitrary: tc)
apply(simp add: t-steps.simps)
apply(simp add: t-steps.simps)
done

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 lex-square ::
    ((nat × nat × nat × nat) × (nat × nat × nat × nat)) set
  where lex-square ≡ less-than <*lex*> lex-triple

fun abc-inc-stage1 :: t-conf ⇒ nat ⇒ nat ⇒ nat
  where
  abc-inc-stage1 (s, l, r) ss n =
          (if s = 0 then 0
           else if s ≤ ss+2*n+1 then 5
           else if s≤ ss+2*n+5 then 4
           else if s ≤ ss+2*n+7 then 3
           else if s = ss+2*n+8 then 2
           else 1)

fun abc-inc-stage2 :: t-conf ⇒ nat ⇒ nat ⇒ nat
  where
  abc-inc-stage2 (s, l, r) ss n =
              (if s ≤ ss + 2*n + 1 then 0
               else if s = ss + 2*n + 2 then length r
               else if s = ss + 2*n + 3 then length r
               else if s = ss + 2*n + 4 then length r
               else if s = ss + 2*n + 5 then
                              if r ≠ [] then length r
                              else 1
               else if s = ss+2*n+6 then length l
               else if s = ss+2*n+7 then length l
               else 0)

fun abc-inc-stage3 :: t-conf ⇒ nat ⇒ nat ⇒ block list ⇒  nat
  where
```

```
  abc-inc-stage3 (s, l, r) ss n ires = (
            if s = ss + 2*n + 3 then 4
            else if s = ss + 2*n + 4 then 3
            else if s = ss + 2*n + 5 then
                 if r ≠ [] ∧ hd r = Oc then 2
                 else 1
            else if s = ss + 2*n + 2 then 0
            else if s = ss + 2*n + 6 then
                    if l = Bk # ires ∧ r ≠ [] ∧  hd r = Oc then 2
                    else 1
            else if s = ss + 2*n + 7 then
                    if r ≠ [] ∧ hd r = Oc then 3
                    else 0
            else ss+2*n+9 − s)
```

fun *abc-inc-stage4* :: *t-conf* ⇒ *nat* ⇒ *nat* ⇒ *block list* ⇒ *nat*
where

```
  abc-inc-stage4 (s, l, r) ss n ires =
          (if s ≤ ss+2*n+1 ∧ (s − ss) mod 2 = 0 then
              if (r≠[] ∧ hd r = Oc) then 0
              else 1
           else if (s ≤ ss+2*n+1 ∧ (s − ss) mod 2 = Suc 0)
                                           then length r
           else if s = ss + 2*n + 6 then
                if l = Bk # ires ∧ hd r = Bk then 0
                else Suc (length l)
           else 0)
```

fun *abc-inc-measure* :: (*t-conf* × *nat* × *nat* × *block list*) ⇒
(*nat* × *nat* × *nat* × *nat*)
where

```
  abc-inc-measure (c, ss, n, ires) =
    (abc-inc-stage1 c ss n, abc-inc-stage2 c ss n,
     abc-inc-stage3 c ss n ires, abc-inc-stage4 c ss n ires)
```

definition *abc-inc-LE* :: (((*nat* × *block list* × *block list*) × *nat* × *nat* × *block list*) × ((*nat* × *block list* × *block list*) × *nat* × *nat* × *block list*)) *set*
where *abc-inc-LE* ≡ (*inv-image lex-square abc-inc-measure*)

lemma *wf-lex-triple*: *wf lex-triple*
by (*auto intro*:*wf-lex-prod simp*:*lex-triple-def lex-pair-def*)

lemma *wf-lex-square*: *wf lex-square*
by (*auto intro*:*wf-lex-triple simp*:*lex-triple-def lex-square-def lex-pair-def*)

lemma *wf-abc-inc-le*[*intro*]: *wf abc-inc-LE*
by(*auto intro*:*wf-inv-image wf-lex-square simp*:*abc-inc-LE-def*)

```
declare inc-inv.simps[simp del]

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), simp)
apply(drule-tac f = f in wf-inv-image)
apply(simp add: inv-image-def)
apply(erule wf-induct, simp)
apply(erule-tac x = x in allE)
apply(erule-tac x = n in allE, erule-tac x = n in allE)
apply(erule-tac x = Suc x in allE, simp)
apply(rule-tac allI)
apply(induct-tac n, simp)
apply(erule-tac x = na in allE, 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;
    ∀ n. ((¬ P (f n) ∧ Q (f n)) ⟶ (Q (f (Suc n)) ∧ (f (Suc n), (f n)) ∈ LE));
    Q (f 0); ¬ P (f 0)⟧
  ⟹ ∃ 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(rule-tac x = na in exI, auto)
apply(rule halt-lemma2''', simp, simp, simp)
apply(erule-tac halt-lemma2'', simp)
done

lemma [simp]:
  ⟦ly = layout-of aprog; abc-fetch as aprog = Some (Inc n)⟧
   ⟹ start-of ly (Suc as) = start-of ly as + 2*n +9
apply(case-tac as, auto simp: abc-fetch.simps start-of.simps
```

layout-of.simps length-of.simps split: *if-splits*)
done

lemma *inc-inv-init*:
⟦*abc-fetch as aprog* = *Some* (*Inc n*);
crsp-l ly (*as*, *am*) (*start-of ly as*, *l*, *r*) *ires*; *ly* = *layout-of aprog*⟧
⟹ *inc-inv ly n* (*as*, *am*) (*start-of ly as*, *l*, *r*) *ires*
apply(*auto simp*: *crsp-l.simps inc-inv.simps*
inv-locate-a.simps at-begin-fst-bwtn.simps
at-begin-fst-awtn.simps at-begin-norm.simps)
apply(*auto intro*: *startof-not0*)
done

lemma *inc-inv-stop-pre*[*simp*]:
⟦*ly* = *layout-of aprog*; *inc-inv ly n* (*as*, *am*) (*s*, *l*, *r*) *ires*;
s = *start-of ly as*; *abc-fetch as aprog* = *Some* (*Inc n*)⟧
⟹ (∀ *na*. ¬ (λ((*s*, *l*, *r*), *ss*, *n'*, *ires'*). *s* = *start-of ly* (*Suc as*))
(*t-steps* (*s*, *l*, *r*) (*ci ly* (*start-of ly as*)
(*Inc n*), *start-of ly as* − *Suc 0*) *na*, *s*, *n*, *ires*) ∧
(λ((*s*, *l*, *r*), *ss*, *n'*, *ires'*). *inc-inv ly n* (*as*, *am*) (*s*, *l*, *r*) *ires'*)
(*t-steps* (*s*, *l*, *r*) (*ci ly* (*start-of ly as*)
(*Inc n*), *start-of ly as* − *Suc 0*) *na*, *s*, *n*, *ires*) ⟶
(λ((*s*, *l*, *r*), *ss*, *n'*, *ires'*). *inc-inv ly n* (*as*, *am*) (*s*, *l*, *r*) *ires'*)
(*t-steps* (*s*, *l*, *r*) (*ci ly* (*start-of ly as*)
(*Inc n*), *start-of ly as* − *Suc 0*) (*Suc na*), *s*, *n*, *ires*) ∧
((*t-steps* (*s*, *l*, *r*) (*ci ly* (*start-of ly as*) (*Inc n*),
start-of ly as − *Suc 0*) (*Suc na*), *s*, *n*, *ires*),
t-steps (*s*, *l*, *r*) (*ci ly* (*start-of ly as*)
(*Inc n*), *start-of ly as* − *Suc 0*) *na*, *s*, *n*, *ires*) ∈ *abc-inc-LE*)
apply(*rule allI*, *rule impI*, *simp add*: *t-steps-ind*,
rule conjI, *erule-tac conjE*)
apply(*rule-tac inc-inv-step*, *simp*, *simp*, *simp*)
apply(*case-tac t-steps* (*start-of* (*layout-of aprog*) *as*, *l*, *r*) (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Inc n*), *start-of* (*layout-of aprog*) *as* − *Suc 0*)
na, *simp*)
proof −
fix *na*
assume *h1*: *abc-fetch as aprog* = *Some* (*Inc n*)
¬ (λ(*s*, *l*, *r*) (*ss*, *n'*, *ires'*). *s* = *start-of* (*layout-of aprog*) *as* + *2* ∗ *n* + *9*)
(*t-steps* (*start-of* (*layout-of aprog*) *as*, *l*, *r*) (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Inc n*), *start-of* (*layout-of aprog*) *as* − *Suc 0*)
na)
(*start-of* (*layout-of aprog*) *as*, *n*, *ires*) ∧
inc-inv (*layout-of aprog*) *n* (*as*, *am*) (*t-steps* (*start-of* (*layout-of aprog*) *as*, *l*,
r)
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*) (*Inc n*), *start-of* (*layout-of*
aprog) *as* − *Suc 0*) *na*) *ires*
from *h1* **have** *h2*: *start-of* (*layout-of aprog*) *as* > *0*
apply(*rule-tac startof-not0*)

done
from *h1* **and** *h2* **show** ((*t-step* (*t-steps* (*start-of* (*layout-of aprog*) *as*, *l*, *r*) (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Inc n*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *na*)
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*) (*Inc n*), *start-of* (*layout-of aprog*) *as* − *Suc 0*),
start-of (*layout-of aprog*) *as*, *n*, *ires*),
t-steps (*start-of* (*layout-of aprog*) *as*, *l*, *r*)
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*) (*Inc n*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *na*,
start-of (*layout-of aprog*) *as*, *n*, *ires*)
∈ *abc-inc-LE*
apply(*case-tac* (*t-steps* (*start-of* (*layout-of aprog*) *as*, *l*, *r*)
(*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Inc n*),
start-of (*layout-of aprog*) *as* − *Suc 0*) *na*), *simp*)
apply(*case-tac a* = *0*,
auto split:*if-splits simp add*:*t-step.simps inc-inv.simps*,
tactic ⟪ *ALLGOALS* (*resolve-tac* [@{*thm fetch-intro*}]) ⟫)
apply(*simp-all add*:*fetch-simps new-tape.simps*)
apply(*auto simp add*: *abc-inc-LE-def*
lex-square-def lex-triple-def lex-pair-def
inv-after-write.simps inv-after-move.simps inv-after-clear.simps
inv-on-left-moving.simps inv-on-left-moving-norm.simps split: *if-splits*)
done
qed

lemma *inc-inv-stop-pre1*:
⟦
ly = *layout-of aprog*;
abc-fetch as aprog = *Some* (*Inc n*);
s = *start-of ly as*;
inc-inv ly n (*as*, *am*) (*s*, *l*, *r*) *ires*
⟧ ⟹
(∃ *stp* > *0*. (λ (*s'*, *l'*, *r'*).
s' = *start-of ly* (*Suc as*) ∧
inc-inv ly n (*as*, *am*) (*s'*, *l'*, *r'*) *ires*)
(*t-steps* (*s*, *l*, *r*) (*ci ly* (*start-of ly as*) (*Inc n*),
start-of ly as − *Suc 0*) *stp*))
apply(*insert halt-lemma2*[*of abc-inc-LE*
λ ((*s*, *l*, *r*), *ss*, *n'*, *ires'*). *s* = *start-of ly* (*Suc as*)
(λ *stp*. (*t-steps* (*s*, *l*, *r*)
(*ci ly* (*start-of ly as*) (*Inc n*),
start-of ly as − *Suc 0*) *stp*, *s*, *n*, *ires*))
λ ((*s*, *l*, *r*), *ss*, *n'*). *inc-inv ly n* (*as*, *am*) (*s*, *l*, *r*) *ires*])
apply(*insert wf-abc-inc-le*)
apply(*insert inc-inv-stop-pre*[*of ly aprog n as am s l r ires*], *simp*)
apply(*simp only*: *t-steps.simps*, *auto*)

apply(*rule-tac x = na* **in** *exI*)
apply(*case-tac* (*t-steps* (*start-of* (*layout-of aprog*) *as*, *l*, *r*)
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Inc n*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *na*), *simp*)
apply(*case-tac na*, *simp add*: *t-steps.simps*, *simp*)
done

lemma *inc-inv-stop*:
assumes *program-and-layout*:
— There is an Abacus program *aprog* and its layout is *ly*:
ly = *layout-of aprog*
and *an-instruction*:
— There is an instruction *Inc n* at postion *as* of *aprog*
abc-fetch as aprog = *Some* (*Inc n*)
and *the-start-state*:
— According to *ly* and *as*, the start state of the TM compiled from this *Inc n* instruction should be *s*:
s = *start-of ly as*
and *inv*:
— Invariant holds on configuration (*s*, *l*, *r*)
inc-inv ly n (*as*, *am*) (*s*, *l*, *r*) *ires*
shows — After *stp* steps of execution, the compiled TM reaches the start state of next compiled TM and the invariant still holds.
(∃ *stp* > *0*. (λ (*s'*, *l'*, *r'*).
s' = *start-of ly* (*Suc as*) ∧
inc-inv ly n (*as*, *am*) (*s'*, *l'*, *r'*) *ires*)
(*t-steps* (*s*, *l*, *r*) (*ci ly* (*start-of ly as*) (*Inc n*),
start-of ly as − *Suc 0*) *stp*))
proof −
from *inc-inv-stop-pre1* [*OF* *program-and-layout an-instruction the-start-state inv*]
show *?thesis* **.**
qed

lemma *inc-inv-stop-cond*:
⟦*ly* = *layout-of aprog*;
s' = *start-of ly* (*as* + *1*);
inc-inv ly n (*as*, *lm*) (*s'*, (*l'*, *r'*)) *ires*;
abc-fetch as aprog = *Some* (*Inc n*)⟧ ⟹
crsp-l ly (*Suc as*, *abc-lm-s lm n* (*Suc* (*abc-lm-v lm n*)))
(*s'*, *l'*, *r'*) *ires*
apply(*subgoal-tac s'* = *start-of ly as* + *2*∗*n* + *9*, *simp*)
apply(*auto simp*: *inc-inv.simps inv-stop.simps crsp-l.simps*)
done

lemma *inc-crsp-ex-pre*:
⟦*ly* = *layout-of aprog*;
crsp-l ly (*as*, *am*) *tc ires*;
abc-fetch as aprog = *Some* (*Inc n*)⟧

$\Longrightarrow \exists\, stp > 0.\ crsp\text{-}l\ ly\ (abc\text{-}step\text{-}l\ (as,\ am)\ (Some\ (Inc\ n)))$
$(t\text{-}steps\ tc\ (ci\ ly\ (start\text{-}of\ ly\ as)\ (Inc\ n),$
$start\text{-}of\ ly\ as - Suc\ 0)\ stp)\ ires$

proof(*case-tac tc*, *simp add*: *abc-step-l.simps*)
fix *a b c*
assume *h1*: *ly = layout-of aprog*
crsp-l (layout-of aprog) (as, am) (a, b, c) ires
abc-fetch as aprog = Some (Inc n)
hence *h2*: *a = start-of ly as*
by(*auto simp*: *crsp-l.simps*)
from *h1* **and** *h2* **have** *h3*:
inc-inv ly n (as, am) (start-of ly as, b, c) ires
by(*rule-tac inc-inv-init*, *simp*, *simp*, *simp*)
from *h1* **and** *h2* **and** *h3* **have** *h4*:
$(\exists\ stp > 0.\ (\lambda\ (s', l', r').\ s' =$
$start\text{-}of\ ly\ (Suc\ as) \wedge inc\text{-}inv\ ly\ n\ (as,\ am)\ (s',\ l',\ r')\ ires)$
$(t\text{-}steps\ (a,\ b,\ c)\ (ci\ ly\ (start\text{-}of\ ly\ as)$
$(Inc\ n),\ start\text{-}of\ ly\ as - Suc\ 0)\ stp))$
apply(*rule-tac inc-inv-stop*, *auto*)
done
from *h1* **and** *h2* **and** *h3* **and** *h4* **show**
$\exists\, stp > 0.\ crsp\text{-}l\ (layout\text{-}of\ aprog)$
$(Suc\ as,\ abc\text{-}lm\text{-}s\ am\ n\ (Suc\ (abc\text{-}lm\text{-}v\ am\ n)))$
$(t\text{-}steps\ (a,\ b,\ c)\ (ci\ (layout\text{-}of\ aprog)$
$(start\text{-}of\ (layout\text{-}of\ aprog)\ as)\ (Inc\ n),$
$start\text{-}of\ (layout\text{-}of\ aprog)\ as - Suc\ 0)\ stp)\ ires$
apply(*erule-tac exE*)
apply(*rule-tac x = stp* **in** *exI*)
apply(*case-tac (t-steps (a, b, c) (ci (layout-of aprog)*
(start-of (layout-of aprog) as) (Inc n),
start-of (layout-of aprog) as − Suc 0) stp), *simp*)
apply(*rule-tac inc-inv-stop-cond*, *auto*)
done
qed

The total correctness of the compilaton of *Inc n* instruction.

lemma *inc-crsp-ex*:
assumes *layout*:
— For any Abacus program *aprog*, assuming its layout is *ly*
ly = layout-of aprog
and *corresponds*:
— Abacus configuration (*as*, *am*) is in correspondence with TM configuration *tc*
crsp-l ly (as, am) tc ires
and *inc*:
— There is an instruction *Inc n* at postion *as* of *aprog*
abc-fetch as aprog = Some (Inc n)
shows
— After *stp* steps of execution, the TM compiled from this *Inc n* stops with a configuration which corresponds to the Abacus configuration obtained from the

execution of this *Inc n* instruction.
∃ *stp* > *0*. *crsp-l ly* (*abc-step-l* (*as*, *am*) (*Some* (*Inc n*)))
(*t-steps tc* (*ci ly* (*start-of ly as*) (*Inc n*),
start-of ly as − *Suc 0*) *stp*) *ires*
proof −
from *inc-crsp-ex-pre* [*OF layout corresponds inc*] **show** *?thesis* **.**
qed

The lemmas in this section lead to the correctness of the compilation of *Dec n e* instruction using the same techniques as *Inc n*.

type-synonym *dec-inv-t* = (*nat* ∗ *nat list*) ⇒ *t-conf* ⇒ *block 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 ss ≤ s ∧ s < ss + 2∗n then
if (s − ss) mod 2 = 0 then
inv-locate-a (as, am) ((s − ss) div 2, l, r) ires
∨ inv-locate-a (as, am′′) ((s − ss) div 2, l, r) ires
else
inv-locate-b (as, am) ((s − ss) div 2, l, r) ires
∨ inv-locate-b (as, am′′) ((s − ss) div 2, l, r) ires
else if s = ss + 2 ∗ n then
inv-locate-a (as, am) (n, l, r) ires
∨ inv-locate-a (as, am′′) (n, l, r) ires
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)

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 then False
else if ss ≤ s ∧ s < ss + 2∗n then
if (s − ss) mod 2 = 0 then
inv-locate-a (as, am) ((s − ss) div 2, l, r) ires
else inv-locate-b (as, am) ((s − ss) div 2, l, r) ires
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)

lemma *dec-fetch-locate-a-o*:
⟦*start-of ly as* ≤ *a*;
a < *start-of ly as* + *2* ∗ *n*; *start-of ly as* > *0*;
a − *start-of ly as* = *2* ∗ *q*⟧
⟹ *fetch* (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*Suc* (*2* ∗ *q*)) *Oc* = (*R*, *a* + *1*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append Suc-pre)
apply(*subgoal-tac* (*findnth n* ! *Suc* (*4* ∗ *q*)) =
findnth (*Suc q*) ! (*4* ∗ *q* + *1*))
apply(*simp add*: *findnth.simps nth-append*)
apply(*subgoal-tac* *findnth n* !(*4* ∗ *q* + *1*) =
findnth (*Suc q*) ! (*4* ∗ *q* + *1*), *simp*)
apply(*rule-tac findnth-nth*, *auto*)
done

lemma *dec-fetch-locate-a-b*:
⟦*start-of ly as* ≤ *a*;
a < *start-of ly as* + *2* ∗ *n*;
start-of ly as > *0*;
a − *start-of ly as* = *2* ∗ *q*⟧
⟹ *fetch* (*ci* (*layout-of aprog*) (*start-of ly as*) (*Dec n e*))
(*Suc* (*2* ∗ *q*)) *Bk* = (*W1*, *a*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append)
apply(*subgoal-tac* (*findnth n* ! (*4* ∗ *q*)) =
findnth (*Suc q*) ! (*4* ∗ *q*))
apply(*simp add*: *findnth.simps nth-append*)
apply(*subgoal-tac* *findnth n* !(*4* ∗ *q* + *0*) =
findnth (*Suc q*) ! (*4* ∗ *q* + *0*), *simp*)

apply(*rule-tac findnth-nth*, *auto*)
done

lemma *dec-fetch-locate-b-o*:
$[\![$*start-of ly as* $\leq$ *a*;
a $<$ *start-of ly as* $+$ *2* $*$ *n*;
(*a* $-$ *start-of ly as*) *mod 2* $=$ *Suc 0*;
start-of ly as$>$ *0*$]\!]$
$\Longrightarrow$ *fetch* (*ci* (*layout-of aprog*) (*start-of ly as*) (*Dec n e*))
(*Suc* (*a* $-$ *start-of ly as*)) *Oc* $=$ (*R*, *a*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append)
apply(*subgoal-tac* $\exists$ *q*. (*a* $-$ *start-of ly as*) $=$ *2* $*$ *q* $+$ *1*, *auto*)
apply(*subgoal-tac* (*findnth n* ! *Suc* (*Suc* (*Suc* (*4* $*$ *q*)))) $=$
findnth (*Suc q*) ! (*4* $*$ *q* $+$ *3*))
apply(*simp add*: *findnth.simps nth-append*)
apply(*subgoal-tac* *findnth n* ! (*4* $*$ *q* $+$ *3*) $=$
findnth (*Suc q*) ! (*4* $*$ *q* $+$ *3*), *simp add*: *add3-Suc*)
apply(*rule-tac findnth-nth*, *auto*)
done

lemma *dec-fetch-locate-b-b*:
$[\![\neg$ *a* $<$ *start-of ly as*;
a $<$ *start-of ly as* $+$ *2* $*$ *n*;
(*a* $-$ *start-of ly as*) *mod 2* $=$ *Suc 0*;
start-of ly as $>$ *0*$]\!]$
$\Longrightarrow$ *fetch* (*ci* (*layout-of aprog*) (*start-of ly as*) (*Dec n e*))
(*Suc* (*a* $-$ *start-of ly as*)) *Bk* $=$ (*R*, *a* $+$ *1*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append)
apply(*subgoal-tac* $\exists$ *q*. (*a* $-$ *start-of ly as*) $=$ *2* $*$ *q* $+$ *1*, *auto*)
apply(*subgoal-tac* (*findnth n* ! *Suc* ((*Suc* (*4* $*$ *q*)))) $=$
findnth (*Suc q*) ! (*4* $*$ *q* $+$ *2*))
apply(*simp add*: *findnth.simps nth-append*)
apply(*subgoal-tac* *findnth n* ! (*4* $*$ *q* $+$ *2*) $=$
findnth (*Suc q*) ! (*4* $*$ *q* $+$ *2*), *simp*)
apply(*rule-tac findnth-nth*, *auto*)
done

lemma *dec-fetch-locate-n-a-o*:
start-of ly as $>$ *0* $\Longrightarrow$ *fetch* (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*Suc* (*2* $*$ *n*)) *Oc*
$=$ (*R*, *start-of ly as* $+$ *2*$*$*n* $+$ *1*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tdec-b-def)
done

lemma *dec-fetch-locate-n-a-b*:
start-of ly as $>$ *0* $\Longrightarrow$ *fetch* (*ci* (*layout-of aprog*)

(*start-of ly as*) (*Dec n e*)) (*Suc* (*2* ∗ *n*)) *Bk*
= (*W1*, *start-of ly as* + *2*∗*n*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tdec-b-def)
done

lemma *dec-fetch-locate-n-b-o*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*Suc* (*Suc* (*2* ∗ *n*))) *Oc*
= (*R*, *start-of ly as* + *2*∗*n* + *2*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tdec-b-def)
done

lemma *dec-fetch-locate-n-b-b*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*Suc* (*Suc* (*2* ∗ *n*))) *Bk*
= (*L*, *start-of ly as* + *2*∗*n* + *13*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tdec-b-def)
done

lemma *dec-fetch-first-on-right-move-o*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*Suc* (*Suc* (*Suc* (*2* ∗ *n*)))) *Oc*
= (*R*, *start-of ly as* + *2*∗*n* + *2*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tdec-b-def)
done

lemma *dec-fetch-first-on-right-move-b*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*) (*start-of ly as*) (*Dec n e*))
(*Suc* (*Suc* (*Suc* (*2* ∗ *n*)))) *Bk*
= (*L*, *start-of ly as* + *2*∗*n* + *3*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append tdec-b-def)
done

lemma [*simp*]: *fetch x* (*a* + *2* ∗ *n*) *b* = *fetch x* (*2* ∗ *n* + *a*) *b*
thm *arg-cong*
apply(*rule-tac x* = *a* + *2*∗*n* **and** *y* = *2*∗*n* + *a* **in** *arg-cong*, *simp*)
done

lemma *dec-fetch-first-after-clear-o*:

start-of ly as > *0* ⟹ *fetch* (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *4*) *Oc*
= (*W0*, *start-of ly as* + *2*∗*n* + *3*)
apply(*auto simp*: *ci.simps findnth.simps tshift.simps*
tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *4* = *Suc* (*2*∗*n* + *3*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-first-after-clear-b*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *4*) *Bk*
= (*R*, *start-of ly as* + *2*∗*n* + *4*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *4*= *Suc* (*2*∗*n* + *3*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-right-move-b*:
start-of ly as > *0* ⟹ *fetch* (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *5*) *Bk*
= (*R*, *start-of ly as* + *2*∗*n* + *5*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *5*= *Suc* (*2*∗*n* + *4*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-check-right-move-b*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *6*) *Bk*
= (*L*, *start-of ly as* + *2*∗*n* + *6*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *6* = *Suc* (*2*∗*n* + *5*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-check-right-move-o*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*) (*start-of ly as*)
(*Dec n e*)) (*2* ∗ *n* + *6*) *Oc*
= (*L*, *start-of ly as* + *2*∗*n* + *7*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *6* = *Suc* (*2*∗*n* + *5*), *simp only*: *fetch.simps*)

apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-left-move-b*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *7*) *Bk*
= (*L*, *start-of ly as* + *2*∗*n* + *10*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *7* = *Suc* (*2*∗*n* + *6*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-after-write-b*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *8*) *Bk*
= (*W1*, *start-of ly as* + *2*∗*n* + *7*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *8* = *Suc* (*2*∗*n* + *7*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-after-write-o*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *8*) *Oc*
= (*R*, *start-of ly as* + *2*∗*n* + *8*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *8* = *Suc* (*2*∗*n* + *7*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-on-right-move-b*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *9*) *Bk*
= (*L*, *start-of ly as* + *2*∗*n* + *9*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *9* = *Suc* (*2*∗*n* + *8*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-on-right-move-o*:
start-of ly as > *0* ⟹

fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *9*) *Oc*
= (*R*, *start-of ly as* + *2*∗*n* + *8*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *9* = *Suc* (*2*∗*n* + *8*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-after-clear-b*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *10*) *Bk*
= (*R*, *start-of ly as* + *2*∗*n* + *4*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *10* = *Suc* (*2*∗*n* + *9*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-after-clear-o*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *10*) *Oc*
= (*W0*, *start-of ly as* + *2*∗*n* + *9*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *10*= *Suc* (*2*∗*n* + *9*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-on-left-move1-o*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *11*) *Oc*
= (*L*, *start-of ly as* + *2*∗*n* + *10*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *11*= *Suc* (*2*∗*n* + *10*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-on-left-move1-b*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *11*) *Bk*
= (*L*, *start-of ly as* + *2*∗*n* + *11*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)

apply(*subgoal-tac 2∗n + 11 = Suc (2∗n + 10)*,
simp only: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-check-left-move1-o*:
start-of ly as > 0 ⟹
fetch (ci (layout-of aprog)
(start-of ly as) (Dec n e)) (2 ∗ n + 12) Oc
= (L, start-of ly as + 2∗n + 10)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2∗n + 12= Suc (2∗n + 11)*, *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-check-left-move1-b*:
start-of ly as > 0 ⟹
fetch (ci (layout-of aprog)
(start-of ly as) (Dec n e)) (2 ∗ n + 12) Bk
= (R, start-of ly as + 2∗n + 12)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2∗n + 12 = Suc (2∗n + 11)*,
simp only: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-after-left-move1-b*:
start-of ly as > 0 ⟹
fetch (ci (layout-of aprog)
(start-of ly as) (Dec n e)) (2 ∗ n + 13) Bk
= (R, start-of ly as + 2∗n + 16)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2∗n + 13 = Suc (2∗n + 12)*,
simp only: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-on-left-move2-o*:
start-of ly as > 0 ⟹
fetch (ci (layout-of aprog)
(start-of ly as) (Dec n e)) (2 ∗ n + 14) Oc
= (L, start-of ly as + 2∗n + 13)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2∗n + 14 = Suc (2∗n + 13)*,
simp only: *fetch.simps*)

apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-on-left-move2-b*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *14*) *Bk*
= (*L*, *start-of ly as* + *2*∗*n* + *14*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *14* = *Suc* (*2*∗*n* + *13*),
simp only: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-check-left-move2-o*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *15*) *Oc*
= (*L*, *start-of ly as* + *2*∗*n* + *13*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *15* = *Suc* (*2*∗*n* + *14*),
simp only: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-check-left-move2-b*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2* ∗ *n* + *15*) *Bk*
= (*R*, *start-of ly as* + *2*∗*n* + *15*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *15*= *Suc* (*2*∗*n* + *14*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-after-left-move2-b*:
⟦*ly* = *layout-of aprog*;
abc-fetch as aprog = *Some* (*Dec n e*);
start-of ly as > *0*⟧ ⟹
fetch (*ci* (*layout-of aprog*) (*start-of ly as*)
(*Dec n e*)) (*2* ∗ *n* + *16*) *Bk*
= (*R*, *start-of ly e*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac 2*∗*n* + *16* = *Suc* (*2*∗*n* + *15*),
simp only: *fetch.simps*)

apply(*auto simp*: *nth-of.simps nth-append*)
done

lemma *dec-fetch-next-state*:
start-of ly as > *0* ⟹
fetch (*ci* (*layout-of aprog*)
(*start-of ly as*) (*Dec n e*)) (*2*∗ *n* + *17*) *b*
= (*Nop*, *0*)
apply(*case-tac b*)
apply(*auto simp*: *ci.simps findnth.simps*
tshift.simps tdec-b-def add3-Suc)
apply(*subgoal-tac* [!] *2*∗*n* + *17* = *Suc* (*2*∗*n* + *16*),
simp-all only: *fetch.simps*)
apply(*auto simp*: *nth-of.simps nth-append*)
done

lemmas *dec-fetch-simps* =
dec-fetch-locate-a-o dec-fetch-locate-a-b dec-fetch-locate-b-o
dec-fetch-locate-b-b dec-fetch-locate-n-a-o
dec-fetch-locate-n-a-b dec-fetch-locate-n-b-o
dec-fetch-locate-n-b-b dec-fetch-first-on-right-move-o
dec-fetch-first-on-right-move-b dec-fetch-first-after-clear-b
dec-fetch-first-after-clear-o dec-fetch-right-move-b
dec-fetch-on-right-move-b dec-fetch-on-right-move-o
dec-fetch-after-clear-b dec-fetch-after-clear-o
dec-fetch-check-right-move-b dec-fetch-check-right-move-o
dec-fetch-left-move-b dec-fetch-on-left-move1-b
dec-fetch-on-left-move1-o dec-fetch-check-left-move1-b
dec-fetch-check-left-move1-o dec-fetch-after-left-move1-b
dec-fetch-on-left-move2-b dec-fetch-on-left-move2-o
dec-fetch-check-left-move2-o dec-fetch-check-left-move2-b
dec-fetch-after-left-move2-b dec-fetch-after-write-b
dec-fetch-after-write-o dec-fetch-next-state

lemma [*simp*]:
⟦*start-of ly as* ≤ *a*;
a < *start-of ly as* + *2* ∗ *n*;
(*a* − *start-of ly as*) *mod 2* = *Suc 0*;
inv-locate-b (*as*, *am*) ((*a* − *start-of ly as*) *div 2*, *aaa*, *Bk* # *xs*) *ires*⟧
⟹ ¬ *Suc a* < *start-of ly as* + *2* ∗ *n* ⟶
inv-locate-a (*as*, *am*) (*n*, *Bk* # *aaa*, *xs*) *ires*
apply(*insert locate-b-2-locate-a*[*of a ly as n am aaa xs*], *simp*)
done

lemma [*simp*]:
⟦*start-of ly as* ≤ *a*;
a < *start-of ly as* + *2* ∗ *n*;
(*a* − *start-of ly as*) *mod 2* = *Suc 0*;

inv-locate-b (*as*, *am*) ((*a* − *start-of ly as*) *div 2*, *aaa*, []) *ires*⟧
⟹ ¬ *Suc a* < *start-of ly as* + *2* ∗ *n* ⟶
inv-locate-a (*as*, *am*) (*n*, *Bk* # *aaa*, []) *ires*
apply(*insert locate-b-2-locate-a-B*[*of a ly as n am aaa*], *simp*)
done

lemma *exp-ind*: $a^{Suc\ b} = a^b$ @ [*a*]
apply(*simp only*: *exponent-def rep-ind*)
done

lemma [*simp*]:
inv-locate-b (*as*, *am*) (*n*, *l*, *Oc* # *r*) *ires*
⟹ *dec-first-on-right-moving n* (*as*, *abc-lm-s am n* (*abc-lm-v am n*))
(*Suc* (*Suc* (*start-of ly as* + *2* ∗ *n*)), *Oc* # *l*, *r*) *ires*
apply(*simp only*: *inv-locate-b.simps*
dec-first-on-right-moving.simps in-middle.simps
abc-lm-s.simps abc-lm-v.simps)
apply(*erule-tac exE*)+
apply(*erule conjE*)+
apply(*case-tac n* < *length am*, *simp*)
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 ml* **in** *exI*, *rule-tac conjI*, *rule-tac* [*1*−*2*] *impI*)
prefer *3*
apply(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = *lm2* **in** *exI*,
rule-tac x = *m* **in** *exI*, *simp*)
apply(*subgoal-tac Suc n* − *length am* = *Suc* (*n* − *length am*),
simp only:*exponent-def rep-ind*, *simp*)
apply(*rule-tac x* = *Suc ml* **in** *exI*, *simp-all*)
apply(*rule-tac* [!] *x* = *mr* − *1* **in** *exI*, *simp-all*)
apply(*case-tac* [!] *mr*, *auto*)
done

lemma [*simp*]:
⟦*inv-locate-b* (*as*, *am*) (*n*, *l*, *r*) *ires*; *l* ≠ []⟧ ⟹
¬ *inv-on-left-moving-in-middle-B* (*as*, *abc-lm-s am n* (*abc-lm-v am n*))
(*s*, *tl l*, *hd l* # *r*) *ires*
apply(*auto simp*: *inv-locate-b.simps*
inv-on-left-moving-in-middle-B.simps in-middle.simps)
apply(*case-tac* [!] *ml*, *auto split*: *if-splits*)
done

lemma [*simp*]: *inv-locate-b* (*as*, *am*) (*n*, *l*, *r*) *ires* ⟹ *l* ≠ []
apply(*auto simp*: *inv-locate-b.simps in-middle.simps split*: *if-splits*)
done

lemma [*simp*]: ⟦*inv-locate-b* (*as*, *am*) (*n*, *l*, *Bk* # *r*) *ires*; *n* < *length am*⟧

$\Longrightarrow$ *inv-on-left-moving-norm* (*as*, *am*) (*s*, *tl l*, *hd l* # *Bk* # *r*) *ires*
apply(*simp only*: *inv-locate-b.simps inv-on-left-moving-norm.simps in-middle.simps*)
apply(*erule-tac exE*)+
apply(*erule-tac conjE*)+
apply(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = *lm2* **in** *exI*, *rule-tac x* = *m* **in** *exI*, *simp*)
apply(*rule-tac x* = *ml* − *1* **in** *exI*, *auto*)
apply(*rule-tac* [!] *x* = *Suc mr* **in** *exI*)
apply(*case-tac* [!] *mr*, *auto*)
done

lemma [*simp*]: ⟦*inv-locate-b* (*as*, *am*) (*n*, *l*, *Bk* # *r*) *ires*; ¬ *n* < *length am*⟧
$\Longrightarrow$ *inv-on-left-moving-norm* (*as*, *am* @ *replicate* (*n* − *length am*) *0* @ [*0*]) (*s*, *tl l*, *hd l* # *Bk* # *r*) *ires*
apply(*simp only*: *inv-locate-b.simps inv-on-left-moving-norm.simps in-middle.simps*)
apply(*erule-tac exE*)+
apply(*erule-tac conjE*)+
apply(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = *lm2* **in** *exI*, *rule-tac x* = *m* **in** *exI*, *simp*)
apply(*subgoal-tac Suc n* − *length am* = *Suc* (*n* − *length am*), *simp only*: *rep-ind exponent-def*, *simp-all*)
apply(*rule-tac x* = *Suc mr* **in** *exI*, *auto*)
done

lemma *inv-locate-b-2-on-left-moving*[*simp*]:
⟦*inv-locate-b* (*as*, *am*) (*n*, *l*, *Bk* # *r*) *ires*⟧
$\Longrightarrow$ (*l* = [] ⟶ *inv-on-left-moving* (*as*, *abc-lm-s am n* (*abc-lm-v am n*)) (*s*, [], *Bk* # *Bk* # *r*) *ires*) ∧
(*l* ≠ [] ⟶ *inv-on-left-moving* (*as*, *abc-lm-s am n* (*abc-lm-v am n*)) (*s*, *tl l*, *hd l* # *Bk* # *r*) *ires*)
apply(*subgoal-tac l*≠[])
apply(*subgoal-tac* ¬ *inv-on-left-moving-in-middle-B* (*as*, *abc-lm-s am n* (*abc-lm-v am n*)) (*s*, *tl l*, *hd l* # *Bk* # *r*) *ires*)
apply(*simp add*:*inv-on-left-moving.simps abc-lm-s.simps abc-lm-v.simps split*: *if-splits*, *auto*)
done

lemma [*simp*]:
inv-locate-b (*as*, *am*) (*n*, *l*, []) *ires* $\Longrightarrow$
inv-locate-b (*as*, *am*) (*n*, *l*, [*Bk*]) *ires*
apply(*auto simp*: *inv-locate-b.simps in-middle.simps*)
apply(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = [] **in** *exI*, *rule-tac x* = *Suc* (*length lm1*) − *length am* **in** *exI*, *rule-tac x* = *m* **in** *exI*, *simp*)
apply(*rule-tac x* = *ml* **in** *exI*, *rule-tac x* = *mr* **in** *exI*)
apply(*auto*)
done

lemma *nil-2-nil*: <*lm*::*nat list*> = [] ⟹ *lm* = []
apply(*auto simp*: *tape-of-nl-abv*)
apply(*case-tac lm*, *simp*)
apply(*case-tac list*, *auto simp*: *tape-of-nat-list.simps*)
done

lemma *inv-locate-b-2-on-left-moving-b*[*simp*]:
inv-locate-b (*as*, *am*) (*n*, *l*, []) *ires*
⟹ (*l* = [] ⟶ *inv-on-left-moving* (*as*,
abc-lm-s am n (*abc-lm-v am n*)) (*s*, [], [*Bk*]) *ires*) ∧
(*l* ≠ [] ⟶ *inv-on-left-moving* (*as*, *abc-lm-s am n*
(*abc-lm-v am n*)) (*s*, *tl l*, [*hd l*]) *ires*)
apply(*insert inv-locate-b-2-on-left-moving*[*of as am n l* [] *ires s*])
apply(*simp only*: *inv-on-left-moving.simps*, *simp*)
apply(*subgoal-tac* ¬ *inv-on-left-moving-in-middle-B*
(*as*, *abc-lm-s am n* (*abc-lm-v am n*)) (*s*, *tl l*, [*hd l*]) *ires*, *simp*)
apply(*simp only*: *inv-on-left-moving-norm.simps*)
apply(*erule-tac exE*)+
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* = *ml* **in** *exI*,
rule-tac x = *mr* **in** *exI*, *simp*)
apply(*case-tac mr*, *simp*, *simp*, *case-tac nat*, *auto intro*: *nil-2-nil*)
done

lemma [*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(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = *lm2* **in** *exI*,
rule-tac x = *m* **in** *exI*, *simp*)
apply(*rule-tac x* = *Suc ml* **in** *exI*,
rule-tac x = *mr* − *1* **in** *exI*, *auto*)
apply(*case-tac* [!] *mr*, *auto*)
done

lemma [*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 [*elim*]:
⟦¬ *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 [*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(*case-tac n* < *length am*)
apply(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = *lm2* **in** *exI*,
rule-tac x = *m* − *1* **in** *exI*, *auto simp*:)
apply(*case-tac* [!] *mr*, *auto*)
done

lemma [*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(*case-tac n* < *length am*, *simp*)
apply(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = *m* − *1* **in** *exI*, *auto*)
apply(*case-tac* [*1*−*2*] *mr*, *auto*)
apply(*case-tac* [*1*−*2*] *m*, *auto simp*: *dec-first-on-right-moving.simps split*: *if-splits*)
done

lemma [*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 [*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 [*simp*]: ⟦*dec-after-clear* (*as*, *am*) (*s*, *l*, []) *ires*⟧
⟹ *dec-right-move* (*as*, *am*) (*s'*, *Bk* # *l*, []) *ires*
apply(*auto simp*: *dec-after-clear.simps dec-right-move.simps*)
done

lemma [*simp*]: $\exists$ *rn*. *a*::*block*^{rn} = []
apply(*rule-tac x* = *0* **in** *exI*, *simp*)
done

lemma [*simp*]: ⟦*dec-after-clear* (*as*, *am*) (*s*, *l*, []) *ires*⟧
$\Longrightarrow$ *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 [*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*⟧
$\Longrightarrow$ *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 [*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*)
apply(*rule-tac iffI*)
apply(*erule-tac* [!] *exE*)+
apply(*erule-tac* [*2*] *exE*)
apply(*rule-tac* [!] *x* = *lm1* **in** *exI*, *rule-tac x* = [] **in** *exI*,
rule-tac [!] *x* = *m* **in** *exI*, *auto*)
apply(*auto intro*: *nil-2-nil*)
done

lemma [*simp*]: ⟦*dec-right-move* (*as*, *am*) (*s*, *l*, []) *ires*⟧
$\Longrightarrow$ *dec-check-right-move* (*as*, *am*) (*s*, *Bk* # *l*, []) *ires*
apply(*insert dec-right-move-2-check-right-move*[*of as am s l* [] *s'*],
simp)
done

lemma [*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 [*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(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = *lm2* **in** *exI*,
rule-tac x = *m* **in** *exI*, *auto*)
done

lemma [*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(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = *m* **in** *exI*, *auto*)
apply(*auto intro*: *BkCons-nil nil-2-nil dest*: *BkCons-nil*)
apply(*rule-tac x* = *Suc rn* **in** *exI*)
apply(*auto intro*: *BkCons-nil nil-2-nil dest*: *BkCons-nil*)
done

lemma [*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(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = *m* **in** *exI*, *auto*)
apply(*auto intro*: *BkCons-nil nil-2-nil dest*: *BkCons-nil*)
done

lemma [*simp*]: *dec-left-move* (*as*, *am*) (*s*, *aaa*, *Oc* # *xs*) *ires* = *False*
apply(*auto simp*: *dec-left-move.simps inv-after-move.simps*)
apply(*case-tac* [!] *rn*, *auto*)
done

lemma [*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 *tape-of-nl-abv-cons-ex*[*simp*]:
$\exists$ *lna*. *Oc* # Oc^{m} @ *Bk* # <*rev lm1*> @ Bk^{ln} = <*m* # *rev lm1*> @ Bk^{lna}
apply(*case-tac lm1*=[], *auto simp*: *tape-of-nl-abv*
tape-of-nat-list.simps)
apply(*rule-tac x* = *ln* **in** *exI*, *simp*)
apply(*simp add*: *tape-of-nat-list-cons exponent-def*)
done

lemma [*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*, *simp*, *auto simp*: *tape-of-nat-def*)
done

lemma [*simp*]: *inv-on-left-moving-in-middle-B* (*as*, [*m*])
(*s′*, *Oc* # Oc^{m} @ *Bk* # *Bk* # *ires*, [*Bk*]) *ires*
apply(*simp add*: *inv-on-left-moving-in-middle-B.simps*)
apply(*rule-tac x* = [*m*] **in** *exI*, *simp*, *auto simp*: *tape-of-nat-def*)
done

lemma [*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*, *auto*)
done

lemma [*simp*]: *lm1* ≠ [] ⟹
inv-on-left-moving-in-middle-B (*as*, *lm1* @ [*m*]) (*s'*,
Oc # Oc^{m} @ *Bk* # <*rev lm1*> @ *Bk* # *Bk* # *ires*, [*Bk*]) *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*, *auto*)
done

lemma [*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 [*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 [*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(*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*, *simp*, *simp*)
apply(*case-tac list* = [],
auto simp: *tape-of-nl-abv tape-of-nat-list.simps split*: *if-splits*)
apply(*case-tac rn*, *auto*)
apply(*case-tac rev lm1*, *simp*, *simp add*: *tape-of-nat-list.simps*)
apply(*case-tac rn*, *auto*)
apply(*case-tac list*, *simp-all add*: *tape-of-nat-list.simps*, *auto*)
apply(*case-tac rev lm1*, *simp*, *simp add*: *tape-of-nat-list.simps*)
apply(*case-tac list*, *simp-all add*: *tape-of-nat-list.simps*, *auto*)
done

lemma [*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 [*simp*]: *dec-after-write* (*as*, *am*) (*s*, *aaa*, []) *ires*

$\Longrightarrow$ *dec-after-write* (*as*, *am*) (*s'*, *aaa*, [*Oc*]) *ires*
apply(*auto simp*: *dec-after-write.simps*)
done

lemma [*simp*]: *dec-on-right-moving* (*as*, *am*) (*s*, *l*, *Oc* # *r*) *ires*
$\Longrightarrow$ *dec-on-right-moving* (*as*, *am*) (*s'*, *Oc* # *l*, *r*) *ires*
apply(*simp only*: *dec-on-right-moving.simps*)
apply(*erule-tac exE*)+
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 [*simp*]: *dec-on-right-moving* (*as*, *am*) (*s*, *l*, *r*) *ires* $\Longrightarrow$ $l \neq []$
apply(*auto simp*: *dec-on-right-moving.simps split*: *if-splits*)
done

lemma [*simp*]: *dec-on-right-moving* (*as*, *am*) (*s*, *l*, *Bk* # *r*) *ires*
$\Longrightarrow$ *dec-after-clear* (*as*, *am*) (*s'*, *tl l*, *hd l* # *Bk* # *r*) *ires*
apply(*auto simp*: *dec-on-right-moving.simps dec-after-clear.simps*)
apply(*case-tac* [!] *mr*, *auto split*: *if-splits*)
done

lemma [*simp*]: *dec-on-right-moving* (*as*, *am*) (*s*, *l*, []) *ires*
$\Longrightarrow$ *dec-after-clear* (*as*, *am*) (*s'*, *tl l*, [*hd l*]) *ires*
apply(*auto simp*: *dec-on-right-moving.simps dec-after-clear.simps*)
apply(*case-tac mr*, *simp-all split*: *if-splits*)
apply(*rule-tac x* = *lm1* **in** *exI*, *simp*)
done

lemma *start-of-le*: $a < b \Longrightarrow$ *start-of ly a* $\leq$ *start-of ly b*
proof(*induct b arbitrary*: *a*, *simp*, *case-tac a* = *b*, *simp*)
fix *b a*
show *start-of ly b* $\leq$ *start-of ly* (*Suc b*)
apply(*case-tac b*::*nat*,
simp add: *start-of.simps*, *simp add*: *start-of.simps*)
done
next
fix *b a*
assume *h1*: $\bigwedge a.\ a < b \Longrightarrow$ *start-of ly a* $\leq$ *start-of ly b*
$a <$ *Suc b* $a \neq b$
hence $a < b$
by(*simp*)
from *h1* **and** *this* **have** *h2*: *start-of ly a* $\leq$ *start-of ly b*
by(*drule-tac h1*, *simp*)
from *h2* **show** *start-of ly a* $\leq$ *start-of ly* (*Suc b*)
proof −

```
    have start-of ly b ≤ start-of ly (Suc b)
      apply(case-tac b::nat,
            simp add: start-of.simps, simp add: start-of.simps)
      done
    from h2 and this show start-of ly a ≤ start-of ly (Suc b)
      by simp
  qed
qed

lemma start-of-dec-length[simp]:
  ⟦abc-fetch a aprog = Some (Dec n e)⟧ ⟹
    start-of (layout-of aprog) (Suc a)
         = start-of (layout-of aprog) a + 2*n + 16
apply(case-tac a, auto simp: abc-fetch.simps start-of.simps
                             layout-of.simps length-of.simps
                      split: if-splits)
done

lemma start-of-ge:
 ⟦abc-fetch a aprog = Some (Dec n e); a < e⟧ ⟹
  start-of (layout-of aprog) e >
             start-of (layout-of aprog) a + 2*n + 15
apply(case-tac e = Suc a,
      simp add: start-of.simps abc-fetch.simps layout-of.simps
                length-of.simps split: if-splits)
apply(subgoal-tac Suc a < e, drule-tac a = Suc a
             and ly = layout-of aprog in start-of-le)
apply(subgoal-tac start-of (layout-of aprog) (Suc a)
         = start-of (layout-of aprog) a + 2*n + 16, simp)
apply(rule-tac start-of-dec-length, simp)
apply(arith)
done

lemma starte-not-equal[simp]:
 ⟦abc-fetch as aprog = Some (Dec n e); ly = layout-of aprog⟧
   ⟹ (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)
```

apply(*case-tac e = as*, *simp*)
apply(*case-tac e < as*)
apply(*drule-tac a = e* **and** *b = as* **and** *ly = ly* **in** *start-of-le*, *simp*)
apply(*drule-tac a = as* **and** *e = e* **in** *start-of-ge*, *simp*, *simp*)
done

lemma [*simp*]: ⟦*abc-fetch as aprog = Some (Dec n e)*; *ly = layout-of aprog*⟧
⟹ (*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)
apply(*insert starte-not-equal*[*of as aprog n e ly*],
simp del: *starte-not-equal*)
apply(*erule-tac conjE*)+
apply(*rule-tac conjI*, *simp del*: *starte-not-equal*)+
apply(*rule not-sym*, *simp*)
done

lemma [*simp*]: *start-of* (*layout-of aprog*) *as* > *0* ⟹
fetch (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n as*)) (*Suc 0*) *Oc* =
(*R*, *Suc* (*start-of* (*layout-of aprog*) *as*))

apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append
Suc-pre tdec-b-def)
apply(*insert findnth-nth*[*of 0 n Suc 0*], *simp*)
apply(*simp add*: *findnth.simps*)
done

lemma *start-of-inj*[*simp*]:
⟦*abc-fetch as aprog = Some (Dec n e)*; *e ≠ as*; *ly = layout-of aprog*⟧
⟹ *start-of ly as ≠ start-of ly e*
apply(*case-tac e < as*)
apply(*case-tac as*, *simp*, *simp*)
apply(*case-tac e = nat*, *simp add*: *start-of.simps*
layout-of.simps length-of.simps)
apply(*subgoal-tac e < length aprog*, *simp add*: *length-of.simps*
split: *abc-inst.splits*)

apply(*simp add*: *abc-fetch.simps split*: *if-splits*)
apply(*subgoal-tac e* < *nat*, *drule-tac a* = *e* **and** *b* = *nat*
and *ly* =*ly* **in** *start-of-le*, *simp*)
apply(*subgoal-tac start-of ly nat* < *start-of ly* (*Suc nat*),
simp, *simp add*: *start-of.simps layout-of.simps*)
apply(*subgoal-tac nat* < *length aprog*, *simp*)
apply(*case-tac aprog* ! *nat*, *auto simp*: *length-of.simps*)
apply(*simp add*: *abc-fetch.simps split*: *if-splits*)
apply(*subgoal-tac e* > *as*, *drule-tac start-of-ge*, *auto*)
done

lemma [*simp*]: ⟦*abc-fetch as aprog* = *Some* (*Dec n e*); *e* < *as*⟧
⟹ *Suc* (*start-of* (*layout-of aprog*) *e*) −
start-of (*layout-of aprog*) *as* = *0*
apply(*frule-tac ly* = *layout-of aprog* **in** *start-of-le*, *simp*)
apply(*subgoal-tac start-of* (*layout-of aprog*) *as* ≠
start-of (*layout-of aprog*) *e*, *arith*)
apply(*rule start-of-inj*, *auto*)
done

lemma [*simp*]:
⟦*abc-fetch as aprog* = *Some* (*Dec n e*);
0 < *start-of* (*layout-of aprog*) *as*⟧
⟹ (*fetch* (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*)) (*Suc* (*start-of* (*layout-of aprog*) *e*) −
start-of (*layout-of aprog*) *as*) *Oc*)
= (*if e* = *as then* (*R*, *start-of* (*layout-of aprog*) *as* + *1*)
else (*Nop*, *0*))
apply(*auto split*: *if-splits*)
apply(*case-tac e* < *as*, *simp add*: *fetch.simps*)
apply(*subgoal-tac e* > *as*)
apply(*drule start-of-ge*, *simp*,
auto simp: *fetch.simps ci-length nth-of.simps*)
apply(*subgoal-tac*
length (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*)) *div 2*= *length-of* (*Dec n e*))
defer
apply(*simp add*: *ci-length*)
apply(*subgoal-tac*
length (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*)) *mod 2* = *0*, *auto simp*: *length-of.simps*)
done

lemma [*simp*]:
start-of (*layout-of aprog*) *as* > *0* ⟹
fetch (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n as*)) (*Suc 0*) *Bk*
= (*W1*, *start-of* (*layout-of aprog*) *as*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps nth-of.simps*

tshift.simps nth-append Suc-pre tdec-b-def)
apply(*insert findnth-nth*[*of 0 n 0*], *simp*)
apply(*simp add*: *findnth.simps*)
done

lemma [*simp*]:
⟦*abc-fetch as aprog* = *Some* (*Dec n e*);
0 < *start-of* (*layout-of aprog*) *as*⟧
⟹ (*fetch* (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*)) (*Suc* (*start-of* (*layout-of aprog*) *e*) −
start-of (*layout-of aprog*) *as*) *Bk*)
= (*if e* = *as then* (*W1*, *start-of* (*layout-of aprog*) *as*)
else (*Nop*, *0*))
apply(*auto split*: *if-splits*)
apply(*case-tac e* < *as*, *simp add*: *fetch.simps*)
apply(*subgoal-tac* *e* > *as*)
apply(*drule start-of-ge*, *simp*, *auto simp*: *fetch.simps*
ci-length nth-of.simps)
apply(*subgoal-tac*
length (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*)) *div 2*= *length-of* (*Dec n e*))
defer
apply(*simp add*: *ci-length*)
apply(*subgoal-tac*
length (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*)) *mod 2* = *0*, *auto simp*: *length-of.simps*)
apply(*simp add*: *ci.simps tshift.simps tdec-b-def*)
done

lemma [*simp*]:
inv-stop (*as*, *abc-lm-s am n* (*abc-lm-v am n*)) (*s*, *l*, *r*) *ires* ⟹ *l* ≠ []
apply(*auto simp*: *inv-stop.simps*)
done

lemma [*simp*]:
⟦*abc-fetch as aprog* = *Some* (*Dec n e*); *e* ≠ *as*; *ly* = *layout-of aprog*⟧
⟹ (¬ (*start-of ly as* ≤ *start-of ly e* ∧
start-of ly e < *start-of ly as* + *2* ∗ *n*))
∧ *start-of ly e* ≠ *start-of ly as* + *2*∗*n* ∧
start-of ly e ≠ *Suc* (*start-of ly as* + *2*∗*n*)
apply(*case-tac e* < *as*)
apply(*drule-tac ly* = *ly* **in** *start-of-le*, *simp*)
apply(*case-tac n*, *simp*, *drule start-of-inj*, *simp*, *simp*, *simp*, *simp*)
apply(*drule-tac start-of-ge*, *simp*, *simp*)
done

lemma [*simp*]:
⟦*abc-fetch as aprog* = *Some* (*Dec n e*); *start-of ly as* ≤ *s*;
s < *start-of ly as* + *2* ∗ *n*; *ly* = *layout-of aprog*⟧

⟹ *Suc s ≠ start-of ly e*
apply(*case-tac e = as*, *simp*)
apply(*case-tac e < as*)
apply(*drule-tac a = e* **and** *b = as* **and** *ly = ly* **in** *start-of-le*, *simp*)
apply(*drule-tac start-of-ge*, *auto*)
done

lemma [*simp*]: ⟦*abc-fetch as aprog = Some* (*Dec n e*);
ly = layout-of aprog⟧
⟹ *Suc* (*start-of ly as + 2 ∗ n*) ≠ *start-of ly e*
apply(*case-tac e = as*, *simp*)
apply(*case-tac e < as*)
apply(*drule-tac a = e* **and** *b = as* **and** *ly = ly* **in** *start-of-le*, *simp*)
apply(*drule-tac start-of-ge*, *auto*)
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 exponent-def*)
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: *rep-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(*case-tac mr = 0*, *simp-all add*: *exponent-def split*: *if-splits*)
apply(*subgoal-tac Suc* (*length lm1*) *− length am =*
Suc (*length lm1 − length am*),
simp add: *rep-ind del*: *replicate.simps*, *simp*)
done

lemma [*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(*insert inv-locate-b-2-on-left-moving*[*of as am n aaa xs ires s*], *simp*)
done

lemma [*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(*insert inv-locate-b-2-on-left-moving-b*[*of as am n aaa ires s*], *simp*)
done

lemma [*simp*]: ⟦*am* ! *n* = (*0*::*nat*); *n* < *length am*⟧ ⟹ *am*[*n* := *0*] = *am*
apply(*simp add*: *list-update-same-conv*)
done

lemma [*simp*]: ⟦*abc-lm-v am n* = *0*;
inv-locate-b (*as*, *abc-lm-s am n 0*) (*n*, *Oc* # *aaa*, *xs*) *ires*⟧
⟹ *inv-locate-b* (*as*, *am*) (*n*, *Oc* # *aaa*, *xs*) *ires*
apply(*simp only*: *inv-locate-b.simps in-middle.simps abc-lm-s.simps*
abc-lm-v.simps)
apply(*erule-tac exE*)+
apply(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = *lm2* **in** *exI*, *simp*)
apply(*case-tac n* < *length am*, *simp-all*)
apply(*erule-tac conjE*)+
apply(*rule-tac x* = *tn* **in** *exI*, *rule-tac x* = *m* **in** *exI*, *simp*)
apply(*rule-tac x* = *ml* **in** *exI*, *rule-tac x* = *mr* **in** *exI*, *simp*)
defer
apply(*rule-tac x* = *Suc n* − *length am* **in** *exI*, *rule-tac x* = *m* **in** *exI*)
apply(*subgoal-tac Suc n* − *length am* = *Suc* (*n* − *length am*))
apply(*simp add*: *exponent-def rep-ind del*: *replicate.simps*, *auto*)
done

lemma [*intro*]: ⟦*abc-lm-v* (*a* # *list*) *0* = *0*⟧ ⟹ *a* = *0*
apply(*simp add*: *abc-lm-v.simps split*: *if-splits*)
done

lemma [*simp*]:
inv-stop (*as*, *abc-lm-s am n 0*)
(*start-of* (*layout-of aprog*) *e*, *aaa*, *Oc* # *xs*) *ires*
⟹ *inv-locate-a* (*as*, *abc-lm-s am n 0*) (*0*, *aaa*, *Oc* # *xs*) *ires*
apply(*simp add*: *inv-locate-a.simps*)
apply(*rule disjI1*)
apply(*auto simp*: *inv-stop.simps at-begin-norm.simps*)
done

lemma [*simp*]:
⟦*abc-lm-v am 0* = *0*;
inv-stop (*as*, *abc-lm-s am 0 0*)
(*start-of* (*layout-of aprog*) *e*, *aaa*, *Oc* # *xs*) *ires*⟧ ⟹
inv-locate-b (*as*, *am*) (*0*, *Oc* # *aaa*, *xs*) *ires*
apply(*auto simp*: *inv-stop.simps inv-locate-b.simps*
in-middle.simps abc-lm-s.simps)
apply(*case-tac am* = [], *simp*)
apply(*rule-tac x* = [] **in** *exI*, *rule-tac x* = *Suc 0* **in** *exI*,
rule-tac x = *0* **in** *exI*, *simp*)
apply(*rule-tac x* = *Suc 0* **in** *exI*, *rule-tac x* = *0* **in** *exI*,
simp add: *tape-of-nl-abv tape-of-nat-list.simps*, *auto*)
apply(*case-tac rn*, *auto*)
apply(*rule-tac x* = *tl am* **in** *exI*, *rule-tac x* = *0* **in** *exI*,
rule-tac x = *hd am* **in** *exI*, *simp*)

apply(*rule-tac x = Suc 0* **in** *exI*, *rule-tac x = hd am* **in** *exI*, *simp*)
apply(*case-tac am*, *simp*, *simp*)
apply(*subgoal-tac a = 0*, *case-tac list*,
auto simp: *tape-of-nat-list.simps tape-of-nl-abv*)
apply(*case-tac rn*, *auto*)
done

lemma [*simp*]:
⟦*inv-stop* (*as*, *abc-lm-s am n 0*)
(*start-of* (*layout-of aprog*) *e*, *aaa*, *Oc # xs*) *ires*⟧
⟹ *inv-locate-b* (*as*, *am*) (*0*, *Oc # aaa*, *xs*) *ires* ∨
inv-locate-b (*as*, *abc-lm-s am n 0*) (*0*, *Oc # aaa*, *xs*) *ires*
apply(*simp*)
done

lemma [*simp*]:
⟦*abc-lm-v am n = 0*;
inv-stop (*as*, *abc-lm-s am n 0*)
(*start-of* (*layout-of aprog*) *e*, *aaa*, *Oc # xs*) *ires*⟧
⟹ ¬ *Suc 0* < *2* ∗ *n* ⟶ *e* = *as* ⟶
inv-locate-b (*as*, *am*) (*n*, *Oc # aaa*, *xs*) *ires*
apply(*case-tac n*, *simp*, *simp*)
done

lemma *dec-false2*:
inv-stop (*as*, *abc-lm-s am n 0*)
(*start-of* (*layout-of aprog*) *e*, *aaa*, *Bk # xs*) *ires* = *False*
apply(*auto simp*: *inv-stop.simps abc-lm-s.simps*)
apply(*case-tac am*, *simp*, *case-tac n*, *simp add*: *tape-of-nl-abv*)
apply(*case-tac list*, *simp add*: *tape-of-nat-list.simps*)
apply(*simp add*: *tape-of-nat-list.simps* , *simp*)
apply(*case-tac list*[*nat* := *0*],
simp add: *tape-of-nat-list.simps tape-of-nl-abv*)
apply(*simp add*: *tape-of-nat-list.simps*)
apply(*case-tac am* @ *replicate* (*n* − *length am*) *0* @ [*0*], *simp*)
apply(*case-tac list*, *auto simp*: *tape-of-nl-abv*
tape-of-nat-list.simps)
done

lemma *dec-false3*:
inv-stop (*as*, *abc-lm-s am n 0*)
(*start-of* (*layout-of aprog*) *e*, *aaa*, []) *ires* = *False*
apply(*auto simp*: *inv-stop.simps abc-lm-s.simps*)
apply(*case-tac am*, *case-tac n*, *auto*)
apply(*case-tac n*, *auto simp*: *tape-of-nl-abv*)
apply(*case-tac list*::*nat list*,
simp add: *tape-of-nat-list.simps tape-of-nat-list.simps*)
apply(*simp add*: *tape-of-nat-list.simps*)
apply(*case-tac list*[*nat* := *0*],

simp add: *tape-of-nat-list.simps tape-of-nat-list.simps*)
apply(*simp add*: *tape-of-nat-list.simps*)
apply(*case-tac* (*am* @ *replicate* (*n* − *length am*) *0* @ [*0*]), *simp*)
apply(*case-tac list*, *auto simp*: *tape-of-nat-list.simps*)
done

lemma [*simp*]:
fetch (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Dec n e*)) *0 b* = (*Nop*, *0*)
by(*simp add*: *fetch.simps*)

declare *dec-inv-1.simps*[*simp del*]

declare *inv-locate-n-b.simps* [*simp del*]

lemma [*simp*]:
⟦*0* < *abc-lm-v am n*; *0* < *n*;
at-begin-norm (*as*, *am*) (*n*, *aaa*, *Oc* # *xs*) *ires*⟧
⟹ *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(*rule-tac x* = *lm1* **in** *exI*, *simp*)
apply(*case-tac length lm2*, *simp*)
apply(*case-tac rn*, *simp*, *simp*)
apply(*rule-tac x* = *tl lm2* **in** *exI*, *rule-tac x* = *hd lm2* **in** *exI*, *simp*)
apply(*rule conjI*)
apply(*case-tac lm2*, *simp*, *simp*)
apply(*case-tac lm2*, *auto simp*: *tape-of-nl-abv tape-of-nat-list.simps*)
apply(*case-tac* [!] *list*, *auto simp*: *tape-of-nl-abv tape-of-nat-list.simps*)
apply(*case-tac rn*, *auto*)
done
lemma [*simp*]: (∃ *rn*. *Oc* # *xs* = Bk^{rn}) = *False*
apply(*auto*)
apply(*case-tac rn*, *auto simp*:)
done

lemma [*simp*]:
⟦*0* < *abc-lm-v am n*; *0* < *n*;
at-begin-fst-bwtn (*as*, *am*) (*n*, *aaa*, *Oc* # *xs*) *ires*⟧
⟹ *inv-locate-n-b* (*as*, *am*) (*n*, *Oc* # *aaa*, *xs*) *ires*
apply(*simp add*: *at-begin-fst-bwtn.simps inv-locate-n-b.simps*)
done

lemma *Suc-minus*:*length am* + *tn* = *n*
⟹ *Suc tn* = *Suc n* − *length am*
apply(*arith*)
done

lemma [*simp*]:

⟦*0* < *abc-lm-v am n*; *0* < *n*;
at-begin-fst-awtn (*as*, *am*) (*n*, *aaa*, *Oc* # *xs*) *ires*⟧
⟹ *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(*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*: *exponent-def rep-ind del*: *replicate.simps*)
apply(*rule conjI*)+
apply(*auto*)
apply(*case-tac* [!] *rn*, *auto*)
done

lemma [*simp*]:
⟦*0* < *abc-lm-v am n*; *0* < *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*)
done

lemma [*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(*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(*case-tac m*, *auto simp*: *exponent-def*)
apply(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = *lm2* **in** *exI*,
rule-tac x = *m* **in** *exI*,
simp add: *Suc-diff-le rep-ind del*: *replicate.simps*)
apply(*rule-tac x* = *Suc* (*Suc 0*) **in** *exI*,
rule-tac x = *m* − *1* **in** *exI*, *simp*)
apply(*case-tac m*, *auto simp*: *exponent-def*)
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(*rule-tac x* = *lm1* **in** *exI*, *rule-tac x* = *lm2* **in** *exI*,
rule-tac x = *m* **in** *exI*,
simp add: *Suc-diff-le rep-ind del*: *replicate.simps*, *simp*)
done

lemma *dec-false-2*:
⟦*0* < *abc-lm-v am n*; *inv-locate-n-b* (*as*, *am*) (*n*, *aaa*, *Bk* # *xs*) *ires*⟧

```
 ⟹ False
apply(auto simp: inv-locate-n-b.simps abc-lm-v.simps split: if-splits)
apply(case-tac [!] m, auto)
done

lemma dec-false-2-b:
 ⟦0 < abc-lm-v am n; inv-locate-n-b (as, am)
                                (n, aaa, []) ires⟧ ⟹ False
apply(auto simp: inv-locate-n-b.simps abc-lm-v.simps split: if-splits)
apply(case-tac [!] m, auto simp: )
done



thm abc-inc-stage1.simps
fun abc-dec-1-stage1:: t-conf ⇒ 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:: t-conf ⇒ 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 :: t-conf ⇒ nat ⇒ nat ⇒ block list ⇒ nat
  where
  abc-dec-1-stage3 (s, l, r) ss n ires =
        (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 l = Bk # ires ∧ 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 :: (t-conf × nat × nat × block list) ⇒ (nat × nat × nat)
  where
  abc-dec-1-measure (c, ss, n, ires) = (abc-dec-1-stage1 c ss n,
                 abc-dec-1-stage2 c ss n, abc-dec-1-stage3 c ss n ires)
```

definition *abc-dec-1-LE* ::
(((*nat* × *block list* × *block list*) × *nat* ×
nat × *block list*) × ((*nat* × *block list* × *block list*) × *nat* × *nat* × *block list*)) *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 wf-lex-triple simp*:*abc-dec-1-LE-def*)

declare *dec-inv-1.simps*[*simp del*] *dec-inv-2.simps*[*simp del*]

lemma [*elim*]:
⟦*abc-fetch as aprog* = *Some* (*Dec n e*);
start-of (*layout-of aprog*) *as* < *start-of* (*layout-of aprog*) *e*;
start-of (*layout-of aprog*) *e* ≤
Suc (*start-of* (*layout-of aprog*) *as* + *2* ∗ *n*)⟧ ⟹ *False*
apply(*case-tac e* = *as*, *simp*)
apply(*case-tac e* < *as*)
apply(*drule-tac a* = *e* **and** *b* = *as* **and** *ly* = *layout-of aprog* **in**
start-of-le, *simp*)
apply(*drule-tac start-of-ge*, *auto*)
done

lemma [*elim*]: ⟦*abc-fetch as aprog* = *Some* (*Dec n e*);
start-of (*layout-of aprog*) *e*
= *start-of* (*layout-of aprog*) *as* + *2* ∗ *n* + *13*⟧
⟹ *False*
apply(*insert starte-not-equal*[*of as aprog n e layout-of aprog*],
simp)
done

lemma [*elim*]: ⟦*abc-fetch as aprog* = *Some* (*Dec n e*);
start-of (*layout-of aprog*) *e* =
start-of (*layout-of aprog*) *as* + *2* ∗ *n* + *14*⟧
⟹ *False*
apply(*insert starte-not-equal*[*of as aprog n e layout-of aprog*],
simp)
done

lemma [*elim*]:
⟦*abc-fetch as aprog* = *Some* (*Dec n e*);
start-of (*layout-of aprog*) *as* < *start-of* (*layout-of aprog*) *e*;
start-of (*layout-of aprog*) *e* ≤
Suc (*start-of* (*layout-of aprog*) *as* + *2* ∗ *n*)⟧
⟹ *False*
apply(*case-tac e* = *as*, *simp*)
apply(*case-tac e* < *as*)
apply(*drule-tac a* = *e* **and** *b* = *as* **and** *ly* = *layout-of aprog* **in**
start-of-le, *simp*)

apply(*drule-tac start-of-ge*, *auto*)
done

lemma [*elim*]:
 $[\![$*abc-fetch as aprog* = *Some* (*Dec n e*);
  start-of (*layout-of aprog*) *e* =
   start-of (*layout-of aprog*) *as* + *2* ∗ *n* + *13*$]\!]$
  ⟹ *False*
apply(*insert starte-not-equal*[*of as aprog n e layout-of aprog*],
  simp)
done

lemma [*simp*]:
 abc-fetch as aprog = *Some* (*Dec n e*) ⟹
  Suc (*start-of* (*layout-of aprog*) *as*) ≠ *start-of* (*layout-of aprog*) *e*
apply(*case-tac e* = *as*, *simp*)
apply(*case-tac e* < *as*)
apply(*drule-tac a* = *e* **and** *b* = *as* **and** *ly* = (*layout-of aprog*) **in**
  start-of-le, *simp*)
apply(*drule-tac a* = *as* **and** *e* = *e* **in** *start-of-ge*, *simp*, *simp*)
done

lemma [*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 [*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 *dec-inv-stop1-pre*:
 $[\![$*abc-fetch as aprog* = *Some* (*Dec n e*); *abc-lm-v am n* = *0*;
  start-of (*layout-of aprog*) *as* > *0*$]\!]$
 ⟹ ∀ *na*. ¬ (λ(*s*, *l*, *r*) (*ss*, *n′*, *ires′*). *s* = *start-of* (*layout-of aprog*) *e*)
  (*t-steps* (*Suc* (*start-of* (*layout-of aprog*) *as*), *l*, *r*)
   (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
    (*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *na*)
     (*start-of* (*layout-of aprog*) *as*, *n*, *ires*) ∧
  dec-inv-1 (*layout-of aprog*) *n e* (*as*, *am*)
  (*t-steps* (*Suc* (*start-of* (*layout-of aprog*) *as*), *l*, *r*)
   (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
    (*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *na*) *ires*
  ⟶ *dec-inv-1* (*layout-of aprog*) *n e* (*as*, *am*)
  (*t-steps* (*Suc* (*start-of* (*layout-of aprog*) *as*), *l*, *r*)

(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*)
(*Suc na*)) *ires* ∧
((*t-steps* (*Suc* (*start-of* (*layout-of aprog*) *as*), *l*, *r*)
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) (*Suc na*),
start-of (*layout-of aprog*) *as*, *n*, *ires*),
t-steps (*Suc* (*start-of* (*layout-of aprog*) *as*), *l*, *r*)
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *na*,
start-of (*layout-of aprog*) *as*, *n*, *ires*)
∈ *abc-dec-1-LE*
apply(*rule allI*, *rule impI*, *simp add*: *t-steps-ind*)
apply(*case-tac* (*t-steps* (*Suc* (*start-of* (*layout-of aprog*) *as*), *l*, *r*)
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*) (*Dec n e*),
start-of (*layout-of aprog*) *as* − *Suc 0*) *na*), *simp*)
apply(*auto split*:*if-splits simp add*:*t-step.simps dec-inv-1.simps*,
tactic ⟪ *ALLGOALS* (*resolve-tac* [@{*thm fetch-intro*}]) ⟫)
apply(*simp-all add*:*dec-fetch-simps new-tape.simps dec-inv-1.simps*)
apply(*auto simp add*: *abc-dec-1-LE-def lex-square-def*
lex-triple-def lex-pair-def
split: *if-splits*)
apply(*rule dec-false-1*, *simp*, *simp*)
done

lemma *dec-inv-stop1*:
⟦*ly* = *layout-of aprog*;
dec-inv-1 ly n e (*as*, *am*) (*start-of ly as* + *1*, *l*, *r*) *ires*;
abc-fetch as aprog = *Some* (*Dec n e*); *abc-lm-v am n* = *0*⟧ ⟹
(∃ *stp*. (λ (*s′*, *l′*, *r′*). *s′* = *start-of ly e* ∧
dec-inv-1 ly n e (*as*, *am*) (*s′*, *l′* , *r′*) *ires*)
(*t-steps* (*start-of ly as* + *1*, *l*, *r*)
(*ci ly* (*start-of ly as*) (*Dec n e*), *start-of ly as* − *Suc 0*) *stp*))
apply(*insert halt-lemma2*[*of abc-dec-1-LE*
λ ((*s*, *l*, *r*), *ss*, *n′*, *ires′*). *s* = *start-of ly e*
(λ *stp*. (*t-steps* (*start-of ly as* + *1*, *l*, *r*)
(*ci ly* (*start-of ly as*) (*Dec n e*), *start-of ly as* − *Suc 0*)
stp, *start-of ly as*, *n*, *ires*))
λ ((*s*, *l*, *r*), *ss*, *n*, *ires′*). *dec-inv-1 ly n e* (*as*, *am*) (*s*, *l*, *r*) *ires′*],
simp)
apply(*insert wf-dec-le*, *simp*)
apply(*insert dec-inv-stop1-pre*[*of as aprog n e am l r*], *simp*)
apply(*subgoal-tac start-of* (*layout-of aprog*) *as* > *0*,
simp add: *t-steps.simps*)
apply(*erule-tac exE*, *rule-tac x* = *na* **in** *exI*)
apply(*case-tac*
(*t-steps* (*Suc* (*start-of* (*layout-of aprog*) *as*), *l*, *r*)
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *na*),

```
    case-tac b, auto)
apply(rule startof-not0)
done


lemma [simp]:
  ⟦abc-fetch as aprog = Some (Dec n e);
   ly = layout-of aprog⟧ ⟹
          start-of ly (Suc as) = start-of ly as + 2*n + 16
by simp

fun abc-dec-2-stage1 :: t-conf ⇒ 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)

thm new-tape.simps

fun abc-dec-2-stage2 :: t-conf ⇒ 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 :: t-conf ⇒ nat ⇒ nat ⇒ block list ⇒ nat
  where
  abc-dec-2-stage3 (s, l, r) ss n ires =
       (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 l = Bk # ires ∧ 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* :: *t-conf* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where

```
  abc-dec-2-stage4 (s, l, r) ss n =
       (if s = ss + 2*n + 2 then length r
        else if s = ss + 2*n + 8 then length r
        else if s = ss + 2*n + 3 then
           if r ≠ [] ∧ hd r = Oc then 1
           else 0
        else if s = ss + 2*n + 7 then
           if r ≠ [] ∧ hd r = Oc then 0
           else 1
        else if s = ss + 2*n + 9 then
           if r ≠ [] ∧ hd r = Oc then 1
           else 0
        else 0)
```

fun *abc-dec-2-measure* :: (*t-conf* × *nat* × *nat* × *block list*) ⇒
(*nat* × *nat* × *nat* × *nat*)
where

```
  abc-dec-2-measure (c, ss, n, ires) =
     (abc-dec-2-stage1 c ss n, abc-dec-2-stage2 c ss n,
      abc-dec-2-stage3 c ss n ires, abc-dec-2-stage4 c ss n)
```

definition *abc-dec-2-LE* ::
(((*nat* × *block list* × *block list*) × *nat* × *nat* × *block list*) ×
((*nat* × *block list* × *block list*) × *nat* × *nat* × *block list*)) *set*
where *abc-dec-2-LE* ≡ (*inv-image lex-square abc-dec-2-measure*)

lemma *wf-dec-2-le*: *wf abc-dec-2-LE*
by(*auto intro*:*wf-inv-image wf-lex-triple wf-lex-square*
simp:*abc-dec-2-LE-def*)

lemma [*simp*]: *dec-after-write* (*as*, *am*) (*s*, *aa*, *r*) *ires*
⟹ *takeWhile* (λ*a*. *a* = *Oc*) *aa* = []
apply(*simp only* : *dec-after-write.simps*)
apply(*erule exE*)+
apply(*erule-tac conjE*)+
apply(*case-tac aa*, *simp*)
apply(*case-tac a*, *simp only*: *takeWhile.simps* , *simp*, *simp split*: *if-splits*)
done

lemma [*simp*]:

⟦*dec-on-right-moving* (*as*, *lm*) (*s*, *aa*, []) *ires*;
length (*takeWhile* (λ*a*. *a* = *Oc*) (*tl aa*))
≠ *length* (*takeWhile* (λ*a*. *a* = *Oc*) *aa*) − *Suc 0*⟧
⟹ *length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl aa*)) <
length (*takeWhile* (λ*a*. *a* = *Oc*) *aa*) − *Suc 0*
apply(*simp only*: *dec-on-right-moving.simps*)
apply(*erule-tac exE*)+
apply(*erule-tac conjE*)+
apply(*case-tac mr*, *auto split*: *if-splits*)
done

lemma [*simp*]:
dec-after-clear (*as*, *abc-lm-s am n* (*abc-lm-v am n* − *Suc 0*))
(*start-of* (*layout-of aprog*) *as* + *2* ∗ *n* + *9*, *aa*, *Bk* # *xs*) *ires*
⟹ *length xs* − *Suc 0* < *length xs* +
length (*takeWhile* (λ*a*. *a* = *Oc*) *aa*)
apply(*simp only*: *dec-after-clear.simps*)
apply(*erule-tac exE*)+
apply(*erule conjE*)+
apply(*simp split*: *if-splits*)
done

lemma [*simp*]:
⟦*dec-after-clear* (*as*, *abc-lm-s am n* (*abc-lm-v am n* − *Suc 0*))
(*start-of* (*layout-of aprog*) *as* + *2* ∗ *n* + *9*, *aa*, []) *ires*⟧
⟹ *Suc 0* < *length* (*takeWhile* (λ*a*. *a* = *Oc*) *aa*)
apply(*simp add*: *dec-after-clear.simps split*: *if-splits*)
done

lemma [*simp*]:
⟦*dec-on-right-moving* (*as*, *am*) (*s*, *aa*, *Bk* # *xs*) *ires*;
Suc (*length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl aa*)))
≠ *length* (*takeWhile* (λ*a*. *a* = *Oc*) *aa*)⟧
⟹ *Suc* (*length* (*takeWhile* (λ*a*. *a* = *Oc*) (*tl aa*)))
< *length* (*takeWhile* (λ*a*. *a* = *Oc*) *aa*)
apply(*simp only*: *dec-on-right-moving.simps*)
apply(*erule exE*)+
apply(*erule conjE*)+
apply(*case-tac ml*, *auto split*: *if-splits*)
done

lemma [*simp*]: *inv-check-left-moving* (*as*, *abc-lm-s am n* (*abc-lm-v am n* − *Suc 0*))
(*start-of* (*layout-of aprog*) *as* + *2* ∗ *n* + *11*, [], *Oc* # *xs*) *ires* = *False*
apply(*simp add*: *inv-check-left-moving.simps inv-check-left-moving-in-middle.simps*)
done

```
lemma dec-inv-stop2-pre:
  ⟦abc-fetch as aprog = Some (Dec n e); abc-lm-v am n > 0⟧ ⟹
    ∀ na. ¬ (λ(s, l, r) (ss, n', ires').
                    s = start-of (layout-of aprog) as + 2 ∗ n + 16)
   (t-steps (Suc (start-of (layout-of aprog) as), l, r)
      (ci (layout-of aprog) (start-of (layout-of aprog) as) (Dec n e),
           start-of (layout-of aprog) as − Suc 0) na)
    (start-of (layout-of aprog) as, n, ires) ∧
 dec-inv-2 (layout-of aprog) n e (as, am)
     (t-steps (Suc (start-of (layout-of aprog) as), l, r)
      (ci (layout-of aprog) (start-of (layout-of aprog) as) (Dec n e),
         start-of (layout-of aprog) as − Suc 0) na) ires
 ⟶
 dec-inv-2 (layout-of aprog) n e (as, am)
     (t-steps (Suc (start-of (layout-of aprog) as), l, r)
      (ci (layout-of aprog) (start-of (layout-of aprog) as) (Dec n e),
            start-of (layout-of aprog) as − Suc 0) (Suc na)) ires ∧
 ((t-steps (Suc (start-of (layout-of aprog) as), l, r)
      (ci (layout-of aprog) (start-of (layout-of aprog) as) (Dec n e),
           start-of (layout-of aprog) as − Suc 0) (Suc na),
            start-of (layout-of aprog) as, n, ires),
  t-steps (Suc (start-of (layout-of aprog) as), l, r)
      (ci (layout-of aprog) (start-of (layout-of aprog) as) (Dec n e),
           start-of (layout-of aprog) as − Suc 0) na,
                  start-of (layout-of aprog) as, n, ires)
   ∈ abc-dec-2-LE
apply(subgoal-tac start-of (layout-of aprog) as > 0)
apply(rule allI, rule impI, simp add: t-steps-ind)
apply(case-tac (t-steps (Suc (start-of (layout-of aprog) as), l, r)
     (ci (layout-of aprog) (start-of (layout-of aprog) as) (Dec n e),
           start-of (layout-of aprog) as − Suc 0) na), simp)
apply(auto split:if-splits simp add:t-step.simps dec-inv-2.simps,
          tactic ⟪ ALLGOALS (resolve-tac [@{thm fetch-intro}]) ⟫)
apply(simp-all add:dec-fetch-simps new-tape.simps dec-inv-2.simps)
apply(auto simp add: abc-dec-2-LE-def lex-square-def lex-triple-def
                    lex-pair-def split: if-splits)
apply(auto intro: dec-false-2-b dec-false-2)
apply(rule startof-not0)
done

lemma dec-stop2:
 ⟦ly = layout-of aprog;
   dec-inv-2 ly n e (as, am) (start-of ly as + 1, l, r) ires;
   abc-fetch as aprog = Some (Dec n e);
   abc-lm-v am n > 0⟧ ⟹
  (∃ stp. (λ (s', l', r'). s' = start-of ly (Suc as) ∧
   dec-inv-2 ly n e (as, am) (s', l', r') ires)
      (t-steps (start-of ly as+1, l, r) (ci ly (start-of ly as)
                        (Dec n e), start-of ly as − Suc 0) stp))
```

```
apply(insert halt-lemma2[of abc-dec-2-LE
      λ ((s, l, r), ss, n', ires'). s = start-of ly (Suc as)
      (λ stp. (t-steps (start-of ly as + 1, l, r)
       (ci ly (start-of ly as) (Dec n e), start-of ly as − Suc 0) stp,
             start-of ly as, n, ires))
      (λ ((s, l, r), ss, n, ires'). dec-inv-2 ly n e (as, am) (s, l, r) ires')])
apply(insert wf-dec-2-le, simp)
apply(insert dec-inv-stop2-pre[of as aprog n e am l r],
      simp add: t-steps.simps)
apply(erule-tac exE)
apply(rule-tac x = na in exI)
apply(case-tac (t-steps (Suc (start-of (layout-of aprog) as), l, r)
(ci (layout-of aprog) (start-of (layout-of aprog) as) (Dec n e),
          start-of (layout-of aprog) as − Suc 0) na),
      case-tac b, auto)
done

lemma dec-inv-stop-cond1:
  ⟦ly = layout-of aprog;
    dec-inv-1 ly n e (as, lm) (s, (l, r)) ires; s = start-of ly e;
    abc-fetch as aprog = Some (Dec n e); abc-lm-v lm n = 0⟧
   ⟹ crsp-l ly (e, abc-lm-s lm n 0) (s, l, r) ires
apply(simp add: dec-inv-1.simps split: if-splits)
apply(auto simp: crsp-l.simps inv-stop.simps )
done

lemma dec-inv-stop-cond2:
   ⟦ly = layout-of aprog; s = start-of ly (Suc as);
     dec-inv-2 ly n e (as, lm) (s, (l, r)) ires;
     abc-fetch as aprog = Some (Dec n e);
     abc-lm-v lm n > 0⟧
   ⟹ crsp-l ly (Suc as,
               abc-lm-s lm n (abc-lm-v lm n − Suc 0)) (s, l, r) ires
apply(simp add: dec-inv-2.simps split: if-splits)
apply(auto simp: crsp-l.simps inv-stop.simps )
done

lemma [simp]: (case Bk^rn of [] ⇒ Bk |
             Bk # xs ⇒ Bk | Oc # xs ⇒ Oc) = Bk
apply(case-tac rn, auto)
done

lemma [simp]: t-steps tc (p,off) (m + n) =
               t-steps (t-steps tc (p, off) m) (p, off) n
apply(induct m arbitrary: n)
apply(simp add: t-steps.simps)
proof −
  fix m n
  assume h1: ⋀n. t-steps tc (p, off) (m + n) =
```

t-steps (*t-steps tc* (*p*, *off*) *m*) (*p*, *off*) *n*
hence *h2*: *t-steps tc* (*p*, *off*) (*Suc m* + *n*) =
t-steps tc (*p*, *off*) (*m* + *Suc n*)
by *simp*
from *h1* **and** *this* **show**
t-steps tc (*p*, *off*) (*Suc m* + *n*) =
t-steps (*t-steps tc* (*p*, *off*) (*Suc m*)) (*p*, *off*) *n*
proof(*simp only*: *h2*, *simp add*: *t-steps.simps*)
have *h3*: (*t-step* (*t-steps tc* (*p*, *off*) *m*) (*p*, *off*)) =
(*t-steps* (*t-step tc* (*p*, *off*)) (*p*, *off*) *m*)
apply(*simp add*: *t-steps.simps*[*THEN sym*] *t-steps-ind*[*THEN sym*])
done
from *h3* **show**
t-steps (*t-step* (*t-steps tc* (*p*, *off*) *m*) (*p*, *off*)) (*p*, *off*) *n* = *t-steps* (*t-steps* (*t-step tc* (*p*, *off*)) (*p*, *off*) *m*) (*p*, *off*) *n*
by *simp*
qed
qed

lemma [*simp*]: *abc-fetch as aprog* = *Some* (*Dec n e*) $\Longrightarrow$
Suc (*start-of* (*layout-of aprog*) *as*) $\neq$
start-of (*layout-of aprog*) *e*
apply(*case-tac e* = *as*, *simp*)
apply(*case-tac e* < *as*)
apply(*drule-tac a* = *e* **and** *b* = *as* **and** *ly* = *layout-of aprog*
in *start-of-le*, *simp*)
apply(*drule-tac start-of-ge*, *auto*)
done

lemma [*simp*]: *inv-locate-b* (*as*, []) (*0*, *Oc* # *Bk* # *Bk* # *ires*, $Bk^{rn\ -\ Suc\ 0}$) *ires*
apply(*auto simp*: *inv-locate-b.simps in-middle.simps*)
apply(*rule-tac x* = [] **in** *exI*, *rule-tac x* = *Suc 0* **in** *exI*,
rule-tac x = *0* **in** *exI*, *simp*)
apply(*rule-tac x* = *Suc 0* **in** *exI*, *rule-tac x* = *0* **in** *exI*, *auto*)
apply(*case-tac rn*, *simp*, *case-tac nat*, *auto*)
done

lemma [*simp*]:
inv-locate-n-b (*as*, []) (*0*, *Oc* # *Bk* # *Bk* # *ires*, $Bk^{rn\ -\ Suc\ 0}$) *ires*
apply(*auto simp*: *inv-locate-n-b.simps in-middle.simps*)
apply(*case-tac rn*, *simp*, *case-tac nat*, *auto*)
done

lemma [*simp*]:
abc-fetch as aprog = *Some* (*Dec n e*) $\Longrightarrow$
dec-inv-1 (*layout-of aprog*) *n e* (*as*, [])
(*Suc* (*start-of* (*layout-of aprog*) *as*), *Oc* # *Bk* # *Bk* # *ires*, $Bk^{rn\ -\ Suc\ 0}$) *ires*
$\wedge$
dec-inv-2 (*layout-of aprog*) *n e* (*as*, [])

(*Suc* (*start-of* (*layout-of aprog*) *as*), *Oc* # *Bk* # *Bk* # *ires*, $Bk^{rn - Suc\ 0}$) *ires*
apply(*simp add*: *dec-inv-1.simps dec-inv-2.simps*)
apply(*case-tac n*, *auto*)
done

lemma [*simp*]:
⟦*am* ≠ []; <*am*> = *Oc* # *r'*;
abc-fetch as aprog = *Some* (*Dec n e*)⟧
⟹ *inv-locate-b* (*as*, *am*) (*0*, *Oc* # *Bk* # *Bk* # *ires*, *r'* @ Bk^{rn}) *ires*
apply(*auto simp*: *inv-locate-b.simps in-middle.simps*)
apply(*rule-tac x* = *tl am* **in** *exI*, *rule-tac x* = *0* **in** *exI*,
rule-tac x = *hd am* **in** *exI*, *simp*)
apply(*rule-tac x* = *Suc 0* **in** *exI*)
apply(*rule-tac x* = *hd am* **in** *exI*, *simp*)
apply(*case-tac am*, *simp*, *case-tac list*, *auto simp*: *tape-of-nl-abv tape-of-nat-list.simps*)
apply(*case-tac rn*, *auto*)
done

lemma [*simp*]:
⟦<*am*> = *Oc* # *r'*; *abc-fetch as aprog* = *Some* (*Dec n e*)⟧ ⟹
inv-locate-n-b (*as*, *am*) (*0*, *Oc* # *Bk* # *Bk* # *ires*, *r'* @ Bk^{rn}) *ires*
apply(*auto simp*: *inv-locate-n-b.simps*)
apply(*rule-tac x* = *tl am* **in** *exI*, *rule-tac x* = *hd am* **in** *exI*, *auto*)
apply(*case-tac* [!] *am*, *auto simp*: *tape-of-nl-abv tape-of-nat-list.simps*)
apply(*case-tac* [!]*list*, *auto simp*: *tape-of-nl-abv tape-of-nat-list.simps*)
apply(*case-tac rn*, *simp*, *simp*)
apply(*erule-tac x* = *nat* **in** *allE*, *simp*)
done

lemma [*simp*]:
⟦*am* ≠ [];
<*am*> = *Oc* # *r'*;
abc-fetch as aprog = *Some* (*Dec n e*)⟧ ⟹
dec-inv-1 (*layout-of aprog*) *n e* (*as*, *am*)
(*Suc* (*start-of* (*layout-of aprog*) *as*),
Oc # *Bk* # *Bk* # *ires*, *r'* @ Bk^{rn}) *ires* ∧
dec-inv-2 (*layout-of aprog*) *n e* (*as*, *am*)
(*Suc* (*start-of* (*layout-of aprog*) *as*),
Oc # *Bk* # *Bk* # *ires*, *r'* @ Bk^{rn}) *ires*
apply(*simp add*: *dec-inv-1.simps dec-inv-2.simps*)
apply(*case-tac n*, *auto*)
done

lemma [*simp*]: *am* ≠ [] ⟹ ∃ *r'*. <*am*::*nat list*> = *Oc* # *r'*
apply(*case-tac am*, *simp*, *case-tac list*)
apply(*auto simp*: *tape-of-nl-abv tape-of-nat-list.simps*)
done

lemma [*simp*]: *start-of* (*layout-of aprog*) *as* > *0* ⟹

(*fetch* (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Dec n e*)) (*Suc 0*) *Bk*)
= (*W1*, *start-of* (*layout-of aprog*) *as*)
apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append Suc-pre tdec-b-def)
thm *findnth-nth*
apply(*insert findnth-nth*[*of 0 n 0*], *simp*)
apply(*simp add*: *findnth.simps*)
done

lemma [*simp*]:
start-of (*layout-of aprog*) *as* > 0
$\Longrightarrow$ (*t-step* (*start-of* (*layout-of aprog*) *as*, *Bk* # *Bk* # *ires*, Bk^{rn})
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*) (*Dec n e*),
start-of (*layout-of aprog*) *as* $-$ *Suc 0*))
= (*start-of* (*layout-of aprog*) *as*, *Bk* # *Bk* # *ires*, *Oc* # $Bk^{rn- \, Suc\ 0}$)
apply(*simp add*: *t-step.simps*)
apply(*case-tac start-of* (*layout-of aprog*) *as*,
auto simp: *new-tape.simps*)
apply(*case-tac rn*, *auto*)
done

lemma [*simp*]: *start-of* (*layout-of aprog*) *as* $> 0 \Longrightarrow$
(*fetch* (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*)) (*Suc 0*) *Oc*)
= (*R*, *Suc* (*start-of* (*layout-of aprog*) *as*))

apply(*auto simp*: *ci.simps findnth.simps fetch.simps*
nth-of.simps tshift.simps nth-append
Suc-pre tdec-b-def)
apply(*insert findnth-nth*[*of 0 n Suc 0*], *simp*)
apply(*simp add*: *findnth.simps*)
done

lemma [*simp*]: *start-of* (*layout-of aprog*) *as* $> 0 \Longrightarrow$
(*t-step* (*start-of* (*layout-of aprog*) *as*, *Bk* # *Bk* # *ires*, *Oc* # $Bk^{rn\ -\ Suc\ 0}$)
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*) (*Dec n e*),
start-of (*layout-of aprog*) *as* $-$ *Suc 0*)) =
(*Suc* (*start-of* (*layout-of aprog*) *as*), *Oc* # *Bk* # *Bk* # *ires*, $Bk^{rn-Suc\ 0}$)
apply(*simp add*: *t-step.simps*)
apply(*case-tac start-of* (*layout-of aprog*) *as*,
auto simp: *new-tape.simps*)
done

lemma [*simp*]: *start-of* (*layout-of aprog*) *as* $> 0 \Longrightarrow$
t-step (*start-of* (*layout-of aprog*) *as*, *Bk* # *Bk* # *ires*, *Oc* # r' @ Bk^{rn})
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*) (*Dec n e*),
start-of (*layout-of aprog*) *as* $-$ *Suc 0*) =
(*Suc* (*start-of* (*layout-of aprog*) *as*), *Oc* # *Bk* # *Bk* # *ires*, r' @ Bk^{rn})

apply(*simp add*: *t-step.simps*)
apply(*case-tac start-of* (*layout-of aprog*) *as*,
auto simp: *new-tape.simps*)
done

lemma *crsp-next-state*:
⟦*crsp-l* (*layout-of aprog*) (*as*, *am*) *tc ires*;
abc-fetch as aprog = *Some* (*Dec n e*)⟧
⟹ ∃ *stp'* > *0*. (λ (*s*, *l*, *r*).
(*s* = *Suc* (*start-of* (*layout-of aprog*) *as*)
∧ (*dec-inv-1* (*layout-of aprog*) *n e* (*as*, *am*) (*s*, *l*, *r*) *ires*)
∧ (*dec-inv-2* (*layout-of aprog*) *n e* (*as*, *am*) (*s*, *l*, *r*)) *ires*))
(*t-steps tc* (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *stp'*)
apply(*subgoal-tac start-of* (*layout-of aprog*) *as* > *0*)
apply(*case-tac tc*, *case-tac b*, *auto simp*: *crsp-l.simps*)
apply(*case-tac am* = [], *simp*)
apply(*rule-tac x* = *Suc* (*Suc 0*) **in** *exI*, *simp add*: *t-steps.simps*)
proof−
fix *rn*
assume *h1*: *am* ≠ [] *abc-fetch as aprog* = *Some* (*Dec n e*)
start-of (*layout-of aprog*) *as* > *0*
hence *h2*: ∃ *r'*. <*am*> = *Oc* # *r'*
by *simp*
from *h1* **and** *h2* **show**
∃*stp'*>*0*. *case t-steps* (*start-of* (*layout-of aprog*) *as*, *Bk* # *Bk* # *ires*, <*am*> @ Bk^{rn})
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*) (*Dec n e*),
start-of (*layout-of aprog*) *as* − *Suc 0*) *stp' of*
(*s*, *ab*) ⇒ *s* = *Suc* (*start-of* (*layout-of aprog*) *as*) ∧
dec-inv-1 (*layout-of aprog*) *n e* (*as*, *am*) (*s*, *ab*) *ires* ∧
dec-inv-2 (*layout-of aprog*) *n e* (*as*, *am*) (*s*, *ab*) *ires*
proof(*erule-tac exE*, *simp*, *rule-tac x* = *Suc 0* **in** *exI*,
simp add: *t-steps.simps*)
qed
next
assume *abc-fetch as aprog* = *Some* (*Dec n e*)
thus *0* < *start-of* (*layout-of aprog*) *as*
apply(*insert startof-not0*[*of layout-of aprog as*], *simp*)
done
qed

lemma *dec-crsp-ex1*:
⟦*crsp-l* (*layout-of aprog*) (*as*, *am*) *tc ires*;
abc-fetch as aprog = *Some* (*Dec n e*);
abc-lm-v am n = *0*⟧
⟹ ∃*stp* > *0*. *crsp-l* (*layout-of aprog*) (*e*, *abc-lm-s am n 0*)
(*t-steps tc* (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *stp*) *ires*

proof −
assume *h1*: *crsp-l* (*layout-of aprog*) (*as*, *am*) *tc ires*
abc-fetch as aprog = *Some* (*Dec n e*) *abc-lm-v am n* = *0*
hence *h2*: ∃ *stp′* > *0*. (λ (*s*, *l*, *r*).
(*s* = *Suc* (*start-of* (*layout-of aprog*) *as*) ∧
(*dec-inv-1* (*layout-of aprog*) *n e* (*as*, *am*) (*s*, *l*, *r*)) *ires*))
(*t-steps tc* (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *stp′*)
apply(*insert crsp-next-state*[*of aprog as am tc ires n e*], *auto*)
done
from *h1* **and** *h2* **show**
∃ *stp* > *0*. *crsp-l* (*layout-of aprog*) (*e*, *abc-lm-s am n 0*)
(*t-steps tc* (*ci* (*layout-of aprog*) (*start-of*
(*layout-of aprog*) *as*) (*Dec n e*),
start-of (*layout-of aprog*) *as* − *Suc 0*) *stp*) *ires*
proof(*erule-tac exE*, *case-tac* (*t-steps tc* (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Dec n e*), *start-of*
(*layout-of aprog*) *as* − *Suc 0*) *stp′*), *simp*)
fix *stp′ a b c*
assume *h3*: *stp′* > *0* ∧ *a* = *Suc* (*start-of* (*layout-of aprog*) *as*) ∧
dec-inv-1 (*layout-of aprog*) *n e* (*as*, *am*) (*a*, *b*, *c*) *ires*
abc-fetch as aprog = *Some* (*Dec n e*) *abc-lm-v am n* = *0*
t-steps tc (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *stp′*
= (*Suc* (*start-of* (*layout-of aprog*) *as*), *b*, *c*)
thus ∃ *stp* > *0*. *crsp-l* (*layout-of aprog*) (*e*, *abc-lm-s am n 0*)
(*t-steps tc* (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *stp*) *ires*
proof(*erule-tac conjE*, *simp*)
assume *dec-inv-1* (*layout-of aprog*) *n e* (*as*, *am*)
(*Suc* (*start-of* (*layout-of aprog*) *as*), *b*, *c*) *ires*
abc-fetch as aprog = *Some* (*Dec n e*)
abc-lm-v am n = *0*
t-steps tc (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Dec n e*),
start-of (*layout-of aprog*) *as* − *Suc 0*) *stp′*
= (*Suc* (*start-of* (*layout-of aprog*) *as*), *b*, *c*)
hence *h4*: ∃ *stp*. (λ(*s′*, *l′*, *r′*). *s′* =
start-of (*layout-of aprog*) *e* ∧
dec-inv-1 (*layout-of aprog*) *n e* (*as*, *am*) (*s′*, *l′*, *r′*) *ires*)
(*t-steps* (*start-of* (*layout-of aprog*) *as* + *1*, *b*, *c*)
(*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Dec n e*),
start-of (*layout-of aprog*) *as* − *Suc 0*) *stp*)
apply(*rule-tac dec-inv-stop1*, *auto*)
done
from *h3* **and** *h4* **show** *?thesis*
apply(*erule-tac exE*)
apply(*rule-tac x* = *stp′* + *stp* **in** *exI*, *simp*)

apply(*case-tac* (*t-steps* (*Suc* (*start-of* (*layout-of aprog*) *as*),
b, *c*) (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Dec n e*),
start-of (*layout-of aprog*) *as* − *Suc 0*) *stp*),
simp)
apply(*rule-tac dec-inv-stop-cond1*, *auto*)
done
qed
qed
qed

lemma *dec-crsp-ex2*:
⟦*crsp-l* (*layout-of aprog*) (*as*, *am*) *tc ires*;
abc-fetch as aprog = *Some* (*Dec n e*);
0 < *abc-lm-v am n*⟧
⟹ ∃ *stp* > *0*. *crsp-l* (*layout-of aprog*)
(*Suc as*, *abc-lm-s am n* (*abc-lm-v am n* − *Suc 0*))
(*t-steps tc* (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *stp*) *ires*
proof −
assume *h1*:
crsp-l (*layout-of aprog*) (*as*, *am*) *tc ires*
abc-fetch as aprog = *Some* (*Dec n e*)
abc-lm-v am n > *0*
hence *h2*:
∃ *stp′* > *0*. (λ (*s*, *l*, *r*). (*s* = *Suc* (*start-of* (*layout-of aprog*) *as*)
∧ (*dec-inv-2* (*layout-of aprog*) *n e* (*as*, *am*) (*s*, *l*, *r*)) *ires*))
(*t-steps tc* (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *stp′*)
apply(*insert crsp-next-state*[*of aprog as am tc ires n e*], *auto*)
done
from *h1* **and** *h2* **show**
∃ *stp* >*0*. *crsp-l* (*layout-of aprog*)
(*Suc as*, *abc-lm-s am n* (*abc-lm-v am n* − *Suc 0*))
(*t-steps tc* (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *stp*) *ires*
proof(*erule-tac exE*,
case-tac
(*t-steps tc* (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *stp′*), *simp*)
fix *stp′ a b c*
assume *h3*: *0* < *stp′* ∧ *a* = *Suc* (*start-of* (*layout-of aprog*) *as*) ∧
dec-inv-2 (*layout-of aprog*) *n e* (*as*, *am*) (*a*, *b*, *c*) *ires*
abc-fetch as aprog = *Some* (*Dec n e*)
abc-lm-v am n > *0*
t-steps tc (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Dec n e*),
start-of (*layout-of aprog*) *as* − *Suc 0*) *stp′*
= (*Suc* (*start-of* (*layout-of aprog*) *as*), *b*, *c*)

thus *?thesis*
proof(*erule-tac conjE*, *simp*)
assume
dec-inv-2 (*layout-of aprog*) *n e* (*as*, *am*)
(*Suc* (*start-of* (*layout-of aprog*) *as*), *b*, *c*) *ires*
abc-fetch as aprog = *Some* (*Dec n e*) *abc-lm-v am n* > *0*
t-steps tc (*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *stp′*
= (*Suc* (*start-of* (*layout-of aprog*) *as*), *b*, *c*)
hence *h4*:
∃ *stp*. (λ(*s′*, *l′*, *r′*). *s′* = *start-of* (*layout-of aprog*) (*Suc as*) ∧
dec-inv-2 (*layout-of aprog*) *n e* (*as*, *am*) (*s′*, *l′*, *r′*) *ires*)
(*t-steps* (*start-of* (*layout-of aprog*) *as* + *1*, *b*, *c*)
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *stp*)
apply(*rule-tac dec-stop2*, *auto*)
done
from *h3* **and** *h4* **show** *?thesis*
apply(*erule-tac exE*)
apply(*rule-tac x* = *stp′* + *stp* **in** *exI*, *simp*)
apply(*case-tac*
(*t-steps* (*Suc* (*start-of* (*layout-of aprog*) *as*), *b*, *c*)
(*ci* (*layout-of aprog*) (*start-of* (*layout-of aprog*) *as*)
(*Dec n e*), *start-of* (*layout-of aprog*) *as* − *Suc 0*) *stp*)
,*simp*)
apply(*rule-tac dec-inv-stop-cond2*, *auto*)
done
qed
qed
qed

lemma *dec-crsp-ex-pre*:
⟦*ly* = *layout-of aprog*; *crsp-l ly* (*as*, *am*) *tc ires*;
abc-fetch as aprog = *Some* (*Dec n e*)⟧
⟹ ∃ *stp* > *0*. *crsp-l ly* (*abc-step-l* (*as*, *am*) (*Some* (*Dec n e*)))
(*t-steps tc* (*ci* (*layout-of aprog*) (*start-of ly as*) (*Dec n e*),
start-of ly as − *Suc 0*) *stp*) *ires*
apply(*auto simp*: *abc-step-l.simps intro*: *dec-crsp-ex2 dec-crsp-ex1*)
done

lemma *dec-crsp-ex*:
assumes *layout*: — There is an Abacus program *aprog* with layout *ly*
ly = *layout-of aprog*
and *dec*: — There is an *Dec n e* instruction at postion *as* of *aprog*
abc-fetch as aprog = *Some* (*Dec n e*)
and *correspond*:
— Abacus configuration (*as*, *am*) is in correspondence with TM configuration *tc*
crsp-l ly (*as*, *am*) *tc ires*
shows

$\exists$ *stp* > *0*. *crsp-l ly* (*abc-step-l* (*as*, *am*) (*Some* (*Dec n e*)))
(*t-steps tc* (*ci* (*layout-of aprog*) (*start-of ly as*) (*Dec n e*),
start-of ly as − *Suc 0*) *stp*) *ires*

proof −
from *dec-crsp-ex-pre layout dec correspond* **show** *?thesis* **by** *blast*
qed

lemma *goto-fetch*:
fetch (*ci* (*layout-of aprog*)
(*start-of* (*layout-of aprog*) *as*) (*Goto n*)) (*Suc 0*) *b*
= (*Nop*, *start-of* (*layout-of aprog*) *n*)
apply(*auto simp*: *ci.simps fetch.simps nth-of.simps*
split: *block.splits*)
done

Correctness of complied *Goto n*

lemma *goto-crsp-ex-pre*:
⟦*ly* = *layout-of aprog*;
crsp-l ly (*as*, *am*) *tc ires*;
abc-fetch as aprog = *Some* (*Goto n*)⟧
⟹ $\exists$ *stp* > *0*. *crsp-l ly* (*abc-step-l* (*as*, *am*) (*Some* (*Goto n*)))
(*t-steps tc* (*ci* (*layout-of aprog*) (*start-of ly as*) (*Goto n*),
start-of ly as − *Suc 0*) *stp*) *ires*
apply(*rule-tac x* = *1* **in** *exI*)
apply(*simp add*: *abc-step-l.simps t-steps.simps t-step.simps*)
apply(*case-tac tc*, *simp*)
apply(*subgoal-tac a* = *start-of* (*layout-of aprog*) *as*, *auto*)
apply(*subgoal-tac start-of* (*layout-of aprog*) *as* > *0*, *simp*)
apply(*auto simp*: *goto-fetch new-tape.simps crsp-l.simps*)
apply(*rule startof-not0*)
done

lemma *goto-crsp-ex*:
assumes *layout*: *ly* = *layout-of aprog*
and *goto*: *abc-fetch as aprog* = *Some* (*Goto n*)
and *correspondence*: *crsp-l ly* (*as*, *am*) *tc ires*
shows $\exists$ *stp*>*0*. *crsp-l ly* (*abc-step-l* (*as*, *am*) (*Some* (*Goto n*)))
(*t-steps tc* (*ci* (*layout-of aprog*) (*start-of ly as*) (*Goto n*),
start-of ly as − *Suc 0*) *stp*) *ires*
proof −
from *goto-crsp-ex-pre* **and** *layout goto correspondence* **show** *?thesis* **by** *blast*
qed

8.4 The correctness of the compiler

declare *abc-step-l.simps*[*simp del*]

```
lemma tm-crsp-ex:
    ⟦ly = layout-of aprog;
     crsp-l ly (as, am) tc ires;
     as < length aprog;
     abc-fetch as aprog = Some ins⟧
  ⟹ ∃ n > 0. crsp-l ly (abc-step-l (as,am) (Some ins))
       (t-steps tc (ci (layout-of aprog) (start-of ly as)
         (ins), (start-of ly as) − (Suc 0)) n) ires
apply(case-tac ins, simp)
apply(auto intro: inc-crsp-ex-pre dec-crsp-ex goto-crsp-ex)
done

lemma start-of-pre:
  n < length aprog ⟹ start-of (layout-of aprog) n
                 = start-of (layout-of (butlast aprog)) n
apply(induct n, simp add: start-of.simps, simp)
apply(simp add: layout-of.simps start-of.simps)
apply(subgoal-tac n < length aprog − Suc 0, simp)
apply(subgoal-tac (aprog ! n) = (butlast aprog ! n), simp)
proof −
  fix n
  assume h1: Suc n < length aprog
  thus aprog ! n = butlast aprog ! n
    apply(case-tac length aprog, simp, simp)
    apply(insert nth-append[of butlast aprog [last aprog] n])
    apply(subgoal-tac (butlast aprog @ [last aprog]) = aprog)
    apply(simp split: if-splits)
    apply(rule append-butlast-last-id, case-tac aprog, simp, simp)
    done
next
  fix n
  assume Suc n < length aprog
  thus n < length aprog − Suc 0
    apply(case-tac aprog, simp, simp)
    done
qed

lemma zip-eq: xs = ys ⟹ zip xs zs = zip ys zs
by simp

lemma tpairs-of-append-iff: length aprog = Suc n ⟹
    tpairs-of aprog = tpairs-of (butlast aprog) @
                [(start-of (layout-of aprog) n, aprog ! n)]
apply(simp add: tpairs-of.simps)
apply(insert zip-append[of map (start-of (layout-of aprog)) [0..<n]
   butlast aprog [start-of (layout-of aprog) n] [last aprog]])
apply(simp del: zip-append)
apply(subgoal-tac (butlast aprog @ [last aprog]) = aprog, auto)
apply(rule-tac zip-eq, auto)
```

```
apply(rule-tac start-of-pre, simp)
apply(insert last-conv-nth[of aprog], case-tac aprog, simp, simp)
apply(rule append-butlast-last-id, case-tac aprog, simp, simp)
done

lemma [simp]: list-all (λ(n, tm). abacus.t-ncorrect (ci layout n tm))
      (zip (map (start-of layout) [0..<length aprog]) aprog)
proof(induct length aprog arbitrary: aprog, simp)
  fix x aprog
  assume ind: ⋀aprog. x = length aprog ⟹
      list-all (λ(n, tm). abacus.t-ncorrect (ci layout n tm))
        (zip (map (start-of layout) [0..<length aprog]) aprog)
  and h: Suc x = length (aprog::abc-inst list)
  have g1: list-all (λ(n, tm). abacus.t-ncorrect (ci layout n tm))
    (zip (map (start-of layout) [0..<length (butlast aprog)])
                                          (butlast aprog))
    using h
    apply(rule-tac ind, auto)
    done
  have g2: (map (start-of layout) [0..<length aprog]) =
                map (start-of layout) ([0..<length aprog − 1]
      @ [length aprog − 1])
    using h
    apply(case-tac aprog, simp, simp)
    done
  have ∃ xs a. aprog = xs @ [a]
    using h
    apply(rule-tac x = butlast aprog in exI,
       rule-tac x = last aprog in exI)
    apply(case-tac aprog = [], simp, simp)
    done
  from this obtain xs where ∃ a. aprog = xs @ [a] ..
  from this obtain a where g3: aprog = xs @ [a] ..
  from g1 and g2 and g3 show list-all (λ(n, tm).
                          abacus.t-ncorrect (ci layout n tm))
          (zip (map (start-of layout) [0..<length aprog]) aprog)
    apply(simp)
    apply(auto simp: t-ncorrect.simps ci.simps  tshift.simps
       tinc-b-def tdec-b-def split: abc-inst.splits)
    apply arith+
    done
qed

lemma [intro]: abc2t-correct aprog
apply(simp add: abc2t-correct.simps tpairs-of.simps
        split: abc-inst.splits)
done

lemma as-out: ⟦ly = layout-of aprog; tprog = tm-of aprog;
```

```
            crsp-l ly (as, am) tc ires; length aprog ≤ as⟧
         ⟹ abc-step-l (as, am) (abc-fetch as aprog) = (as, am)
apply(simp add: abc-fetch.simps abc-step-l.simps)
done

lemma tm-merge-ex:
  ⟦crsp-l (layout-of aprog) (as, am) tc ires;
    as < length aprog;
    abc-fetch as aprog = Some a;
    abc2t-correct aprog;
    crsp-l (layout-of aprog) (abc-step-l (as, am) (Some a))
     (t-steps tc (ci (layout-of aprog) (start-of (layout-of aprog) as)
        a, start-of (layout-of aprog) as − Suc 0) n) ires;
    n > 0⟧
   ⟹ ∃ stp > 0. crsp-l (layout-of aprog) (abc-step-l (as, am)
                    (Some a)) (t-steps tc (tm-of aprog, 0) stp) ires
apply(case-tac (t-steps tc (ci (layout-of aprog)
           (start-of (layout-of aprog) as) a,
            start-of (layout-of aprog) as − Suc 0) n), simp)
apply(case-tac (abc-step-l (as, am) (Some a)), simp)
proof −
  fix aa b c aaa ba
  assume h:
  crsp-l (layout-of aprog) (as, am) tc ires
  as < length aprog
  abc-fetch as aprog = Some a
  crsp-l (layout-of aprog) (aaa, ba) (aa, b, c) ires
  abc2t-correct aprog
  n>0
  t-steps tc (ci (layout-of aprog) (start-of (layout-of aprog) as) a,
     start-of (layout-of aprog) as − Suc 0) n = (aa, b, c)
   abc-step-l (as, am) (Some a) = (aaa, ba)
  hence aa = start-of (layout-of aprog) aaa
    apply(simp add: crsp-l.simps)
    done
  from this and h show
  ∃ stp > 0. crsp-l (layout-of aprog) (aaa, ba)
                      (t-steps tc (tm-of aprog, 0) stp) ires
    apply(insert tms-out-ex[of layout-of aprog aprog
             tm-of aprog as am tc ires a n aa b c aaa ba], auto)
    done
qed

lemma crsp-inside:
  ⟦ly = layout-of aprog;
    tprog = tm-of aprog;
    crsp-l ly (as, am) tc ires;
    as < length aprog⟧ ⟹
    (∃ stp > 0. crsp-l ly (abc-step-l (as,am) (abc-fetch as aprog))
```

(*t-steps tc* (*tprog*, *0*) *stp*) *ires*)
apply(*case-tac abc-fetch as aprog*, *simp add*: *abc-fetch.simps*)
proof −
fix *a*
assume *ly* = *layout-of aprog*
tprog = *tm-of aprog*
crsp-l ly (*as*, *am*) *tc ires*
as < *length aprog*
abc-fetch as aprog = *Some a*
thus ∃ *stp* > *0*. *crsp-l ly* (*abc-step-l* (*as*, *am*)
(*abc-fetch as aprog*)) (*t-steps tc* (*tprog*, *0*) *stp*) *ires*
proof(*insert tm-crsp-ex*[*of ly aprog as am tc ires a*],
auto intro: *tm-merge-ex*)
qed
qed

lemma *crsp-outside*:
⟦*ly* = *layout-of aprog*; *tprog* = *tm-of aprog*;
crsp-l ly (*as*, *am*) *tc ires*; *as* ≥ *length aprog*⟧
⟹ (∃ *stp*. *crsp-l ly* (*abc-step-l* (*as*,*am*) (*abc-fetch as aprog*))
(*t-steps tc* (*tprog*, *0*) *stp*) *ires*)
apply(*subgoal-tac abc-step-l* (*as*, *am*) (*abc-fetch as aprog*)
= (*as*, *am*), *simp*)
apply(*rule-tac x* = *0* **in** *exI*, *simp add*: *t-steps.simps*)
apply(*rule as-out*, *simp*+)
done

Single-step correntess of the compiler.

lemma *astep-crsp-pre*:
⟦*ly* = *layout-of aprog*;
tprog = *tm-of aprog*;
crsp-l ly (*as*, *am*) *tc ires*⟧
⟹ (∃ *stp*. *crsp-l ly* (*abc-step-l* (*as*,*am*)
(*abc-fetch as aprog*)) (*t-steps tc* (*tprog*, *0*) *stp*) *ires*)
apply(*case-tac as* < *length aprog*)
apply(*drule-tac crsp-inside*, *auto*)
apply(*rule-tac crsp-outside*, *simp*+)
done

Single-step correntess of the compiler.

lemma *astep-crsp-pre1*:
⟦*ly* = *layout-of aprog*;
tprog = *tm-of aprog*;
crsp-l ly (*as*, *am*) *tc ires*⟧
⟹ (∃ *stp*. *crsp-l ly* (*abc-step-l* (*as*,*am*)
(*abc-fetch as aprog*)) (*t-steps tc* (*tprog*, *0*) *stp*) *ires*)
apply(*case-tac as* < *length aprog*)
apply(*drule-tac crsp-inside*, *auto*)
apply(*rule-tac crsp-outside*, *simp*+)

done

lemma *astep-crsp*:
 assumes *layout*:
 — There is a Abacus program *aprog* with layout *ly*
 ly = *layout-of aprog*
 and *compiled*:
 — *tprog* is the TM compiled from *aprog*
 tprog = *tm-of aprog*
 and *corresponds*:
 — Abacus configuration (*as*, *am*) is in correspondence with TM configuration *tc*
 crsp-l ly (*as*, *am*) *tc ires*
 — One step execution of *aprog* can be simulated by multi-step execution of *tprog*

 shows (∃ *stp*. *crsp-l ly* (*abc-step-l* (*as*,*am*)
 (*abc-fetch as aprog*)) (*t-steps tc* (*tprog*, *0*) *stp*) *ires*)
proof −
 from *astep-crsp-pre1* [*OF layout compiled corresponds*] **show** *?thesis* **.**
qed

lemma *steps-crsp-pre*:
 ⟦*ly* = *layout-of aprog*; *tprog* = *tm-of aprog*;
 crsp-l ly ac tc ires; *ac*′ = *abc-steps-l ac aprog n*⟧ ⟹
 (∃ *n*′. *crsp-l ly ac*′ (*t-steps tc* (*tprog*, *0*) *n*′) *ires*)
apply(*induct n arbitrary*: *ac*′ *ac tc*, *simp add*: *abc-steps-l.simps*)
apply(*rule-tac x* = *0* **in** *exI*)
apply(*case-tac ac*, *simp add*: *abc-steps-l.simps t-steps.simps*)
apply(*case-tac ac*, *simp add*: *abc-steps-l.simps*)
apply(*subgoal-tac*
 (∃ *stp*. *crsp-l ly* (*abc-step-l* (*a*, *b*)
 (*abc-fetch a aprog*)) (*t-steps tc* (*tprog*, *0*) *stp*) *ires*))
apply(*erule exE*)
apply(*subgoal-tac*
 ∃ *n*′. *crsp-l* (*layout-of aprog*)
 (*abc-steps-l* (*abc-step-l* (*a*, *b*) (*abc-fetch a aprog*)) *aprog n*)
 (*t-steps* ((*t-steps tc* (*tprog*, *0*) *stp*)) (*tm-of aprog*, *0*) *n*′) *ires*)
apply(*erule exE*)
apply(*subgoal-tac*
 t-steps (*t-steps tc* (*tprog*, *0*) *stp*) (*tm-of aprog*, *0*) *n*′ =
 t-steps tc (*tprog*, *0*) (*stp* + *n*′))
apply(*rule-tac x* = *stp* + *n*′ **in** *exI*, *simp*)
apply(*auto intro*: *astep-crsp simp*: *t-step-add*)
done

Multi-step correctess of the compiler.

lemma *steps-crsp*:
 assumes *layout*:
 — There is an Abacus program *aprog* with layout *ly*
 ly = *layout-of aprog*

and *compiled*:
— *tprog* is the TM compiled from *aprog*
tprog = *tm-of aprog*
and *correspond*:
— Abacus configuration *ac* is in correspondence with TM configuration *tc*
crsp-l ly ac tc ires
and *execution*:
— *ac'* is the configuration obtained from *n*-step execution of *aprog* starting from configuration *ac*
ac' = *abc-steps-l ac aprog n*
— There exists steps *n'* steps, after these steps of execution, the Turing configuration such obtained is in correspondence with *ac'*
shows (∃ *n'*. *crsp-l ly ac'* (*t-steps tc* (*tprog*, *0*) *n'*) *ires*)
proof −
from *steps-crsp-pre* [*OF layout compiled correspond execution*] **show** *?thesis* **.**
qed

8.5 The Mop-up machine

fun *mop-bef* :: *nat* ⇒ *tprog*
where
mop-bef 0 = [] |
mop-bef (*Suc n*) = *mop-bef n* @
[(*R*, *2*∗*n* + *3*), (*W0*, *2*∗*n* + *2*), (*R*, *2*∗*n* + *1*), (*W1*, *2*∗*n* + *2*)]

definition *mp-up* :: *tprog*
where
mp-up ≡ [(*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 *tMp* :: *nat* ⇒ *nat* ⇒ *tprog*
where
tMp n off = *tshift* (*mop-bef n* @ *tshift mp-up* (*2*∗*n*)) *off*

declare *mp-up-def*[*simp del*] *tMp.simps*[*simp del*] *mop-bef.simps*[*simp del*]

lemma *tm-append-step*:
⟦*t-ncorrect tp1*; *t-step tc* (*tp1*, *0*) = (*s*, *l*, *r*); *s* ≠ *0*⟧
⟹ *t-step tc* (*tp1* @ *tp2*, *0*) = (*s*, *l*, *r*)
apply(*simp add*: *t-step.simps*)
apply(*case-tac tc*, *simp*)
apply(*case-tac*
(*fetch tp1 a* (*case c of* [] ⇒ *Bk* |
Bk # *xs* ⇒ *Bk* | *Oc* # *xs* ⇒ *Oc*)), *simp*)
apply(*case-tac a*, *simp add*: *fetch.simps*)
apply(*simp add*: *fetch.simps*)
apply(*case-tac c*, *simp*)
apply(*case-tac* [!] *ab*::*block*)

```
apply(auto simp: nth-of.simps nth-append t-ncorrect.simps
          split: if-splits)
done

lemma state0-ind: t-steps (0, l, r) (tp, 0) stp = (0, l, r)
apply(induct stp, simp add: t-steps.simps)
apply(simp add: t-steps.simps t-step.simps fetch.simps new-tape.simps)
done

lemma tm-append-steps:
 ⟦t-ncorrect tp1; t-steps tc (tp1, 0) stp = (s, l ,r); s ≠ 0⟧
  ⟹ t-steps tc (tp1 @ tp2, 0) stp = (s, l, r)
apply(induct stp arbitrary: tc s l r)
apply(case-tac tc,  simp)
apply(simp add: t-steps.simps)
proof −
  fix stp tc s l r
  assume h1: ⋀tc s l r. ⟦t-ncorrect tp1; t-steps tc (tp1, 0) stp =
   (s, l, r); s ≠ 0⟧ ⟹ t-steps tc (tp1 @ tp2, 0) stp = (s, l, r)
    and h2: t-steps tc (tp1, 0) (Suc stp) = (s, l, r) s ≠ 0
           t-ncorrect tp1
  thus t-steps tc (tp1 @ tp2, 0) (Suc stp) = (s, l, r)
    apply(simp add: t-steps.simps)
    apply(case-tac (t-step tc (tp1, 0)), simp)
    proof−
      fix a b c
      assume g1: ⋀tc s l r. ⟦t-steps tc (tp1, 0) stp = (s, l, r);
               0 < s⟧ ⟹ t-steps tc (tp1 @ tp2, 0) stp = (s, l, r)
 and g2: t-step tc (tp1, 0) = (a, b, c)
               t-steps (a, b, c) (tp1, 0) stp = (s, l, r)
               0 < s
               t-ncorrect tp1
      hence g3: a > 0
 apply(case-tac a::nat, auto simp: t-steps.simps)
 apply(simp add: state0-ind)
 done
      from g1 and g2 and this have g4:
                   (t-step tc (tp1 @ tp2, 0)) = (a, b, c)
 apply(rule-tac tm-append-step, simp, simp, simp)
 done
      from g1 and g2 and g3 and g4 show
          t-steps (t-step tc (tp1 @ tp2, 0)) (tp1 @ tp2, 0) stp
                                                   = (s, l, r)
 apply(simp)
 done
    qed
qed

lemma shift-fetch:
```

```
⟦n < length tp;
 (tp:: (taction × nat) list) ! n = (aa, ba);
  ba ≠ 0⟧
  ⟹ (tshift tp (length tp div 2)) ! n =
                    (aa , ba + length tp div 2)
apply(simp add: tshift.simps)
done

lemma tshift-length-equal: length (tshift tp q) = length tp
apply(auto simp: tshift.simps)
done

thm nth-of.simps

lemma [simp]: t-ncorrect tp ⟹ 2 * (length tp div 2) = length tp
apply(auto simp: t-ncorrect.simps)
done

lemma  tm-append-step-equal':
  ⟦t-ncorrect tp; t-ncorrect tp'; off = length tp div 2⟧ ⟹
   (λ (s, l, r). ((λ (s', l', r').
     (s'≠ 0 ⟶ (s = s' + off ∧ l = l' ∧ r = r')))
        (t-step (a, b, c) (tp', 0))))
              (t-step (a + off, b, c) (tp @ tshift tp' off, 0))
apply(simp add: t-step.simps)
apply(case-tac a, simp add: fetch.simps)
apply(case-tac
(fetch tp' a (case c of [] ⇒ Bk | Bk # xs ⇒ Bk | Oc # xs ⇒ Oc)),
 simp)
apply(case-tac
(fetch (tp @ tshift tp' (length tp div 2))
       (Suc (nat + length tp div 2))
          (case c of [] ⇒ Bk | Bk # xs ⇒ Bk | Oc # xs ⇒ Oc)),
 simp)
apply(case-tac (new-tape aa (b, c)),
      case-tac (new-tape aaa (b, c)), simp,
      rule impI, simp add: fetch.simps split: block.splits option.splits)
apply (auto simp: nth-of.simps t-ncorrect.simps
                  nth-append tshift-length-equal tshift.simps split: if-splits)
done


lemma  tm-append-step-equal:
 ⟦t-ncorrect tp; t-ncorrect tp'; off = length tp div 2;
  t-step (a, b, c) (tp', 0) = (aa, ab, bb);  aa ≠ 0⟧
 ⟹ t-step (a + length tp div 2, b, c)
       (tp @ tshift tp' (length tp div 2), 0)
                        = (aa + length tp div 2, ab, bb)
apply(insert tm-append-step-equal'[of tp tp' off a b c], simp)
```

```
apply(case-tac (t-step (a + length tp div 2, b, c)
                  (tp @ tshift tp′ (length tp div 2), 0)), simp)
done

lemma tm-append-steps-equal:
 ⟦t-ncorrect tp; t-ncorrect tp′; off = length tp div 2⟧ ⟹
   (λ (s, l, r). ((λ (s′, l′, r′). ((s′≠ 0 ⟶ s = s′ + off ∧ l = l′
                 ∧ r = r′))) (t-steps (a, b, c) (tp′, 0) stp)))
   (t-steps (a + off, b, c) (tp @ tshift tp′ off, 0) stp)
apply(induct stp arbitrary : a b c, simp add: t-steps.simps)
apply(simp add: t-steps.simps)
apply(case-tac (t-step (a, b, c) (tp′, 0)), simp)
apply(case-tac aa = 0, simp add: state0-ind)
apply(subgoal-tac (t-step (a + length tp div 2, b, c)
                   (tp @ tshift tp′ (length tp div 2), 0))
  = (aa + length tp div 2, ba, ca), simp)
apply(rule tm-append-step-equal, auto)
done


type-synonym mopup-type = t-conf ⇒ nat list ⇒ nat ⇒ block 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 = <abc-lm-v 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 (abc-lm-v 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.*
$m = Suc\ (abc\text{-}lm\text{-}v\ lm\ n) \wedge l = Bk^{lnr}\ @\ Oc^{m}\ @\ Bk^{lnl}\ @\ Bk\ \#\ Bk\ \#\ ires$
∧
$(r = \text{<}ml\text{>}\ @\ Bk^{rn}))$

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.*
$m = Suc\ (abc\text{-}lm\text{-}v\ lm\ n) \wedge$
$l = Bk^{lnr}\ @\ Oc^{m}\ @\ Bk^{lnl}\ @\ Bk\ \#\ Bk\ \#\ ires \wedge$
$(r = Bk\ \#\ \text{<}ml\text{>}\ @\ Bk^{rn} \vee$
$r = Bk\ \#\ Bk\ \#\ \text{<}ml\text{>}\ @\ Bk^{rn}))$

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.*
$m = Suc\ (abc\text{-}lm\text{-}v\ lm\ n) \wedge$
$l = Bk^{lnr}\ @\ Oc^{m}\ @\ Bk^{lnl}\ @\ Bk\ \#\ Bk\ \#\ ires \wedge$
$(r = \text{<}ml\text{>}\ @\ Bk^{rn} \vee r = Bk\ \#\ \text{<}ml\text{>}\ @\ Bk^{rn}))$

fun *mopup-left-moving* :: *mopup-type*
where
mopup-left-moving (s, l, r) *lm n ires* =
(∃ *lnl lnr rn m.*
$m = Suc\ (abc\text{-}lm\text{-}v\ lm\ n) \wedge$
$((l = Bk^{lnr}\ @\ Oc^{m}\ @\ Bk^{lnl}\ @\ Bk\ \#\ Bk\ \#\ ires \wedge r = Bk^{rn}) \vee$
$(l = Oc^{m\ -\ 1}\ @\ Bk^{lnl}\ @\ Bk\ \#\ Bk\ \#\ ires \wedge r = Oc\ \#\ Bk^{rn})))$

fun *mopup-jump-over2* :: *mopup-type*
where
mopup-jump-over2 (s, l, r) *lm n ires* =
(∃ *ln rn m1 m2.*
$m1 + m2 = Suc\ (abc\text{-}lm\text{-}v\ lm\ n)$
$\wedge\ r \neq []$
$\wedge\ (hd\ r = Oc \longrightarrow (l = Oc^{m1}\ @\ Bk^{ln}\ @\ Bk\ \#\ Bk\ \#\ ires \wedge r = Oc^{m2}\ @$
$Bk^{rn}))$
$\wedge\ (hd\ r = Bk \longrightarrow (l = Bk^{ln}\ @\ Bk\ \#\ ires \wedge r = Bk\ \#\ Oc^{m1\ +\ m2}\ @$
$Bk^{rn})))$

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)

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-fetch-0*[*simp*]:
(*fetch* (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) *0 b*) = (*Nop*, *0*)
by(*simp add*: *fetch.simps*)

lemma *mop-bef-length*[*simp*]: *length* (*mop-bef n*) = *4* ∗ *n*
apply(*induct n*, *simp add*: *mop-bef.simps*, *simp add*: *mop-bef.simps*)
done

thm *findnth-nth*
lemma *mop-bef-nth*:
⟦*q* < *n*; *x* < *4*⟧ ⟹ *mop-bef n* ! (*4* ∗ *q* + *x*) =
mop-bef (*Suc q*) ! ((*4* ∗ *q*) + *x*)
apply(*induct n*, *simp*)
apply(*case-tac q* < *n*, *simp add*: *mop-bef.simps*, *auto*)
apply(*simp add*: *nth-append*)
apply(*subgoal-tac q* = *n*, *simp*)
apply(*arith*)
done

lemma *fetch-bef-erase-a-o*[*simp*]:
⟦*0* < *s*; *s* ≤ *2* ∗ *n*; *s mod 2* = *Suc 0*⟧
⟹ (*fetch* (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) *s Oc*) = (*W0*, *s* + *1*)
apply(*subgoal-tac* ∃ *q*. *s* = *2*∗*q* + *1*, *auto*)
apply(*subgoal-tac length* (*mop-bef n*) = *4*∗*n*)
apply(*auto simp*: *fetch.simps nth-of.simps nth-append*)
apply(*subgoal-tac mop-bef n* ! (*4* ∗ *q* + *1*) =
mop-bef (*Suc q*) ! ((*4* ∗ *q*) + *1*),
simp add: *mop-bef.simps nth-append*)
apply(*rule mop-bef-nth*, *auto*)
done

lemma *fetch-bef-erase-a-b*[*simp*]:
 $[\![n < length\ lm;\ 0 < s;\ s \leq 2 * n;\ s\ mod\ 2 = Suc\ 0]\!]$
 $\Longrightarrow$ (*fetch* (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) *s Bk*) = (*R*, *s* + *2*)
apply(*subgoal-tac* ∃ *q*. *s* = *2*∗*q* + *1*, *auto*)
apply(*subgoal-tac length* (*mop-bef n*) = *4*∗*n*)
apply(*auto simp*: *fetch.simps nth-of.simps nth-append*)
apply(*subgoal-tac mop-bef n* ! (*4* ∗ *q* + *0*) =
 mop-bef (*Suc q*) ! ((*4* ∗ *q* + *0*)),
 simp add: *mop-bef.simps nth-append*)
apply(*rule mop-bef-nth*, *auto*)
done

lemma *fetch-bef-erase-b-b*:
 $[\![n < length\ lm;\ 0 < s;\ s \leq 2 * n;\ s\ mod\ 2 = 0]\!] \Longrightarrow$
 (*fetch* (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) *s Bk*) = (*R*, *s* − *1*)
apply(*subgoal-tac* ∃ *q*. *s* = *2* ∗ *q*, *auto*)
apply(*case-tac qa*, *simp*, *simp*)
apply(*auto simp*: *fetch.simps nth-of.simps nth-append*)
apply(*subgoal-tac mop-bef n* ! (*4* ∗ *nat* + *2*) =
 mop-bef (*Suc nat*) ! ((*4* ∗ *nat*) + *2*),
 simp add: *mop-bef.simps nth-append*)
apply(*rule mop-bef-nth*, *auto*)
done

lemma *fetch-jump-over1-o*:
 fetch (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) (*Suc* (*2* ∗ *n*)) *Oc*
 = (*R*, *Suc* (*2* ∗ *n*))
apply(*subgoal-tac length* (*mop-bef n*) = *4* ∗ *n*)
apply(*auto simp*: *fetch.simps nth-of.simps mp-up-def nth-append*
 tshift.simps)
done

lemma *fetch-jump-over1-b*:
 fetch (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) (*Suc* (*2* ∗ *n*)) *Bk*
 = (*R*, *Suc* (*Suc* (*2* ∗ *n*)))
apply(*subgoal-tac length* (*mop-bef n*) = *4* ∗ *n*)
apply(*auto simp*: *fetch.simps nth-of.simps mp-up-def*
 nth-append tshift.simps)
done

lemma *fetch-aft-erase-a-o*:
 fetch (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) (*Suc* (*Suc* (*2* ∗ *n*))) *Oc*
 = (*W0*, *Suc* (*2* ∗ *n* + *2*))
apply(*subgoal-tac length* (*mop-bef n*) = *4* ∗ *n*)
apply(*auto simp*: *fetch.simps nth-of.simps mp-up-def*
 nth-append tshift.simps)
done

lemma *fetch-aft-erase-a-b*:
fetch (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) (*Suc* (*Suc* (*2* ∗ *n*))) *Bk*
= (*L*, *Suc* (*2* ∗ *n* + *4*))
apply(*subgoal-tac length* (*mop-bef n*) = *4* ∗ *n*)
apply(*auto simp*: *fetch.simps nth-of.simps mp-up-def*
nth-append tshift.simps)
done

lemma *fetch-aft-erase-b-b*:
fetch (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) (*2*∗*n* + *3*) *Bk*
= (*R*, *Suc* (*2* ∗ *n* + *3*))
apply(*subgoal-tac length* (*mop-bef n*) = *4* ∗ *n*)
apply(*subgoal-tac 2*∗*n* + *3* = *Suc* (*2*∗*n* + *2*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps mp-up-def nth-append tshift.simps*)
done

lemma *fetch-aft-erase-c-o*:
fetch (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) (*2* ∗ *n* + *4*) *Oc*
= (*W0*, *Suc* (*2* ∗ *n* + *2*))
apply(*subgoal-tac length* (*mop-bef n*) = *4* ∗ *n*)
apply(*subgoal-tac 2*∗*n* + *4* = *Suc* (*2*∗*n* + *3*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps mp-up-def nth-append tshift.simps*)
done

lemma *fetch-aft-erase-c-b*:
fetch (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) (*2* ∗ *n* + *4*) *Bk*
= (*R*, *Suc* (*2* ∗ *n* + *1*))
apply(*subgoal-tac length* (*mop-bef n*) = *4* ∗ *n*)
apply(*subgoal-tac 2*∗*n* + *4* = *Suc* (*2*∗*n* + *3*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps mp-up-def nth-append tshift.simps*)
done

lemma *fetch-left-moving-o*:
(*fetch* (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) (*2* ∗ *n* + *5*) *Oc*)
= (*L*, *2*∗*n* + *6*)
apply(*subgoal-tac length* (*mop-bef n*) = *4* ∗ *n*)
apply(*subgoal-tac 2*∗*n* + *5* = *Suc* (*2*∗*n* + *4*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps mp-up-def nth-append tshift.simps*)
done

lemma *fetch-left-moving-b*:
(*fetch* (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) (*2* ∗ *n* + *5*) *Bk*)
= (*L*, *2*∗*n* + *5*)
apply(*subgoal-tac length* (*mop-bef n*) = *4* ∗ *n*)
apply(*subgoal-tac 2*∗*n* + *5* = *Suc* (*2*∗*n* + *4*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps mp-up-def nth-append tshift.simps*)
done

lemma *fetch-jump-over2-b*:

(*fetch* (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) (*2* ∗ *n* + *6*) *Bk*)
= (*R*, *0*)
apply(*subgoal-tac length* (*mop-bef n*) = *4* ∗ *n*)
apply(*subgoal-tac 2*∗*n* + *6* = *Suc* (*2*∗*n* + *5*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps mp-up-def nth-append tshift.simps*)
done

lemma *fetch-jump-over2-o*:
(*fetch* (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)) (*2* ∗ *n* + *6*) *Oc*)
= (*L*, *2*∗*n* + *6*)
apply(*subgoal-tac length* (*mop-bef n*) = *4* ∗ *n*)
apply(*subgoal-tac 2*∗*n* + *6* = *Suc* (*2*∗*n* + *5*), *simp only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps mp-up-def nth-append tshift.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

lemma [*simp*]:
⟦*n* < *length lm*; *0* < *s*; *s mod 2* = *Suc 0*;
mopup-bef-erase-a (*s*, *l*, *Oc* # *xs*) *lm n ires*;
Suc s ≤ *2* ∗ *n*⟧ ⟹
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*)
apply(*rule-tac x* = *m* − *1* **in** *exI*, *rule-tac x* = *rn* **in** *exI*)
apply(*case-tac m*, *simp*, *simp*)
done

lemma *mopup-false1*:
⟦*0* < *s*; *s* ≤ *2* ∗ *n*; *s mod 2* = *Suc 0*; ¬ *Suc s* ≤ *2* ∗ *n*⟧
⟹ *RR*
apply(*arith*)
done

lemma [*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 *drop-abc-lm-v-simp*[*simp*]:
n < *length lm* ⟹ *drop n lm* = *abc-lm-v lm n* # *drop* (*Suc n*) *lm*
apply(*auto simp*: *abc-lm-v.simps*)

apply(*drule drop-Suc-conv-tl*, *simp*)
done
lemma [*simp*]: ($\exists$ *rna*. Bk^{rn} = *Bk* # Bk^{rna}) $\vee$ Bk^{rn} = []
apply(*case-tac rn*, *simp*, *auto*)
done

lemma [*simp*]: $\exists$ *lna*. *Bk* # Bk^{ln} = Bk^{lna}
apply(*rule-tac x* = *Suc ln* **in** *exI*, *auto*)
done

lemma *mopup-bef-erase-a-2-jump-over*[*simp*]:
⟦*n* < *length lm*; *0* < *s*; *s mod 2* = *Suc 0*;
mopup-bef-erase-a (*s*, *l*, *Bk* # *xs*) *lm n ires*; *Suc s* = *2* ∗ *n*⟧
⟹ *mopup-jump-over1* (*Suc* (*2* ∗ *n*), *Bk* # *l*, *xs*) *lm n ires*
apply(*auto simp*: *mopup-bef-erase-a.simps mopup-jump-over1.simps*)
apply(*case-tac m*, *simp*)
apply(*rule-tac x* = *Suc ln* **in** *exI*, *rule-tac x* = *0* **in** *exI*,
simp add: *tape-of-nl-abv*)
apply(*case-tac drop* (*Suc n*) *lm*, *auto simp*: *tape-of-nat-list.simps*)
done

lemma *Suc-Suc-div*: ⟦*0* < *s*; *s mod 2* = *Suc 0*; *Suc* (*Suc s*) ≤ *2* ∗ *n*⟧
⟹ (*Suc* (*Suc* (*s div 2*))) ≤ *n*
apply(*arith*)
done

lemma *mopup-bef-erase-a-2-a*[*simp*]:
⟦*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*⟧ ⟹
mopup-bef-erase-a (*Suc* (*Suc s*), *Bk* # *l*, *xs*) *lm n ires*
apply(*auto simp*: *mopup-bef-erase-a.simps*)
apply(*subgoal-tac drop* (*Suc* (*Suc* (*s div 2*))) *lm* ≠ [])
apply(*case-tac m*, *simp*)
apply(*rule-tac x* = *Suc* (*abc-lm-v lm* (*Suc* (*s div 2*))) **in** *exI*,
rule-tac x = *rn* **in** *exI*, *simp*, *simp*)
apply(*subgoal-tac* (*Suc* (*Suc* (*s div 2*))) ≤ *n*, *simp*,
rule-tac Suc-Suc-div, *auto*)
done

lemma *mopup-false2*:
⟦*n* < *length lm*; *0* < *s*; *s* ≤ *2* ∗ *n*;
s mod 2 = *Suc 0*; *Suc s* ≠ *2* ∗ *n*;
¬ *Suc* (*Suc s*) ≤ *2* ∗ *n*⟧ ⟹ *RR*
apply(*arith*)
done

lemma [*simp*]:
⟦*n* < *length lm*; *0* < *s*; *s* ≤ *2* ∗ *n*;

```
  s mod 2 = Suc 0;
  mopup-bef-erase-a (s, l, Bk # xs) lm n ires;
  r = Bk # xs⟧
 ⟹ (Suc s = 2 * n ⟶
          mopup-jump-over1 (Suc (2 * n), Bk # l, xs) lm n ires) ∧
    (Suc s ≠ 2 * n ⟶
     (Suc (Suc s) ≤ 2 * n ⟶
        mopup-bef-erase-a (Suc (Suc s), Bk # l, xs) lm n ires) ∧
     (¬ Suc (Suc s) ≤ 2 * n ⟶
        mopup-aft-erase-a (Suc (Suc s), Bk # l, xs) lm n ires))
apply(auto elim: mopup-false2)
done

lemma [simp]: mopup-bef-erase-a (s, l, []) lm n ires ⟹
                    mopup-bef-erase-a (s, l, [Bk]) lm n ires
apply(auto simp: mopup-bef-erase-a.simps)
done

lemma [simp]:
  ⟦n < length lm; 0 < s; s ≤ 2 * n; s mod 2 = Suc 0;
    mopup-bef-erase-a (s, l, []) lm n ires; r = []⟧
   ⟹ (Suc s = 2 * n ⟶
          mopup-jump-over1 (Suc (2 * n), Bk # l, []) lm n ires) ∧
      (Suc s ≠ 2 * n ⟶
          (Suc (Suc s) ≤ 2 * n ⟶
             mopup-bef-erase-a (Suc (Suc s), Bk # l, []) lm n ires) ∧
          (¬ Suc (Suc s) ≤ 2 * n ⟶
             mopup-aft-erase-a (Suc (Suc s), Bk # l, []) lm n ires))
apply(auto)
done

lemma mopup-bef-erase-b (s, l, Oc # xs) lm n ires ⟹ l ≠ []
apply(auto simp: mopup-bef-erase-b.simps)
done

lemma [simp]: mopup-bef-erase-b (s, l, Oc # xs) lm n ires = False
apply(auto simp: mopup-bef-erase-b.simps )
done

lemma [simp]: ⟦0 < s; s ≤ 2 *n; s mod 2 ≠ Suc 0⟧ ⟹
                              (s − Suc 0) mod 2 = Suc 0
apply(arith)
done

lemma [simp]: ⟦0 < s; s ≤ 2 *n; s mod 2 ≠ Suc 0⟧ ⟹
                               s − Suc 0 ≤ 2 * n
apply(simp)
done
```

```
lemma [simp]: ⟦0 < s; s ≤ 2 *n; s mod 2 ≠ Suc 0⟧ ⟹ ¬ s ≤ Suc 0
apply(arith)
done

lemma [simp]: ⟦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
apply(auto simp: mopup-bef-erase-b.simps mopup-bef-erase-a.simps)
done

lemma [simp]: ⟦mopup-bef-erase-b (s, l, []) lm n ires⟧ ⟹
                 mopup-bef-erase-a (s − Suc 0, Bk # l, []) lm n ires
apply(auto simp: mopup-bef-erase-b.simps mopup-bef-erase-a.simps)
done

lemma [simp]:
  ⟦n < length lm;
   mopup-jump-over1 (Suc (2 * n), l, Oc # xs) lm n ires;
   r = Oc # xs⟧
  ⟹ mopup-jump-over1 (Suc (2 * n), Oc # l, xs) lm n ires
apply(auto simp: 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)
apply(case-tac m2, simp, simp, rule-tac x = rn in exI, simp)
apply(rule-tac x = ln in exI, rule-tac x = Suc m1 in exI,
      rule-tac x = m2 − 1 in exI)
apply(case-tac m2, simp, simp)
done

lemma mopup-jump-over1-2-aft-erase-a[simp]:
 ⟦n < length lm; mopup-jump-over1 (Suc (2 * n), l, Bk # xs) lm n ires⟧
  ⟹ mopup-aft-erase-a (Suc (Suc (2 * n)), Bk # l, xs) lm n ires
apply(simp only: mopup-jump-over1.simps mopup-aft-erase-a.simps)
apply(erule-tac exE)+
apply(rule-tac x = ln in exI, rule-tac x = Suc 0 in exI)
apply(case-tac m2, simp)
apply(rule-tac x = rn in exI, rule-tac x = drop (Suc n) lm in exI,
      simp)
apply(simp)
done

lemma [simp]:
 ⟦n < length lm; mopup-jump-over1 (Suc (2 * n), l, []) lm n ires⟧ ⟹
    mopup-aft-erase-a (Suc (Suc (2 * n)), Bk # l, []) lm n ires
apply(rule mopup-jump-over1-2-aft-erase-a, simp)
apply(auto simp: mopup-jump-over1.simps)
apply(rule-tac x = ln in exI, rule-tac x = m1 in exI,
      rule-tac x = m2 in exI, simp add: )
```

apply(*rule-tac x* = *0* **in** *exI*, *auto*)
done

lemma [*simp*]:
⟦*n* < *length lm*;
mopup-aft-erase-a (*Suc* (*Suc* (*2* ∗ *n*)), *l*, *Oc* # *xs*) *lm n ires*⟧
⟹ *mopup-aft-erase-b* (*Suc* (*Suc* (*Suc* (*2* ∗ *n*))), *l*, *Bk* # *xs*) *lm n ires*
apply(*auto simp*: *mopup-aft-erase-a.simps mopup-aft-erase-b.simps*)
apply(*case-tac ml*, *simp*, *case-tac rn*, *simp*, *simp*)
apply(*case-tac list*, *auto simp*: *tape-of-nl-abv*
tape-of-nat-list.simps)
apply(*case-tac a*, *simp*, *rule-tac x* = *rn* **in** *exI*,
rule-tac x = [] **in** *exI*,
simp add: *tape-of-nat-list.simps*, *simp*)
apply(*rule-tac x* = *rn* **in** *exI*, *rule-tac x* = [*nat*] **in** *exI*,
simp add: *tape-of-nat-list.simps*)
apply(*case-tac a*, *simp*, *rule-tac x* = *rn* **in** *exI*,
rule-tac x = *aa* # *lista* **in** *exI*, *simp*, *simp*)
apply(*rule-tac x* = *rn* **in** *exI*, *rule-tac x* = *nat* # *aa* # *lista* **in** *exI*,
simp add: *tape-of-nat-list.simps*)
done

lemma [*simp*]:
mopup-aft-erase-a (*Suc* (*Suc* (*2* ∗ *n*)), *l*, *Bk* # *xs*) *lm n ires* ⟹ *l* ≠ []
apply(*auto simp*: *mopup-aft-erase-a.simps*)
done

lemma [*simp*]:
⟦*n* < *length lm*;
mopup-aft-erase-a (*Suc* (*Suc* (*2* ∗ *n*)), *l*, *Bk* # *xs*) *lm n ires*⟧
⟹ *mopup-left-moving* (*5* + *2* ∗ *n*, *tl l*, *hd l* # *Bk* # *xs*) *lm n ires*
apply(*simp only*: *mopup-aft-erase-a.simps mopup-left-moving.simps*)
apply(*erule exE*)+
apply(*case-tac lnr*, *simp*)
apply(*rule-tac x* = *lnl* **in** *exI*, *simp*, *rule-tac x* = *rn* **in** *exI*, *simp*)
apply(*subgoal-tac ml* = [], *simp*)
apply(*rule-tac xs* = *xs* **and** *rn* = *rn* **in** *BkCons-nil*, *simp*, *auto*)
apply(*subgoal-tac ml* = [], *auto*)
apply(*rule-tac xs* = *xs* **and** *rn* = *rn* **in** *BkCons-nil*, *simp*)
done

lemma [*simp*]:
mopup-aft-erase-a (*Suc* (*Suc* (*2* ∗ *n*)), *l*, []) *lm n ires* ⟹ *l* ≠ []
apply(*simp only*: *mopup-aft-erase-a.simps*)
apply(*erule exE*)+
apply(*auto*)
done

lemma [*simp*]:

⟦n < *length lm*; *mopup-aft-erase-a* (*Suc* (*Suc* (*2* ∗ *n*)), *l*, []) *lm n ires*⟧
⟹ *mopup-left-moving* (*5* + *2* ∗ *n*, *tl l*, [*hd l*]) *lm n ires*
apply(*simp only*: *mopup-aft-erase-a.simps mopup-left-moving.simps*)
apply(*erule exE*)+
apply(*subgoal-tac ml* = [] ∧ *rn* = *0*, *erule conjE*, *erule conjE*, *simp*)
apply(*case-tac lnr*, *simp*, *rule-tac x* = *lnl* **in** *exI*, *simp*,
rule-tac x = *0* **in** *exI*, *simp*)
apply(*rule-tac x* = *lnl* **in** *exI*, *rule-tac x* = *nat* **in** *exI*,
rule-tac x = *Suc 0* **in** *exI*, *simp*)
apply(*case-tac ml*, *simp*, *case-tac rn*, *simp*, *simp*)
apply(*case-tac list*, *auto simp*: *tape-of-nl-abv tape-of-nat-list.simps*)
done

lemma [*simp*]: *mopup-aft-erase-b* (*2* ∗ *n* + *3*, *l*, *Oc* # *xs*) *lm n ires* = *False*
apply(*auto simp*: *mopup-aft-erase-b.simps*)
done

lemma [*simp*]:
⟦*n* < *length lm*;
mopup-aft-erase-c (*2* ∗ *n* + *4*, *l*, *Oc* # *xs*) *lm n ires*⟧
⟹ *mopup-aft-erase-b* (*Suc* (*Suc* (*Suc* (*2* ∗ *n*))), *l*, *Bk* # *xs*) *lm n ires*
apply(*auto simp*: *mopup-aft-erase-c.simps mopup-aft-erase-b.simps*)
apply(*case-tac ml*, *simp*, *case-tac rn*, *simp*, *simp*, *simp*)
apply(*case-tac list*, *auto simp*: *tape-of-nl-abv*
tape-of-nat-list.simps tape-of-nat-abv)
apply(*case-tac a*, *rule-tac x* = *rn* **in** *exI*,
rule-tac x = [] **in** *exI*, *simp add*: *tape-of-nat-list.simps*)
apply(*rule-tac x* = *rn* **in** *exI*, *rule-tac x* = [*nat*] **in** *exI*,
simp add: *tape-of-nat-list.simps*)
apply(*case-tac a*, *simp*, *rule-tac x* = *rn* **in** *exI*,
rule-tac x = *aa* # *lista* **in** *exI*, *simp*)
apply(*rule-tac x* = *rn* **in** *exI*, *rule-tac x* = *nat* # *aa* # *lista* **in** *exI*,
simp add: *tape-of-nat-list.simps*)
done

lemma *mopup-aft-erase-c-aft-erase-a*[*simp*]:
⟦*n* < *length lm*; *mopup-aft-erase-c* (*2* ∗ *n* + *4*, *l*, *Bk* # *xs*) *lm n ires*⟧
⟹ *mopup-aft-erase-a* (*Suc* (*Suc* (*2* ∗ *n*)), *Bk* # *l*, *xs*) *lm n ires*
apply(*simp only*: *mopup-aft-erase-c.simps mopup-aft-erase-a.simps*)
apply(*erule-tac exE*)+
apply(*erule conjE*, *erule conjE*, *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(*rule-tac x* = *nat* **in** *exI*, *rule-tac x* = [] **in** *exI*, *simp*)
apply(*rule-tac xs* = *xs* **and** *rn* = *rn* **in** *BkCons-nil*, *simp*, *simp*,
rule conjI)
apply(*rule-tac x* = *lnl* **in** *exI*, *rule-tac x* = *Suc lnr* **in** *exI*, *simp*)
apply(*rule-tac x* = *rn* **in** *exI*, *rule-tac x* = *ml* **in** *exI*, *simp*)

done

lemma [*simp*]:
⟦*n* < *length lm*; *mopup-aft-erase-c* (*2* ∗ *n* + *4*, *l*, []) *lm n ires*⟧
⟹ *mopup-aft-erase-a* (*Suc* (*Suc* (*2* ∗ *n*)), *Bk* # *l*, []) *lm n ires*
apply(*rule mopup-aft-erase-c-aft-erase-a*, *simp*)
apply(*simp only*: *mopup-aft-erase-c.simps*)
apply(*erule exE*)+
apply(*rule-tac x* = *lnl* **in** *exI*, *rule-tac x* = *lnr* **in** *exI*, *simp add*:)
apply(*rule-tac x* = *0* **in** *exI*, *rule-tac x* = [] **in** *exI*, *simp*)
done

lemma *mopup-aft-erase-b-2-aft-erase-c*[*simp*]:
⟦*n* < *length lm*; *mopup-aft-erase-b* (*2* ∗ *n* + *3*, *l*, *Bk* # *xs*) *lm n ires*⟧
⟹ *mopup-aft-erase-c* (*4* + *2* ∗ *n*, *Bk* # *l*, *xs*) *lm n ires*
apply(*auto simp*: *mopup-aft-erase-b.simps mopup-aft-erase-c.simps*)
apply(*rule-tac x* = *lnl* **in** *exI*, *rule-tac x* = *Suc lnr* **in** *exI*, *simp*)
apply(*rule-tac x* = *lnl* **in** *exI*, *rule-tac x* = *Suc lnr* **in** *exI*, *simp*)
done

lemma [*simp*]:
⟦*n* < *length lm*; *mopup-aft-erase-b* (*2* ∗ *n* + *3*, *l*, []) *lm n ires*⟧
⟹ *mopup-aft-erase-c* (*4* + *2* ∗ *n*, *Bk* # *l*, []) *lm n ires*
apply(*rule-tac mopup-aft-erase-b-2-aft-erase-c*, *simp*)
apply(*simp add*: *mopup-aft-erase-b.simps*)
done

lemma [*simp*]:
mopup-left-moving (*2* ∗ *n* + *5*, *l*, *Oc* # *xs*) *lm n ires* ⟹ *l* ≠ []
apply(*auto simp*: *mopup-left-moving.simps*)
done

lemma [*simp*]:
⟦*n* < *length lm*; *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(*case-tac rn*, *simp*, *simp add*:)
apply(*case-tac hd l*, *simp add*:)
apply(*case-tac abc-lm-v lm n*, *simp*)
apply(*rule-tac x* = *lnl* **in** *exI*, *rule-tac x* = *rn* **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*], *simp*)
apply(*case-tac abc-lm-v lm n*, *simp*)
apply(*case-tac lnl*, *simp*, *simp*)
apply(*rule-tac x* = *lnl* **in** *exI*, *rule-tac x* = *rn* **in** *exI*)
apply(*rule-tac x* = *nat* **in** *exI*, *rule-tac x* = *Suc* (*Suc 0*) **in** *exI*, *simp*)
done

lemma [*simp*]: *mopup-left-moving* (*2* ∗ *n* + *5*, *l*, *xs*) *lm n ires* ⟹ *l* ≠ []
apply(*auto simp*: *mopup-left-moving.simps*)
done

lemma [*simp*]:
⟦*n* < *length lm*; *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(*case-tac lnr*, *simp*)
apply(*rule-tac x* = *lnl* **in** *exI*, *rule-tac x* = *0* **in** *exI*,
rule-tac x = *rn* **in** *exI*, *simp*, *simp*)
apply(*rule-tac x* = *lnl* **in** *exI*, *rule-tac x* = *nat* **in** *exI*, *simp*)
done

lemma [*simp*]:
⟦*n* < *length lm*; *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(*case-tac lnr*, *simp*)
apply(*rule-tac x* = *lnl* **in** *exI*, *rule-tac x* = *0* **in** *exI*,
rule-tac x = *0* **in** *exI*, *simp*, *auto*)
done

lemma [*simp*]:
mopup-jump-over2 (*2* ∗ *n* + *6*, *l*, *Oc* # *xs*) *lm n ires* ⟹ *l* ≠ []
apply(*auto simp*: *mopup-jump-over2.simps*)
done

lemma [*intro*]: ∃ *lna*. *Bk* # Bk^{ln} = Bk^{lna} @ [*Bk*]
apply(*simp only*: *exp-ind*[*THEN sym*], *auto*)
done

lemma [*simp*]:
⟦*n* < *length lm*; *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 add*: , *erule conjE*, *erule-tac conjE*)
apply(*case-tac m1*, *simp*)
apply(*rule-tac x* = *ln* **in** *exI*, *rule-tac x* = *rn* **in** *exI*,
rule-tac x = *0* **in** *exI*, *simp*)
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 = *nat* **in** *exI*, *rule-tac x* = *Suc m2* **in** *exI*, *simp*)
done

lemma [*simp*]: ∃ *rna*. *Oc* # Oc^{a} @ Bk^{rn} = <*a*> @ Bk^{rna}
apply(*case-tac a*, *auto simp*: *tape-of-nat-abv*)
done

lemma [*simp*]:
⟦*n* < *length lm*; *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*)
done

lemma [*simp*]: *mopup-jump-over2* (*2* ∗ *n* + *6*, *l*, []) *lm n ires* = *False*
apply(*simp only*: *mopup-jump-over2.simps*, *simp*)
done

lemma *mopup-inv-step*:
⟦*n* < *length lm*; *mopup-inv* (*s*, *l*, *r*) *lm n ires*⟧
⟹ *mopup-inv* (*t-step* (*s*, *l*, *r*)
((*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)), *0*)) *lm n ires*
apply(*auto split*:*if-splits simp add*:*t-step.simps*,
tactic ⟪ *ALLGOALS* (*resolve-tac* [@{*thm fetch-intro*}]) ⟫)
apply(*simp-all add*: *mopupfetchs new-tape.simps*)
done

declare *mopup-inv.simps*[*simp del*]

lemma *mopup-inv-steps*:
⟦*n* < *length lm*; *mopup-inv* (*s*, *l*, *r*) *lm n ires*⟧ ⟹
mopup-inv (*t-steps* (*s*, *l*, *r*)
((*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)), *0*) *stp*) *lm n ires*
apply(*induct stp*, *simp add*: *t-steps.simps*)
apply(*simp add*: *t-steps-ind*)
apply(*case-tac* (*t-steps* (*s*, *l*, *r*)
(*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*) *stp*), *simp*)
apply(*rule-tac mopup-inv-step*, *simp*, *simp*)
done

lemma [*simp*]:
⟦*n* < *length lm*; *Suc 0* ≤ *n*⟧ ⟹
mopup-bef-erase-a (*Suc 0*, Bk^{ln} @ *Bk* # *Bk* # *ires*, <*lm*> @ Bk^{rn}) *lm*
n ires
apply(*auto simp*: *mopup-bef-erase-a.simps abc-lm-v.simps*)
apply(*case-tac lm*, *simp*, *case-tac list*, *simp*, *simp*)
apply(*rule-tac x* = *Suc a* **in** *exI*, *rule-tac x* = *rn* **in** *exI*, *simp*)
done

lemma [*simp*]:
lm ≠ [] ⟹ *mopup-jump-over1* (*Suc 0*, Bk^{ln} @ *Bk* # *Bk* # *ires*, <*lm*> @ Bk^{rn})
lm 0 ires
apply(*auto simp*: *mopup-jump-over1.simps*)

```
apply(rule-tac x = ln in exI, rule-tac x = 0 in exI, simp add: )
apply(case-tac lm, simp, simp add: abc-lm-v.simps)
apply(case-tac rn, simp)
apply(case-tac list, rule-tac disjI2,
      simp add: tape-of-nl-abv tape-of-nat-list.simps)
apply(rule-tac disjI1,
      simp add: tape-of-nl-abv tape-of-nat-list.simps )
apply(rule-tac disjI1, case-tac list,
      simp add: tape-of-nl-abv tape-of-nat-list.simps,
      rule-tac x = nat in exI, simp)
apply(simp add: tape-of-nl-abv tape-of-nat-list.simps )
done

lemma mopup-init:
 ⟦n < length lm; crsp-l ly (as, lm) (ac, l, r) ires⟧ ⟹
                              mopup-inv (Suc 0, l, r) lm n ires
apply(auto simp: crsp-l.simps mopup-inv.simps)
apply(case-tac n, simp, auto simp: mopup-bef-erase-a.simps )
apply(rule-tac x = Suc (hd lm) in exI, rule-tac x = rn in exI, simp)
apply(case-tac lm, simp, case-tac list, simp, case-tac lista, simp add: abc-lm-v.simps)
apply(simp add: tape-of-nl-abv tape-of-nat-list.simps abc-lm-v.simps)
apply(simp add: mopup-jump-over1.simps)
apply(rule-tac x = 0 in exI, rule-tac x = 0 in exI, auto)
apply(case-tac [!] n, simp-all)
apply(case-tac [!] lm, simp, case-tac list, simp)
apply(case-tac rn, simp add: tape-of-nl-abv tape-of-nat-list.simps abc-lm-v.simps)
apply(erule-tac x = nat in allE, simp add: tape-of-nl-abv tape-of-nat-list.simps
abc-lm-v.simps)
apply(simp add: abc-lm-v.simps, auto)
apply(case-tac list, simp-all add: tape-of-nl-abv tape-of-nat-list.simps abc-lm-v.simps)

apply(erule-tac x = rn in allE, simp-all)
done

fun abc-mopup-stage1 :: t-conf ⇒ 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 :: t-conf ⇒ 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* :: *t-conf* ⇒ *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)

fun *abc-mopup-measure* :: (*t-conf* × *nat*) ⇒ (*nat* × *nat* × *nat*)
where
abc-mopup-measure (*c*, *n*) =
(*abc-mopup-stage1 c n*, *abc-mopup-stage2 c n*,
abc-mopup-stage3 c n)

definition *abc-mopup-LE* ::
(((*nat* × *block list* × *block list*) × *nat*) ×
((*nat* × *block list* × *block list*) × *nat*)) *set*
where
abc-mopup-LE ≡ (*inv-image lex-triple abc-mopup-measure*)

lemma *wf-abc-mopup-le*[*intro*]: *wf abc-mopup-LE*
by(*auto intro*:*wf-inv-image wf-lex-triple simp*:*abc-mopup-LE-def*)

lemma [*simp*]: *mopup-bef-erase-a* (*a*, *aa*, []) *lm n ires* = *False*
apply(*auto simp*: *mopup-bef-erase-a.simps*)
done

lemma [*simp*]: *mopup-bef-erase-b* (*a*, *aa*, []) *lm n ires* = *False*
apply(*auto simp*: *mopup-bef-erase-b.simps*)
done

lemma [*simp*]: *mopup-aft-erase-b* (*2* ∗ *n* + *3*, *aa*, []) *lm n ires* = *False*
apply(*auto simp*: *mopup-aft-erase-b.simps*)
done

lemma *mopup-halt-pre*:
⟦*n* < *length lm*; *mopup-inv* (*Suc 0*, *l*, *r*) *lm n ires*; *wf abc-mopup-LE*⟧
⟹ ∀ *na*. ¬ (λ(*s*, *l*, *r*) *n*. *s* = *0*) (*t-steps* (*Suc 0*, *l*, *r*)
(*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*) *na*) *n* ⟶
((*t-steps* (*Suc 0*, *l*, *r*) (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*)
(*Suc na*), *n*),

t-steps (*Suc 0*, *l*, *r*) (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*)
na, *n*) ∈ *abc-mopup-LE*
apply(*rule allI*, *rule impI*, *simp add*: *t-steps-ind*)
apply(*subgoal-tac mopup-inv* (*t-steps* (*Suc 0*, *l*, *r*)
(*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*) *na*) *lm n ires*)
apply(*case-tac* (*t-steps* (*Suc 0*, *l*, *r*)
(*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*) *na*), *simp*)
proof −
fix *na a b c*
assume *n* < *length lm mopup-inv* (*a*, *b*, *c*) *lm n ires 0* < *a*
thus ((*t-step* (*a*, *b*, *c*) (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*), *n*),
(*a*, *b*, *c*), *n*) ∈ *abc-mopup-LE*
apply(*auto split*:*if-splits simp add*:*t-step.simps mopup-inv.simps*,
tactic ⟪ *ALLGOALS* (*resolve-tac* [@{*thm fetch-intro*}]) ⟫)
apply(*simp-all add*: *mopupfetchs new-tape.simps abc-mopup-LE-def*
lex-triple-def lex-pair-def)
done
next
fix *na*
assume *n* < *length lm mopup-inv* (*Suc 0*, *l*, *r*) *lm n ires*
thus *mopup-inv* (*t-steps* (*Suc 0*, *l*, *r*)
(*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*) *na*) *lm n ires*
apply(*rule mopup-inv-steps*)
done
qed

lemma *mopup-halt*: ⟦*n* < *length lm*; *crsp-l ly* (*as*, *lm*) (*s*, *l*, *r*) *ires*⟧ =⇒
∃ *stp*. (λ (*s*, *l*, *r*). *s* = *0*) (*t-steps* (*Suc 0*, *l*, *r*)
((*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)), *0*) *stp*)
apply(*subgoal-tac mopup-inv* (*Suc 0*, *l*, *r*) *lm n ires*)
apply(*insert wf-abc-mopup-le*)
apply(*insert halt-lemma*[*of abc-mopup-LE*
λ ((*s*, *l*, *r*), *n*). *s* = *0*
λ *stp*. (*t-steps* (*Suc 0*, *l*, *r*) ((*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*))
, *0*) *stp*, *n*)], *auto*)
apply(*insert mopup-halt-pre*[*of n lm l r*], *simp*, *erule exE*)
apply(*rule-tac x* = *na* **in** *exI*, *case-tac* (*t-steps* (*Suc 0*, *l*, *r*)
(*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*) *na*), *simp*)
apply(*rule-tac mopup-init*, *auto*)
done

lemma *mopup-halt-conf-pre*:
⟦*n* < *length lm*; *crsp-l ly* (*as*, *lm*) (*s*, *l*, *r*) *ires*⟧
=⇒ ∃ *na*. (λ (*s'*, *l'*, *r'*). *s'* = *0* ∧ *mopup-stop* (*s'*, *l'*, *r'*) *lm n ires*)
(*t-steps* (*Suc 0*, *l*, *r*)
((*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)), *0*) *na*)
apply(*subgoal-tac* ∃ *stp*. (λ (*s*, *l*, *r*). *s* = *0*)
(*t-steps* (*Suc 0*, *l*, *r*) ((*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)), *0*) *stp*),

erule exE)
apply(*rule-tac x = stp* **in** *exI*,
case-tac (*t-steps* (*Suc 0*, *l*, *r*)
(*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*) *stp*), *simp*)
apply(*subgoal-tac mopup-inv* (*Suc 0*, *l*, *r*) *lm n ires*)
apply(*subgoal-tac mopup-inv* (*t-steps* (*Suc 0*, *l*, *r*)
(*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*) *stp*) *lm n ires*, *simp*)
apply(*simp only*: *mopup-inv.simps*)
apply(*rule-tac mopup-inv-steps*, *simp*, *simp*)
apply(*rule-tac mopup-init*, *simp*, *simp*)
apply(*rule-tac mopup-halt*, *simp*, *simp*)
done

lemma *mopup-halt-conf*:
assumes *len*: $n < length\ lm$
and *correspond*: *crsp-l ly* (*as*, *lm*) (*s*, *l*, *r*) *ires*
shows
$\exists\ na.\ (\lambda\ (s', l', r').\ ((\exists\ ln\ rn.\ s' = 0 \wedge l' = Bk^{ln}\ @\ Bk\ \#\ Bk\ \#\ ires$
$\wedge\ r' = Oc^{Suc\ (abc\text{-}lm\text{-}v\ lm\ n)}\ @\ Bk^{rn})))$
(*t-steps* (*Suc 0*, *l*, *r*)
((*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*)), *0*) *na*)
using *len correspond mopup-halt-conf-pre*[*of n lm ly as s l r ires*]
apply(*simp add*: *mopup-stop.simps tape-of-nat-abv tape-of-nat-list.simps*)
done

8.6 Final results about Abacus machine

lemma *mopup-halt-bef*: ⟦$n < length\ lm$; *crsp-l ly* (*as*, *lm*) (*s*, *l*, *r*) *ires*⟧
$\Longrightarrow \exists\ stp.\ (\lambda(s, l, r).\ s \neq 0 \wedge ((\lambda\ (s', l', r').\ s' = 0)$
(*t-step* (*s*, *l*, *r*) (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*))))
(*t-steps* (*Suc 0*, *l*, *r*) (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*) *stp*)
apply(*insert mopup-halt*[*of n lm ly as s l r ires*], *simp*, *erule-tac exE*)
proof −
fix *stp*
assume $n < length\ lm$
crsp-l ly (*as*, *lm*) (*s*, *l*, *r*) *ires*
$(\lambda(s, l, r).\ s = 0)$
(*t-steps* (*Suc 0*, *l*, *r*)
(*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*) *stp*)
thus $\exists\ stp.\ (\lambda(s, ab).\ 0 < s \wedge (\lambda(s', l', r').\ s' = 0)$
(*t-step* (*s*, *ab*) (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*)))
(*t-steps* (*Suc 0*, *l*, *r*) (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*) *stp*)
proof(*induct stp*, *simp add*: *t-steps.simps*, *simp*)
fix *stpa*
assume *h1*:
$(\lambda(s, l, r).\ s = 0)$ (*t-steps* (*Suc 0*, *l*, *r*)
(*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*) *stpa*) ⟹
$\exists\ stp.\ (\lambda(s, ab).\ 0 < s \wedge (\lambda(s', l', r').\ s' = 0)$
(*t-step* (*s*, *ab*) (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*), *0*)))

```
          (t-steps (Suc 0, l, r)
            (mop-bef n @ tshift mp-up (2 ∗ n), 0) stp)
      and h2:
        (λ(s, l, r). s = 0) (t-steps (Suc 0, l, r)
                  (mop-bef n @ tshift mp-up (2 ∗ n), 0) (Suc stpa))
        n < length lm
        crsp-l ly (as, lm) (s, l, r) ires
    thus ∃ stp. (λ(s, ab). 0 < s ∧ (λ(s′, l′, r′). s′ = 0)
          (t-step (s, ab) (mop-bef n @ tshift mp-up (2 ∗ n), 0))) (
            t-steps (Suc 0, l, r)
              (mop-bef n @ tshift mp-up (2 ∗ n), 0) stp)
      apply(case-tac (λ(s, l, r). s = 0) (t-steps (Suc 0, l, r)
                  (mop-bef n @ tshift mp-up (2 ∗ n), 0) stpa),
          simp)
      apply(rule-tac x = stpa in exI)
      apply(case-tac (t-steps (Suc 0, l, r)
                      (mop-bef n @ tshift mp-up (2 ∗ n), 0) stpa),
          simp add: t-steps-ind)
      done
  qed
qed

lemma tshift-nth-state0: ⟦n < length tp; tp ! n = (a, 0)⟧
    ⟹ tshift tp off ! n = (a, 0)
apply(induct n, case-tac tp, simp)
apply(auto simp: tshift.simps)
done

lemma shift-length: length (tshift tp n) = length tp
apply(auto simp: tshift.simps)
done

lemma even-Suc-le: ⟦y mod 2 = 0; 2 ∗ x < y⟧ ⟹ Suc (2 ∗ x) < y
by arith

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

lemma tm-append-fetch-equal:
  ⟦t-ncorrect (tm-of aprog); s′> 0;
    fetch (mop-bef n @ tshift mp-up (2 ∗ n)) s′ b = (a, 0)⟧
⟹ fetch (tm-of aprog @ tshift (mop-bef n @ tshift mp-up (2 ∗ n))
    (length (tm-of aprog) div 2)) (s′ + length (tm-of aprog) div 2) b
  = (a, 0)
apply(case-tac s′, simp)
apply(auto simp: fetch.simps nth-of.simps t-ncorrect.simps shift-length nth-append
                tshift.simps split: list.splits block.splits split: if-splits)
done
```

```
lemma [simp]:
  ⟦t-ncorrect (tm-of aprog);
    t-step (s', l', r') (mop-bef n @ tshift mp-up (2 * n), 0) =
                                        (0, l'', r''); s' > 0⟧
  ⟹ t-step (s' + length (tm-of aprog) div 2, l', r')
        (tm-of aprog @ tshift (mop-bef n @ tshift mp-up (2 * n))
          (length (tm-of aprog) div 2), 0) = (0, l'', r'')
apply(simp add: t-step.simps)
apply(subgoal-tac
   (fetch (mop-bef n @ tshift mp-up (2 * n)) s'
              (case r' of [] ⇒ Bk | Bk # xs ⇒ Bk | Oc # xs ⇒ Oc))
  = (fetch (tm-of aprog @ tshift (mop-bef n @ tshift mp-up (2 * n))
       (length (tm-of aprog) div 2)) (s' + length (tm-of aprog) div 2)
    (case r' of [] ⇒ Bk | Bk # xs ⇒ Bk | Oc # xs ⇒ Oc)), simp)
apply(case-tac (fetch (mop-bef n @ tshift mp-up (2 * n)) s'
       (case r' of [] ⇒ Bk | Bk # xs ⇒ Bk | Oc # xs ⇒ Oc)), simp)
apply(drule-tac tm-append-fetch-equal, auto)
done

lemma [intro]:
  start-of (layout-of aprog) (length aprog) − Suc 0 =
                                   length (tm-of aprog) div 2
apply(subgoal-tac  abc2t-correct aprog)
apply(insert pre-lheq[of concat (take (length aprog)
       (tms-of aprog)) length aprog aprog], simp add: tm-of.simps)
by auto

lemma tm-append-stop-step:
  ⟦t-ncorrect (tm-of aprog);
    t-ncorrect (mop-bef n @ tshift mp-up (2 * n)); n < length lm;
   (t-steps (Suc 0, l, r) (mop-bef n @ tshift mp-up (2 * n), 0) stp) =
                       (s', l', r');
    s' ≠ 0;
    t-step (s', l', r') (mop-bef n @ tshift mp-up (2 * n), 0)
                                            = (0, l'', r'')⟧
     ⟹
(t-steps ((start-of (layout-of aprog) (length aprog), l, r))
  (tm-of aprog @ tshift (mop-bef n @ tshift mp-up (2 * n))
   (start-of (layout-of aprog) (length aprog) − Suc 0), 0) (Suc stp))
  = (0, l'', r'')
apply(insert tm-append-steps-equal[of tm-of aprog
      (mop-bef n @ tshift mp-up (2 * n))
      (start-of (layout-of aprog) (length aprog) − Suc 0)
      Suc 0 l r stp], simp)
apply(subgoal-tac (start-of (layout-of aprog) (length aprog) − Suc 0)
            = (length (tm-of aprog) div 2), simp add: t-steps-ind)
apply(case-tac
 (t-steps (start-of (layout-of aprog) (length aprog), l, r)
      (tm-of aprog @ tshift (mop-bef n @ tshift mp-up (2 * n))
```

```
        (length (tm-of aprog) div 2), 0) stp), simp)
apply(subgoal-tac start-of (layout-of aprog) (length aprog) > 0,
      case-tac start-of (layout-of aprog) (length aprog),
      simp, simp)
apply(rule startof-not0, auto)
done

lemma start-of-out-range:
as ≥ length aprog ⟹
  start-of (layout-of aprog) as =
          start-of (layout-of aprog) (length aprog)
apply(induct as, simp)
apply(case-tac length aprog = Suc as, simp)
apply(simp add: start-of.simps)
done

lemma [intro]: t-ncorrect (tm-of aprog)
apply(simp add: tm-of.simps)
apply(insert tms-mod2[of length aprog aprog],
                              simp add: t-ncorrect.simps)
done

lemma abacus-turing-eq-halt-case-pre:
  ⟦ly = layout-of aprog;
    tprog = tm-of aprog;
    crsp-l ly ac tc ires;
    n < length am;
    abc-steps-l ac aprog stp = (as, am);
    mop-ss = start-of ly (length aprog);
    as ≥ length aprog⟧
    ⟹ ∃ stp. (λ (s, l, r). s = 0 ∧ mopup-inv (0, l, r) am n ires)
            (t-steps tc (tprog @ (tMp n (mop-ss − 1)), 0) stp)
apply(insert steps-crsp[of ly aprog tprog ac tc ires (as, am) stp], auto)
apply(case-tac (t-steps tc (tm-of aprog, 0) n'),
      simp add: tMp.simps)
apply(subgoal-tac t-ncorrect (mop-bef n @ tshift mp-up (2 ∗ n)))
proof −
  fix n' a b c
  assume h1:
    crsp-l (layout-of aprog) ac tc ires
    abc-steps-l ac aprog stp = (as, am)
    length aprog ≤ as
    crsp-l (layout-of aprog) (as, am) (a, b, c) ires
    t-steps tc (tm-of aprog, 0) n' = (a, b, c)
    n < length am
    t-ncorrect (mop-bef n @ tshift mp-up (2 ∗ n))
  hence h2:
  t-steps tc (tm-of aprog @ tshift (mop-bef n @ tshift mp-up (2 ∗ n))
    (start-of (layout-of aprog) (length aprog) − Suc 0), 0) n'
```

```
                        = (a, b, c)
   apply(rule-tac tm-append-steps, simp)
   apply(simp add: crsp-l.simps, auto)
   apply(simp add: crsp-l.simps)
   apply(rule startof-not0)
   done
 from h1 have h3:
 ∃ stp. (λ(s, l, r). s ≠ 0 ∧ ((λ (s′, l′, r′). s′ = 0)
       (t-step (s, l, r) (mop-bef n @ tshift mp-up (2 ∗ n), 0))))
     (t-steps (Suc 0, b, c)
          (mop-bef n @ tshift mp-up (2 ∗ n), 0) stp)
   apply(rule-tac mopup-halt-bef, auto)
   done
 from h1 and h2 and h3 show
   ∃ stp. case t-steps tc (tm-of aprog @ abacus.tshift (mop-bef n @ abacus.tshift
mp-up (2 ∗ n))
   (start-of (layout-of aprog) (length aprog) − Suc 0), 0) stp of (s, ab)
   ⇒ s = 0 ∧ mopup-inv (0, ab) am n ires
 proof(erule-tac exE,
   case-tac (t-steps (Suc 0, b, c)
          (mop-bef n @ tshift mp-up (2 ∗ n), 0) stpa), simp,
   case-tac (t-step (aa, ba, ca)
          (mop-bef n @ tshift mp-up (2 ∗ n), 0)), simp)
   fix stpa aa ba ca aaa baa caa
   assume g1: 0 < aa ∧ aaa = 0
     t-steps (Suc 0, b, c)
     (mop-bef n @ tshift mp-up (2 ∗ n), 0) stpa = (aa, ba,ca)
     t-step (aa, ba, ca) (mop-bef n @ tshift mp-up (2 ∗ n), 0)
     = (0, baa, caa)
   from h1 and this have g2:
     t-steps (start-of (layout-of aprog) (length aprog), b, c)
       (tm-of aprog @ tshift (mop-bef n @ tshift mp-up (2 ∗ n))
         (start-of (layout-of aprog) (length aprog) − Suc 0), 0)
            (Suc stpa) = (0, baa, caa)
     apply(rule-tac tm-append-stop-step, auto)
     done
   from h1 and h2 and g1 and this show ?thesis
     apply(rule-tac x = n′ + Suc stpa in exI)
     apply(simp add: t-steps-ind del: t-steps.simps)
     apply(subgoal-tac a = start-of (layout-of aprog)
                                 (length aprog), simp)
     apply(insert mopup-inv-steps[of n am Suc 0 b c ires Suc stpa],
          simp add: t-steps-ind)
     apply(subgoal-tac mopup-inv (Suc 0, b, c) am n ires, simp)
     apply(rule-tac mopup-init, simp, simp)
     apply(simp add: crsp-l.simps)
     apply(erule-tac start-of-out-range)
     done
 qed
```

next
  show *t-ncorrect* (*mop-bef n* @ *tshift mp-up* (*2* ∗ *n*))
    apply(*auto simp*: *t-ncorrect.simps tshift.simps mp-up-def*)
    done
qed

One of the main theorems about Abacus compilation.

lemma *abacus-turing-eq-halt-case*:
  assumes *layout*:
  — There is an Abacus program *aprog* with layout *ly*:
  ly = *layout-of aprog*
  and *complied*:
  — The TM compiled from *aprog* is *tprog*:
  tprog = *tm-of aprog*
  and *correspond*:
  — TM configuration *tc* and Abacus configuration *ac* are in correspondence:
  crsp-l ly ac tc ires
  and *halt-state*:
  — *as* is a program label outside the range of *aprog*. So if Abacus is in such a state, it is in halt state:
  as ≥ *length aprog*
  and *abc-exec*:
  — Supposing after *stp* step of execution, Abacus program *aprog* reaches such a halt state:
  abc-steps-l ac aprog stp = (*as*, *am*)
  and *rs-len*:
  — *n* is a memory address in the range of Abacus memory *am*:
  n < *length am*
  and *mopup-start*:
  — The startling label for mopup mahines, according to the layout and Abacus program should be *mop-ss*:
  mop-ss = *start-of ly* (*length aprog*)
  shows
  — After *stp* steps of execution of the TM composed of *tprog* and the mopup TM (*tMp n* (*mop-ss* − *1*)) will halt and gives rise to a configuration which only hold the content of memory cell *n*:
  ∃ *stp*. (λ (*s*, *l*, *r*). ∃ *ln rn*. *s* = *0* ∧ *l* = Bk^{ln} @ *Bk* # *Bk* # *ires*
    ∧ *r* = $Oc^{Suc\ (abc\text{-}lm\text{-}v\ am\ n)}$ @ Bk^{rn})
      (*t-steps tc* (*tprog* @ (*tMp n* (*mop-ss* − *1*)), *0*) *stp*)
proof −
  from *layout complied correspond halt-state abc-exec rs-len mopup-start*
    and *abacus-turing-eq-halt-case-pre* [*of ly aprog tprog ac tc ires n am stp as mop-ss*]
  show *?thesis*
    apply(*simp add*: *mopup-inv.simps mopup-stop.simps tape-of-nat-abv*)
    done
qed

lemma *abc-unhalt-case-zero*:

```
⟦crsp-l (layout-of aprog) ac tc ires;
  n < length am;
  ∀ stp. (λ(as, am). as < length aprog) (abc-steps-l ac aprog stp)⟧
 ⟹ ∃ astp bstp. 0 ≤ bstp ∧
          crsp-l (layout-of aprog) (abc-steps-l ac aprog astp)
               (t-steps tc (tm-of aprog, 0) bstp) ires
apply(rule-tac x = Suc 0 in exI)
apply(case-tac  abc-steps-l ac aprog (Suc 0), simp)
proof −
  fix a b
  assume crsp-l (layout-of aprog) ac tc ires
        abc-steps-l ac aprog (Suc 0) = (a, b)
  thus ∃ bstp. crsp-l (layout-of aprog) (a, b)
              (t-steps tc (tm-of aprog, 0) bstp) ires
    apply(insert steps-crsp[of layout-of aprog aprog
                  tm-of aprog ac tc ires (a, b) Suc 0], auto)
    done
qed

declare abc-steps-l.simps[simp del]

lemma abc-steps-ind:
 let (as, am) = abc-steps-l ac aprog stp in
   abc-steps-l ac aprog (Suc stp) =
              abc-step-l (as, am) (abc-fetch as aprog)
proof(simp)
  show (λ(as, am). abc-steps-l ac aprog (Suc stp) =
        abc-step-l (as, am) (abc-fetch as aprog))
              (abc-steps-l ac aprog stp)
  proof(induct stp arbitrary: ac)
    fix ac
    show (λ(as, am). abc-steps-l ac aprog (Suc 0) =
          abc-step-l (as, am) (abc-fetch as aprog))
                (abc-steps-l ac aprog 0)
      apply(case-tac ac:: nat × nat list,
          simp add: abc-steps-l.simps)
      apply(case-tac (abc-step-l (a, b) (abc-fetch a aprog)),
          simp add: abc-steps-l.simps)
      done
  next
    fix stp ac
    assume h1:
      (⋀ac. (λ(as, am). abc-steps-l ac aprog (Suc stp) =
                         abc-step-l (as, am) (abc-fetch as aprog))
          (abc-steps-l ac aprog stp))
    thus
      (λ(as, am). abc-steps-l ac aprog (Suc (Suc stp)) =
             abc-step-l (as, am) (abc-fetch as aprog))
                        (abc-steps-l ac aprog (Suc stp))
```

```
      apply(case-tac ac::nat × nat list, simp)
      apply(subgoal-tac
            abc-steps-l (a, b) aprog (Suc (Suc stp)) =
             abc-steps-l (abc-step-l (a, b) (abc-fetch a aprog))
                                                  aprog (Suc stp), simp)
      apply(case-tac (abc-step-l (a, b) (abc-fetch a aprog)), simp)
    proof -
      fix a b aa ba
      assume h2: abc-step-l (a, b) (abc-fetch a aprog) = (aa, ba)
      from h1 and h2  show
      (λ(as, am). abc-steps-l (aa, ba) aprog (Suc stp) =
           abc-step-l (as, am) (abc-fetch as aprog))
                  (abc-steps-l (a, b) aprog (Suc stp))
 apply(insert h1[of (aa, ba)])
 apply(simp add: abc-steps-l.simps)
 apply(insert h2, simp)
 done
    next
      fix a b
      show
        abc-steps-l (a, b) aprog (Suc (Suc stp)) =
         abc-steps-l (abc-step-l (a, b) (abc-fetch a aprog))
                                                       aprog (Suc stp)
 apply(simp only: abc-steps-l.simps)
 done
    qed
  qed
qed

lemma abc-unhalt-case-induct:
  ⟦crsp-l (layout-of aprog) ac tc ires;
    n < length am;
    ∀ stp. (λ(as, am). as < length aprog) (abc-steps-l ac aprog stp);
    stp ≤ bstp;
    crsp-l (layout-of aprog) (abc-steps-l ac aprog astp)
                           (t-steps tc (tm-of aprog, 0) bstp) ires⟧
 ⟹ ∃ astp bstp. Suc stp ≤ bstp ∧ crsp-l (layout-of aprog)
       (abc-steps-l ac aprog astp) (t-steps tc (tm-of aprog, 0) bstp) ires
apply(rule-tac x = Suc astp in exI)
apply(case-tac abc-steps-l ac aprog astp)
proof -
  fix a b
  assume
    ∀ stp. (λ(as, am). as < length aprog)
                 (abc-steps-l ac aprog stp)
    stp ≤ bstp
    crsp-l (layout-of aprog) (abc-steps-l ac aprog astp)
      (t-steps tc (tm-of aprog, 0) bstp) ires
    abc-steps-l ac aprog astp = (a, b)
```

thus
$\exists$ *bstp*$\geq$*Suc stp. crsp-l* (*layout-of aprog*)
(*abc-steps-l ac aprog* (*Suc astp*))
(*t-steps tc* (*tm-of aprog*, *0*) *bstp*) *ires*
apply(*insert crsp-inside*[*of layout-of aprog aprog*
tm-of aprog a b (*t-steps tc* (*tm-of aprog*, *0*) *bstp*) *ires*], *auto*)
apply(*erule-tac x* = *astp* **in** *allE*, *auto*)
apply(*rule-tac x* = *bstp* + *stpa* **in** *exI*, *simp*)
apply(*insert abc-steps-ind*[*of ac aprog astp*], *simp*)
done
qed

lemma *abc-unhalt-case*:
⟦*crsp-l* (*layout-of aprog*) *ac tc ires*;
$\forall$ *stp*. (λ(*as*, *am*). *as* < *length aprog*) (*abc-steps-l ac aprog stp*)⟧
$\Longrightarrow$ ($\exists$ *astp bstp*. *bstp* $\geq$ *stp* $\wedge$
crsp-l (*layout-of aprog*) (*abc-steps-l ac aprog astp*)
(*t-steps tc* (*tm-of aprog*, *0*) *bstp*) *ires*)
apply(*induct stp*)
apply(*rule-tac abc-unhalt-case-zero*, *auto*)
apply(*rule-tac abc-unhalt-case-induct*, *auto*)
done

lemma *abacus-turing-eq-unhalt-case-pre*:
⟦*ly* = *layout-of aprog*;
tprog = *tm-of aprog*;
crsp-l ly ac tc ires;
$\forall$ *stp*. ((λ (*as*, *am*). *as* < *length aprog*)
(*abc-steps-l ac aprog stp*));
mop-ss = *start-of ly* (*length aprog*)⟧
$\Longrightarrow$ ($\neg$ ($\exists$ *stp*. (λ (*s*, *l*, *r*). *s* = *0*)
(*t-steps tc* (*tprog* @ (*tMp n* (*mop-ss* − *1*)), *0*) *stp*)))
apply(*auto*)
proof −
fix *stp a b*
assume *h1*:
crsp-l (*layout-of aprog*) *ac tc ires*
$\forall$ *stp*. (λ(*as*, *am*). *as* < *length aprog*) (*abc-steps-l ac aprog stp*)
t-steps tc (*tm-of aprog* @ *tMp n* (*start-of* (*layout-of aprog*)
(*length aprog*) − *Suc 0*), *0*) *stp* = (*0*, *a*, *b*)
thus *False*
proof(*insert abc-unhalt-case*[*of aprog ac tc ires stp*], *auto*,
case-tac (*abc-steps-l ac aprog astp*),
case-tac (*t-steps tc* (*tm-of aprog*, *0*) *bstp*), *simp*)
fix *astp bstp aa ba aaa baa c*
assume *h2*:
abc-steps-l ac aprog astp = (*aa*, *ba*) *stp* $\leq$ *bstp*
t-steps tc (*tm-of aprog*, *0*) *bstp* = (*aaa*, *baa*, *c*)
crsp-l (*layout-of aprog*) (*aa*, *ba*) (*aaa*, *baa*, *c*) *ires*

hence *h3*:
t-steps tc (*tm-of aprog* @ *tMp n*
(*start-of* (*layout-of aprog*) (*length aprog*) − *Suc 0*), *0*) *bstp*
= (*aaa*, *baa*, *c*)
apply(*intro tm-append-steps*, *auto*)
apply(*simp add*: *crsp-l.simps*, *rule startof-not0*)
done
from *h2* **have** *h4*: ∃ *diff*. *bstp* = *stp* + *diff*
apply(*rule-tac x* = *bstp* − *stp* **in** *exI*, *simp*)
done
from *h4* **and** *h3* **and** *h2* **and** *h1* **show** *?thesis*
apply(*auto*)
apply(*simp add*: *state0-ind crsp-l.simps*)
apply(*subgoal-tac start-of* (*layout-of aprog*) *aa* > *0*, *simp*)
apply(*rule startof-not0*)
done
qed
qed

lemma *abacus-turing-eq-unhalt-case*:
assumes *layout*:
— There is an Abacus program *aprog* with layout *ly*:
ly = *layout-of aprog*
and *compiled*:
— The TM compiled from *aprog* is *tprog*:
tprog = *tm-of aprog*
and *correspond*:
— TM configuration *tc* and Abacus configuration *ac* are in correspondence:
crsp-l ly ac tc ires
and *abc-unhalt*:
— If, no matter how many steps the Abacus program *aprog* executes, it may never reach a halt state.
∀ *stp*. ((λ (*as*, *am*). *as* < *length aprog*)
(*abc-steps-l ac aprog stp*))
and *mopup-start*: *mop-ss* = *start-of ly* (*length aprog*)
shows
— The the TM composed of TM *tprog* and the moupup TM may never reach a halt state as well.
¬ (∃ *stp*. (λ (*s*, *l*, *r*). *s* = *0*)
(*t-steps tc* (*tprog* @ (*tMp n* (*mop-ss* − *1*)), *0*) *stp*))
using *layout compiled correspond abc-unhalt mopup-start*
apply(*rule-tac abacus-turing-eq-unhalt-case-pre*, *auto*)
done

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 [*intro*]: *abc-list-crsp* (*lm* @ 0^m) *lm*

```
apply(auto simp: abc-list-crsp-def)
done

lemma abc-list-crsp-lm-v:
  abc-list-crsp lma lmb ⟹ abc-lm-v lma n = abc-lm-v lmb n
apply(auto simp: abc-list-crsp-def abc-lm-v.simps
              nth-append exponent-def)
done

lemma  rep-app-cons-iff:
  k < n ⟹ replicate n a[k:=b] =
        replicate k a @ b # replicate (n − k − 1) a
apply(induct n arbitrary: k, simp)
apply(simp split:nat.splits)
done

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)
apply(auto simp: abc-list-crsp-def abc-lm-v.simps abc-lm-s.simps)
apply(simp-all add: list-update-append, auto simp: exponent-def)
proof −
  fix na
  assume h: m < length lmb + na  ¬ m < length lmb
  hence m − length lmb < na by simp
  hence replicate na 0[(m− length lmb):= n] =
          replicate (m − length lmb) 0 @ n #
             replicate (na − (m − length lmb) − 1) 0
    apply(erule-tac rep-app-cons-iff)
    done
  thus ∃nb. replicate na 0[m − length lmb := n] =
                replicate (m − length lmb) 0 @ n # replicate nb 0 ∨
                replicate (m − length lmb) 0 @ [n] =
                replicate na 0[m − length lmb := n] @ replicate nb 0
    apply(auto)
    done
next
  fix na
  assume h: ¬ m < length lmb + na
  show
    ∃nb. replicate na 0 @ replicate (m − (length lmb + na)) 0 @ [n] =
          replicate (m − length lmb) 0 @ n # replicate nb 0 ∨
         replicate (m − length lmb) 0 @ [n] =
           replicate na 0 @
           replicate (m − (length lmb + na)) 0 @ n # replicate nb 0
    apply(rule-tac x = 0 in exI, simp, auto)
    using h
    apply(simp add: replicate-add[THEN sym])
    done
```

next
  fix *na*
  assume *h*: ¬ *m* < *length lma m* < *length lma* + *na*
  hence *m* − *length lma* < *na* **by** *simp*
  hence
    replicate na 0[(*m*− *length lma*):= *n*] = *replicate* (*m* − *length lma*)
              0 @ *n* # *replicate* (*na* − (*m* − *length lma*) − *1*) *0*
    apply(*erule-tac rep-app-cons-iff*)
    done
  thus ∃ *nb*. *replicate* (*m* − *length lma*) *0* @ [*n*] =
              replicate na 0[*m* − *length lma* := *n*] @ *replicate nb 0*
          ∨ *replicate na 0*[*m* − *length lma* := *n*] =
              replicate (*m* − *length lma*) *0* @ *n* # *replicate nb 0*
    apply(*auto*)
    done
next
  fix *na*
  assume ¬ *m* < *length lma* + *na*
  thus ∃ *nb*. *replicate* (*m* − *length lma*) *0* @ [*n*] =
          replicate na 0 @
          replicate (*m* − (*length lma* + *na*)) *0* @ *n* # *replicate nb 0*
        ∨ *replicate na 0* @
            replicate (*m* − (*length lma* + *na*)) *0* @ [*n*] =
          replicate (*m* − *length lma*) *0* @ *n* # *replicate nb 0*
    apply(*rule-tac x* = *0* **in** *exI*, *simp*, *auto*)
    apply(*simp add*: *replicate-add*[*THEN sym*])
    done
qed

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(*case-tac i*, *auto simp*: *abc-step-l.simps*
      abc-list-crsp-lm-s abc-list-crsp-lm-v Let-def
                  split: *abc-inst.splits if-splits*)
done

lemma *abc-steps-red*:
  abc-steps-l ac aprog stp = (*as*, *am*) ⟹
    abc-steps-l ac aprog (*Suc stp*) =
        abc-step-l (*as*, *am*) (*abc-fetch as aprog*)
using *abc-steps-ind*[*of ac aprog stp*]
apply(*simp*)
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
apply(induct stp arbitrary: a lm', simp add: abc-steps-l.simps, auto)
apply(case-tac abc-steps-l (0, lm @ 0^m) aprog stp,
   simp add: abc-steps-ind)
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-steps-l (0, lm @ 0^m) aprog (Suc stp) = (a, lm')
         abc-steps-l (0, lm @ 0^m) aprog stp = (aa, b)
         aprog ≠ []
  hence g1: abc-steps-l (0, lm @ 0^m) aprog (Suc stp)
        = abc-step-l (aa, b) (abc-fetch aa aprog)
    apply(rule-tac abc-steps-red, simp)
    done
  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-steps-red, simp)
    done
  show ∃ lma. abc-steps-l (0, lm) aprog (Suc stp) = (a, lma) ∧
           abc-list-crsp lm' lma
    using g1 g2 g3 h
    apply(auto)
    apply(case-tac abc-step-l (aa, b) (abc-fetch aa aprog),
        case-tac abc-step-l (aa, lma) (abc-fetch aa aprog), simp)
    apply(rule-tac abc-list-crsp-step, auto)
    done
qed

lemma [simp]: (case ca of [] ⇒ Bk | Bk # xs ⇒ Bk | Oc # xs ⇒ Oc) =
              (case ca of [] ⇒ Bk | x # xs ⇒ x)
by(case-tac ca, simp-all, case-tac a, simp, simp)

lemma steps-eq: length t mod 2 = 0 ⟹
                   t-steps c (t, 0) stp = steps c t stp
apply(induct stp)
apply(simp add: steps.simps t-steps.simps)
apply(simp add:tstep-red t-steps-ind)
apply(case-tac steps c t stp, simp)
apply(auto simp: t-step.simps tstep.simps)
```

done

lemma *crsp-l-start*: *crsp-l ly* (*0*, *lm*) (*Suc 0*, *Bk* # *Bk* # *ires*, <*lm*> @ Bk^{rn})
ires
apply(*simp add*: *crsp-l.simps*, *auto simp*: *start-of.simps*)
done

lemma *t-ncorrect-app*: ⟦*t-ncorrect t1*; *t-ncorrect t2*⟧ ⟹
t-ncorrect (*t1* @ *t2*)
apply(*simp add*: *t-ncorrect.simps*, *auto*)
done

lemma [*simp*]:
(*length* (*tm-of aprog*) +
length (*tMp n* (*start-of ly* (*length aprog*) − *Suc 0*))) *mod 2* = *0*
apply(*subgoal-tac*
t-ncorrect (*tm-of aprog* @ *tMp n*
(*start-of ly* (*length aprog*) − *Suc 0*)))
apply(*simp add*: *t-ncorrect.simps*)
apply(*rule-tac t-ncorrect-app*,
auto simp: *tMp.simps t-ncorrect.simps tshift.simps mp-up-def*)
apply(*subgoal-tac*
t-ncorrect (*tm-of aprog*), *simp add*: *t-ncorrect.simps*)
apply(*auto*)
done

lemma [*simp*]: *takeWhile* (λ*a*. *a* = *Oc*)
(*replicate rs Oc* @ *replicate rn Bk*) = *replicate rs Oc*
apply(*induct rs*, *auto*)
apply(*induct rn*, *auto*)
done

lemma *abacus-turing-eq-halt′*:
⟦*ly* = *layout-of aprog*;
tprog = *tm-of aprog*;
n < *length am*;
abc-steps-l (*0*, *lm*) *aprog stp* = (*as*, *am*);
mop-ss = *start-of ly* (*length aprog*);
as ≥ *length aprog*⟧
⟹ ∃ *stp m l*. *steps* (*Suc 0*, *Bk* # *Bk* # *ires*, <*lm*> @ Bk^{rn})
(*tprog* @ (*tMp n* (*mop-ss* − *1*))) *stp*
= (*0*, Bk^{m} @ *Bk* # *Bk* # *ires*, $Oc^{Suc\ (abc\text{-}lm\text{-}v\ am\ n)}$ @ Bk^{l})
apply(*drule-tac tc* = (*Suc 0*, *Bk* # *Bk* # *ires*, <*lm*> @ Bk^{rn}) **in**
abacus-turing-eq-halt-case, *auto intro*: *crsp-l-start*)
apply(*subgoal-tac*
length (*tm-of aprog* @ *tMp n*
(*start-of ly* (*length aprog*) − *Suc 0*)) *mod 2* = *0*)
apply(*simp add*: *steps-eq*)
apply(*rule-tac x* = *stpa* **in** *exI*,

simp add: *exponent-def*, *auto*)
done

lemma *list-length*: $xs = ys \Longrightarrow length\ xs = length\ ys$
by *simp*
lemma [*elim*]: $tinres\ (Bk^m)\ b \Longrightarrow \exists\, m.\ b = Bk^m$
apply(*auto simp*: *tinres-def*)
apply(*rule-tac* $x = m{-}n$ **in** *exI*,
auto simp: *exponent-def replicate-add*[*THEN sym*])
apply(*case-tac* $m < n$, *auto*)
apply(*drule-tac list-length*, *auto*)
apply(*subgoal-tac* $\exists\ d.\ m = d + n$, *auto simp*: *replicate-add*)
apply(*rule-tac* $x = m - n$ **in** *exI*, *simp*)
done
lemma [*intro*]: *tinres* [*Bk*] (Bk^k)
apply(*auto simp*: *tinres-def exponent-def*)
apply(*case-tac k*, *auto*)
apply(*rule-tac* $x = Suc\ 0$ **in** *exI*, *simp*)
done

lemma *abacus-turing-eq-halt-pre*:
⟦*ly* = *layout-of aprog*;
tprog = *tm-of aprog*;
n < *length am*;
abc-steps-l (*0*, *lm*) *aprog stp* = (*as*, *am*);
mop-ss = *start-of ly* (*length aprog*);
as ≥ *length aprog*⟧
⟹ ∃ *stp m l. steps* (*Suc 0*, *Bk* # *Bk* # *ires*, <*lm*> @ Bk^{rn})
(*tprog* @ (*tMp n* (*mop-ss* − *1*))) *stp*
= (*0*, Bk^m @ *Bk* # *Bk* # *ires*, $Oc^{Suc\ (abc\text{-}lm\text{-}v\ am\ n)}$ @ Bk^l)
using *abacus-turing-eq-halt′*
apply(*simp*)
done

Main theorem for the case when the original Abacus program does halt.

lemma *abacus-turing-eq-halt*:
assumes *layout*:
ly = *layout-of aprog*
— There is an Abacus program *aprog* with layout *ly*:
and *compiled*: *tprog* = *tm-of aprog*
— The TM compiled from *aprog* is *tprog*:
and *halt-state*:
— *as* is a program label outside the range of *aprog*. So if Abacus is in such a state, it is in halt state:
as ≥ *length aprog*
and *abc-exec*:
— Supposing after *stp* step of execution, Abacus program *aprog* reaches such a halt state:

abc-steps-l (*0*, *lm*) *aprog stp* = (*as*, *am*)
and *rs-locate*:
— *n* is a memory address in the range of Abacus memory *am*:
n < *length am*
and *mopup-start*:
— The startling label for mopup mahines, according to the layout and Abacus program should be *mop-ss*:
mop-ss = *start-of ly* (*length aprog*)
shows
— After *stp* steps of execution of the TM composed of *tprog* and the mopup TM (*tMp n* (*mop-ss* − *1*)) will halt and gives rise to a configuration which only hold the content of memory cell *n*:
∃ *stp m l*. *steps* (*Suc 0*, *Bk* # *Bk* # *ires*, <*lm*> @ Bk^{rn}) (*tprog* @ (*tMp n* (*mop-ss* − *1*))) *stp*
= (*0*, Bk^{m} @ *Bk* # *Bk* # *ires*, $Oc^{Suc\ (abc\text{-}lm\text{-}v\ am\ n)}$ @ Bk^{l})
using *layout compiled halt-state abc-exec rs-locate mopup-start*
by(*rule-tac abacus-turing-eq-halt-pre*, *auto*)

lemma *abacus-turing-eq-uhalt'*:
⟦*ly* = *layout-of aprog*;
tprog = *tm-of aprog*;
∀ *stp*. ((λ (*as*, *am*). *as* < *length aprog*)
(*abc-steps-l* (*0*, *lm*) *aprog stp*));
mop-ss = *start-of ly* (*length aprog*)⟧
⟹ (¬ (∃ *stp*. *isS0* (*steps* (*Suc 0*, [*Bk*, *Bk*], <*lm*>)
(*tprog* @ (*tMp n* (*mop-ss* − *1*))) *stp*)))
apply(*drule-tac tc* = (*Suc 0*, [*Bk*, *Bk*], <*lm*>) **and** *n* = *n* **and** *ires* = [] **in**
abacus-turing-eq-unhalt-case, *auto intro*: *crsp-l-start*)
apply(*simp add*: *crsp-l.simps start-of.simps*)
apply(*erule-tac x* = *stp* **in** *allE*, *erule-tac x* = *stp* **in** *allE*)
apply(*subgoal-tac*
length (*tm-of aprog* @ *tMp n*
(*start-of ly* (*length aprog*) − *Suc 0*)) *mod 2* = *0*)
apply(*simp add*: *steps-eq*, *auto simp*: *isS0-def*)
done

Main theorem for the case when the original Abacus program does not halt.

lemma *abacus-turing-eq-uhalt*:
assumes *layout*:
— There is an Abacus program *aprog* with layout *ly*:
ly = *layout-of aprog*
and *compiled*:
— The TM compiled from *aprog* is *tprog*:
tprog = *tm-of aprog*
and *abc-unhalt*:
— If, no matter how many steps the Abacus program *aprog* executes, it may never reach a halt state.
∀ *stp*. ((λ (*as*, *am*). *as* < *length aprog*)
(*abc-steps-l* (*0*, *lm*) *aprog stp*))

and *mop-start*: *mop-ss* = *start-of ly* (*length aprog*)
shows
— The the TM composed of TM *tprog* and the moupup TM may never reach a halt state as well.
¬ (∃ *stp*. *isS0* (*steps* (*Suc 0*, [*Bk*, *Bk*], <*lm*>)
(*tprog* @ (*tMp n* (*mop-ss* − *1*))) *stp*))
using *abacus-turing-eq-uhalt′*
layout compiled abc-unhalt mop-start
by(*auto*)

end

theory *rec-def*
imports *Main*
begin

9 Recursive functions

Datatype of recursive operators.

datatype *recf* =
— The zero function, which always resturns *0* as result.
z |
— The successor function, which increments its arguments.
s |
— The projection function, where *id i j* returns the *j*-th argment out of the *i* arguments.
id nat nat |
— The compostion operator, where "*Cn n f* [*g1*; *g2*; ... ;*gm*] computes *f* (*g1*(*x1*, *x2*, ..., *xn*), *g2*(*x1*, *x2*, ..., *xn*), ... , *gm*(*x1*, *x2*, ... , *xn*)) for input argments *x1*, ..., *xn*.
Cn nat recf recf list |
— The primitive resursive operator, where *Pr n f g* computes: *Pr n f g* (*x1*, *x2*, ..., *xn*−*1*, *0*) = *f*(*x1*, ..., *xn*−*1*) and *Pr n f g* (*x1*, *x2*, ..., *xn*−*1*, *k′*) = *g*(*x1*, *x2*, ..., *xn*−*1*, *k*, *Pr n f g* (*x1*, ..., *xn*−*1*, *k*)).
Pr nat recf recf |
— The minimization operator, where *Mn n f* (*x1*, *x2*, ... , *xn*) computes the first i such that *f* (*x1*, ..., *xn*, *i*) = *0* and for all *j*, *f* (*x1*, *x2*, ..., *xn*, *j*) > *0*.
Mn nat recf

The semantis of recursive operators is given by an inductively defined relation as follows, where *rec-calc-rel R* [*x1*, *x2*, ..., *xn*] *r* means the computation of *R* over input arguments [*x1*, *x2*, ..., *xn* terminates and gives rise to a result *r*

inductive *rec-calc-rel* :: *recf* ⇒ *nat list* ⇒ *nat* ⇒ *bool*
where
calc-z: *rec-calc-rel z* [*n*] *0* |

calc-s: *rec-calc-rel s* [*n*] (*Suc n*) |
calc-id: ⟦*length args* = *i*; *j* < *i*; *args*!*j* = *r*⟧ ⟹ *rec-calc-rel* (*id i j*) *args r* |
calc-cn: ⟦*length args* = *n*;
∀ *k* < *length gs*. *rec-calc-rel* (*gs* ! *k*) *args* (*rs* ! *k*);
length rs = *length gs*;
rec-calc-rel f rs r⟧
⟹ *rec-calc-rel* (*Cn n f gs*) *args r* |
calc-pr-zero:
⟦*length args* = *n*;
rec-calc-rel f args r0 ⟧
⟹ *rec-calc-rel* (*Pr n f g*) (*args* @ [*0*]) *r0* |
calc-pr-ind:
⟦ *length args* = *n*;
rec-calc-rel (*Pr n f g*) (*args* @ [*k*]) *rk*;
rec-calc-rel g (*args* @ [*k*] @ [*rk*]) *rk'*⟧
⟹ *rec-calc-rel* (*Pr n f g*) (*args* @ [*Suc k*]) *rk'* |
calc-mn: ⟦*length args* = *n*;
rec-calc-rel f (*args*@[*r*]) *0*;
∀ *i* < *r*. (∃ *ri*. *rec-calc-rel f* (*args*@[*i*]) *ri* ∧ *ri* ≠ *0*)⟧
⟹ *rec-calc-rel* (*Mn n f*) *args r*

inductive-cases *calc-pr-reverse*:
rec-calc-rel (*Pr n f g*) (*lm*) *rSucy*

inductive-cases *calc-z-reverse*: *rec-calc-rel z lm x*

inductive-cases *calc-s-reverse*: *rec-calc-rel s lm x*

inductive-cases *calc-id-reverse*: *rec-calc-rel* (*id m n*) *lm x*

inductive-cases *calc-cn-reverse*: *rec-calc-rel* (*Cn n f gs*) *lm x*

inductive-cases *calc-mn-reverse*:*rec-calc-rel* (*Mn n f*) *lm x*
end
theory *recursive*
imports *Main rec-def abacus*
begin

10 Compiling from recursive functions to Abacus machines

Some auxilliary Abacus machines used to construct the result Abacus machines.

get-paras-num recf returns the arity of recursive function *recf*.

fun *get-paras-num* :: *recf* ⇒ *nat*
where
get-paras-num z = *1* |

get-paras-num s = *1* |
get-paras-num (*id m n*) = *m* |
get-paras-num (*Cn n f gs*) = *n* |
get-paras-num (*Pr n f g*) = *Suc n* |
get-paras-num (*Mn n f*) = *n*

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 *empty* :: *nat* ⇒ *nat* ⇒ *abc-prog*
where
empty m n = [*Dec m 3*, *Inc n*, *Goto 0*]

fun *abc-inst-shift* :: *abc-inst* ⇒ *nat* ⇒ *abc-inst*
where
abc-inst-shift (*Inc m*) *n* = *Inc m* |
abc-inst-shift (*Dec m e*) *n* = *Dec m* (*e* + *n*) |
abc-inst-shift (*Goto m*) *n* = *Goto* (*m* + *n*)

fun *abc-shift* :: *abc-inst list* ⇒ *nat* ⇒ *abc-inst list*
where
abc-shift xs n = *map* (λ *x*. *abc-inst-shift x n*) *xs*

fun *abc-append* :: *abc-inst list* ⇒ *abc-inst list* ⇒
abc-inst list (**infixl** [+] *60*)
where
abc-append al bl = (*let al-len* = *length al in*
al @ *abc-shift bl al-len*)

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* [+] *empty* (*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 [+] *empty 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.

function *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*) (*f* # *gs*) *in*
let (*fprog*, *fpara*, *fn*) = *hd cied-gs 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* (*tl cied-gs*) *pstr* [+] *mv-boxes 0 qstr n* [+]
mv-boxes pstr 0 (*length gs*) [+] *fprog* [+]
empty fpara pstr [+] *empty-boxes* (*length gs*) [+]
empty 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*
(*empty n p* [+] *fprog* [+] *empty 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) )
  by pat-completeness auto
termination
proof
term size
  show wf (measure size) by auto
next
  fix n f gs x
  assume (x::recf) ∈ set (f # gs)
  thus (x, Cn n f gs) ∈ measure size
    by(induct gs, auto)
next
  fix n f g
  show (f, Pr n f g) ∈ measure size by auto
next
  fix n f g x xa y xb ya
  show (g, Pr n f g) ∈ measure size by auto
next
  fix n f
  show (f, Mn n f) ∈ measure size by auto
qed

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] abc-append.simps[simp del]
        empty.simps[simp del] addition.simps[simp del]

thm rec-calc-rel.induct

declare abc-steps-l.simps[simp del] abc-fetch.simps[simp del]
        abc-step-l.simps[simp del]

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 rec-calc-inj-case-z:
  ⟦rec-calc-rel z l x; rec-calc-rel z l y⟧ ⟹ x = y
apply(auto elim: calc-z-reverse)
done

lemma  rec-calc-inj-case-s:
  ⟦rec-calc-rel s l x; rec-calc-rel s l y⟧ ⟹ x = y
apply(auto elim: calc-s-reverse)
done

lemma rec-calc-inj-case-id:
  ⟦rec-calc-rel (recf.id nat1 nat2) l x;
    rec-calc-rel (recf.id nat1 nat2) l y⟧ ⟹ x = y
apply(auto elim: calc-id-reverse)
done

lemma rec-calc-inj-case-mn:
  assumes ind: ⋀ l x y. ⟦rec-calc-rel f l x; rec-calc-rel f l y⟧
          ⟹ x = y
  and h: rec-calc-rel (Mn n f) l x rec-calc-rel (Mn n f) l y
  shows x = y
  apply(insert h)
  apply(elim  calc-mn-reverse)
  apply(case-tac x > y, simp)
  apply(erule-tac x = y in allE, auto)
proof −
  fix v va
  assume rec-calc-rel f (l @ [y]) 0
    rec-calc-rel f (l @ [y]) v
    0 < v
  thus False
    apply(insert ind[of l @ [y] 0 v], simp)
    done
next
  fix v va
  assume
    rec-calc-rel f (l @ [x]) 0
    ∀ x<y. ∃ v. rec-calc-rel f (l @ [x]) v ∧ 0 < v ¬ y < x
  thus x = y
    apply(erule-tac x = x in allE)
    apply(case-tac x = y, auto)
    apply(drule-tac y = v in ind, simp, simp)
    done
```

qed

lemma *rec-calc-inj-case-pr*:
assumes *f-ind*:
$\bigwedge l\ x\ y.\ [\![rec\text{-}calc\text{-}rel\ f\ l\ x;\ rec\text{-}calc\text{-}rel\ f\ l\ y]\!] \Longrightarrow x = y$
and *g-ind*:
$\bigwedge x\ xa\ y\ xb\ ya\ l\ xc\ yb.$
$[\![x = rec\text{-}ci\ f;\ (xa,\ y) = x;\ (xb,\ ya) = y;$
$rec\text{-}calc\text{-}rel\ g\ l\ xc;\ rec\text{-}calc\text{-}rel\ g\ l\ yb]\!] \Longrightarrow xc = yb$
and *h*: *rec-calc-rel* (*Pr n f g*) *l x rec-calc-rel* (*Pr n f g*) *l y*
shows $x = y$
apply(*case-tac rec-ci f*)
proof −
fix *a b c*
assume *rec-ci f* = (*a*, *b*, *c*)
hence *ng-ind*:
$\bigwedge\ l\ xc\ yb.\ [\![rec\text{-}calc\text{-}rel\ g\ l\ xc;\ rec\text{-}calc\text{-}rel\ g\ l\ yb]\!]$
$\Longrightarrow xc = yb$
apply(*insert g-ind*[*of* (*a*, *b*, *c*) *a* (*b*, *c*) *b c*], *simp*)
done
from *h* **show** $x = y$
apply(*erule-tac calc-pr-reverse*, *erule-tac calc-pr-reverse*)
apply(*erule f-ind*, *simp*, *simp*)
apply(*erule-tac calc-pr-reverse*, *simp*, *simp*)
proof −
fix *la ya ry laa yaa rya*
assume *k1*: *rec-calc-rel g* (*la* @ [*ya*, *ry*]) *x*
rec-calc-rel g (*la* @ [*ya*, *rya*]) *y*
and *k2*: *rec-calc-rel* (*Pr* (*length la*) *f g*) (*la* @ [*ya*]) *ry*
rec-calc-rel (*Pr* (*length la*) *f g*) (*la* @ [*ya*]) *rya*
from *k2* **have** *ry* = *rya*
apply(*induct ya arbitrary*: *ry rya*)
apply(*erule-tac calc-pr-reverse*,
erule-tac calc-pr-reverse, *simp*)
apply(*erule f-ind*, *simp*, *simp*, *simp*)
apply(*erule-tac calc-pr-reverse*, *simp*)
apply(*erule-tac rSucy* = *rya* **in** *calc-pr-reverse*, *simp*, *simp*)
proof −
fix *ya ry rya l y ryb laa yb ryc*
assume *ind*:
$\bigwedge ry\ rya.\ [\![rec\text{-}calc\text{-}rel\ (Pr\ (length\ l)\ f\ g)\ (l\ @\ [y])\ ry;$
$rec\text{-}calc\text{-}rel\ (Pr\ (length\ l)\ f\ g)\ (l\ @\ [y])\ rya]\!] \Longrightarrow ry = rya$
and *j*: *rec-calc-rel* (*Pr* (*length l*) *f g*) (*l* @ [*y*]) *ryb*
rec-calc-rel g (*l* @ [*y*, *ryb*]) *ry*
rec-calc-rel (*Pr* (*length l*) *f g*) (*l* @ [*y*]) *ryc*
rec-calc-rel g (*l* @ [*y*, *ryc*]) *rya*
from *j* **show** *ry* = *rya*
apply(*insert ind*[*of ryb ryc*], *simp*)
apply(*insert ng-ind*[*of l* @ [*y*, *ryc*] *ry rya*], *simp*)

```
 done
   qed
   from k1 and this show x = y
     apply(simp)
     apply(insert ng-ind[of la @ [ya, rya] x y], simp)
     done
 qed
qed

lemma Suc-nth-part-eq:
 ∀ k<Suc (length list). (a # xs) ! k = (aa # list) ! k
     ⟹ ∀ k<(length list). (xs) ! k = (list) ! k
apply(rule allI, rule impI)
apply(erule-tac x = Suc k in allE, simp)
done


lemma list-eq-intro:
 ⟦length xs = length ys; ∀ k < length xs. xs ! k = ys ! k⟧
 ⟹ xs = ys
apply(induct xs arbitrary: ys, simp)
apply(case-tac ys, simp, simp)
proof −
 fix a xs ys aa list
 assume ind:
   ⋀ys. ⟦length list = length ys; ∀ k<length ys. xs ! k = ys ! k⟧
   ⟹ xs = ys
   and h: length xs = length list
   ∀ k<Suc (length list). (a # xs) ! k = (aa # list) ! k
 from h show a = aa ∧ xs = list
   apply(insert ind[of list], simp)
   apply(frule Suc-nth-part-eq, simp)
   apply(erule-tac x = 0 in allE, simp)
   done
qed

lemma rec-calc-inj-case-cn:
 assumes ind:
 ⋀x l xa y.
 ⟦x = f ∨ x ∈ set gs; rec-calc-rel x l xa; rec-calc-rel x l y⟧
 ⟹ xa = y
 and h: rec-calc-rel (Cn n f gs) l x
       rec-calc-rel (Cn n f gs) l y
 shows x = y
 apply(insert h, elim  calc-cn-reverse)
 apply(subgoal-tac rs = rsa)
 apply(rule-tac x = f and l = rsa and xa = x and y = y in ind,
      simp, simp, simp)
 apply(intro list-eq-intro, simp, rule allI, rule impI)
```

apply(*erule-tac* $x = k$ **in** *allE*, *rule-tac* $x = k$ **in** *allE*, *simp*, *simp*)
apply(*rule-tac* $x = gs ! k$ **in** *ind*, *simp*, *simp*, *simp*)
done

lemma *rec-calc-inj*:
⟦*rec-calc-rel f l x*;
rec-calc-rel f l y⟧ $\Longrightarrow x = y$
apply(*induct f arbitrary*: *l x y rule*: *rec-ci.induct*)
apply(*simp add*: *rec-calc-inj-case-z*)
apply(*simp add*: *rec-calc-inj-case-s*)
apply(*simp add*: *rec-calc-inj-case-id*, *simp*)
apply(*erule rec-calc-inj-case-cn*,*simp*, *simp*)
apply(*erule rec-calc-inj-case-pr*, *auto*)
apply(*erule rec-calc-inj-case-mn*, *auto*)
done

lemma *calc-rel-reverse-ind-step-ex*:
⟦*rec-calc-rel* (*Pr n f g*) (*lm* @ [*Suc x*]) *rs*⟧
$\Longrightarrow \exists$ *rs*. *rec-calc-rel* (*Pr n f g*) (*lm* @ [*x*]) *rs*
apply(*erule calc-pr-reverse*, *simp*, *simp*)
apply(*rule-tac* $x = rk$ **in** *exI*, *simp*)
done

lemma [*simp*]: $Suc\ x \le y \Longrightarrow Suc\ (y - Suc\ x) = y - x$
by *arith*

lemma *calc-pr-para-not-null*:
rec-calc-rel (*Pr n f g*) *lm rs* $\Longrightarrow lm \neq []$
apply(*erule calc-pr-reverse*, *simp*, *simp*)
done

lemma *calc-pr-less-ex*:
⟦*rec-calc-rel* (*Pr n f g*) *lm rs*; $x \le$ *last lm*⟧ $\Longrightarrow$
$\exists$*rs*. *rec-calc-rel* (*Pr n f g*) (*butlast lm* @ [*last lm* $-$ *x*]) *rs*
apply(*subgoal-tac lm* $\neq$ [])
apply(*induct x*, *rule-tac* $x = rs$ **in** *exI*, *simp*, *simp*, *erule exE*)
apply(*rule-tac* $rs = xa$ **in** *calc-rel-reverse-ind-step-ex*, *simp*)
apply(*simp add*: *calc-pr-para-not-null*)
done

lemma *calc-pr-zero-ex*:
rec-calc-rel (*Pr n f g*) *lm rs* $\Longrightarrow$
$\exists$*rs*. *rec-calc-rel f* (*butlast lm*) *rs*
apply(*drule-tac* $x = $ *last lm* **in** *calc-pr-less-ex*, *simp*,
erule-tac exE, *simp*)
apply(*erule-tac calc-pr-reverse*, *simp*)
apply(*rule-tac* $x = rs$ **in** *exI*, *simp*, *simp*)
done

lemma *abc-steps-ind*:
  abc-steps-l (*as*, *am*) *ap* (*Suc stp*) =
          abc-steps-l (*abc-steps-l* (*as*, *am*) *ap stp*) *ap* (*Suc 0*)
apply(*insert abc-steps-add*[*of as am ap stp Suc 0*], *simp*)
done

lemma *abc-steps-zero*: *abc-steps-l asm ap 0* = *asm*
apply(*case-tac asm*, *simp add*: *abc-steps-l.simps*)
done

lemma *abc-append-nth*:
  n < *length ap* + *length bp* ⟹
      (*ap* [+] *bp*) ! *n* =
        (*if n* < *length ap then ap* ! *n*
         else abc-inst-shift (*bp* ! (*n* − *length ap*)) (*length ap*))
apply(*simp add*: *abc-append.simps nth-append map-nth split*: *if-splits*)
done

lemma *abc-state-keep*:
  as ≥ *length bp* ⟹ *abc-steps-l* (*as*, *lm*) *bp stp* = (*as*, *lm*)
apply(*induct stp*, *simp add*: *abc-steps-zero*)
apply(*simp add*: *abc-steps-ind*)
apply(*simp add*: *abc-steps-zero*)
apply(*simp add*: *abc-steps-l.simps abc-fetch.simps abc-step-l.simps*)
done

lemma *abc-halt-equal*:
  ⟦*abc-steps-l* (*0*, *lm*) *bp stpa* = (*length bp*, *lm1*);
    abc-steps-l (*0*, *lm*) *bp stpb* = (*length bp*, *lm2*)⟧ ⟹ *lm1* = *lm2*
apply(*case-tac stpa* − *stpb* > *0*)
apply(*insert abc-steps-add*[*of 0 lm bp stpb stpa* − *stpb*], *simp*)
apply(*insert abc-state-keep*[*of bp length bp lm2 stpa* − *stpb*],
      simp, *simp add*: *abc-steps-zero*)
apply(*insert abc-steps-add*[*of 0 lm bp stpa stpb* − *stpa*], *simp*)
apply(*insert abc-state-keep*[*of bp length bp lm1 stpb* − *stpa*],
      simp)
done

lemma *abc-halt-point-ex*:
  ⟦∃ *stp*. *abc-steps-l* (*0*, *lm*) *bp stp* = (*bs*, *lm′*);
    bs = *length bp*; *bp* ≠ []⟧
  ⟹ ∃ *stp*. (λ (*s*, *l*). *s* < *bs* ∧
              (*abc-steps-l* (*s*, *l*) *bp* (*Suc 0*)) = (*bs*, *lm′*))
      (*abc-steps-l* (*0*, *lm*) *bp stp*)
apply(*erule-tac exE*)
proof −
  fix *stp*

```
  assume bs = length bp
         abc-steps-l (0, lm) bp stp = (bs, lm')
         bp ≠ []
  thus
    ∃ stp. (λ(s, l). s < bs ∧
      abc-steps-l (s, l) bp (Suc 0) = (bs, lm'))
                      (abc-steps-l (0, lm) bp stp)
    apply(induct stp, simp add: abc-steps-zero, simp)
  proof −
    fix stpa
    assume ind:
     abc-steps-l (0, lm) bp stpa = (length bp, lm')
       ⟹ ∃ stp. (λ(s, l). s < length bp  ∧ abc-steps-l (s, l) bp
             (Suc 0) = (length bp, lm')) (abc-steps-l (0, lm) bp stp)
    and h: abc-steps-l (0, lm) bp (Suc stpa) = (length bp, lm')
           abc-steps-l (0, lm) bp stp = (length bp, lm')
           bp ≠ []
    from h show
      ∃ stp. (λ(s, l). s < length bp ∧ abc-steps-l (s, l) bp (Suc 0)
                  = (length bp, lm')) (abc-steps-l (0, lm) bp stp)
      apply(case-tac abc-steps-l (0, lm) bp stpa,
            case-tac a = length bp)
      apply(insert ind, simp)
      apply(subgoal-tac b = lm', simp)
      apply(rule-tac abc-halt-equal, simp, simp)
      apply(rule-tac x = stpa in exI, simp add: abc-steps-ind)
      apply(simp add: abc-steps-zero)
      apply(rule classical, simp add: abc-steps-l.simps
                              abc-fetch.simps abc-step-l.simps)
      done
  qed
qed


lemma abc-append-empty-r[simp]: [] [+] ab = ab
apply(simp add: abc-append.simps abc-inst-shift.simps)
apply(induct ab, simp, simp)
apply(case-tac a, simp-all add: abc-inst-shift.simps)
done

lemma abc-append-empty-l[simp]:  ab [+] [] = ab
apply(simp add: abc-append.simps abc-inst-shift.simps)
done


lemma abc-append-length[simp]:
  length (ap [+] bp) = length ap + length bp
apply(simp add: abc-append.simps)
done
```

lemma *abc-append-commute*: *as* [+] *bs* [+] *cs* = *as* [+] (*bs* [+] *cs*)
apply(*simp add*: *abc-append.simps abc-shift.simps abc-inst-shift.simps*)
apply(*induct cs*, *simp*, *simp*)
apply(*case-tac a*, *auto simp*: *abc-inst-shift.simps*)
done

lemma *abc-halt-point-step*[*simp*]:
⟦*a* < *length bp*; *abc-steps-l* (*a*, *b*) *bp* (*Suc 0*) = (*length bp*, *lm′*)⟧
⟹ *abc-steps-l* (*length ap* + *a*, *b*) (*ap* [+] *bp* [+] *cp*) (*Suc 0*) =
(*length ap* + *length bp*, *lm′*)
apply(*simp add*: *abc-steps-l.simps abc-fetch.simps abc-append-nth*)
apply(*case-tac bp* ! *a*,
auto simp: *abc-steps-l.simps abc-step-l.simps*)
done

lemma *abc-step-state-in*:
⟦*bs* < *length bp*; *abc-steps-l* (*a*, *b*) *bp* (*Suc 0*) = (*bs*, *l*)⟧
⟹ *a* < *length bp*
apply(*simp add*: *abc-steps-l.simps abc-fetch.simps*)
apply(*rule-tac classical*,
simp add: *abc-step-l.simps abc-steps-l.simps*)
done

lemma *abc-append-state-in-exc*:
⟦*bs* < *length bp*; *abc-steps-l* (*0*, *lm*) *bp stpa* = (*bs*, *l*)⟧
⟹ *abc-steps-l* (*length ap*, *lm*) (*ap* [+] *bp* [+] *cp*) *stpa* =
(*length ap* + *bs*, *l*)
apply(*induct stpa arbitrary*: *bs l*, *simp add*: *abc-steps-zero*)
proof −
fix *stpa bs l*
assume *ind*:
⋀*bs l*. ⟦*bs* < *length bp*; *abc-steps-l* (*0*, *lm*) *bp stpa* = (*bs*, *l*)⟧
⟹ *abc-steps-l* (*length ap*, *lm*) (*ap* [+] *bp* [+] *cp*) *stpa* =
(*length ap* + *bs*, *l*)
and *h*: *bs* < *length bp*
abc-steps-l (*0*, *lm*) *bp* (*Suc stpa*) = (*bs*, *l*)
from *h* **show**
abc-steps-l (*length ap*, *lm*) (*ap* [+] *bp* [+] *cp*) (*Suc stpa*) =
(*length ap* + *bs*, *l*)
apply(*simp add*: *abc-steps-ind*)
apply(*case-tac* (*abc-steps-l* (*0*, *lm*) *bp stpa*), *simp*)
proof −
fix *a b*
assume *g*: *abc-steps-l* (*0*, *lm*) *bp stpa* = (*a*, *b*)
abc-steps-l (*a*, *b*) *bp* (*Suc 0*) = (*bs*, *l*)
from *h* **and** *g* **have** *k1*: *a* < *length bp*
apply(*simp add*: *abc-step-state-in*)

```
      done
    from h and g and k1 show
   abc-steps-l (abc-steps-l (length ap, lm) (ap [+] bp [+] cp) stpa)
            (ap [+] bp [+] cp) (Suc 0) = (length ap + bs, l)
      apply(insert ind[of a b], simp)
      apply(simp add: abc-steps-l.simps abc-fetch.simps
                  abc-append-nth)
      apply(case-tac bp ! a, auto simp:
                           abc-steps-l.simps abc-step-l.simps)
      done
  qed
qed

lemma [simp]: abc-steps-l (0, am) [] stp = (0, am)
apply(induct stp, simp add: abc-steps-zero)
apply(simp add: abc-steps-ind)
apply(simp add: abc-steps-zero abc-steps-l.simps
            abc-fetch.simps abc-step-l.simps)
done

lemma abc-append-exc1:
  ⟦∃ stp. abc-steps-l (0, lm) bp stp = (bs, lm′);
    bs = length bp;
    as = length ap⟧
    ⟹ ∃ stp. abc-steps-l (as, lm) (ap [+] bp [+] cp) stp
                                        = (as + bs, lm′)
apply(case-tac bp = [], erule-tac exE, simp,
      rule-tac x = 0 in exI, simp add: abc-steps-zero)
apply(frule-tac abc-halt-point-ex, simp, simp,
      erule-tac exE, erule-tac exE)
apply(rule-tac x = stpa + Suc 0 in exI)
apply(case-tac (abc-steps-l (0, lm) bp stpa),
      simp add: abc-steps-ind)
apply(subgoal-tac
  abc-steps-l (length ap, lm) (ap [+] bp [+] cp) stpa
                              = (length ap + a, b), simp)
apply(simp add: abc-steps-zero)
apply(rule-tac abc-append-state-in-exc, simp, simp)
done

lemma abc-append-exc3:
  ⟦∃ stp. abc-steps-l (0, am) bp stp = (bs, bm); ss = length ap⟧
   ⟹  ∃ stp. abc-steps-l (ss, am) (ap [+] bp) stp = (bs + ss, bm)
apply(erule-tac exE)
proof −
  fix stp
  assume h: abc-steps-l (0, am) bp stp = (bs, bm) ss = length ap
  thus ∃stp. abc-steps-l (ss, am) (ap [+] bp) stp = (bs + ss, bm)
  proof(induct stp arbitrary: bs bm)
```

```
    fix bs bm
    assume abc-steps-l (0, am) bp 0 = (bs, bm)
    thus ∃ stp. abc-steps-l (ss, am) (ap [+] bp) stp = (bs + ss, bm)
      apply(rule-tac x = 0 in exI, simp add: abc-steps-l.simps)
      done
  next
    fix stp bs bm
    assume ind:
      ⋀bs bm. ⟦abc-steps-l (0, am) bp stp = (bs, bm);
                ss = length ap⟧ ⟹
          ∃ stp. abc-steps-l (ss, am) (ap [+] bp) stp = (bs + ss, bm)
    and g: abc-steps-l (0, am) bp (Suc stp) = (bs, bm)
    from g show
      ∃ stp. abc-steps-l (ss, am) (ap [+] bp) stp = (bs + ss, bm)
      apply(insert abc-steps-add[of 0 am bp stp Suc 0], simp)
      apply(case-tac (abc-steps-l (0, am) bp stp), simp)
    proof −
      fix a b
      assume (bs, bm) = abc-steps-l (a, b) bp (Suc 0)
             abc-steps-l (0, am) bp (Suc stp) =
                       abc-steps-l (a, b) bp (Suc 0)
              abc-steps-l (0, am) bp stp = (a, b)
      thus ?thesis
 apply(insert ind[of a b], simp add: h, erule-tac exE)
 apply(rule-tac x = Suc stp in exI)
 apply(simp only: abc-steps-ind, simp add: abc-steps-zero)
      proof −
 fix stp
 assume (bs, bm) = abc-steps-l (a, b) bp (Suc 0)
 thus abc-steps-l (a + length ap, b) (ap [+] bp) (Suc 0)
                                         = (bs + length ap, bm)
   apply(simp add: abc-steps-l.simps abc-steps-zero
                        abc-fetch.simps split: if-splits)
   apply(case-tac bp ! a,
                simp-all add: abc-inst-shift.simps abc-append-nth
                   abc-steps-l.simps abc-steps-zero abc-step-l.simps)
   apply(auto)
   done
      qed
    qed
  qed
qed

lemma abc-add-equal:
  ⟦ap ≠ [];
    abc-steps-l (0, am) ap astp = (a, b);
    a < length ap⟧
     ⟹ (abc-steps-l (0, am) (ap @ bp) astp) = (a, b)
apply(induct astp arbitrary: a b, simp add: abc-steps-l.simps, simp)
```

apply(*simp add*: *abc-steps-ind*)
apply(*case-tac* (*abc-steps-l* (*0*, *am*) *ap astp*))
proof −
  fix *astp a b aa ba*
  assume *ind*:
    ⋀*a b*. ⟦*abc-steps-l* (*0*, *am*) *ap astp* = (*a*, *b*);
          a < *length ap*⟧ ⟹
              abc-steps-l (*0*, *am*) (*ap* @ *bp*) *astp* = (*a*, *b*)
  and *h*: *abc-steps-l* (*abc-steps-l* (*0*, *am*) *ap astp*) *ap* (*Suc 0*)
                                                    = (*a*, *b*)
      a < *length ap*
      abc-steps-l (*0*, *am*) *ap astp* = (*aa*, *ba*)
  from *h* **show** *abc-steps-l* (*abc-steps-l* (*0*, *am*) (*ap* @ *bp*) *astp*)
                              (*ap* @ *bp*) (*Suc 0*) = (*a*, *b*)
    apply(*insert ind*[*of aa ba*], *simp*)
    apply(*subgoal-tac aa* < *length ap*, *simp*)
    apply(*simp add*: *abc-steps-l.simps abc-fetch.simps*
              nth-append abc-steps-zero)
    apply(*rule abc-step-state-in*, *auto*)
    done
qed

lemma *abc-add-exc1*:
  ⟦∃ *astp*. *abc-steps-l* (*0*, *am*) *ap astp* = (*as*, *bm*); *as* = *length ap*⟧
  ⟹ ∃ *stp*. *abc-steps-l* (*0*, *am*) (*ap* @ *bp*) *stp* = (*as*, *bm*)
apply(*case-tac ap* = [], *simp*,
    rule-tac x = *0* **in** *exI*, *simp add*: *abc-steps-zero*)
apply(*drule-tac abc-halt-point-ex*, *simp*, *simp*)
apply(*erule-tac exE*, *case-tac* (*abc-steps-l* (*0*, *am*) *ap astp*), *simp*)
apply(*rule-tac x* = *Suc astp* **in** *exI*, *simp add*: *abc-steps-ind*, *auto*)
apply(*frule-tac bp* = *bp* **in** *abc-add-equal*, *simp*, *simp*, *simp*)
apply(*simp add*: *abc-steps-l.simps abc-steps-zero*
        abc-fetch.simps nth-append)
done

declare *abc-shift.simps*[*simp del*]

lemma *abc-append-exc2*:
  ⟦∃ *astp*. *abc-steps-l* (*0*, *am*) *ap astp* = (*as*, *bm*); *as* = *length ap*;
    ∃ *bstp*. *abc-steps-l* (*0*, *bm*) *bp bstp* = (*bs*, *bm′*); *bs* = *length bp*;
    cs = *as* + *bs*; *bp* ≠ []⟧
  ⟹ ∃ *stp*. *abc-steps-l* (*0*, *am*) (*ap* [+] *bp*) *stp* = (*cs*, *bm′*)
apply(*insert abc-append-exc1*[*of bm bp bs bm′ as ap* []], *simp*)
apply(*drule-tac bp* = *abc-shift bp* (*length ap*) **in** *abc-add-exc1*, *simp*)
apply(*subgoal-tac ap* @ *abc-shift bp* (*length ap*) = *ap* [+] *bp*,
    simp, *auto*)
apply(*rule-tac x* = *stpa* + *stp* **in** *exI*, *simp add*: *abc-steps-add*)
apply(*simp add*: *abc-append.simps*)

done
lemma *exp-length*[*simp*]: *length* $(a^b) = b$
by(*simp add*: *exponent-def*)
lemma *exponent-add-iff*: a^b @ a^c @ *xs* = $a^{b\,+\,c}$ @ *xs*
apply(*auto simp*: *exponent-def replicate-add*)
done
lemma *exponent-cons-iff*: *a* # a^c @ *xs* = $a^{Suc\ c}$ @ *xs*
apply(*auto simp*: *exponent-def replicate-add*)
done

lemma [*simp*]: *length lm* = *n* ⟹
abc-steps-l (*Suc 0*, *lm* @ *Suc x* # *0* # *suf-lm*)
[*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)] (*Suc* (*Suc 0*))
= (*3*, *lm* @ *Suc x* # *0* # *suf-lm*)
apply(*simp add*: *abc-steps-l.simps abc-fetch.simps*
abc-step-l.simps abc-lm-v.simps abc-lm-s.simps
nth-append list-update-append)
done

lemma [*simp*]:
length lm = *n* ⟹
abc-steps-l (*Suc 0*, *lm* @ *Suc x* # *Suc y* # *suf-lm*)
[*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)] (*Suc* (*Suc 0*))
= (*Suc 0*, *lm* @ *Suc x* # *y* # *suf-lm*)
apply(*simp add*: *abc-steps-l.simps abc-fetch.simps*
abc-step-l.simps abc-lm-v.simps abc-lm-s.simps
nth-append list-update-append)
done

lemma *pr-cycle-part-middle-inv*:
⟦*length lm* = *n*⟧ ⟹
∃ *stp*. *abc-steps-l* (*0*, *lm* @ *x* # *y* # *suf-lm*)
[*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)] *stp*
= (*3*, *lm* @ *Suc x* # *0* # *suf-lm*)
proof −
assume *h*: *length lm* = *n*
hence *k1*: ∃ *stp*. *abc-steps-l* (*0*, *lm* @ *x* # *y* # *suf-lm*)
[*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)] *stp*
= (*Suc 0*, *lm* @ *Suc x* # *y* # *suf-lm*)
apply(*rule-tac x* = *Suc 0* **in** *exI*)
apply(*simp add*: *abc-steps-l.simps abc-step-l.simps*
abc-lm-v.simps abc-lm-s.simps nth-append
list-update-append abc-fetch.simps)
done
from *h* **have** *k2*:
∃ *stp*. *abc-steps-l* (*Suc 0*, *lm* @ *Suc x* # *y* # *suf-lm*)
[*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)] *stp*
= (*3*, *lm* @ *Suc x* # *0* # *suf-lm*)

```
  apply(induct y)
  apply(rule-tac x = Suc (Suc 0) in exI, simp, simp,
      erule-tac exE)
  apply(rule-tac x = Suc (Suc 0) + stp in exI,
      simp only: abc-steps-add, simp)
  done
 from k1 and k2 show
  ∃ stp. abc-steps-l (0, lm @ x # y # suf-lm)
                 [Inc n, Dec (Suc n) 3, Goto (Suc 0)] stp
  = (3, lm @ Suc x # 0 # suf-lm)
  apply(erule-tac exE, erule-tac exE)
  apply(rule-tac x = stp + stpa in exI, simp add: abc-steps-add)
  done
qed

lemma [simp]:
 length lm = Suc n ⟹
 (abc-steps-l (length ap, lm @ x # Suc y # suf-lm)
        (ap @ [Dec (Suc (Suc n)) 0, Inc (Suc n), Goto (length ap)])
             (Suc (Suc (Suc 0))))
 = (length ap, lm @ Suc x # y # suf-lm)
apply(simp add: abc-steps-l.simps abc-fetch.simps abc-step-l.simps
      abc-lm-v.simps list-update-append nth-append abc-lm-s.simps)
done

lemma switch-para-inv:
 assumes bp-def:bp =  ap @ [Dec (Suc (Suc n)) 0, Inc (Suc n), Goto ss]
 and h: rec-ci (Pr n f g) = (aprog, rs-pos, a-md)
      ss = length ap
      length lm = Suc n
 shows  ∃stp. abc-steps-l (ss, lm @ x # y # suf-lm) bp stp =
                    (0, lm @ (x + y) # 0 # suf-lm)
apply(induct y arbitrary: x)
apply(rule-tac x = Suc 0 in exI,
 simp add: bp-def empty.simps abc-steps-l.simps
        abc-fetch.simps h abc-step-l.simps
        abc-lm-v.simps list-update-append nth-append
        abc-lm-s.simps)
proof -
 fix y x
 assume ind:
  ⋀x. ∃stp. abc-steps-l (ss, lm @ x # y # suf-lm) bp stp =
                      (0, lm @ (x + y) # 0 # suf-lm)
 show ∃stp. abc-steps-l (ss, lm @ x # Suc y # suf-lm) bp stp =
                    (0, lm @ (x + Suc y) # 0 # suf-lm)
  apply(insert ind[of Suc x], erule-tac exE)
  apply(rule-tac x = Suc (Suc (Suc 0)) + stp in exI,
      simp only: abc-steps-add bp-def h)
  apply(simp add: h)
```

done
qed

lemma [*simp*]:
$length\ lm = rs\text{-}pos \wedge Suc\ (Suc\ rs\text{-}pos) < a\text{-}md \wedge 0 < rs\text{-}pos \Longrightarrow a\text{-}md - Suc\ 0 < Suc\ (Suc\ (Suc\ (a\text{-}md + length\ suf\text{-}lm - Suc\ (Suc\ (Suc\ 0))))))$
apply(*arith*)
done

lemma [*simp*]:
$Suc\ (Suc\ rs\text{-}pos) < a\text{-}md \wedge 0 < rs\text{-}pos \Longrightarrow \neg\ a\text{-}md - Suc\ 0 < rs\text{-}pos - Suc\ 0$
apply(*arith*)
done

lemma [*simp*]:
$Suc\ (Suc\ rs\text{-}pos) < a\text{-}md \wedge 0 < rs\text{-}pos \Longrightarrow \neg\ a\text{-}md - rs\text{-}pos < Suc\ (Suc\ (a\text{-}md - Suc\ (Suc\ rs\text{-}pos)))$
apply(*arith*)
done

lemma *butlast-append-last*: $lm \neq [] \Longrightarrow lm = butlast\ lm\ @\ [last\ lm]$
apply(*auto*)
done

lemma [*simp*]: $rec\text{-}ci\ (Pr\ n\ f\ g) = (aprog,\ rs\text{-}pos,\ a\text{-}md) \Longrightarrow (Suc\ (Suc\ rs\text{-}pos)) < a\text{-}md$
apply(*simp add*: *rec-ci.simps*)
apply(*case-tac rec-ci f*, *simp*)
apply(*case-tac rec-ci g*, *simp*)
apply(*arith*)
done

lemma *ci-pr-para-eq*: $rec\text{-}ci\ (Pr\ n\ f\ g) = (aprog,\ rs\text{-}pos,\ a\text{-}md) \Longrightarrow rs\text{-}pos = Suc\ n$
apply(*simp add*: *rec-ci.simps*)
apply(*case-tac rec-ci g*, *case-tac rec-ci f*, *simp*)
done

lemma [*intro*]:
$[\![rec\text{-}ci\ z = (aprog,\ rs\text{-}pos,\ a\text{-}md);\ rec\text{-}calc\text{-}rel\ z\ lm\ xs]\!] \Longrightarrow length\ lm = rs\text{-}pos$
apply(*simp add*: *rec-ci.simps rec-ci-z-def*)
apply(*erule-tac calc-z-reverse*, *simp*)
done

lemma [*intro*]:
⟦*rec-ci s* = (*aprog*, *rs-pos*, *a-md*); *rec-calc-rel s lm xs*⟧
⟹ *length lm* = *rs-pos*
apply(*simp add*: *rec-ci.simps rec-ci-s-def*)
apply(*erule-tac calc-s-reverse*, *simp*)
done

lemma [*intro*]:
⟦*rec-ci* (*recf.id nat1 nat2*) = (*aprog*, *rs-pos*, *a-md*);
rec-calc-rel (*recf.id nat1 nat2*) *lm xs*⟧ ⟹ *length lm* = *rs-pos*
apply(*simp add*: *rec-ci.simps rec-ci-id.simps*)
apply(*erule-tac calc-id-reverse*, *simp*)
done

lemma [*intro*]:
⟦*rec-ci* (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*);
rec-calc-rel (*Cn n f gs*) *lm xs*⟧ ⟹ *length lm* = *rs-pos*
apply(*erule-tac calc-cn-reverse*, *simp*)
apply(*simp add*: *rec-ci.simps*)
apply(*case-tac rec-ci f*, *simp*)
done

lemma [*intro*]:
⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*);
rec-calc-rel (*Pr n f g*) *lm xs*⟧ ⟹ *length lm* = *rs-pos*
apply(*erule-tac calc-pr-reverse*, *simp*)
apply(*drule-tac ci-pr-para-eq*, *simp*, *simp*)
apply(*drule-tac ci-pr-para-eq*, *simp*)
done

lemma [*intro*]:
⟦*rec-ci* (*Mn n f*) = (*aprog*, *rs-pos*, *a-md*);
rec-calc-rel (*Mn n f*) *lm xs*⟧ ⟹ *length lm* = *rs-pos*
apply(*erule-tac calc-mn-reverse*)
apply(*simp add*: *rec-ci.simps*)
apply(*case-tac rec-ci f*, *simp*)
done

lemma *para-pattern*:
⟦*rec-ci f* = (*aprog*, *rs-pos*, *a-md*); *rec-calc-rel f lm xs*⟧
⟹ *length lm* = *rs-pos*
apply(*case-tac f*, *auto*)
done

lemma *ci-pr-g-paras*:
⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*);
rec-ci g = (*a*, *aa*, *ba*);
rec-calc-rel (*Pr n f g*) (*lm* @ [*x*]) *rs*; *x* > *0*⟧ ⟹
aa = *Suc rs-pos*

apply(*erule calc-pr-reverse*, *simp*)
apply(*subgoal-tac length* (*args* @ [*k*, *rk*]) = *aa*, *simp*)
apply(*subgoal-tac rs-pos* = *Suc n*, *simp*)
apply(*simp add*: *ci-pr-para-eq*)
apply(*erule para-pattern*, *simp*)
done

lemma *ci-pr-g-md-less*:
⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*);
rec-ci g = (*a*, *aa*, *ba*)⟧ ⟹ *ba* < *a-md*
apply(*simp add*: *rec-ci.simps*)
apply(*case-tac rec-ci f*, *auto*)
done

lemma [*intro*]: *rec-ci z* = (*ap*, *rp*, *ad*) ⟹ *rp* < *ad*
by(*simp add*: *rec-ci.simps*)

lemma [*intro*]: *rec-ci s* = (*ap*, *rp*, *ad*) ⟹ *rp* < *ad*
by(*simp add*: *rec-ci.simps*)

lemma [*intro*]: *rec-ci* (*recf.id nat1 nat2*) = (*ap*, *rp*, *ad*) ⟹ *rp* < *ad*
by(*simp add*: *rec-ci.simps*)

lemma [*intro*]: *rec-ci* (*Cn n f gs*) = (*ap*, *rp*, *ad*) ⟹ *rp* < *ad*
apply(*simp add*: *rec-ci.simps*)
apply(*case-tac rec-ci f*, *simp*)
done

lemma [*intro*]: *rec-ci* (*Pr n f g*) = (*ap*, *rp*, *ad*) ⟹ *rp* < *ad*
apply(*simp add*: *rec-ci.simps*)
by(*case-tac rec-ci f*, *case-tac rec-ci g*, *auto*)

lemma [*intro*]: *rec-ci* (*Mn n f*) = (*ap*, *rp*, *ad*) ⟹ *rp* < *ad*
apply(*simp add*: *rec-ci.simps*)
apply(*case-tac rec-ci f*, *simp*)
apply(*arith*)
done

lemma *ci-ad-ge-paras*: *rec-ci f* = (*ap*, *rp*, *ad*) ⟹ *ad* > *rp*
apply(*case-tac f*, *auto*)
done

lemma [*elim*]: ⟦*a* [+] *b* = []; *a* ≠ [] ∨ *b* ≠ []⟧ ⟹ *RR*
apply(*auto simp*: *abc-append.simps abc-shift.simps*)
done

lemma [*intro*]: *rec-ci z* = ([], *aa*, *ba*) ⟹ *False*
by(*simp add*: *rec-ci.simps rec-ci-z-def*)

lemma [*intro*]: *rec-ci s* = ([], *aa*, *ba*) $\Longrightarrow$ *False*
by(*auto simp*: *rec-ci.simps rec-ci-s-def addition.simps*)

lemma [*intro*]: *rec-ci* (*id m n*) = ([], *aa*, *ba*) $\Longrightarrow$ *False*
by(*auto simp*: *rec-ci.simps rec-ci-id.simps addition.simps*)

lemma [*intro*]: *rec-ci* (*Cn n f gs*) = ([], *aa*, *ba*) $\Longrightarrow$ *False*
apply(*case-tac rec-ci f*, *auto simp*: *rec-ci.simps abc-append.simps*)
apply(*simp add*: *abc-shift.simps empty.simps*)
done

lemma [*intro*]: *rec-ci* (*Pr n f g*) = ([], *aa*, *ba*) $\Longrightarrow$ *False*
apply(*simp add*: *rec-ci.simps*)
apply(*case-tac rec-ci f*, *case-tac rec-ci g*)
by(*auto*)

lemma [*intro*]: *rec-ci* (*Mn n f*) = ([], *aa*, *ba*) $\Longrightarrow$ *False*
apply(*case-tac rec-ci f*, *auto simp*: *rec-ci.simps*)
done

lemma *rec-ci-not-null*: *rec-ci g* = (*a*, *aa*, *ba*) $\Longrightarrow$ *a* $\neq$ []
by(*case-tac g*, *auto*)

lemma *calc-pr-g-def*:
⟦*rec-calc-rel* (*Pr rs-pos f g*) (*lm* @ [*Suc x*]) *rsa*;
  rec-calc-rel (*Pr rs-pos f g*) (*lm* @ [*x*]) *rsxa*⟧
$\Longrightarrow$ *rec-calc-rel g* (*lm* @ [*x*, *rsxa*]) *rsa*
apply(*erule-tac calc-pr-reverse*, *simp*, *simp*)
apply(*subgoal-tac rsxa* = *rk*, *simp*)
apply(*erule-tac rec-calc-inj*, *auto*)
done

lemma *ci-pr-md-def*:
⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*);
  rec-ci g = (*a*, *aa*, *ba*); *rec-ci f* = (*ab*, *ac*, *bc*)⟧
$\Longrightarrow$ *a-md* = *Suc* (*max* (*n* + *3*) (*max bc ba*))
by(*simp add*: *rec-ci.simps*)

lemma *ci-pr-f-paras*:
⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*);
  rec-calc-rel (*Pr n f g*) *lm rs*;
  rec-ci f = (*ab*, *ac*, *bc*)⟧ $\Longrightarrow$ *ac* = *rs-pos* − *Suc 0*
apply(*subgoal-tac* ∃ *rs*. *rec-calc-rel f* (*butlast lm*) *rs*,
  erule-tac exE)
apply(*drule-tac f* = *f* **and** *lm* = *butlast lm* **in** *para-pattern*,
  simp, *simp*)
apply(*drule-tac para-pattern*, *simp*)
apply(*subgoal-tac lm* $\neq$ [], *simp*)
apply(*erule-tac calc-pr-reverse*, *simp*, *simp*)

apply(*erule calc-pr-zero-ex*)
done

lemma *ci-pr-md-ge-f*: ⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*);
rec-ci f = (*ab*, *ac*, *bc*)⟧ ⟹ *Suc bc* ≤ *a-md*
apply(*case-tac rec-ci g*)
apply(*simp add*: *rec-ci.simps*, *auto*)
done

lemma *ci-pr-md-ge-g*: ⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*);
rec-ci g = (*ab*, *ac*, *bc*)⟧ ⟹ *bc* < *a-md*
apply(*case-tac rec-ci f*)
apply(*simp add*: *rec-ci.simps*, *auto*)
done

lemma *rec-calc-rel-def0*:
⟦*rec-calc-rel* (*Pr n f g*) *lm rs*; *rec-calc-rel f* (*butlast lm*) *rsa*⟧
⟹ *rec-calc-rel* (*Pr n f g*) (*butlast lm* @ [*0*]) *rsa*
apply(*rule-tac calc-pr-zero*, *simp*)
apply(*erule-tac calc-pr-reverse*, *simp*, *simp*, *simp*)
done

lemma [*simp*]: *length* (*empty m n*) = *3*
by (*auto simp*: *empty.simps*)

lemma [*simp*]: ⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*); *rec-calc-rel* (*Pr n f g*)
lm rs⟧
⟹ *rs-pos* = *Suc n*
apply(*simp add*: *ci-pr-para-eq*)
done

lemma [*simp*]: ⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*); *rec-calc-rel* (*Pr n f g*)
lm rs⟧
⟹ *length lm* = *Suc n*
apply(*subgoal-tac rs-pos* = *Suc n*, *rule-tac para-pattern*, *simp*, *simp*)
apply(*case-tac rec-ci f*, *case-tac rec-ci g*, *simp add*: *rec-ci.simps*)
done

lemma [*simp*]: *rec-ci* (*Pr n f g*) = (*a*, *rs-pos*, *a-md*) ⟹ *Suc* (*Suc n*) < *a-md*
apply(*case-tac rec-ci f*, *case-tac rec-ci g*, *simp add*: *rec-ci.simps*)
apply *arith*
done

lemma [*simp*]: *rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*) ⟹ *0* < *rs-pos*
apply(*case-tac rec-ci f*, *case-tac rec-ci g*)
apply(*simp add*: *rec-ci.simps*)
done

lemma [*simp*]: *Suc* (*Suc rs-pos*) < *a-md* ⟹
butlast lm @ (*last lm* − *xa*) # (*rsa*::*nat*) # *0* # $0^{a\text{-}md\ -\ Suc\ (Suc\ (Suc\ rs\text{-}pos))}$
@ *xa* # *suf-lm* =
butlast lm @ (*last lm* − *xa*) # *rsa* # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *xa* # *suf-lm*
apply(*simp add*: *exp-ind-def* [*THEN sym*])
done

lemma *pr-cycle-part-ind*:
assumes *g-ind*:
⋀*lm rs suf-lm*. *rec-calc-rel g lm rs* ⟹
∃ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{ba\ -\ aa}$ @ *suf-lm*) *a stp* =
(*length a*, *lm* @ *rs* # $0^{ba\ -\ Suc\ aa}$ @ *suf-lm*)
and *ap-def*:
ap = ([*Dec* (*a-md* − *Suc 0*) (*length a* + *7*)] [+]
(*a* [+] [*Inc* (*rs-pos* − *Suc 0*), *Dec rs-pos 3*, *Goto* (*Suc 0*)])) @
[*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length a* + *4*)]
and *h*: *rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*)
rec-calc-rel (*Pr n f g*)
(*butlast lm* @ [*last lm* − *Suc xa*]) *rsxa*
Suc xa ≤ *last lm*
rec-ci g = (*a*, *aa*, *ba*)
rec-calc-rel (*Pr n f g*) (*butlast lm* @ [*last lm* − *xa*]) *rsa*
lm ≠ []
shows
∃ *stp*. *abc-steps-l*
(*0*, *butlast lm* @ (*last lm* − *Suc xa*) # *rsxa* #
$0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *Suc xa* # *suf-lm*) *ap stp* =
(*0*, *butlast lm* @ (*last lm* − *xa*) # *rsa*
$0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *xa* # *suf-lm*)
proof −
have *k1*: ∃ *stp*. *abc-steps-l* (*0*, *butlast lm* @ (*last lm* − *Suc xa*) #
rsxa # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *Suc xa* # *suf-lm*) *ap stp* =
(*length a* + *4*, *butlast lm* @ (*last lm* − *xa*) # *0* # *rsa* #
$0^{a\text{-}md\ -\ Suc\ (Suc\ (Suc\ rs\text{-}pos))}$ @ *xa* # *suf-lm*)
apply(*simp add*: *ap-def*, *rule-tac abc-add-exc1*)
apply(*rule-tac as* = *Suc 0* **and**
bm = *butlast lm* @ (*last lm* − *Suc xa*) #
rsxa # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *xa* # *suf-lm* **in** *abc-append-exc2*,
auto)
proof −
show
∃ *astp*. *abc-steps-l* (*0*, *butlast lm* @ (*last lm* − *Suc xa*) # *rsxa*
$0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *Suc xa* # *suf-lm*)
[*Dec* (*a-md* − *Suc 0*)(*length a* + *7*)] *astp* =
(*Suc 0*, *butlast lm* @ (*last lm* − *Suc xa*) #
rsxa # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *xa* # *suf-lm*)
apply(*rule-tac x* = *Suc 0* **in** *exI*,

simp add: *abc-steps-l.simps abc-step-l.simps*
abc-fetch.simps)
apply(*subgoal-tac length lm* = *Suc n* ∧ *rs-pos* = *Suc n* ∧
a-md > *Suc* (*Suc rs-pos*))
apply(*simp add*: *abc-lm-v.simps nth-append abc-lm-s.simps*)
apply(*insert nth-append*[*of*
(*last lm* − *Suc xa*) # *rsxa* # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$
Suc xa # *suf-lm* (*a-md* − *rs-pos*)], *simp*)
apply(*simp add*: *list-update-append del*: *list-update.simps*)
apply(*insert list-update-append*[*of* (*last lm* − *Suc xa*) # *rsxa* #
$0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$
Suc xa # *suf-lm a-md* − *rs-pos xa*], *simp*)
apply(*case-tac a-md*, *simp*, *simp*)
apply(*insert h*, *simp*)
apply(*insert para-pattern*[*of Pr n f g aprog rs-pos a-md*
(*butlast lm* @ [*last lm* − *Suc xa*]) *rsxa*], *simp*)
done
next
show ∃ *bstp*. *abc-steps-l* (*0*, *butlast lm* @ (*last lm* − *Suc xa*) #
rsxa # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *xa* # *suf-lm*) (*a* [+]
[*Inc* (*rs-pos* − *Suc 0*), *Dec rs-pos 3*, *Goto* (*Suc 0*)]) *bstp* =
(*3* + *length a*, *butlast lm* @ (*last lm* − *xa*) # *0* # *rsa* #
$0^{a\text{-}md\ -\ Suc\ (Suc\ (Suc\ rs\text{-}pos))}$ @ *xa* # *suf-lm*)
apply(*rule-tac as* = *length a* **and**
bm = *butlast lm* @ (*last lm* − *Suc xa*) # *rsxa* # *rsa* #
$0^{a\text{-}md\ -\ Suc\ (Suc\ (Suc\ rs\text{-}pos))}$ @ *xa* # *suf-lm*
in *abc-append-exc2*, *simp-all*)
proof −
from *h* **have** *j1*: *aa* = *Suc rs-pos* ∧ *a-md* > *ba* ∧ *ba* > *Suc rs-pos*
apply(*insert h*)
apply(*insert ci-pr-g-paras*[*of n f g aprog rs-pos*
a-md a aa ba butlast lm last lm − *xa rsa*], *simp*)
apply(*drule-tac ci-pr-md-ge-g*, *auto*)
apply(*erule-tac ci-ad-ge-paras*)
done
from *h* **have** *j2*: *rec-calc-rel g* (*butlast lm* @
[*last lm* − *Suc xa*, *rsxa*]) *rsa*
apply(*rule-tac calc-pr-g-def*, *simp*, *simp*)
done
from *j1* **and** *j2* **show**
∃ *astp*. *abc-steps-l* (*0*, *butlast lm* @ (*last lm* − *Suc xa*) #
rsxa # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *xa* # *suf-lm*) *a astp* =
(*length a*, *butlast lm* @ (*last lm* − *Suc xa*) # *rsxa* # *rsa*
$0^{a\text{-}md\ -\ Suc\ (Suc\ (Suc\ rs\text{-}pos))}$ @ *xa* # *suf-lm*)
apply(*insert g-ind*[*of*
butlast lm @ (*last lm* − *Suc xa*) # [*rsxa*] *rsa*
$0^{a\text{-}md\ -\ ba\ -\ Suc\ 0}$ @ *xa* # *suf-lm*], *simp*, *auto*)
apply(*simp add*: *exponent-add-iff*)

```
 apply(rule-tac x = stp in exI, simp add: numeral-3-eq-3)
 done
   next
     from h have j3: length lm = rs-pos ∧ rs-pos > 0
 apply(rule-tac conjI)
 apply(drule-tac lm = (butlast lm @ [last lm − Suc xa])
                     and xs = rsxa in para-pattern, simp, simp, simp)
       done
     from h have j4: Suc (last lm − Suc xa) = last lm − xa
 apply(case-tac last lm, simp, simp)
 done
     from j3 and j4 show
     ∃ bstp. abc-steps-l (0, butlast lm @ (last lm − Suc xa) # rsxa #
                rsa # 0^(a-md − Suc (Suc (Suc rs-pos))) @ xa # suf-lm)
          [Inc (rs-pos − Suc 0), Dec rs-pos 3, Goto (Suc 0)] bstp =
       (3, butlast lm @ (last lm − xa) # 0 # rsa #
                0^(a-md − Suc (Suc (Suc rs-pos))) @ xa # suf-lm)
 apply(insert pr-cycle-part-middle-inv[of butlast lm
         rs-pos − Suc 0 (last lm − Suc xa) rsxa
         rsa # 0^(a-md − Suc (Suc (Suc rs-pos))) @ xa # suf-lm], simp)
 done
   qed
  qed
  from h have k2:
    ∃ stp. abc-steps-l (length a + 4, butlast lm @ (last lm − xa) # 0
          # rsa # 0^(a-md − Suc (Suc (Suc rs-pos))) @ xa # suf-lm) ap stp =
     (0, butlast lm @ (last lm − xa) # rsa # 0^(a-md − Suc (Suc rs-pos)) @ xa #
suf-lm)
   apply(insert switch-para-inv[of ap
     ([Dec (a-md − Suc 0) (length a + 7)] [+]
     (a [+] [Inc (rs-pos − Suc 0), Dec rs-pos 3, Goto (Suc 0)]))
     n length a + 4 f g aprog rs-pos a-md
     butlast lm @ [last lm − xa] 0 rsa
     0^(a-md − Suc (Suc (Suc rs-pos))) @ xa # suf-lm])
   apply(simp add: h ap-def)
   apply(subgoal-tac length lm = Suc n ∧ Suc (Suc rs-pos) < a-md,
         simp)
   apply(insert h, simp)
   apply(frule-tac lm = (butlast lm @ [last lm − Suc xa])
     and xs = rsxa in para-pattern, simp, simp)
   done
  from k1 and k2 show ?thesis
   apply(auto)
   apply(rule-tac x = stp + stpa in exI, simp add: abc-steps-add)
   done
qed

lemma ci-pr-ex1:
```

⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*);
  rec-ci g = (*a*, *aa*, *ba*);
  rec-ci f = (*ab*, *ac*, *bc*)⟧
⟹ ∃ *ap bp*. *length ap* = *6* + *length ab* ∧
  aprog = *ap* [+] *bp* ∧
  bp = ([*Dec* (*a-md* − *Suc 0*) (*length a* + *7*)] [+] (*a* [+]
    [*Inc* (*rs-pos* − *Suc 0*), *Dec rs-pos 3*, *Goto* (*Suc 0*)])) @
    [*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length a* + *4*)]
apply(*simp add*: *rec-ci.simps*)
apply(*rule-tac x* = *recursive.empty n* (*max* (*Suc* (*Suc* (*Suc n*)))
  (*max bc ba*)) [+] *ab* [+] *recursive.empty n* (*Suc n*) **in** *exI*,
  simp)
apply(*auto simp add*: *abc-append-commute add3-Suc*)
done

lemma *pr-cycle-part*:
⟦⋀*lm rs suf-lm*. *rec-calc-rel g lm rs* ⟹
  ∃ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{ba\ -\ aa}$ @ *suf-lm*) *a stp* =
    (*length a*, *lm* @ *rs* # $0^{ba\ -\ Suc\ aa}$ @ *suf-lm*);
rec-ci (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*);
rec-calc-rel (*Pr n f g*) *lm rs*;
rec-ci g = (*a*, *aa*, *ba*);
rec-calc-rel (*Pr n f g*) (*butlast lm* @ [*last lm* − *x*]) *rsx*;
rec-ci f = (*ab*, *ac*, *bc*);
lm ≠ [];
x ≤ *last lm*⟧ ⟹
∃ *stp*. *abc-steps-l* (*6* + *length ab*, *butlast lm* @ (*last lm* − *x*) #
    rsx # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *x* # *suf-lm*) *aprog stp* =
(*6* + *length ab*, *butlast lm* @ *last lm* # *rs* #
    $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *0* # *suf-lm*)
proof −
assume *g-ind*:
  ⋀*lm rs suf-lm*. *rec-calc-rel g lm rs* ⟹
  ∃ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{ba\ -\ aa}$ @ *suf-lm*) *a stp* =
    (*length a*, *lm* @ *rs* # $0^{ba\ -\ Suc\ aa}$ @ *suf-lm*)
  and *h*: *rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*)
    rec-calc-rel (*Pr n f g*) *lm rs*
    rec-ci g = (*a*, *aa*, *ba*)
    rec-calc-rel (*Pr n f g*) (*butlast lm* @ [*last lm* − *x*]) *rsx*
    lm ≠ []
    x ≤ *last lm*
    rec-ci f = (*ab*, *ac*, *bc*)
from *h* **show**
  ∃ *stp*. *abc-steps-l* (*6* + *length ab*, *butlast lm* @ (*last lm* − *x*) #
    rsx # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *x* # *suf-lm*) *aprog stp* =
  (*6* + *length ab*, *butlast lm* @ *last lm* # *rs* #
    $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *0* # *suf-lm*)
proof(*induct x arbitrary*: *rsx*, *simp-all*)

fix *rsxa*
assume *rec-calc-rel* (*Pr n f g*) *lm rsxa*
rec-calc-rel (*Pr n f g*) *lm rs*
from *h* **and** *this* **have** *rs* = *rsxa*
apply(*subgoal-tac lm* ≠ [] ∧ *rs-pos* = *Suc n*, *simp*)
apply(*rule-tac rec-calc-inj*, *simp*, *simp*)
apply(*simp*)
done
thus ∃ *stp*. *abc-steps-l* (*6* + *length ab*, *butlast lm* @ *last lm* # *rsxa* # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *0* # *suf-lm*) *aprog stp* =
(*6* + *length ab*, *butlast lm* @ *last lm* # *rs* # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *0* # *suf-lm*)
by(*rule-tac x* = *0* **in** *exI*, *simp add*: *abc-steps-l.simps*)
next
fix *xa rsxa*
assume *ind*:
⋀*rsx*. *rec-calc-rel* (*Pr n f g*) (*butlast lm* @ [*last lm* − *xa*]) *rsx*
⟹ ∃ *stp*. *abc-steps-l* (*6* + *length ab*, *butlast lm* @ (*last lm* − *xa*) # *rsx* # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *xa* # *suf-lm*) *aprog stp* =
(*6* + *length ab*, *butlast lm* @ *last lm* # *rs* # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *0* # *suf-lm*)
and *g*: *rec-calc-rel* (*Pr n f g*)
(*butlast lm* @ [*last lm* − *Suc xa*]) *rsxa*
Suc xa ≤ *last lm*
rec-ci (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*)
rec-calc-rel (*Pr n f g*) *lm rs*
rec-ci g = (*a*, *aa*, *ba*)
rec-ci f = (*ab*, *ac*, *bc*) *lm* ≠ []
from *g* **have** *k1*:
∃ *rs*. *rec-calc-rel* (*Pr n f g*) (*butlast lm* @ [*last lm* − *xa*]) *rs*
apply(*rule-tac rs* = *rs* **in** *calc-pr-less-ex*, *simp*, *simp*)
done
from *g* **and** *this* **show**
∃ *stp*. *abc-steps-l* (*6* + *length ab*,
butlast lm @ (*last lm* − *Suc xa*) # *rsxa* #
$0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *Suc xa* # *suf-lm*) *aprog stp* =
(*6* + *length ab*, *butlast lm* @ *last lm* # *rs* #
$0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *0* # *suf-lm*)
proof(*erule-tac exE*)
fix *rsa*
assume *k2*: *rec-calc-rel* (*Pr n f g*)
(*butlast lm* @ [*last lm* − *xa*]) *rsa*
from *g* **and** *k2* **have**
∃ *stp*. *abc-steps-l* (*6* + *length ab*, *butlast lm* @
(*last lm* − *Suc xa*) # *rsxa* #
$0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *Suc xa* # *suf-lm*) *aprog stp*
= (*6* + *length ab*, *butlast lm* @ (*last lm* − *xa*) # *rsa* #
$0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *xa* # *suf-lm*)

proof −
from *g* **have** *k2-1*:
∃ *ap bp*. *length ap* = *6* + *length ab* ∧
aprog = *ap* [+] *bp* ∧
bp = ([*Dec* (*a-md* − *Suc 0*) (*length a* + *7*)] [+]
(*a* [+] [*Inc* (*rs-pos* − *Suc 0*), *Dec rs-pos 3*,
Goto (*Suc 0*)])) @
[*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length a* + *4*)]
apply(*rule-tac ci-pr-ex1*, *auto*)
done
from *k2-1* **and** *k2* **and** *g* **show** *?thesis*
proof(*erule-tac exE*, *erule-tac exE*)
fix *ap bp*
assume
length ap = *6* + *length ab* ∧
aprog = *ap* [+] *bp* ∧ *bp* =
([*Dec* (*a-md* − *Suc 0*) (*length a* + *7*)] [+]
(*a* [+] [*Inc* (*rs-pos* − *Suc 0*), *Dec rs-pos 3*,
Goto (*Suc 0*)])) @
[*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length a* + *4*)]
from *g* **and** *this* **and** *k2* **and** *g-ind* **show** *?thesis*
apply(*insert abc-append-exc3*[*of*
butlast lm @ (*last lm* − *Suc xa*) # *rsxa* #
$0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *Suc xa* # *suf-lm bp 0*
butlast lm @ (*last lm* − *xa*) # *rsa* #
$0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *xa* # *suf-lm length ap ap*],
simp)
apply(*subgoal-tac*
∃ *stp*. *abc-steps-l* (*0*, *butlast lm* @ (*last lm* − *Suc xa*)
rsxa # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *Suc xa*
suf-lm) *bp stp* =
(*0*, *butlast lm* @ (*last lm* − *xa*) # *rsa* #
$0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *xa* # *suf-lm*),
simp, *erule-tac conjE*, *erule conjE*)
apply(*erule pr-cycle-part-ind*, *auto*)
done
qed
qed
from *g* **and** *k2* **and** *this* **show** *?thesis*
apply(*erule-tac exE*)
apply(*insert ind*[*of rsa*], *simp*)
apply(*erule-tac exE*)
apply(*rule-tac x* = *stp* + *stpa* **in** *exI*,
simp add: *abc-steps-add*)
done
qed
qed
qed

lemma *ci-pr-length*:
  ⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*);
    rec-ci g = (*a*, *aa*, *ba*);
    rec-ci f = (*ab*, *ac*, *bc*)⟧
    $\Longrightarrow$ *length aprog* = *13* + *length ab* + *length a*
apply(*auto simp*: *rec-ci.simps*)
done

thm *empty.simps*
term *max*
fun *empty-inv* :: *nat* × *nat list* ⇒ *nat* ⇒ *nat* ⇒ *nat list* ⇒ *bool*
  where
  empty-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 *empty-stage1* :: *nat* × *nat list* ⇒ *nat* ⇒ *nat*
  where
  empty-stage1 (*as*, *lm*) *m* =
          (*if as* = *3 then 0*
           else 1)

fun *empty-stage2* :: *nat* × *nat list* ⇒ *nat* ⇒ *nat*
  where
  empty-stage2 (*as*, *lm*) *m* = (*lm* ! *m*)

fun *empty-stage3* :: *nat* × *nat list* ⇒ *nat* ⇒ *nat*
  where
  empty-stage3 (*as*, *lm*) *m* = (*if as* = *1 then 3*
                              else if as = *2 then 2*
                              else if as = *0 then 1*
                              else 0)

fun *empty-measure* :: ((*nat* × *nat list*) × *nat*) ⇒ (*nat* × *nat* × *nat*)
  where
  empty-measure ((*as*, *lm*), *m*) =
     (*empty-stage1* (*as*, *lm*) *m*, *empty-stage2* (*as*, *lm*) *m*,
      empty-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 *empty-LE* ::
(((*nat* × *nat list*) × *nat*) × ((*nat* × *nat list*) × *nat*)) *set*
where
empty-LE ≡ (*inv-image lex-triple empty-measure*)

lemma *wf-lex-triple*: *wf lex-triple*
by (*auto intro*:*wf-lex-prod simp*:*lex-triple-def lex-pair-def*)

lemma *wf-empty-le*[*intro*]: *wf empty-LE*
by(*auto intro*:*wf-inv-image wf-lex-triple simp*: *empty-LE-def*)

declare *empty-inv*.*simps*[*simp del*]

lemma *empty-inv-init*:
⟦*m* < *length initlm*; *n* < *length initlm*; *m* ≠ *n*⟧ ⟹
empty-inv (*0*, *initlm*) *m n initlm*
apply(*simp add*: *abc-steps-l*.*simps empty-inv*.*simps*)
apply(*rule-tac x* = *initlm* ! *m* **in** *exI*,
rule-tac x = *initlm* ! *n* **in** *exI*, *simp*)
done

lemma [*simp*]: *abc-fetch 0* (*recursive*.*empty m n*) = *Some* (*Dec m 3*)
apply(*simp add*: *empty*.*simps abc-fetch*.*simps*)
done

lemma [*simp*]: *abc-fetch* (*Suc 0*) (*recursive*.*empty m n*) =
Some (*Inc n*)
apply(*simp add*: *empty*.*simps abc-fetch*.*simps*)
done

lemma [*simp*]: *abc-fetch 2* (*recursive*.*empty m n*) = *Some* (*Goto 0*)
apply(*simp add*: *empty*.*simps abc-fetch*.*simps*)
done

lemma [*simp*]: *abc-fetch 3* (*recursive*.*empty m n*) = *None*
apply(*simp add*: *empty*.*simps abc-fetch*.*simps*)
done

lemma [*simp*]:
⟦*m* ≠ *n*; *m* < *length initlm*; *n* < *length initlm*;

```
  k + l = initlm ! m + initlm ! n; k ≤ initlm ! m; 0 < k⟧
 ⟹ ∃ka la. initlm[m := k, n := l, m := k − Suc 0] =
    initlm[m := ka, n := la] ∧
    Suc (ka + la) = initlm ! m + initlm ! n ∧
    ka < initlm ! m
apply(rule-tac x = k − Suc 0 in exI, rule-tac x = l in exI,
    simp, auto)
apply(subgoal-tac
    initlm[m := k, n := l, m := k − Suc 0] =
     initlm[n := l, m := k, m := k − Suc 0])
apply(simp add: list-update-overwrite )
apply(simp add: list-update-swap)
apply(simp add: list-update-swap)
done

lemma [simp]:
 ⟦m ≠ n; m < length initlm; n < length initlm;
  Suc (k + l) = initlm ! m + initlm ! n;
  k < initlm ! m⟧
  ⟹ ∃ka la. initlm[m := k, n := l, n := Suc l] =
          initlm[m := ka, n := la] ∧
          ka + la = initlm ! m + initlm ! n ∧
          ka ≤ initlm ! m
apply(rule-tac x = k in exI, rule-tac x = Suc l in exI, auto)
done

lemma [simp]:
 ⟦length initlm > max m n; m ≠ n⟧ ⟹
  ∀na. ¬ (λ(as, lm) m. as = 3)
   (abc-steps-l (0, initlm) (recursive.empty m n) na) m ∧
  empty-inv (abc-steps-l (0, initlm)
        (recursive.empty m n) na) m n initlm ⟶
  empty-inv (abc-steps-l (0, initlm)
        (recursive.empty m n) (Suc na)) m n initlm ∧
  ((abc-steps-l (0, initlm) (recursive.empty m n) (Suc na), m),
   abc-steps-l (0, initlm) (recursive.empty m n) na, m) ∈ empty-LE
apply(rule allI, rule impI, simp add: abc-steps-ind)
apply(case-tac (abc-steps-l (0, initlm) (recursive.empty m n) na),
    simp)
apply(auto split:if-splits simp add:abc-steps-l.simps empty-inv.simps)
apply(auto simp add: empty-LE-def lex-triple-def lex-pair-def
           abc-step-l.simps abc-steps-l.simps
           empty-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 empty-inv-halt:
 ⟦length initlm > max m n; m ≠ n⟧ ⟹
```

∃ *stp*. (λ (*as*, *lm*). *as* = *3* ∧
empty-inv (*as*, *lm*) *m n initlm*)
(*abc-steps-l* (*0*::*nat*, *initlm*) (*empty m n*) *stp*)
apply(*insert halt-lemma2*[*of empty-LE*
λ ((*as*, *lm*), *m*). *as* = (*3*::*nat*)
λ *stp*. (*abc-steps-l* (*0*, *initlm*) (*recursive.empty m n*) *stp*, *m*)
λ ((*as*, *lm*), *m*). *empty-inv* (*as*, *lm*) *m n initlm*])
apply(*insert wf-empty-le*, *simp add*: *empty-inv-init abc-steps-zero*)
apply(*erule-tac exE*)
apply(*rule-tac x* = *na* **in** *exI*)
apply(*case-tac* (*abc-steps-l* (*0*, *initlm*) (*recursive.empty m n*) *na*),
simp, *auto*)
done

lemma *empty-halt-cond*:
⟦*m* ≠ *n*; *empty-inv* (*a*, *b*) *m n lm*; *a* = *3*⟧ ⟹
b = *lm*[*n* := *lm* ! *m* + *lm* ! *n*, *m* := *0*]
apply(*simp add*: *empty-inv.simps*, *auto*)
apply(*simp add*: *list-update-swap*)
done

lemma *empty-ex*:
⟦*length lm* > *max m n*; *m* ≠ *n*⟧ ⟹
∃ *stp*. *abc-steps-l* (*0*::*nat*, *lm*) (*empty m n*) *stp*
= (*3*, (*lm*[*n* := (*lm* ! *m* + *lm* ! *n*)])[*m* := *0*::*nat*])
apply(*drule empty-inv-halt*, *simp*, *erule-tac exE*)
apply(*rule-tac x* = *stp* **in** *exI*)
apply(*case-tac abc-steps-l* (*0*, *lm*) (*recursive.empty m n*) *stp*,
simp)
apply(*erule-tac empty-halt-cond*, *auto*)
done

lemma [*simp*]:
⟦*a-md* = *Suc* (*max* (*Suc* (*Suc n*)) (*max bc ba*));
length lm = *rs-pos* ∧ *rs-pos* = *n* ∧ *n* > *0*⟧
⟹ *n* − *Suc 0* < *length lm* +
(*Suc* (*max* (*Suc* (*Suc n*)) (*max bc ba*)) − *rs-pos* + *length suf-lm*) ∧
Suc (*Suc n*) < *length lm* + (*Suc* (*max* (*Suc* (*Suc n*)) (*max bc ba*)) −
rs-pos + *length suf-lm*) ∧ *bc* < *length lm* + (*Suc* (*max* (*Suc* (*Suc n*))
(*max bc ba*)) − *rs-pos* + *length suf-lm*) ∧ *ba* < *length lm* +
(*Suc* (*max* (*Suc* (*Suc n*)) (*max bc ba*)) − *rs-pos* + *length suf-lm*)
apply(*arith*)
done

lemma [*simp*]:
⟦*a-md* = *Suc* (*max* (*Suc* (*Suc n*)) (*max bc ba*));
length lm = *rs-pos* ∧ *rs-pos* = *n* ∧ *n* > *0*⟧
⟹ *n* − *Suc 0* < *Suc* (*length suf-lm* + *max* (*Suc* (*Suc n*)) (*max bc ba*)) ∧
Suc n < *length suf-lm* + *max* (*Suc* (*Suc n*)) (*max bc ba*) ∧

$bc < Suc\ (length\ suf\text{-}lm + max\ (Suc\ (Suc\ n))\ (max\ bc\ ba)) \wedge$
$ba < Suc\ (length\ suf\text{-}lm + max\ (Suc\ (Suc\ n))\ (max\ bc\ ba))$
apply(*arith*)
done

lemma [*simp*]: $n - Suc\ 0 \neq max\ (Suc\ (Suc\ n))\ (max\ bc\ ba)$
apply(*arith*)
done

lemma [*simp*]:
$a\text{-}md \geq Suc\ bc \wedge rs\text{-}pos > 0 \wedge bc \geq rs\text{-}pos \Longrightarrow$
$bc - (rs\text{-}pos - Suc\ 0) + a\text{-}md - Suc\ bc = Suc\ (a\text{-}md - rs\text{-}pos - Suc\ 0)$
apply(*arith*)
done

lemma [*simp*]: $length\ lm = n \wedge rs\text{-}pos = n \wedge 0 < rs\text{-}pos \wedge$
$Suc\ rs\text{-}pos < a\text{-}md$
$\Longrightarrow n - Suc\ 0 < Suc\ (Suc\ (a\text{-}md + length\ suf\text{-}lm - Suc\ (Suc\ 0)))$
$\wedge\ n < Suc\ (Suc\ (a\text{-}md + length\ suf\text{-}lm - Suc\ (Suc\ 0)))$
apply(*arith*)
done

lemma [*simp*]: $length\ lm = n \wedge rs\text{-}pos = n \wedge 0 < rs\text{-}pos \wedge$
$Suc\ rs\text{-}pos < a\text{-}md \Longrightarrow n - Suc\ 0 \neq n$
by *arith*

lemma *ci-pr-ex2*:
$[\![rec\text{-}ci\ (Pr\ n\ f\ g) = (aprog,\ rs\text{-}pos,\ a\text{-}md);$
$rec\text{-}calc\text{-}rel\ (Pr\ n\ f\ g)\ lm\ rs;$
$rec\text{-}ci\ g = (a,\ aa,\ ba);$
$rec\text{-}ci\ f = (ab,\ ac,\ bc)]\!]$
$\Longrightarrow \exists\ ap\ bp.\ aprog = ap\ [+]\ bp\ \wedge$
$ap = empty\ n\ (max\ (Suc\ (Suc\ (Suc\ n)))\ (max\ bc\ ba))$
apply(*simp add*: *rec-ci.simps*)
apply(*rule-tac* $x = (ab\ [+]\ (recursive.empty\ n\ (Suc\ n)\ [+]$
$([Dec\ (max\ (n + 3)\ (max\ bc\ ba))\ (length\ a + 7)]$
$[+]\ (a\ [+]\ [Inc\ n,\ Dec\ (Suc\ n)\ 3,\ Goto\ (Suc\ 0)]))\ @$
$[Dec\ (Suc\ (Suc\ n))\ 0,\ Inc\ (Suc\ n),\ Goto\ (length\ a + 4)]))$ **in** *exI*, *auto*)
apply(*simp add*: *abc-append-commute add3-Suc*)
done

lemma [*simp*]:
$max\ (Suc\ (Suc\ (Suc\ n)))\ (max\ bc\ ba) - n <$
$Suc\ (max\ (Suc\ (Suc\ (Suc\ n)))\ (max\ bc\ ba)) - n$
apply(*arith*)
done
lemma *exp-nth*[*simp*]: $n < m \Longrightarrow a^m\ !\ n = a$
apply(*simp add*: *exponent-def*)
done

lemma [*simp*]: *length lm* = *n* ∧ *rs-pos* = *n* ∧ *0* < *n* ⟹
lm[*n* − *Suc 0* := *0*::*nat*] = *butlast lm* @ [*0*]
apply(*auto*)
apply(*insert list-update-append*[*of butlast lm* [*last lm*]
length lm − *Suc 0 0*], *simp*)
done

lemma [*simp*]: ⟦*length lm* = *n*; *0* < *n*⟧ ⟹ *lm* ! (*n* − *Suc 0*) = *last lm*
apply(*insert nth-append*[*of butlast lm* [*last lm*] *n* − *Suc 0*],
simp)
apply(*insert butlast-append-last*[*of lm*], *auto*)
done
lemma *exp-suc-iff*: a^b @ [*a*] = $a^{b + Suc\ 0}$
apply(*simp add*: *exponent-def rep-ind del*: *replicate.simps*)
done

lemma *less-not-less*[*simp*]: *n* > *0* ⟹ ¬ *n* < *n* − *Suc 0*
by *auto*

lemma [*simp*]:
Suc n < *length suf-lm* + *max* (*Suc* (*Suc n*)) (*max bc ba*) ∧
bc < *Suc* (*length suf-lm* + *max* (*Suc* (*Suc n*))
(*max bc ba*)) ∧
ba < *Suc* (*length suf-lm* + *max* (*Suc* (*Suc n*)) (*max bc ba*))
by *arith*

lemma [*simp*]: *length lm* = *n* ∧ *rs-pos* = *n* ∧ *n* > *0* ⟹
(*lm* @ $0^{Suc\ (max\ (Suc\ (Suc\ n))\ (max\ bc\ ba)) - n}$ @ *suf-lm*)
[*max* (*Suc* (*Suc n*)) (*max bc ba*) :=
(*lm* @ $0^{Suc\ (max\ (Suc\ (Suc\ n))\ (max\ bc\ ba)) - n}$ @ *suf-lm*) ! (*n* − *Suc 0*) +
(*lm* @ $0^{Suc\ (max\ (Suc\ (Suc\ n))\ (max\ bc\ ba)) - n}$ @ *suf-lm*) !
max (*Suc* (*Suc n*)) (*max bc ba*), *n* − *Suc 0* := *0*::*nat*]
= *butlast lm* @ *0* # $0^{max\ (Suc\ (Suc\ n))\ (max\ bc\ ba) - n}$ @ *last lm* # *suf-lm*
apply(*simp add*: *nth-append exp-nth list-update-append*)
apply(*insert list-update-append*[*of* $0^{(max\ (Suc\ (Suc\ n))\ (max\ bc\ ba)) - n}$
[*0*] *max* (*Suc* (*Suc n*)) (*max bc ba*) − *n last lm*], *simp*)
apply(*simp add*: *exp-suc-iff Suc-diff-le del*: *list-update.simps*)
done

lemma *exp-eq*: (*a* = *b*) = (c^a = c^b)
apply(*auto simp*: *exponent-def*)
done

lemma [*simp*]:
⟦*length lm* = *n*; *0* < *n*; *Suc n* < *a-md*⟧ ⟹
(*butlast lm* @ *rsa* # $0^{a\text{-}md - Suc\ n}$ @ *last lm* # *suf-lm*)
[*n* := (*butlast lm* @ *rsa* # $0^{a\text{-}md - Suc\ n}$ @ *last lm* # *suf-lm*) !

$(n - Suc\ 0) + (butlast\ lm\ @\ rsa\ \#\ (0::nat)^{a\text{-}md\ -\ Suc\ n}\ @$
$last\ lm\ \#\ suf\text{-}lm)\ !\ n,\ n - Suc\ 0 := 0]$
$= butlast\ lm\ @\ 0\ \#\ rsa\ \#\ 0^{a\text{-}md\ -\ Suc\ (Suc\ n)}\ @\ last\ lm\ \#\ suf\text{-}lm$
apply(*simp add*: *nth-append exp-nth list-update-append*)
apply(*case-tac a-md − Suc n*, *simp*, *simp add*: *exponent-def*)
done

lemma [*simp*]:
$Suc\ (Suc\ rs\text{-}pos) \leq a\text{-}md \wedge length\ lm = rs\text{-}pos \wedge 0 < rs\text{-}pos$
$\Longrightarrow a\text{-}md - Suc\ 0 <$
$Suc\ (Suc\ (Suc\ (a\text{-}md + length\ suf\text{-}lm - Suc\ (Suc\ (Suc\ 0)))))$
by *arith*

lemma [*simp*]:
$Suc\ (Suc\ rs\text{-}pos) \leq a\text{-}md \wedge length\ lm = rs\text{-}pos \wedge 0 < rs\text{-}pos \Longrightarrow$
$\neg\ a\text{-}md - Suc\ 0 < rs\text{-}pos - Suc\ 0$
by *arith*

lemma [*simp*]: $Suc\ (Suc\ rs\text{-}pos) \leq a\text{-}md \Longrightarrow$
$\neg\ a\text{-}md - Suc\ 0 < rs\text{-}pos - Suc\ 0$
by *arith*

lemma [*simp*]: $[\![Suc\ (Suc\ rs\text{-}pos) \leq a\text{-}md]\!] \Longrightarrow$
$\neg\ a\text{-}md - rs\text{-}pos < Suc\ (Suc\ (a\text{-}md - Suc\ (Suc\ rs\text{-}pos)))$
by *arith*

lemma [*simp*]:
$Suc\ (Suc\ rs\text{-}pos) \leq a\text{-}md \wedge length\ lm = rs\text{-}pos \wedge 0 < rs\text{-}pos$
$\Longrightarrow (abc\text{-}lm\text{-}v\ (butlast\ lm\ @\ last\ lm\ \#\ rs\ \#\ 0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}\ @$
$0\ \#\ suf\text{-}lm)\ (a\text{-}md - Suc\ 0) = 0 \longrightarrow$
$abc\text{-}lm\text{-}s\ (butlast\ lm\ @\ last\ lm\ \#\ rs\ \#\ 0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}\ @$
$0\ \#\ suf\text{-}lm)\ (a\text{-}md - Suc\ 0)\ 0 =$
$lm\ @\ rs\ \#\ 0^{a\text{-}md\ -\ Suc\ rs\text{-}pos}\ @\ suf\text{-}lm) \wedge$
$abc\text{-}lm\text{-}v\ (butlast\ lm\ @\ last\ lm\ \#\ rs\ \#\ 0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}\ @$
$0\ \#\ suf\text{-}lm)\ (a\text{-}md - Suc\ 0) = 0$
apply(*simp add*: *abc-lm-v.simps nth-append abc-lm-s.simps*)
apply(*insert nth-append*[*of last lm* $\#$ *rs* $\#$ $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$
$0\ \#\ suf\text{-}lm\ (a\text{-}md - rs\text{-}pos)$], *auto*)
apply(*simp only*: *exp-suc-iff*)
apply(*subgoal-tac a-md − Suc 0 < a-md + length suf-lm*, *simp*)
apply(*case-tac lm* = [], *auto*)
done

lemma *pr-prog-ex*[*simp*]: $[\![rec\text{-}ci\ (Pr\ n\ f\ g) = (aprog,\ rs\text{-}pos,\ a\text{-}md);$
$rec\text{-}ci\ g = (a,\ aa,\ ba);\ rec\text{-}ci\ f = (ab,\ ac,\ bc)]\!]$
$\Longrightarrow \exists\ cp.\ aprog = recursive.empty\ n\ (max\ (n + 3)$
$(max\ bc\ ba))\ [+]\ cp$
apply(*simp add*: *rec-ci.simps*)

apply(*rule-tac x* = (*ab* [+] (*recursive.empty n* (*Suc n*) [+]
([*Dec* (*max* (*n* + *3*) (*max bc ba*)) (*length a* + *7*)]
[+] (*a* [+] [*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)]))
@ [*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length a* + *4*)])) **in** *exI*)
apply(*auto simp*: *abc-append-commute*)
done

lemma [*simp*]: *empty m n* ≠ []
by (*simp add*: *empty.simps*)

lemma [*intro*]:
⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*);
rec-ci f = (*ab*, *ac*, *bc*)⟧ ⟹
∃ *ap*. (∃ *cp*. *aprog* = *ap* [+] *ab* [+] *cp*) ∧ *length ap* = *3*
apply(*case-tac rec-ci g*, *simp add*: *rec-ci.simps*)
apply(*rule-tac x* = *empty n*
(*max* (*n* + *3*) (*max bc c*)) **in** *exI*, *simp*)
apply(*rule-tac x* = *recursive.empty n* (*Suc n*) [+]
([*Dec* (*max* (*n* + *3*) (*max bc c*)) (*length a* + *7*)]
[+] *a* [+] [*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)])
@ [*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length a* + *4*)] **in** *exI*,
auto)
apply(*simp add*: *abc-append-commute*)
done

lemma [*intro*]:
⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*);
rec-ci g = (*a*, *aa*, *ba*);
rec-ci f = (*ab*, *ac*, *bc*)⟧ ⟹
∃ *ap*. (∃ *cp*. *aprog* = *ap* [+] *recursive.empty n* (*Suc n*) [+] *cp*)
∧ *length ap* = *3* + *length ab*
apply(*simp add*: *rec-ci.simps*)
apply(*rule-tac x* = *recursive.empty n* (*max* (*n* + *3*)
(*max bc ba*)) [+] *ab* **in** *exI*, *simp*)
apply(*rule-tac x* = ([*Dec* (*max* (*n* + *3*) (*max bc ba*))
(*length a* + *7*)] [+] *a* [+]
[*Inc n*, *Dec* (*Suc n*) *3*, *Goto* (*Suc 0*)]) @
[*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length a* + *4*)] **in** *exI*)
apply(*auto simp*: *abc-append-commute*)
done

lemma [*intro*]:
⟦*rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*);
rec-ci g = (*a*, *aa*, *ba*);
rec-ci f = (*ab*, *ac*, *bc*)⟧
⟹ ∃ *ap*. (∃ *cp*. *aprog* = *ap* [+] ([*Dec* (*a-md* − *Suc 0*) (*length a* + *7*)]
[+] (*a* [+] [*Inc* (*rs-pos* − *Suc 0*), *Dec rs-pos 3*,

Goto (*Suc* *0*)])) @ [*Dec* (*Suc* (*Suc* *n*)) *0*, *Inc* (*Suc* *n*),
Goto (*length* *a* + *4*)] [+] *cp*) ∧
length *ap* = *6* + *length* *ab*
apply(*simp add*: *rec-ci.simps*)
apply(*rule-tac x* = *recursive.empty n*
(*max* (*n* + *3*) (*max bc ba*)) [+] *ab* [+]
recursive.empty n (*Suc n*) **in** *exI*, *simp*)
apply(*rule-tac x* = [] **in** *exI*, *auto*)
apply(*simp add*: *abc-append-commute*)
done

lemma [*simp*]:
n < *Suc* (*max* (*n* + *3*) (*max bc ba*) + *length suf-lm*) ∧
Suc (*Suc n*) < *max* (*n* + *3*) (*max bc ba*) + *length suf-lm* ∧
bc < *Suc* (*max* (*n* + *3*) (*max bc ba*) + *length suf-lm*) ∧
ba < *Suc* (*max* (*n* + *3*) (*max bc ba*) + *length suf-lm*)
by *arith*

lemma [*simp*]: *n* ≠ *max* (*n* + (*3*::*nat*)) (*max bc ba*)
by *arith*

lemma [*simp*]:*length lm* = *Suc n* ⟹ *lm*[*n* := (*0*::*nat*)] = *butlast lm* @ [*0*]
apply(*subgoal-tac* ∃ *xs x*. *lm* = *xs* @ [*x*], *auto simp*: *list-update-append*)
apply(*rule-tac x* = *butlast lm* **in** *exI*, *rule-tac x* = *last lm* **in** *exI*)
apply(*case-tac lm*, *auto*)
done

lemma [*simp*]: *length lm* = *Suc n* ⟹ *lm* ! *n* =*last lm*
apply(*subgoal-tac lm* ≠ [])
apply(*simp add*: *last-conv-nth*, *case-tac lm*, *simp-all*)
done

lemma [*simp*]: *length lm* = *Suc n* ⟹
(*lm* @ (*0*::*nat*)$^{max\ (n\ +\ 3)\ (max\ bc\ ba)\ -\ n}$ @ *suf-lm*)
[*max* (*n* + *3*) (*max bc ba*) := (*lm* @ *0*$^{max\ (n\ +\ 3)\ (max\ bc\ ba)\ -\ n}$ @
suf-lm) ! *n* +
(*lm* @ *0*$^{max\ (n\ +\ 3)\ (max\ bc\ ba)\ -\ n}$ @ *suf-lm*) ! *max* (*n* + *3*) (*max*
bc ba), *n* := *0*]
= *butlast lm* @ *0* # *0*$^{max\ (n\ +\ 3)\ (max\ bc\ ba)\ -\ Suc\ n}$ @ *last lm* # *suf-lm*
apply(*auto simp*: *list-update-append nth-append*)
apply(*subgoal-tac* (*0*$^{max\ (n\ +\ 3)\ (max\ bc\ ba)\ -\ n}$) = *0*$^{max\ (n\ +\ 3)\ (max\ bc\ ba)\ -\ Suc\ n}$
@ [*0*::*nat*])
apply(*simp add*: *list-update-append*)
apply(*simp add*: *exp-suc-iff*)
done

lemma [*simp*]: *Suc* (*Suc n*) < *a-md* ⟹
  n < *Suc* (*Suc* (*a-md* + *length suf-lm* − *2*)) ∧
   n < *Suc* (*a-md* + *length suf-lm* − *2*)
by(*arith*)

lemma [*simp*]: ⟦*length lm* = *Suc n*; *Suc* (*Suc n*) < *a-md*⟧
  ⟹(*butlast lm* @ (*rsa*::*nat*) # $0^{a\text{-}md\ -\ Suc\ (Suc\ n)}$ @ *last lm* # *suf-lm*)
   [*Suc n* := (*butlast lm* @ *rsa* # $0^{a\text{-}md\ -\ Suc\ (Suc\ n)}$ @ *last lm* # *suf-lm*) !
n +
      (*butlast lm* @ *rsa* # $0^{a\text{-}md\ -\ Suc\ (Suc\ n)}$ @ *last lm* # *suf-lm*) ! *Suc*
n, *n* := *0*]
 = *butlast lm* @ *0* # *rsa* # $0^{a\text{-}md\ -\ Suc\ (Suc\ (Suc\ n))}$ @ *last lm* # *suf-lm*
apply(*auto simp*: *list-update-append*)
apply(*subgoal-tac* ($0^{a\text{-}md\ -\ Suc\ (Suc\ n)}$) = (*0*::*nat*) # ($0^{a\text{-}md\ -\ Suc\ (Suc\ (Suc\ n))}$),
simp add: *nth-append*)
apply(*simp add*: *exp-ind-def*[*THEN sym*])
done

lemma *pr-case*:
 assumes *nf-ind*:
 ⋀ *lm rs suf-lm*. *rec-calc-rel f lm rs* ⟹
 ∃ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{bc\ -\ ac}$ @ *suf-lm*) *ab stp* =
     (*length ab*, *lm* @ *rs* # $0^{bc\ -\ Suc\ ac}$ @ *suf-lm*)
 and *ng-ind*: ⋀ *lm rs suf-lm*. *rec-calc-rel g lm rs* ⟹
  ∃ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{ba\ -\ aa}$ @ *suf-lm*) *a stp* =
      (*length a*, *lm* @ *rs* # $0^{ba\ -\ Suc\ aa}$ @ *suf-lm*)
  and *h*: *rec-ci* (*Pr n f g*) = (*aprog*, *rs-pos*, *a-md*) *rec-calc-rel* (*Pr n f g*) *lm rs*
    rec-ci g = (*a*, *aa*, *ba*) *rec-ci f* = (*ab*, *ac*, *bc*)
 shows ∃ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog stp* = (*length*
aprog, *lm* @ *rs* # $0^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
proof −
 from *h* **have** *k1*: ∃ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog stp*
  = (*3*, *butlast lm* @ *0* # $0^{a\text{-}md\ -\ rs\text{-}pos\ -\ 1}$ @ *last lm* # *suf-lm*)
 proof −
  have ∃ *bp cp*. *aprog* = *bp* [+] *cp* ∧ *bp* = *empty n*
      (*max* (*n* + *3*) (*max bc ba*))
   apply(*insert h*, *simp*)
   apply(*erule pr-prog-ex*, *auto*)
   done
  thus *?thesis*
   apply(*erule-tac exE*, *erule-tac exE*, *simp*)
   apply(*subgoal-tac*
     ∃ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*)
      ([] [+] *recursive.empty n*
       (*max* (*n* + *3*) (*max bc ba*)) [+] *cp*) *stp* =
     (*0* + *3*, *butlast lm* @ *0* # $0^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @
              last lm # *suf-lm*), *simp*)
   apply(*rule-tac abc-append-exc1*, *simp-all*)

apply(*insert empty-ex*[*of n* (*max* (*n* + *3*)
(*max bc ba*)) *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*], *simp*)
apply(*subgoal-tac a-md* = *Suc* (*max* (*n* + *3*) (*max bc ba*)),
simp)
apply(*subgoal-tac length lm* = *Suc n* ∧ *rs-pos* = *Suc n*, *simp*)
apply(*insert h*)
apply(*simp add*: *para-pattern ci-pr-para-eq*)
apply(*rule ci-pr-md-def*, *auto*)
done
qed
from *h* **have** *k2*:
∃ *stp*. *abc-steps-l* (*3*, *butlast lm* @ *0* # $0^{a\text{-}md\ -\ rs\text{-}pos\ -\ 1}$ @
last lm # *suf-lm*) *aprog stp*
= (*length aprog*, *lm* @ *rs* # $0^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
proof −
from *h* **have** *k2-1*: ∃ *rs*. *rec-calc-rel f* (*butlast lm*) *rs*
apply(*erule-tac calc-pr-zero-ex*)
done
thus *?thesis*
proof(*erule-tac exE*)
fix *rsa*
assume *k2-2*: *rec-calc-rel f* (*butlast lm*) *rsa*
from *h* **and** *k2-2* **have** *k2-2-1*:
∃ *stp*. *abc-steps-l* (*3*, *butlast lm* @ *0* # $0^{a\text{-}md\ -\ rs\text{-}pos\ -\ 1}$
@ *last lm* # *suf-lm*) *aprog stp*
= (*3* + *length ab*, *butlast lm* @ *rsa* # $0^{a\text{-}md\ -\ rs\text{-}pos\ -\ 1}$ @
last lm # *suf-lm*)
proof −
from *h* **have** *j1*:
∃ *ap bp cp*. *aprog* = *ap* [+] *bp* [+] *cp* ∧ *length ap* = *3* ∧
bp = *ab*
apply(*auto*)
done
from *h* **have** *j2*: *ac* = *rs-pos* − *1*
apply(*drule-tac ci-pr-f-paras*, *simp*, *auto*)
done
from *h* **and** *j2* **have** *j3*: *a-md* ≥ *Suc bc* ∧ *rs-pos* > *0* ∧ *bc* ≥ *rs-pos*
apply(*rule-tac conjI*)
apply(*erule-tac ab* = *ab* **and** *ac* = *ac* **in** *ci-pr-md-ge-f*, *simp*)
apply(*rule-tac context-conjI*)
apply(*simp-all add*: *rec-ci.simps*)
apply(*drule-tac ci-ad-ge-paras*, *drule-tac ci-ad-ge-paras*)
apply(*arith*)
done
from *j1* **and** *j2* **show** *?thesis*
apply(*auto simp del*: *abc-append-commute*)
apply(*rule-tac abc-append-exc1*, *simp-all*)
apply(*insert nf-ind*[*of butlast lm rsa*
$0^{a\text{-}md\ -\ bc\ -\ Suc\ 0}$ @ *last lm* # *suf-lm*],

simp add: *k2-2 j2*, *erule-tac exE*)
apply(*simp add*: *exponent-add-iff j3*)
apply(*rule-tac x* = *stp* **in** *exI*, *simp*)
done
qed
from *h* **have** *k2-2-2*:
∃ *stp*. *abc-steps-l* (*3* + *length ab*, *butlast lm* @ *rsa* #
$0^{a\text{-}md - rs\text{-}pos - 1}$ @ *last lm* # *suf-lm*) *aprog stp*
= (*6* + *length ab*, *butlast lm* @ *0* # *rsa* #
$0^{a\text{-}md - rs\text{-}pos - 2}$ @ *last lm* # *suf-lm*)
proof −
from *h* **have** ∃ *ap bp cp*. *aprog* = *ap* [+] *bp* [+] *cp* ∧
length ap = *3* + *length ab* ∧ *bp* = *recursive.empty n* (*Suc n*)
by *auto*
thus *?thesis*
proof(*erule-tac exE*, *erule-tac exE*, *erule-tac exE*,
erule-tac exE)
fix *ap cp bp apa*
assume *aprog* = *ap* [+] *bp* [+] *cp* ∧ *length ap* = *3* +
length ab ∧ *bp* = *recursive.empty n* (*Suc n*)
thus *?thesis*
apply(*simp del*: *abc-append-commute*)
apply(*subgoal-tac*
∃ *stp*. *abc-steps-l* (*3* + *length ab*,
butlast lm @ *rsa* # $0^{a\text{-}md - Suc\ rs\text{-}pos}$ @
last lm # *suf-lm*) (*ap* [+]
recursive.empty n (*Suc n*) [+] *cp*) *stp* =
((*3* + *length ab*) + *3*, *butlast lm* @ *0* # *rsa* #
$0^{a\text{-}md - Suc\ (Suc\ rs\text{-}pos)}$ @ *last lm* # *suf-lm*), *simp*)
apply(*rule-tac abc-append-exc1*, *simp-all*)
apply(*insert empty-ex*[*of n Suc n*
butlast lm @ *rsa* # $0^{a\text{-}md - Suc\ rs\text{-}pos}$ @
last lm # *suf-lm*], *simp*)
apply(*subgoal-tac length lm* = *Suc n* ∧ *rs-pos* = *Suc n* ∧ *a-md* > *Suc* (*Suc n*), *simp*)
apply(*insert h*, *simp*)
done
qed
qed
from *h* **have** *k2-3*: *lm* ≠ []
apply(*rule-tac calc-pr-para-not-null*, *simp*)
done
from *h* **and** *k2-2* **and** *k2-3* **have** *k2-2-3*:
∃ *stp*. *abc-steps-l* (*6* + *length ab*, *butlast lm* @
(*last lm* − *last lm*) # *rsa* #
$0^{a\text{-}md - (Suc\ (Suc\ rs\text{-}pos))}$ @ *last lm* # *suf-lm*) *aprog stp*
= (*6* + *length ab*, *butlast lm* @ *last lm* # *rs* #
$0^{a\text{-}md - Suc\ (Suc\ (rs\text{-}pos))}$ @ *0* # *suf-lm*)

apply(*rule-tac x = last lm* **and** *g = g* **in** *pr-cycle-part*, *auto*)
apply(*rule-tac ng-ind*, *simp*)
apply(*rule-tac rec-calc-rel-def0*, *simp*, *simp*)
done
from *h* **have** *k2-2-4*:
$\exists$ *stp. abc-steps-l* (*6 + length ab*,
butlast lm @ last lm # rs # $0^{a\text{-}md\ -\ rs\text{-}pos\ -\ 2}$ @
0 # suf-lm) *aprog stp*
= (*13 + length ab + length a*,
lm @ rs # $0^{a\text{-}md\ -\ rs\text{-}pos\ -\ 1}$ @ *suf-lm*)
proof −
from *h* **have**
$\exists$ *ap bp cp. aprog = ap [+] bp [+] cp* $\wedge$
length ap = 6 + length ab $\wedge$
bp = ([*Dec* (*a-md − Suc 0*) (*length a + 7*)] [+]
(*a* [+] [*Inc* (*rs-pos − Suc 0*),
Dec rs-pos 3, *Goto* (*Suc 0*)])) @
[*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*), *Goto* (*length a + 4*)]
by *auto*
thus *?thesis*
apply(*auto*)
apply(*subgoal-tac*
$\exists$ *stp. abc-steps-l* (*6 + length ab*, *butlast lm @*
last lm # rs # $0^{a\text{-}md\ -\ Suc\ (Suc\ rs\text{-}pos)}$ @ *0 # suf-lm*)
(*ap* [+] ([*Dec* (*a-md − Suc 0*) (*length a + 7*)] [+]
(*a* [+] [*Inc* (*rs-pos − Suc 0*), *Dec rs-pos 3*,
Goto (*Suc 0*)])) @ [*Dec* (*Suc* (*Suc n*)) *0*, *Inc* (*Suc n*),
Goto (*length a + 4*)] [+] *cp*) *stp* =
(*6 + length ab +* (*length a + 7*) ,
lm @ rs # $0^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*), *simp*)
apply(*subgoal-tac 13 +* (*length ab + length a*) =
13 + length ab + length a, *simp*)
apply(*arith*)
apply(*rule abc-append-exc1*, *simp-all*)
apply(*rule-tac x = Suc 0* **in** *exI*,
simp add: *abc-steps-l.simps abc-fetch.simps*
nth-append abc-append-nth abc-step-l.simps)
apply(*subgoal-tac a-md > Suc* (*Suc rs-pos*) $\wedge$
length lm = rs-pos $\wedge$ *rs-pos > 0*, *simp*)
apply(*insert h*, *simp*)
apply(*subgoal-tac rs-pos = Suc n*, *simp*, *simp*)
done
qed
from *h* **have** *k2-2-5*: *length aprog = 13 + length ab + length a*
apply(*rule-tac ci-pr-length*, *simp-all*)
done
from *k2-2-1* **and** *k2-2-2* **and** *k2-2-3* **and** *k2-2-4* **and** *k2-2-5*
show *?thesis*
apply(*auto*)

apply(*rule-tac x = stp + stpa + stpb + stpc* **in** *exI*,
simp add: *abc-steps-add*)
done
qed
qed
from *k1* **and** *k2* **show**
$\exists$ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog stp*
= (*length aprog*, *lm* @ *rs* # $0^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
apply(*erule-tac exE*)
apply(*erule-tac exE*)
apply(*rule-tac x = stp + stpa* **in** *exI*)
apply(*simp add*: *abc-steps-add*)
done
qed

thm *rec-calc-rel.induct*

lemma *eq-switch*: $x = y \Longrightarrow y = x$
by *simp*

lemma [*simp*]:
⟦*rec-ci f* = (*a*, *aa*, *ba*);
rec-ci (*Mn n f*) = (*aprog*, *rs-pos*, *a-md*)⟧ $\Longrightarrow$ $\exists$ *bp*. *aprog* = *a* @ *bp*
apply(*simp add*: *rec-ci.simps*)
apply(*rule-tac x* = [*Dec* (*Suc n*) (*length a* + *5*),
Dec (*Suc n*) (*length a* + *3*), *Goto* (*Suc* (*length a*)),
Inc n, *Goto 0*] **in** *exI*, *auto*)
done

lemma *ci-mn-para-eq*[*simp*]:
rec-ci (*Mn n f*) = (*aprog*, *rs-pos*, *a-md*) $\Longrightarrow$ *rs-pos* = *n*
apply(*case-tac rec-ci f*, *simp add*: *rec-ci.simps*)
done

lemma [*simp*]: *rec-ci f* = (*a*, *aa*, *ba*) $\Longrightarrow$ *aa* < *ba*
apply(*simp add*: *ci-ad-ge-paras*)
done

lemma [*simp*]: ⟦*rec-ci f* = (*a*, *aa*, *ba*);
rec-ci (*Mn n f*) = (*aprog*, *rs-pos*, *a-md*)⟧
$\Longrightarrow$ *ba* $\leq$ *a-md*
apply(*simp add*: *rec-ci.simps*)
by *arith*

lemma *mn-calc-f*:
assumes *ind*:
$\bigwedge$*aprog a-md rs-pos rs suf-lm lm*.
⟦*rec-ci f* = (*aprog*, *rs-pos*, *a-md*); *rec-calc-rel f lm rs*⟧
$\Longrightarrow$ $\exists$ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog stp*

```
        = (length aprog, lm @ [rs] @ 0^(a-md - rs-pos - 1) @ suf-lm)
  and h: rec-ci f = (a, aa, ba)
      rec-ci (Mn n f) = (aprog, rs-pos, a-md)
      rec-calc-rel f (lm @ [x]) rsx
      aa = Suc n
  shows ∃ stp. abc-steps-l (0, lm @ x # 0^(a-md - Suc rs-pos) @ suf-lm)
             aprog stp = (length a,
              lm @ x # rsx # 0^(a-md - Suc (Suc rs-pos)) @ suf-lm)
proof -
  from h have k1: ∃ ap bp. aprog = ap @ bp ∧ ap = a
    by simp
  from h have k2: rs-pos = n
    apply(erule-tac ci-mn-para-eq)
    done
  from h and k1 and k2 show ?thesis

  proof(erule-tac exE, erule-tac exE, simp,
      rule-tac abc-add-exc1, auto)
    fix bp
    show
      ∃ astp. abc-steps-l (0, lm @ x # 0^(a-md - Suc n) @ suf-lm) a astp
      = (length a, lm @ x # rsx # 0^(a-md - Suc (Suc n)) @ suf-lm)
      apply(insert ind[of a Suc n ba  lm @ [x] rsx
          0^(a-md - ba) @ suf-lm], simp add: exponent-add-iff h k2)
      apply(subgoal-tac ba > aa ∧ a-md ≥ ba ∧ aa = Suc n,
          insert h, auto)
      done
  qed
qed
thm rec-ci.simps

fun mn-ind-inv ::
  nat × nat list ⇒ nat ⇒ nat ⇒ nat ⇒ nat list ⇒ nat list ⇒ bool
  where
  mn-ind-inv (as, lm') ss x rsx suf-lm lm =
          (if as = ss then lm' = lm @ x # rsx # suf-lm
           else if as = ss + 1 then
                ∃ y. (lm' = lm @ x # y # suf-lm) ∧ y ≤ rsx
           else if as = ss + 2 then
                ∃ y. (lm' = lm @ x # y # suf-lm) ∧ y ≤ rsx
           else if as = ss + 3 then lm' = lm @ x # 0 # suf-lm
           else if as = ss + 4 then lm' = lm @ Suc x # 0 # suf-lm
           else if as = 0 then lm' = lm @ Suc x # 0 # suf-lm
           else False
)

fun mn-stage1 :: nat × nat list ⇒ nat ⇒ nat ⇒ nat
  where
  mn-stage1 (as, lm) ss n =
```

(*if as = 0 then 0*
else if as = ss + 4 then 1
else if as = ss + 3 then 2
else if as = ss + 2 ∨ as = ss + 1 then 3
else if as = ss then 4
else 0
)

fun *mn-stage2* :: *nat* × *nat list* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
mn-stage2 (*as*, *lm*) *ss n* =
(*if as = ss + 1 ∨ as = ss + 2 then* (*lm* ! (*Suc n*))
else 0)

fun *mn-stage3* :: *nat* × *nat list* ⇒ *nat* ⇒ *nat* ⇒ *nat*
where
mn-stage3 (*as*, *lm*) *ss n* = (*if as = ss + 2 then 1 else 0*)

fun *mn-measure* :: ((*nat* × *nat list*) × *nat* × *nat*) ⇒
(*nat* × *nat* × *nat*)
where
mn-measure ((*as*, *lm*), *ss*, *n*) =
(*mn-stage1* (*as*, *lm*) *ss n*, *mn-stage2* (*as*, *lm*) *ss n*,
mn-stage3 (*as*, *lm*) *ss n*)

definition *mn-LE* :: (((*nat* × *nat list*) × *nat* × *nat*) ×
((*nat* × *nat list*) × *nat* × *nat*)) *set*
where *mn-LE* ≡ (*inv-image lex-triple mn-measure*)

thm *halt-lemma2*
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 *mn-inv-init*:
mn-ind-inv (*abc-steps-l* (*length a*, *lm* @ *x* # *rsx* # *suf-lm*) *aprog 0*)
(*length a*) *x rsx suf-lm lm*
apply(*simp add*: *mn-ind-inv.simps abc-steps-zero*)
done

lemma *mn-halt-init*:
rec-ci f = (*a*, *aa*, *ba*) ⟹
¬ (λ(*as*, *lm′*) (*ss*, *n*). *as* = *0*)
(*abc-steps-l* (*length a*, *lm* @ *x* # *rsx* # *suf-lm*) *aprog 0*)
(*length a*, *n*)
apply(*simp add*: *abc-steps-zero*)
apply(*erule-tac rec-ci-not-null*)

done

thm *rec-ci.simps*
lemma [*simp*]:
⟦*rec-ci f* = (*a*, *aa*, *ba*);
rec-ci (*Mn n f*) = (*aprog*, *rs-pos*, *a-md*)⟧
⟹ *abc-fetch* (*length a*) *aprog* =
Some (*Dec* (*Suc n*) (*length a* + *5*))
apply(*simp add*: *rec-ci.simps abc-fetch.simps*,
erule-tac conjE, *erule-tac conjE*, *simp*)
apply(*drule-tac eq-switch*, *drule-tac eq-switch*, *simp*)
done

lemma [*simp*]: ⟦*rec-ci f* = (*a*, *aa*, *ba*); *rec-ci* (*Mn n f*) = (*aprog*, *rs-pos*, *a-md*)⟧
⟹ *abc-fetch* (*Suc* (*length a*)) *aprog* = *Some* (*Dec* (*Suc n*) (*length a* + *3*))
apply(*simp add*: *rec-ci.simps abc-fetch.simps*, *erule-tac conjE*, *erule-tac conjE*, *simp*)
apply(*drule-tac eq-switch*, *drule-tac eq-switch*, *simp add*: *nth-append*)
done

lemma [*simp*]:
⟦*rec-ci f* = (*a*, *aa*, *ba*);
rec-ci (*Mn n f*) = (*aprog*, *rs-pos*, *a-md*)⟧
⟹ *abc-fetch* (*Suc* (*Suc* (*length a*))) *aprog* =
Some (*Goto* (*length a* + *1*))
apply(*simp add*: *rec-ci.simps abc-fetch.simps*,
erule-tac conjE, *erule-tac conjE*, *simp*)
apply(*drule-tac eq-switch*, *drule-tac eq-switch*, *simp add*: *nth-append*)
done

lemma [*simp*]:
⟦*rec-ci f* = (*a*, *aa*, *ba*);
rec-ci (*Mn n f*) = (*aprog*, *rs-pos*, *a-md*)⟧
⟹ *abc-fetch* (*length a* + *3*) *aprog* = *Some* (*Inc n*)
apply(*simp add*: *rec-ci.simps abc-fetch.simps*,
erule-tac conjE, *erule-tac conjE*, *simp*)
apply(*drule-tac eq-switch*, *drule-tac eq-switch*, *simp add*: *nth-append*)
done

lemma [*simp*]: ⟦*rec-ci f* = (*a*, *aa*, *ba*); *rec-ci* (*Mn n f*) = (*aprog*, *rs-pos*, *a-md*)⟧
⟹ *abc-fetch* (*length a* + *4*) *aprog* = *Some* (*Goto 0*)
apply(*simp add*: *rec-ci.simps abc-fetch.simps*, *erule-tac conjE*, *erule-tac conjE*, *simp*)
apply(*drule-tac eq-switch*, *drule-tac eq-switch*, *simp add*: *nth-append*)
done

lemma [*simp*]:
0 < *rsx*
⟹ ∃ *y*. (*lm* @ *x* # *rsx* # *suf-lm*)[*Suc* (*length lm*) := *rsx* − *Suc 0*]

$= lm @ x \# y \# suf\text{-}lm \wedge y \leq rsx$
apply(*case-tac rsx*, *simp*, *simp*)
apply(*rule-tac x = nat* **in** *exI*, *simp add*: *list-update-append*)
done

lemma [*simp*]:
$[\![y \leq rsx;\ 0 < y]\!]$
$\Longrightarrow \exists\, ya.\ (lm @ x \# y \# suf\text{-}lm)[Suc\ (length\ lm) := y - Suc\ 0]$
$= lm @ x \# ya \# suf\text{-}lm \wedge ya \leq rsx$
apply(*case-tac y*, *simp*, *simp*)
apply(*rule-tac x = nat* **in** *exI*, *simp add*: *list-update-append*)
done

lemma *mn-halt-lemma*:
$[\![rec\text{-}ci\ f = (a, aa, ba);$
$rec\text{-}ci\ (Mn\ n\ f) = (aprog, rs\text{-}pos, a\text{-}md);$
$0 < rsx;\ length\ lm = n]\!]$
$\Longrightarrow$
$\forall\, na.\ \neg\ (\lambda(as, lm')\ (ss, n).\ as = 0)$
$(abc\text{-}steps\text{-}l\ (length\ a, lm @ x \# rsx \# suf\text{-}lm)\ aprog\ na)$
$(length\ a, n)$
$\wedge\ mn\text{-}ind\text{-}inv\ (abc\text{-}steps\text{-}l\ (length\ a, lm @ x \# rsx \# suf\text{-}lm)$
$aprog\ na)\ (length\ a)\ x\ rsx\ suf\text{-}lm\ lm$
$\longrightarrow mn\text{-}ind\text{-}inv\ (abc\text{-}steps\text{-}l\ (length\ a, lm @ x \# rsx \# suf\text{-}lm)\ aprog$
$(Suc\ na))\ (length\ a)\ x\ rsx\ suf\text{-}lm\ lm$
$\wedge\ ((abc\text{-}steps\text{-}l\ (length\ a, lm @ x \# rsx \# suf\text{-}lm)\ aprog\ (Suc\ na),$
$length\ a, n),$
$abc\text{-}steps\text{-}l\ (length\ a, lm @ x \# rsx \# suf\text{-}lm)\ aprog\ na,$
$length\ a, n) \in mn\text{-}LE$
apply(*rule allI*, *rule impI*, *simp add*: *abc-steps-ind*)
apply(*case-tac* (*abc-steps-l* (*length a*, *lm* @ *x* # *rsx* # *suf-lm*)
aprog na), *simp*)
apply(*auto split*:*if-splits simp add*:*abc-steps-l.simps*
mn-ind-inv.simps abc-steps-zero)
apply(*auto simp add*: *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
split: *if-splits*)
apply(*drule-tac rec-ci-not-null*, *simp*)
done

lemma *mn-halt*:
$[\![rec\text{-}ci\ f = (a, aa, ba);$
$rec\text{-}ci\ (Mn\ n\ f) = (aprog, rs\text{-}pos, a\text{-}md);$
$0 < rsx;\ length\ lm = n]\!]$
$\Longrightarrow \exists\ stp.\ (\lambda\ (as, lm').\ (as = 0\ \wedge$
$mn\text{-}ind\text{-}inv\ (as, lm')\ (length\ a)\ x\ rsx\ suf\text{-}lm\ lm))$
$(abc\text{-}steps\text{-}l\ (length\ a, lm @ x \# rsx \# suf\text{-}lm)\ aprog\ stp)$
apply(*insert wf-mn-le*)

```
apply(insert halt-lemma2[of mn-LE
  λ ((as, lm'), ss, n). as = 0
  λ stp. (abc-steps-l (length a, lm @ x # rsx # suf-lm) aprog stp,
   length a, n)
   λ ((as, lm'), ss, n). mn-ind-inv (as, lm') ss x rsx suf-lm lm],
   simp)
apply(simp add: mn-halt-init mn-inv-init)
apply(drule-tac x = x and suf-lm = suf-lm in mn-halt-lemma, auto)
apply(rule-tac x = n in exI,
      case-tac (abc-steps-l (length a, lm @ x # rsx # suf-lm)
                             aprog n), simp)
done

lemma [simp]: Suc rs-pos < a-md ⟹
                Suc (a-md − Suc (Suc rs-pos)) = a-md − Suc rs-pos
by arith

term rec-ci

lemma mn-ind-step:
  assumes ind:
  ⋀aprog a-md rs-pos rs suf-lm lm.
  ⟦rec-ci f = (aprog, rs-pos, a-md);
   rec-calc-rel f lm rs⟧ ⟹
  ∃ stp. abc-steps-l (0, lm @ 0^(a-md − rs-pos) @ suf-lm) aprog stp
          = (length aprog, lm @ [rs] @ 0^(a-md − rs-pos − 1) @ suf-lm)
  and h: rec-ci f = (a, aa, ba)
        rec-ci (Mn n f) = (aprog, rs-pos, a-md)
        rec-calc-rel f (lm @ [x]) rsx
        rsx > 0
        Suc rs-pos < a-md
        aa = Suc rs-pos
  shows ∃ stp. abc-steps-l (0, lm @ x # 0^(a-md − Suc rs-pos) @ suf-lm)
             aprog stp = (0, lm @ Suc x # 0^(a-md − Suc rs-pos) @ suf-lm)
thm abc-add-exc1
proof −
  have k1:
    ∃ stp. abc-steps-l (0, lm @ x #  0^(a-md − Suc (rs-pos)) @ suf-lm)
        aprog stp =
      (length a, lm @ x # rsx # 0^(a-md  − Suc (Suc rs-pos)) @ suf-lm)
    apply(insert h)
    apply(auto intro: mn-calc-f ind)
    done
  from h have k2: length lm = n
    apply(subgoal-tac rs-pos = n)
    apply(drule-tac  para-pattern, simp, simp, simp)
    done
  from h have k3: a-md > (Suc rs-pos)
    apply(simp)
```

```
    done
  from k2 and h and k3 have k4:
    ∃ stp. abc-steps-l (length a,
       lm @ x # rsx # 0^(a-md - Suc (Suc rs-pos)) @ suf-lm) aprog stp =
        (0, lm @ Suc x # 0^(a-md - rs-pos - 1) @ suf-lm)
    apply(frule-tac x = x and
       suf-lm = 0^(a-md - Suc (Suc rs-pos)) @ suf-lm in mn-halt, auto)
    apply(rule-tac x = stp in exI,
          simp add: mn-ind-inv.simps rec-ci-not-null exponent-def)
    apply(simp only: replicate.simps[THEN sym], simp)
    done

  from k1 and k4 show ?thesis
    apply(auto)
    apply(rule-tac x = stp + stpa in exI, simp add: abc-steps-add)
    done
qed

lemma [simp]:
  ⟦rec-ci f = (a, aa, ba); rec-ci (Mn n f) = (aprog, rs-pos, a-md);
    rec-calc-rel (Mn n f) lm rs⟧ ⟹ aa = Suc rs-pos
apply(rule-tac calc-mn-reverse, simp)
apply(insert para-pattern [of f a aa ba lm @ [rs] 0], simp)
apply(subgoal-tac rs-pos = length lm, simp)
apply(drule-tac ci-mn-para-eq, simp)
done

lemma [simp]: ⟦rec-ci (Mn n f) = (aprog, rs-pos, a-md);
                rec-calc-rel (Mn n f) lm rs⟧ ⟹ Suc rs-pos < a-md
apply(case-tac rec-ci f)
apply(subgoal-tac c > b ∧ b = Suc rs-pos ∧ a-md ≥ c)
apply(arith, auto)
done

lemma mn-ind-steps:
  assumes ind:
  ⋀aprog a-md rs-pos rs suf-lm lm.
  ⟦rec-ci f = (aprog, rs-pos, a-md); rec-calc-rel f lm rs⟧ ⟹
  ∃ stp. abc-steps-l (0, lm @ 0^(a-md - rs-pos) @ suf-lm) aprog stp =
            (length aprog, lm @ [rs] @ 0^(a-md - rs-pos - 1) @ suf-lm)
  and h: rec-ci f = (a, aa, ba)
  rec-ci (Mn n f) = (aprog, rs-pos, a-md)
  rec-calc-rel (Mn n f) lm rs
  rec-calc-rel f (lm @ [rs]) 0
  ∀ x<rs. (∃ v. rec-calc-rel f (lm @ [x]) v ∧ 0 < v)
  n = length lm
  x ≤ rs
  shows ∃ stp. abc-steps-l (0, lm @ 0 # 0^(a-md - Suc rs-pos) @ suf-lm)
```

```
          aprog stp = (0, lm @ x # 0^(a-md - Suc rs-pos) @ suf-lm)
apply(insert h, induct x,
   rule-tac x = 0 in exI, simp add: abc-steps-zero, simp)
proof -
 fix x
 assume k1:
  ∃stp. abc-steps-l (0, lm @ 0 # 0^(a-md - Suc rs-pos) @ suf-lm)
          aprog stp = (0, lm @ x # 0^(a-md - Suc rs-pos) @ suf-lm)
 and k2: rec-ci (Mn (length lm) f) = (aprog, rs-pos, a-md)
      rec-calc-rel (Mn (length lm) f) lm rs
      rec-calc-rel f (lm @ [rs]) 0
      ∀x<rs.(∃ v. rec-calc-rel f (lm @ [x]) v ∧ v > 0)
      n = length lm
      Suc x ≤ rs
      rec-ci f = (a, aa, ba)
 hence k2:
  ∃stp. abc-steps-l (0, lm @ x # 0^(a-md - rs-pos - 1) @ suf-lm) aprog
         stp = (0, lm @ Suc x # 0^(a-md - rs-pos - 1) @ suf-lm)
  apply(erule-tac x = x in allE)
  apply(auto)
  apply(rule-tac  x = x in mn-ind-step)
  apply(rule-tac ind, auto)
  done
 from k1 and k2 show
  ∃stp. abc-steps-l (0, lm @ 0 # 0^(a-md - Suc rs-pos) @ suf-lm)
      aprog stp = (0, lm @ Suc x # 0^(a-md - Suc rs-pos) @ suf-lm)
  apply(auto)
  apply(rule-tac x = stp + stpa in exI, simp only: abc-steps-add)
  done
qed

lemma [simp]:
⟦rec-ci f = (a, aa, ba);
 rec-ci (Mn n f) = (aprog, rs-pos, a-md);
 rec-calc-rel (Mn n f) lm rs;
 length lm = n⟧
 ⟹ abc-lm-v (lm @ rs # 0^(a-md - Suc rs-pos) @ suf-lm) (Suc n) = 0
apply(auto simp: abc-lm-v.simps nth-append)
done

lemma [simp]:
 ⟦rec-ci f = (a, aa, ba);
  rec-ci (Mn n f) = (aprog, rs-pos, a-md);
  rec-calc-rel (Mn n f) lm rs;
   length lm = n⟧
  ⟹ abc-lm-s (lm @ rs # 0^(a-md - Suc rs-pos) @ suf-lm) (Suc n) 0 =
                   lm @ rs # 0^(a-md - Suc rs-pos) @ suf-lm
apply(auto simp: abc-lm-s.simps list-update-append)
done
```

```
lemma mn-length:
  ⟦rec-ci f = (a, aa, ba);
    rec-ci (Mn n f) = (aprog, rs-pos, a-md)⟧
  ⟹ length aprog = length a + 5
apply(simp add: rec-ci.simps, erule-tac conjE)
apply(drule-tac eq-switch, drule-tac eq-switch, simp)
done

lemma mn-final-step:
  assumes ind:
  ⋀aprog a-md rs-pos rs suf-lm lm.
  ⟦rec-ci f = (aprog, rs-pos, a-md);
  rec-calc-rel f lm rs⟧ ⟹
  ∃ stp. abc-steps-l (0, lm @ 0^(a-md − rs-pos) @ suf-lm) aprog stp =
            (length aprog, lm @ [rs] @ 0^(a-md − rs-pos − 1) @ suf-lm)
  and h: rec-ci f = (a, aa, ba)
         rec-ci (Mn n f) = (aprog, rs-pos, a-md)
         rec-calc-rel (Mn n f) lm rs
         rec-calc-rel f (lm @ [rs]) 0
  shows ∃ stp. abc-steps-l (0, lm @ rs # 0^(a-md − Suc rs-pos) @ suf-lm)
    aprog stp = (length aprog, lm @ rs # 0^(a-md − Suc rs-pos) @ suf-lm)
proof −
  from h and ind have k1:
    ∃ stp.  abc-steps-l (0, lm @ rs # 0^(a-md − Suc rs-pos) @ suf-lm)
        aprog stp = (length a,  lm @ rs # 0^(a-md − Suc rs-pos) @ suf-lm)
    thm mn-calc-f
    apply(insert mn-calc-f[of f a aa ba n aprog
                          rs-pos a-md lm rs 0 suf-lm], simp)
    apply(subgoal-tac aa = Suc n, simp add: exponent-cons-iff)
    apply(subgoal-tac rs-pos = n, simp, simp)
    done
  from h have k2: length lm = n
    apply(subgoal-tac rs-pos = n)
    apply(drule-tac f = Mn n f in para-pattern, simp, simp, simp)
    done
  from h and k2 have k3:
  ∃ stp. abc-steps-l (length a, lm @ rs # 0^(a-md − Suc rs-pos) @ suf-lm)
    aprog stp = (length a + 5, lm @ rs # 0^(a-md − Suc rs-pos) @ suf-lm)
    apply(rule-tac x = Suc 0 in exI,
          simp add: abc-step-l.simps abc-steps-l.simps)
    done
  from h have k4: length aprog = length a + 5
    apply(simp add: mn-length)
    done
  from k1 and k3 and k4 show ?thesis
    apply(auto)
    apply(rule-tac x = stp + stpa in exI, simp add: abc-steps-add)
    done
```

qed

lemma *mn-case*:
  assumes *ind*:
  $\bigwedge$*aprog a-md rs-pos rs suf-lm lm*.
  $[\![$*rec-ci f* = (*aprog*, *rs-pos*, *a-md*); *rec-calc-rel f lm rs*$]\!] \Longrightarrow$
  $\exists$ *stp*. *abc-steps-l* (*0*, *lm* @ *0*$^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog stp* =
      (*length aprog*, *lm* @ [*rs*] @ *0*$^{a\text{-}md\ -\ rs\text{-}pos\ -\ 1}$ @ *suf-lm*)
  and *h*: *rec-ci* (*Mn n f*) = (*aprog*, *rs-pos*, *a-md*)
    rec-calc-rel (*Mn n f*) *lm rs*
  shows $\exists$ *stp*. *abc-steps-l* (*0*, *lm* @ *0*$^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog stp*
  = (*length aprog*, *lm* @ [*rs*] @ *0*$^{a\text{-}md\ -\ rs\text{-}pos\ -\ 1}$ @ *suf-lm*)
apply(*case-tac rec-ci f*, *simp*)
apply(*insert h*, *rule-tac calc-mn-reverse*, *simp*)
proof −
  fix *a b c v*
  assume *h*: *rec-ci f* = (*a*, *b*, *c*)
    rec-ci (*Mn n f*) = (*aprog*, *rs-pos*, *a-md*)
    rec-calc-rel (*Mn n f*) *lm rs*
    rec-calc-rel f (*lm* @ [*rs*]) *0*
    $\forall$ *x*<*rs*. $\exists$ *v*. *rec-calc-rel f* (*lm* @ [*x*]) *v* $\wedge$ *0* < *v*
    n = *length lm*
  hence *k1*:
    $\exists$ *stp*. *abc-steps-l* (*0*, *lm* @ *0* # *0*$^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*) *aprog*
      stp = (*0*, *lm* @ *rs* # *0*$^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
    thm *mn-ind-steps*
    apply(*auto intro*: *mn-ind-steps ind*)
    done
  from *h* **have** *k2*:
    $\exists$ *stp*. *abc-steps-l* (*0*, *lm* @ *rs* # *0*$^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*) *aprog*
      stp = (*length aprog*, *lm* @ *rs* # *0*$^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
    apply(*auto intro*: *mn-final-step ind*)
    done
  from *k1* **and** *k2* **show**
    $\exists$ *stp*. *abc-steps-l* (*0*, *lm* @ *0*$^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog stp* =
  (*length aprog*, *lm* @ *rs* # *0*$^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
    apply(*auto*, *insert h*)
    apply(*subgoal-tac Suc rs-pos* < *a-md*)
    apply(*rule-tac x* = *stp* + *stpa* **in** *exI*,
      simp only: *abc-steps-add exponent-cons-iff*, *simp*, *simp*)
    done
qed

lemma *z-rs*: *rec-calc-rel z lm rs* $\Longrightarrow$ *rs* = *0*
apply(*rule-tac calc-z-reverse*, *auto*)
done

lemma *z-case*:

```
⟦rec-ci z = (aprog, rs-pos, a-md); rec-calc-rel z lm rs⟧
⟹ ∃ stp. abc-steps-l (0, lm @ 0^(a-md − rs-pos) @ suf-lm) aprog stp =
      (length aprog, lm @ [rs] @ 0^(a-md − rs-pos − 1) @ suf-lm)
apply(simp add: rec-ci.simps rec-ci-z-def, auto)
apply(rule-tac x = Suc 0 in exI, simp add: abc-steps-l.simps
                              abc-fetch.simps abc-step-l.simps z-rs)
done
thm addition.simps

thm addition.simps
thm rec-ci-s-def
fun addition-inv :: nat × nat list ⇒ nat ⇒ nat ⇒ nat ⇒
                    nat list ⇒ bool
  where
  addition-inv (as, lm') m n p lm =
       (let sn = lm ! n in
        let sm = lm ! m in
        lm ! p = 0 ∧
           (if as = 0 then ∃ x. x ≤ lm ! m ∧ lm' = lm[m := x,
                               n := (sn + sm − x), p := (sm − x)]
           else if as = 1 then ∃ x. x < lm ! m ∧ lm' = lm[m := x,
                        n := (sn + sm − x − 1), p := (sm − x − 1)]
           else if as = 2 then ∃ x. x < lm ! m ∧ lm' = lm[m := x,
                          n := (sn + sm − x), p := (sm − x − 1)]
           else if as = 3 then ∃ x. x < lm ! m ∧ lm' = lm[m := x,
                              n := (sn + sm − x), p := (sm − x)]
           else if as = 4 then ∃ x. x ≤ lm ! m ∧ lm' = lm[m := x,
                                  n := (sn + sm), p := (sm − x)]
           else if as = 5 then ∃ x. x < lm ! m ∧ lm' = lm[m := x,
                              n := (sn + sm), p := (sm − x − 1)]
           else if as = 6 then ∃ x. x < lm ! m ∧ lm' =
                  lm[m := Suc x, n := (sn + sm), p := (sm − x − 1)]
           else if as = 7 then lm' = lm[m := sm, n := (sn + sm)]
           else False))

fun addition-stage1 :: nat × nat list ⇒ nat ⇒ nat ⇒ nat
  where
  addition-stage1 (as, lm) m p =
         (if as = 0 ∨ as = 1 ∨ as = 2 ∨ as = 3 then 2
          else if as = 4 ∨ as = 5 ∨ as = 6 then 1
          else 0)

fun addition-stage2 :: nat × nat list ⇒ nat ⇒  nat ⇒ nat
  where
  addition-stage2 (as, lm) m p =
             (if 0 ≤ as ∧ as ≤ 3 then lm ! m
              else if 4 ≤ as ∧ as ≤ 6 then lm ! p
              else 0)
```

fun *addition-stage3* :: *nat* × *nat list* ⇒ *nat* ⇒ *nat* ⇒ *nat*
  where
  addition-stage3 (*as*, *lm*) *m p* =
    (*if as* = *1 then 4*
     else if as = *2 then 3*
     else if as = *3 then 2*
     else if as = *0 then 1*
     else if as = *5 then 2*
     else if as = *6 then 1*
     else if as = *4 then 0*
     else 0)

fun *addition-measure* :: ((*nat* × *nat list*) × *nat* × *nat*) ⇒
                                    (*nat* × *nat* × *nat*)
  where
  addition-measure ((*as*, *lm*), *m*, *p*) =
    (*addition-stage1* (*as*, *lm*) *m p*,
     addition-stage2 (*as*, *lm*) *m p*,
     addition-stage3 (*as*, *lm*) *m p*)

definition *addition-LE* :: (((*nat* × *nat list*) × *nat* × *nat*) ×
                      ((*nat* × *nat list*) × *nat* × *nat*)) *set*
  where *addition-LE* ≡ (*inv-image lex-triple addition-measure*)

lemma [*simp*]: *wf addition-LE*
by(*simp add*: *wf-inv-image wf-lex-triple addition-LE-def*)

declare *addition-inv.simps*[*simp del*]

lemma *addition-inv-init*:
  ⟦*m* ≠ *n*; *max m n* < *p*; *length lm* > *p*; *lm* ! *p* = *0*⟧ ⟹
                              addition-inv (*0*, *lm*) *m n p lm*
apply(*simp add*: *addition-inv.simps*)
apply(*rule-tac x* = *lm* ! *m* **in** *exI*, *simp*)
done

thm *addition.simps*

lemma [*simp*]: *abc-fetch 0* (*addition m n p*) = *Some* (*Dec m 4*)
by(*simp add*: *abc-fetch.simps addition.simps*)

lemma [*simp*]: *abc-fetch* (*Suc 0*) (*addition m n p*) = *Some* (*Inc n*)
by(*simp add*: *abc-fetch.simps addition.simps*)

lemma [*simp*]: *abc-fetch 2* (*addition m n p*) = *Some* (*Inc p*)
by(*simp add*: *abc-fetch.simps addition.simps*)

lemma [*simp*]: *abc-fetch 3* (*addition m n p*) = *Some* (*Goto 0*)
by(*simp add*: *abc-fetch.simps addition.simps*)

lemma [*simp*]: *abc-fetch 4* (*addition m n p*) = *Some* (*Dec p 7*)
by(*simp add*: *abc-fetch.simps addition.simps*)

lemma [*simp*]: *abc-fetch 5* (*addition m n p*) = *Some* (*Inc m*)
by(*simp add*: *abc-fetch.simps addition.simps*)

lemma [*simp*]: *abc-fetch 6* (*addition m n p*) = *Some* (*Goto 4*)
by(*simp add*: *abc-fetch.simps addition.simps*)

lemma [*simp*]:
$[\![m \neq n;\ p < \mathit{length}\ lm;\ lm\ !\ p = 0;\ m < p;\ n < p;\ x \leq lm\ !\ m;\ 0 < x]\!]$
$\Longrightarrow \exists\, xa{<}lm\ !\ m.\ lm[m := x,\ n := lm\ !\ n + lm\ !\ m - x,$
$p := lm\ !\ m - x,\ m := x - \mathit{Suc}\ 0] =$
$lm[m := xa,\ n := lm\ !\ n + lm\ !\ m - \mathit{Suc}\ xa,$
$p := lm\ !\ m - \mathit{Suc}\ xa]$
apply(*case-tac x*, *simp*, *simp*)
apply(*rule-tac x* = *nat* **in** *exI*, *simp add*: *list-update-swap*
list-update-overwrite)
done

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

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

lemma *addition-halt-lemma*:
$[\![m \neq n; \mathit{max}\ m\ n < p; \mathit{length}\ lm > p; lm\ !\ p = 0]\!] \Longrightarrow$
$\forall na.\ \neg\ (\lambda(as, lm')\ (m, p).\ as = 7)$
$(\mathit{abc\text{-}steps\text{-}l}\ (0, lm)\ (\mathit{addition}\ m\ n\ p)\ na)\ (m, p) \wedge$
$\mathit{addition\text{-}inv}\ (\mathit{abc\text{-}steps\text{-}l}\ (0, lm)\ (\mathit{addition}\ m\ n\ p)\ na)\ m\ n\ p\ lm$
$\longrightarrow \mathit{addition\text{-}inv}\ (\mathit{abc\text{-}steps\text{-}l}\ (0, lm)\ (\mathit{addition}\ m\ n\ p)$
$(\mathit{Suc}\ na))\ m\ n\ p\ lm$
$\wedge\ ((\mathit{abc\text{-}steps\text{-}l}\ (0, lm)\ (\mathit{addition}\ m\ n\ p)\ (\mathit{Suc}\ na), m, p),$
$\mathit{abc\text{-}steps\text{-}l}\ (0, lm)\ (\mathit{addition}\ m\ n\ p)\ na, m, p) \in \mathit{addition\text{-}LE}$
apply(*rule allI*, *rule impI*, *simp add*: *abc-steps-ind*)
apply(*case-tac* (*abc-steps-l* (*0*, *lm*) (*addition m n p*) *na*), *simp*)
apply(*auto split*:*if-splits simp add*: *addition-inv.simps*
abc-steps-zero)
apply(*simp-all add*: *abc-steps-l.simps abc-steps-zero*)
apply(*auto simp add*: *addition-LE-def lex-triple-def lex-pair-def*
abc-step-l.simps addition-inv.simps
abc-lm-v.simps abc-lm-s.simps nth-append
split: *if-splits*)
apply(*rule-tac* $x = x$ **in** *exI*, *simp*)
done

lemma *addition-ex*:
  ⟦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). as = 7
  λ stp. (abc-steps-l (0, lm) (addition m n p) stp, m, p)
  λ ((as, lm′), m, p). addition-inv (as, lm′) m n p lm],
  simp add: abc-steps-zero addition-inv-init)
apply(drule-tac addition-halt-lemma, simp, simp, simp,
      simp, erule-tac exE)
apply(rule-tac x = na **in** exI,
      case-tac (abc-steps-l (0, lm) (addition m n p) na), auto)
done

lemma [simp]: length (addition m n p) = 7
by (simp add: addition.simps)

lemma [elim]: addition 0 (Suc 0) 2 = [] ⟹ RR
by(simp add: addition.simps)

lemma [simp]: (0^2)[0 := n] = [n, 0::nat]
apply(subgoal-tac 2 = Suc 1,
      simp only: replicate.simps exponent-def)
apply(auto)
done

lemma [simp]:
  ∃ stp. abc-steps-l (0, n # 0^2 @ suf-lm)
    (addition 0 (Suc 0) 2 [+] [Inc (Suc 0)]) stp =
                    (8, n # Suc n # 0 # suf-lm)
apply(rule-tac bm = n # n # 0 # suf-lm **in** abc-append-exc2, auto)
apply(insert addition-ex[of 0 Suc 0 2 n # 0^2 @ suf-lm],
      simp add: nth-append numeral-2-eq-2, erule-tac exE)
apply(rule-tac x = stp **in** exI,
      case-tac (abc-steps-l (0, n # 0^2 @ suf-lm)
                  (addition 0 (Suc 0) 2) stp),
      simp add: addition-inv.simps nth-append list-update-append numeral-2-eq-2)
apply(simp add: nth-append numeral-2-eq-2, erule-tac exE)
apply(rule-tac x = Suc 0 **in** exI,
      simp add: abc-steps-l.simps abc-fetch.simps
      abc-steps-zero abc-step-l.simps abc-lm-s.simps abc-lm-v.simps)
done

lemma *s-case*:
  ⟦rec-ci s = (aprog, rs-pos, a-md); rec-calc-rel s lm rs⟧
  ⟹ ∃ stp. abc-steps-l (0, lm @ $0^{a\text{-}md - rs\text{-}pos}$ @ suf-lm) aprog stp =
          (length aprog, lm @ [rs] @ $0^{a\text{-}md - rs\text{-}pos - 1}$ @ suf-lm)
apply(simp add: rec-ci.simps rec-ci-s-def, auto)

apply(*rule-tac calc-s-reverse*, *auto*)
done

lemma [*simp*]:
  $[\![n < length\ lm;\ lm\ !\ n = rs]\!]$
    $\Longrightarrow \exists\, stp.\ abc\text{-}steps\text{-}l\ (0,\ lm\ @\ 0\ \#\ 0\ \#suf\text{-}lm)$
      $(addition\ n\ (length\ lm)\ (Suc\ (length\ lm)))\ stp$
      $= (7,\ lm\ @\ rs\ \#\ 0\ \#\ suf\text{-}lm)$
apply(*insert addition-ex*[*of n length lm*
  Suc (*length lm*) *lm* @ *0* # *0* # *suf-lm*])
apply(*simp add*: *nth-append*, *erule-tac exE*)
apply(*rule-tac x* = *stp* **in** *exI*)
apply(*case-tac abc-steps-l* (*0*, *lm* @ *0* # *0* # *suf-lm*) (*addition n* (*length lm*)
  (*Suc* (*length lm*))) *stp*, *simp*)
apply(*simp add*: *addition-inv.simps*)
apply(*insert nth-append*[*of lm 0* # *0* # *suf-lm n*], *simp*)
done

lemma [*simp*]: $0^2 = [0,\ 0::nat]$
apply(*auto simp*: *exponent-def numeral-2-eq-2*)
done

lemma *id-case*:
  $[\![rec\text{-}ci\ (id\ m\ n) = (aprog,\ rs\text{-}pos,\ a\text{-}md);$
    $rec\text{-}calc\text{-}rel\ (id\ m\ n)\ lm\ rs]\!]$
  $\Longrightarrow \exists\, stp.\ abc\text{-}steps\text{-}l\ (0,\ lm\ @\ 0^{a\text{-}md\ -\ rs\text{-}pos}\ @\ suf\text{-}lm)\ aprog\ stp =$
    $(length\ aprog,\ lm\ @\ [rs]\ @\ 0^{a\text{-}md\ -\ rs\text{-}pos\ -\ 1}\ @\ suf\text{-}lm)$
apply(*simp add*: *rec-ci.simps rec-ci-id.simps*, *auto*)
apply(*rule-tac calc-id-reverse*, *simp*, *simp*)
done

lemma *list-tl-induct*:
  $[\![P\ [];\ \bigwedge a\ list.\ P\ list \Longrightarrow P\ (list\ @\ [a::'a])]\!] \Longrightarrow$
    $P\ ((list::'a\ list))$
apply(*case-tac length list*, *simp*)
proof −
  fix *nat*
  assume *ind*: $\bigwedge a\ list.\ P\ list \Longrightarrow P\ (list\ @\ [a])$
  and *h*: *length list* = *Suc nat P* []
  from *h* **show** *P list*
  proof(*induct nat arbitrary*: *list*, *case-tac lista*, *simp*, *simp*)
    fix *lista a listaa*
    from *h* **show** *P* [*a*]
      by(*insert ind*[*of* []], *simp add*: *h*)
  next
    fix *nat list*
    assume *nind*: $\bigwedge list.\ [\![length\ list = Suc\ nat;\ P\ []]\!] \Longrightarrow P\ list$
    and *g*: *length* (*list*:: *'a list*) = *Suc* (*Suc nat*)
    from *g* **show** *P* (*list*::*'a list*)

```
    apply(insert nind[of butlast list], simp add: h)
    apply(insert ind[of butlast list last list], simp)
    apply(subgoal-tac butlast list @ [last list] = list, simp)
    apply(case-tac list::'a list, simp, simp)
    done
  qed
qed

thm list.induct

lemma nth-eq-butlast-nth: ⟦length ys > Suc k⟧ ⟹
                                    ys ! k = butlast ys ! k
apply(subgoal-tac ∃ xs y. ys = xs @ [y], auto simp: nth-append)
apply(rule-tac x = butlast ys in exI, rule-tac x = last ys in exI)
apply(case-tac ys = [], simp, simp)
done

lemma [simp]:
⟦∀ k<Suc (length list). rec-calc-rel ((list @ [a]) ! k) lm (ys ! k);
  length ys = Suc (length list)⟧
   ⟹ ∀ k<length list. rec-calc-rel (list ! k) lm (butlast ys ! k)
apply(rule allI, rule impI)
apply(erule-tac  x = k in allE, simp add: nth-append)
apply(subgoal-tac ys ! k = butlast ys ! k, simp)
apply(rule-tac nth-eq-butlast-nth, arith)
done


thm cn-merge-gs.simps
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, simp)
apply(case-tac a, simp add: abc-append-commute)
done

lemma cn-merge-gs-length:
  length (cn-merge-gs (map rec-ci list) pstr) =
     (∑ (ap, pos, n)←map rec-ci list. length ap) + 3 ∗ length list
apply(induct list arbitrary: pstr, simp, simp)
apply(case-tac rec-ci a, simp)
done

lemma [simp]: Suc n ≤ pstr ⟹ pstr + x − n > 0
by arith

lemma [simp]:
  ⟦Suc (pstr + length list) ≤ a-md;
    length ys = Suc (length list);
```

length lm = *n*;
Suc n $\leq$ *pstr*⟧
$\Longrightarrow$ (*ys* ! *length list* # $0^{pstr - Suc\ n}$ @ *butlast ys* @
$0^{a\text{-}md - (pstr + length\ list)}$ @ *suf-lm*) !
(*pstr* + *length list* − *n*) = (*0* :: *nat*)
apply(*insert nth-append*[*of ys* ! *length list* # $0^{pstr - Suc\ n}$ @
butlast ys $0^{a\text{-}md - (pstr + length\ list)}$ @ *suf-lm*
(*pstr* + *length list* − *n*)], *simp add*: *nth-append*)
done

lemma [*simp*]:
⟦*Suc* (*pstr* + *length list*) $\leq$ *a-md*;
length ys = *Suc* (*length list*);
length lm = *n*;
Suc n $\leq$ *pstr*⟧
$\Longrightarrow$ (*lm* @ *last ys* # $0^{pstr - Suc\ n}$ @ *butlast ys* @
$0^{a\text{-}md - (pstr + length\ list)}$ @ *suf-lm*)[*pstr* + *length list* :=
last ys, *n* := *0*] =
lm @ $0{::}nat^{pstr - n}$ @ *ys* @ $0^{a\text{-}md - Suc\ (pstr + length\ list)}$ @ *suf-lm*
apply(*insert list-update-length*[*of*
lm @ *last ys* # $0^{pstr - Suc\ n}$ @ *butlast ys 0*
$0^{a\text{-}md - Suc\ (pstr + length\ list)}$ @ *suf-lm last ys*], *simp*)
apply(*simp add*: *exponent-cons-iff*)
apply(*insert list-update-length*[*of lm*
last ys $0^{pstr - Suc\ n}$ @ *butlast ys* @
last ys # $0^{a\text{-}md - Suc\ (pstr + length\ list)}$ @ *suf-lm 0*], *simp*)
apply(*simp add*: *exponent-cons-iff*)
apply(*case-tac ys* = [], *simp-all add*: *append-butlast-last-id*)
done

lemma *cn-merge-gs-ex*:
⟦$\bigwedge$*x aprog a-md rs-pos rs suf-lm lm*.
⟦*x* $\in$ *set gs*; *rec-ci x* = (*aprog*, *rs-pos*, *a-md*);
rec-calc-rel x lm rs⟧
$\Longrightarrow$ $\exists$ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md - rs\text{-}pos}$ @ *suf-lm*) *aprog stp*
= (*length aprog*, *lm* @ [*rs*] @ $0^{a\text{-}md - rs\text{-}pos - 1}$ @ *suf-lm*);
pstr + *length gs*$\leq$ *a-md*;
$\forall$ *k*<*length gs*. *rec-calc-rel* (*gs* ! *k*) *lm* (*ys* ! *k*);
length ys = *length gs*; *length lm* = *n*;
pstr $\geq$ *Max* (*set* (*Suc n* # *map* (λ(*aprog*, *p*, *n*). *n*) (*map rec-ci gs*)))⟧
$\Longrightarrow$ $\exists$ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md - n}$ @ *suf-lm*)
(*cn-merge-gs* (*map rec-ci gs*) *pstr*) *stp*
= (*listsum* (*map* ((λ(*ap*, *pos*, *n*). *length ap*) $\circ$ *rec-ci*) *gs*) +
3 $*$ *length gs*, *lm* @ $0^{pstr - n}$ @ *ys* @ $0^{a\text{-}md - (pstr + length\ gs)}$ @ *suf-lm*)
apply(*induct gs arbitrary*: *ys rule*: *list-tl-induct*)
apply(*simp add*: *exponent-add-iff*, *simp*)

proof −
fix *a list ys*
assume *ind*: $\bigwedge x$ *aprog a-md rs-pos rs suf-lm lm*.
$[\![x = a \vee x \in set\ list;\ rec\text{-}ci\ x = (aprog,\ rs\text{-}pos,\ a\text{-}md);$
$rec\text{-}calc\text{-}rel\ x\ lm\ rs]\!]$
$\Longrightarrow \exists\, stp.\ abc\text{-}steps\text{-}l\ (0,\ lm\ @\ 0^{a\text{-}md\ -\ rs\text{-}pos}\ @\ suf\text{-}lm)\ aprog\ stp =$
$(length\ aprog,\ lm\ @\ rs\ \#\ 0^{a\text{-}md\ -\ Suc\ rs\text{-}pos}\ @\ suf\text{-}lm)$
and *ind2*:
$\bigwedge ys.\ [\![\bigwedge x\ aprog\ a\text{-}md\ rs\text{-}pos\ rs\ suf\text{-}lm\ lm.$
$[\![x \in set\ list;\ rec\text{-}ci\ x = (aprog,\ rs\text{-}pos,\ a\text{-}md);$
$rec\text{-}calc\text{-}rel\ x\ lm\ rs]\!]$
$\Longrightarrow \exists\, stp.\ abc\text{-}steps\text{-}l\ (0,\ lm\ @\ 0^{a\text{-}md\ -\ rs\text{-}pos}\ @\ suf\text{-}lm)\ aprog\ stp$
$= (length\ aprog,\ lm\ @\ rs\ \#\ 0^{a\text{-}md\ -\ Suc\ rs\text{-}pos}\ @\ suf\text{-}lm);$
$\forall\, k{<}length\ list.\ rec\text{-}calc\text{-}rel\ (list\ !\ k)\ lm\ (ys\ !\ k);$
$length\ ys = length\ list]\!]$
$\Longrightarrow \exists\, stp.\ abc\text{-}steps\text{-}l\ (0,\ lm\ @\ 0^{a\text{-}md\ -\ n}\ @\ suf\text{-}lm)$
$(cn\text{-}merge\text{-}gs\ (map\ rec\text{-}ci\ list)\ pstr)\ stp =$
$(listsum\ (map\ ((\lambda(ap,\ pos,\ n).\ length\ ap) \circ rec\text{-}ci)\ list) +$
$3 * length\ list,$
$lm\ @\ 0^{pstr\ -\ n}\ @\ ys\ @\ 0^{a\text{-}md\ -\ (pstr\ +\ length\ list)}\ @\ suf\text{-}lm)$
and *h*: $Suc\ (pstr + length\ list) \leq a\text{-}md$
$\forall\, k{<}Suc\ (length\ list).$
$rec\text{-}calc\text{-}rel\ ((list\ @\ [a])\ !\ k)\ lm\ (ys\ !\ k)$
$length\ ys = Suc\ (length\ list)$
$length\ lm = n$
$Suc\ n \leq pstr \wedge (\lambda(aprog,\ p,\ n).\ n)\ (rec\text{-}ci\ a) \leq pstr \wedge$
$(\forall\, a{\in}set\ list.\ (\lambda(aprog,\ p,\ n).\ n)\ (rec\text{-}ci\ a) \leq pstr)$
from *h* **have** *k1*:
$\exists\, stp.\ abc\text{-}steps\text{-}l\ (0,\ lm\ @\ 0^{a\text{-}md\ -\ n}\ @\ suf\text{-}lm)$
$(cn\text{-}merge\text{-}gs\ (map\ rec\text{-}ci\ list)\ pstr)\ stp =$
$(listsum\ (map\ ((\lambda(ap,\ pos,\ n).\ length\ ap) \circ rec\text{-}ci)\ list) +$
$3 * length\ list,\ lm\ @\ 0^{pstr\ -\ n}\ @\ butlast\ ys\ @$
$0^{a\text{-}md\ -\ (pstr\ +\ length\ list)}\ @\ suf\text{-}lm)$
apply(*rule-tac ind2*)
apply(*rule-tac ind, auto*)
done
from *k1* **and** *h* **show**
$\exists\, stp.\ abc\text{-}steps\text{-}l\ (0,\ lm\ @\ 0^{a\text{-}md\ -\ n}\ @\ suf\text{-}lm)$
$(cn\text{-}merge\text{-}gs\ (map\ rec\text{-}ci\ list\ @\ [rec\text{-}ci\ a])\ pstr)\ stp =$
$(listsum\ (map\ ((\lambda(ap,\ pos,\ n).\ length\ ap) \circ rec\text{-}ci)\ list) +$
$(\lambda(ap,\ pos,\ n).\ length\ ap)\ (rec\text{-}ci\ a) + (3 + 3 * length\ list),$
$lm\ @\ 0^{pstr\ -\ n}\ @\ ys\ @\ 0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ list)}\ @\ suf\text{-}lm)$
apply(*simp add*: *cn-merge-gs-tl-app*)
thm *abc-append-exc2*
apply(*rule-tac as* =
$(\sum (ap,\ pos,\ n){\leftarrow}map\ rec\text{-}ci\ list.\ length\ ap) + 3 * length\ list$
and $bm = lm\ @\ 0^{pstr\ -\ n}\ @\ butlast\ ys\ @$
$0^{a\text{-}md\ -\ (pstr\ +\ length\ list)}\ @\ suf\text{-}lm$

```
      and bs = (λ(ap, pos, n). length ap) (rec-ci a) + 3
      and bm' = lm @ 0^(pstr - n) @ ys @ 0^(a-md - Suc (pstr + length list)) @
                              suf-lm in abc-append-exc2, simp)
    apply(simp add: cn-merge-gs-length)
  proof -
    from h show
      ∃ bstp. abc-steps-l (0, lm @ 0^(pstr - n) @ butlast ys @
                              0^(a-md - (pstr + length list)) @ suf-lm)
           ((λ(gprog, gpara, gn). gprog [+] recursive.empty gpara
           (pstr + length list)) (rec-ci a)) bstp =
           ((λ(ap, pos, n). length ap) (rec-ci a) + 3,
          lm @ 0^(pstr - n) @ ys @ 0^(a-md - Suc (pstr + length list)) @ suf-lm)
      apply(case-tac rec-ci a, simp)
      apply(rule-tac as = length aa and
                bm = lm @ (ys ! (length list)) #
        0^(pstr - Suc n) @ butlast ys @ 0^(a-md - (pstr + length list)) @ suf-lm
        and bs = 3 and bm' = lm @ 0^(pstr - n) @ ys @
          0^(a-md - Suc (pstr + length list)) @ suf-lm in abc-append-exc2)
    proof -
      fix aa b c
      assume g: rec-ci a = (aa, b, c)
      from h and g have k2: b = n
  apply(erule-tac x = length list in allE, simp)
  apply(subgoal-tac length lm = b, simp)
  apply(rule para-pattern, simp, simp)
  done
      from h and g and this show
        ∃ astp. abc-steps-l (0, lm @ 0^(pstr - n) @ butlast ys @
                      0^(a-md - (pstr + length list)) @ suf-lm) aa astp =
        (length aa, lm @ ys ! length list # 0^(pstr - Suc n) @
                      butlast ys @ 0^(a-md - (pstr + length list)) @ suf-lm)
  apply(subgoal-tac c ≥ Suc n)
  apply(insert ind[of a aa b c lm ys ! length list
      0^(pstr - c) @ butlast ys @ 0^(a-md - (pstr + length list)) @ suf-lm], simp)
  apply(erule-tac x = length list in allE,
              simp add: exponent-add-iff)
  apply(rule-tac Suc-leI, rule-tac ci-ad-ge-paras, simp)
  done
    next
      fix aa b c
      show length aa = length aa by simp
    next
      fix aa b c
      assume rec-ci a = (aa, b, c)
      from h and this show
      ∃ bstp. abc-steps-l (0, lm @ ys ! length list #
        0^(pstr - Suc n) @ butlast ys @ 0^(a-md - (pstr + length list)) @ suf-lm)
              (recursive.empty b (pstr + length list)) bstp =
```

(*3*, *lm* @ $0^{pstr - n}$ @ *ys* @ $0^{a\text{-}md - Suc\ (pstr + length\ list)}$ @ *suf-lm*)
apply(*insert empty-ex* [*of b pstr* + *length list*
lm @ *ys* ! *length list* # $0^{pstr - Suc\ n}$ @ *butlast ys* @
$0^{a\text{-}md - (pstr + length\ list)}$ @ *suf-lm*], *simp*)
apply(*subgoal-tac b* = *n*)
apply(*simp add*: *nth-append split*: *if-splits*)
apply(*erule-tac x* = *length list* **in** *allE*, *simp*)
apply(*drule para-pattern*, *simp*, *simp*)
done
next
fix *aa b c*
show *3* = *length* (*recursive.empty b* (*pstr* + *length list*))
by *simp*
next
fix *aa b aaa ba*
show *length aa* + *3* = *length aa* + *3* **by** *simp*
next
fix *aa b c*
show *empty b* (*pstr* + *length list*) ≠ []
by(*simp add*: *empty.simps*)
qed
next
show (λ(*ap*, *pos*, *n*). *length ap*) (*rec-ci a*) + *3* =
length ((λ(*gprog*, *gpara*, *gn*). *gprog* [+]
recursive.empty gpara (*pstr* + *length list*)) (*rec-ci a*))
by(*case-tac rec-ci a*, *simp*)
next
show *listsum* (*map* ((λ(*ap*, *pos*, *n*). *length ap*) ∘ *rec-ci*) *list*) +
(λ(*ap*, *pos*, *n*). *length ap*) (*rec-ci a*) + (*3* + *3* ∗ *length list*)=
(∑(*ap*, *pos*, *n*)←*map rec-ci list*. *length ap*) + *3* ∗ *length list* +
((λ(*ap*, *pos*, *n*). *length ap*) (*rec-ci a*) + *3*) **by** *simp*
next
show (λ(*gprog*, *gpara*, *gn*). *gprog* [+]
recursive.empty gpara (*pstr* + *length list*)) (*rec-ci a*) ≠ []
by(*case-tac rec-ci a*,
simp add: *abc-append.simps abc-shift.simps*)
qed
qed

declare *drop-abc-lm-v-simp*[*simp del*]

lemma [*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*: *exponent-def rep-ind del*: *replicate.simps*)

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

lemma [*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 [*simp*]:
⟦*Suc n* ≤ *ba* − *aa*; *aa* < *ba*; *length lm1* = *aa*; *length lm2* = *Suc n*;
length lm3 = *ba* − *Suc* (*aa* + *n*)⟧
⟹ (*lm1* @ 0^n @ *last lm2* # *lm3* @ *butlast lm2* @ (*0*::*nat*) # *lm4*)
[*ba* + *n* := *last lm2*, *aa* + *n* := *0*] =
lm1 @ *0* # 0^n @ *lm3* @ *lm2* @ *lm4*
using *list-update-append*[*of lm1* @ 0^n @ *last lm2* # *lm3* @ *butlast lm2 0* # *lm4*
ba + *n last lm2*]
apply(*simp*)
apply(*simp add*: *list-update-append*)
apply(*simp only*: *exponent-cons-iff exp-suc*, *simp*)
apply(*case-tac lm2*, *simp*, *simp*)
done

lemma *mv-boxes-ex*:

```
  ⟦n ≤ ba − aa; ba > aa; length lm1 = aa;
    length (lm2::nat list) = n; length lm3 = ba − aa − n⟧
     ⟹ ∃ stp. abc-steps-l (0, lm1 @ lm2 @ lm3 @ 0^n @ lm4)
       (mv-boxes aa ba n) stp = (3 ∗ n, lm1 @ 0^n @ lm3 @ lm2 @ lm4)
apply(induct n arbitrary: lm2 lm3 lm4, simp)
apply(rule-tac x = 0 in exI, simp add: abc-steps-zero,
              simp add: mv-boxes.simps del: exp-suc-iff)
apply(rule-tac as = 3 ∗n and bm = lm1 @ 0^n @ last lm2 # lm3 @
               butlast lm2 @ 0 # lm4 in abc-append-exc2, simp-all)
apply(simp only: exponent-cons-iff, simp only: exp-suc, simp)
proof −
  fix n lm2 lm3 lm4
  assume ind:
    ⋀lm2 lm3 lm4. ⟦length lm2 = n; length lm3 = ba − (aa + n)⟧ ⟹
    ∃stp. abc-steps-l (0, lm1 @ lm2 @ lm3 @ 0^n @ lm4)
       (mv-boxes aa ba n) stp = (3 ∗ n, lm1 @ 0^n @ lm3 @ lm2 @ lm4)
  and h: Suc n ≤ ba − aa aa < ba length (lm1::nat list) = aa
         length (lm2::nat list) = Suc n
         length (lm3::nat list) = ba − Suc (aa + n)
  from h show
    ∃astp. abc-steps-l (0, lm1 @ lm2 @ lm3 @ 0^n @ 0 # lm4)
                    (mv-boxes aa ba n) astp =
       (3 ∗ n, lm1 @ 0^n @ last lm2 # lm3 @ butlast lm2 @ 0 # lm4)
    apply(insert ind[of butlast lm2 last lm2 # lm3 0 # lm4],
          simp)
    apply(subgoal-tac lm1 @ butlast lm2 @ last lm2 # lm3 @ 0^n @
              0 # lm4 = lm1 @ lm2 @ lm3 @ 0^n @ 0 # lm4, simp, simp)
    apply(case-tac lm2 = [], simp, simp)
    done
next
  fix n lm2 lm3 lm4
  assume h: Suc n ≤ ba − aa
            aa < ba
            length (lm1::nat list) = aa
            length (lm2::nat list) = Suc n
            length (lm3::nat list) = ba − Suc (aa + n)
  thus ∃bstp. abc-steps-l (0, lm1 @ 0^n @ last lm2 # lm3 @
                    butlast lm2 @ 0 # lm4)
                      (recursive.empty (aa + n) (ba + n)) bstp
               = (3, lm1 @ 0 # 0^n @ lm3 @ lm2 @ lm4)
    apply(insert empty-ex[of aa + n ba + n
       lm1 @ 0^n @ last lm2 # lm3 @ butlast lm2 @ 0 # lm4], simp)
    done
qed


lemma [simp]: ⟦Suc n ≤ aa − ba; ba < aa; length lm1 = ba;
       length lm2 = aa − Suc (ba + n); length lm3 = Suc n⟧
   ⟹ (lm1 @ butlast lm3 @ 0 # lm2 @ 0^n @ last lm3 # lm4) ! (aa + n) = last
```

lm3
using *nth-append*[*of lm1* @ *butlast lm3* @ *0* # *lm2* @ 0^n *last lm3* # *lm4 aa* + *n*]
apply(*simp*)
done

lemma [*simp*]: ⟦*Suc n* ≤ *aa* − *ba*; *ba* < *aa*; *length lm1* = *ba*;
length lm2 = *aa* − *Suc* (*ba* + *n*); *length lm3* = *Suc n*⟧
⟹ (*lm1* @ *butlast lm3* @ *0* # *lm2* @ 0^n @ *last lm3* # *lm4*) ! (*ba* + *n*) = *0*
apply(*simp add*: *nth-append*)
done

lemma [*simp*]: ⟦*Suc n* ≤ *aa* − *ba*; *ba* < *aa*; *length lm1* = *ba*;
length lm2 = *aa* − *Suc* (*ba* + *n*); *length lm3* = *Suc n*⟧
⟹ (*lm1* @ *butlast lm3* @ *0* # *lm2* @ 0^n @ *last lm3* # *lm4*)[*ba* + *n* := *last lm3*, *aa* + *n* := *0*]
= *lm1* @ *lm3* @ *lm2* @ *0* # 0^n @ *lm4*
using *list-update-append*[*of lm1* @ *butlast lm3* (*0*::′*a*) # *lm2* @ *0*::′a^n @ *last lm3* # *lm4*]
apply(*simp*)
using *list-update-append*[*of lm1* @ *butlast lm3* @ *last lm3* # *lm2* @ *0*::′a^n
last lm3 # *lm4 aa* + *n 0*]
apply(*simp*)
apply(*simp only*: *exp-ind-def*[*THEN sym*] *exp-suc*, *simp*)
apply(*case-tac lm3*, *simp*, *simp*)
done

lemma *mv-boxes-ex2*:
⟦*n* ≤ *aa* − *ba*;
ba < *aa*;
length (*lm1*::*nat list*) = *ba*;
length (*lm2*::*nat list*) = *aa* − *ba* − *n*;
length (*lm3*::*nat list*) = *n*⟧
⟹ ∃ *stp*. *abc-steps-l* (*0*, *lm1* @ 0^n @ *lm2* @ *lm3* @ *lm4*)
(*mv-boxes aa ba n*) *stp* =
(*3* ∗ *n*, *lm1* @ *lm3* @ *lm2* @ 0^n @ *lm4*)
apply(*induct n arbitrary*: *lm2 lm3 lm4*, *simp*)
apply(*rule-tac x* = *0* **in** *exI*, *simp add*: *abc-steps-zero*,
simp add: *mv-boxes.simps del*: *exp-suc-iff*)
apply(*rule-tac as* = *3* ∗*n* **and** *bm* = *lm1* @ *butlast lm3* @ *0* # *lm2* @
0^n @ *last lm3* # *lm4* **in** *abc-append-exc2*, *simp-all*)
apply(*simp only*: *exponent-cons-iff*, *simp only*: *exp-suc*, *simp*)
proof −
fix *n lm2 lm3 lm4*
assume *ind*:
⋀*lm2 lm3 lm4*. ⟦*length lm2* = *aa* − (*ba* + *n*); *length lm3* = *n*⟧ ⟹
∃ *stp*. *abc-steps-l* (*0*, *lm1* @ 0^n @ *lm2* @ *lm3* @ *lm4*)
(*mv-boxes aa ba n*) *stp* =

```
                  (3 * n, lm1 @ lm3 @ lm2 @ 0^n @ lm4)
  and h: Suc n ≤ aa − ba
       ba < aa
       length (lm1::nat list) = ba
       length (lm2::nat list) = aa − Suc (ba + n)
       length (lm3::nat list) = Suc n
  from h show
    ∃ astp. abc-steps-l (0, lm1 @ 0^n @ 0 # lm2 @ lm3 @ lm4)
      (mv-boxes aa ba n) astp =
        (3 * n, lm1 @ butlast lm3 @ 0 # lm2 @ 0^n @ last lm3 # lm4)
    apply(insert ind[of 0 # lm2 butlast lm3 last lm3 # lm4],
        simp)
    apply(subgoal-tac
      lm1 @ 0^n @ 0 # lm2 @ butlast lm3 @ last lm3 # lm4 =
          lm1 @ 0^n @ 0 # lm2 @ lm3 @ lm4, simp, simp)
    apply(case-tac lm3 = [], simp, simp)
    done
next
  fix n lm2 lm3 lm4
  assume h:
    Suc n ≤ aa − ba
    ba < aa
    length lm1 = ba
    length (lm2::nat list) = aa − Suc (ba + n)
    length (lm3::nat list) = Suc n
  thus
    ∃ bstp. abc-steps-l (0, lm1 @ butlast lm3 @ 0 # lm2 @ 0^n @
                          last lm3 # lm4)
        (recursive.empty (aa + n) (ba + n)) bstp =
            (3, lm1 @ lm3 @ lm2 @ 0 # 0^n @ lm4)
    apply(insert empty-ex[of aa + n ba + n lm1 @ butlast lm3 @
                    0 # lm2 @ 0^n @ last lm3 # lm4], simp)
    done
qed

lemma cn-merge-gs-len:
  length (cn-merge-gs (map rec-ci gs) pstr) =
      (∑ (ap, pos, n)←map rec-ci gs. length ap) + 3 * length gs
apply(induct gs arbitrary: pstr, simp, simp)
apply(case-tac rec-ci a, simp)
done

lemma [simp]: n < pstr ⟹
     Suc (pstr + length ys − n) = Suc (pstr + length ys) − n
by arith

lemma save-paras':
  ⟦length lm = n; pstr > n; a-md > pstr + length ys + n⟧
  ⟹ ∃ stp. abc-steps-l (0, lm @ 0^(pstr − n) @ ys @
```

$0^{a\text{-}md\ -\ pstr\ -\ length\ ys}$ @ *suf-lm*)
(mv-boxes 0 (pstr + Suc (length ys)) n) stp
= (*3* ∗ *n*, 0^{pstr} @ *ys* @ *0* # *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
thm *mv-boxes-ex*
apply(*insert mv-boxes-ex*[*of n pstr* + *Suc* (*length ys*) *0* [] *lm*
$0^{pstr\ -\ n}$ @ *ys* @ [*0*] $0^{a\text{-}md\ -\ pstr\ -\ length\ ys\ -\ n\ -\ Suc\ 0}$ @ *suf-lm*], *simp*)
apply(*erule-tac exE*, *rule-tac x* = *stp* **in** *exI*,
simp add: *exponent-add-iff*)
apply(*simp only*: *exponent-cons-iff*, *simp*)
done

lemma [*simp*]:
(*max ba* (*Max* (*insert ba* (((λ(*aprog*, *p*, *n*). *n*) *o rec-ci*) ' *set gs*))))
= (*Max* (*insert ba* (((λ(*aprog*, *p*, *n*). *n*) *o rec-ci*) ' *set gs*)))
apply(*rule min-max.sup-absorb2*, *auto*)
done

lemma [*simp*]:
((λ(*aprog*, *p*, *n*). *n*) ' *rec-ci* ' *set gs*) =
(((λ(*aprog*, *p*, *n*). *n*) *o rec-ci*) ' *set gs*)
apply(*induct gs*)
apply(*simp*, *simp*)
done

lemma *ci-cn-md-def*:
⟦*rec-ci* (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*);
rec-ci f = (*a*, *aa*, *ba*)⟧
⟹ *a-md* = *max* (*Suc n*) (*Max* (*insert ba* (((λ(*aprog*, *p*, *n*). *n*) *o*
rec-ci) ' *set gs*))) + *Suc* (*length gs*) + *n*
apply(*simp add*: *rec-ci.simps*, *auto*)
done

lemma *save-paras-prog-ex*:
⟦*rec-ci* (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*);
rec-ci f = (*a*, *aa*, *ba*);
pstr = *Max* (*set* (*Suc n* # *ba* # *map* (λ(*aprog*, *p*, *n*). *n*)
(*map rec-ci* (*f* # *gs*))))⟧
⟹ ∃ *ap bp cp*.
aprog = *ap* [+] *bp* [+] *cp* ∧
length ap = (∑(*ap*, *pos*, *n*)←*map rec-ci gs*. *length ap*) +
3 ∗ *length gs* ∧ *bp* = *mv-boxes 0* (*pstr* + *Suc* (*length gs*)) *n*
apply(*simp add*: *rec-ci.simps*)
apply(*rule-tac x* =
cn-merge-gs (*map rec-ci gs*) (*max* (*Suc n*) (*Max* (*insert ba*
(((λ(*aprog*, *p*, *n*). *n*) *o rec-ci*) ' *set gs*)))) **in** *exI*,
simp add: *cn-merge-gs-len*)
apply(*rule-tac x* =
mv-boxes (*max* (*Suc n*) (*Max* (*insert ba* (((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ' *set gs*))))

0 (length gs) [+] a [+]recursive.empty aa (max (Suc n)
(Max (insert ba (((λ(aprog, p, n). n) ∘ rec-ci) ‘ set gs)))) [+]
empty-boxes (length gs) [+] recursive.empty (max (Suc n)
(Max (insert ba (((λ(aprog, p, n). n) ∘ rec-ci) ‘ set gs)))) n [+]
mv-boxes (Suc (max (Suc n) (Max (insert ba (((λ(aprog, p, n). n) ∘ rec-ci)
‘ set gs))) + length gs)) 0 n **in** *exI, auto)*
apply(*simp add: abc-append-commute*)
done

lemma *save-paras*:
$[\![$*rec-ci (Cn n f gs) = (aprog, rs-pos, a-md);*
rs-pos = n;
∀ k<length gs. rec-calc-rel (gs ! k) lm (ys ! k);
length ys = length gs;
length lm = n;
rec-ci f = (a, aa, ba);
pstr = Max (set (Suc n # ba # map (λ(aprog, p, n). n)
(map rec-ci (f # gs))))$]\!]$
⟹ *∃ stp. abc-steps-l ((∑ (ap, pos, n)←map rec-ci gs. length ap) +*
3 ∗ length gs, lm @ $0^{pstr\ -\ n}$ *@ ys @*
$0^{a\text{-}md\ -\ pstr\ -\ length\ ys}$ *@ suf-lm) aprog stp =*
((∑ (ap, pos, n)←map rec-ci gs. length ap) +
3 ∗ length gs + 3 ∗ n,
0^{pstr} *@ ys @ 0 # lm @* $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ *@ suf-lm)*
proof −
assume *h*:
rec-ci (Cn n f gs) = (aprog, rs-pos, a-md)
rs-pos = n
∀ k<length gs. rec-calc-rel (gs ! k) lm (ys ! k)
length ys = length gs
length lm = n
rec-ci f = (a, aa, ba)
and *g*: *pstr = Max (set (Suc n # ba # map (λ(aprog, p, n). n)*
(map rec-ci (f # gs))))
from *h* **and** *g* **have** *k1*:
∃ ap bp cp. aprog = ap [+] bp [+] cp ∧
length ap = (∑ (ap, pos, n)←map rec-ci gs. length ap) +
3 ∗length gs ∧ bp = mv-boxes 0 (pstr + Suc (length ys)) n
apply(*drule-tac save-paras-prog-ex, auto*)
done
from *h* **have** *k2*:
∃ stp. abc-steps-l (0, lm @ $0^{pstr\ -\ n}$ *@ ys @*
$0^{a\text{-}md\ -\ pstr\ -\ length\ ys}$ *@ suf-lm)*
(mv-boxes 0 (pstr + Suc (length ys)) n) stp =
(3 ∗ n, 0^{pstr} *@ ys @ 0 # lm @* $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ *@*
suf-lm)
apply(*rule-tac save-paras′, simp, simp-all add: g*)
apply(*drule-tac a = a* **and** *aa = aa* **and** *ba = ba* **in**
ci-cn-md-def, simp, simp)

```
    done
  from k1 show
    ∃ stp. abc-steps-l ((∑ (ap, pos, n)←map rec-ci gs. length ap) +
        3 ∗ length gs, lm @ 0^(pstr − n) @ ys @
                0^(a-md − pstr − length ys) @ suf-lm) aprog stp =
            ((∑ (ap, pos, n)←map rec-ci gs. length ap) +
              3 ∗ length gs + 3 ∗ n,
              0^pstr @ ys @ 0 # lm @ 0^(a-md − Suc (pstr + length ys + n)) @ suf-lm)
  proof(erule-tac exE, erule-tac exE, erule-tac exE, erule-tac exE)
    fix ap bp apa cp
    assume aprog = ap [+] bp [+] cp ∧ length ap =
            (∑ (ap, pos, n)←map rec-ci gs. length ap) + 3 ∗ length gs
            ∧ bp = mv-boxes 0 (pstr + Suc (length ys)) n
    from this and k2 show ?thesis
      apply(simp)
      apply(rule-tac abc-append-exc1, simp, simp, simp)
      done
  qed
qed

lemma ci-cn-para-eq:
  rec-ci (Cn n f gs) = (aprog, rs-pos, a-md) ⟹ rs-pos = n
apply(simp add: rec-ci.simps, case-tac rec-ci f, simp)
done

lemma calc-gs-prog-ex:
  ⟦rec-ci (Cn n f gs) = (aprog, rs-pos, a-md);
    rec-ci f = (a, aa, ba);
    Max (set (Suc n # ba # map (λ(aprog, p, n). n)
                      (map rec-ci (f # gs)))) = pstr⟧
   ⟹ ∃ ap bp. aprog = ap [+] bp ∧
                ap = cn-merge-gs (map rec-ci gs) pstr
apply(simp add: rec-ci.simps)
apply(rule-tac x = mv-boxes 0 (Suc (max (Suc n)
   (Max (insert ba (((λ(aprog, p, n). n) ∘ rec-ci) ‘ set gs))) + length gs)) n [+]
   mv-boxes (max (Suc n) (Max (insert ba
  (((λ(aprog, p, n). n) ∘ rec-ci) ‘ set gs)))) 0 (length gs) [+]
   a [+] recursive.empty aa (max (Suc n)
    (Max (insert ba (((λ(aprog, p, n). n) ∘ rec-ci) ‘ set gs)))) [+]
   empty-boxes (length gs) [+] recursive.empty (max (Suc n)
    (Max (insert ba (((λ(aprog, p, n). n) ∘ rec-ci) ‘ set gs)))) n [+]
    mv-boxes (Suc (max (Suc n) (Max
    (insert ba (((λ(aprog, p, n). n) ∘ rec-ci) ‘ set gs))) + length gs)) 0 n
   in exI)
apply(auto simp: abc-append-commute)
done

lemma cn-calc-gs:
  assumes ind:
```

$\bigwedge$*x aprog a-md rs-pos rs suf-lm lm.*
⟦*x* ∈ *set gs*;
rec-ci x = (*aprog*, *rs-pos*, *a-md*);
rec-calc-rel x lm rs⟧
⟹ ∃ *stp. abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog stp* =
(*length aprog*, *lm* @ [*rs*] @ $0^{a\text{-}md\ -\ rs\text{-}pos\ -\ 1}$ @ *suf-lm*)
and *h*: *rec-ci* (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*)
∀ *k*<*length gs. rec-calc-rel* (*gs* ! *k*) *lm* (*ys* ! *k*)
length ys = *length gs*
length lm = *n*
rec-ci f = (*a*, *aa*, *ba*)
Max (*set* (*Suc n* # *ba* # *map* (λ(*aprog*, *p*, *n*). *n*)
(*map rec-ci* (*f* # *gs*)))) = *pstr*
shows
∃ *stp. abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog stp* =
(($\sum$(*ap*, *pos*, *n*)←*map rec-ci gs. length ap*) + *3* ∗ *length gs*,
lm @ $0^{pstr\ -\ n}$ @ *ys* @ $0^{a\text{-}md\ -pstr\ -\ length\ ys}$ @ *suf-lm*)
proof −
from *h* **have** *k1*:
∃ *ap bp. aprog* = *ap* [+] *bp* ∧ *ap* =
cn-merge-gs (*map rec-ci gs*) *pstr*
by(*erule-tac calc-gs-prog-ex*, *auto*)
from *h* **have** *j1*: *rs-pos* = *n*
by(*simp add*: *ci-cn-para-eq*)
from *h* **have** *j2*: *a-md* ≥ *pstr*
by(*drule-tac a* = *a* **and** *aa* = *aa* **and** *ba* = *ba* **in**
ci-cn-md-def, *simp*, *simp*)
from *h* **have** *j3*: *pstr* > *n*
by(*auto*)
from *j1* **and** *j2* **and** *j3* **and** *h* **have** *k2*:
∃ *stp. abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*)
(*cn-merge-gs* (*map rec-ci gs*) *pstr*) *stp*
= (($\sum$(*ap*, *pos*, *n*)←*map rec-ci gs. length ap*) + *3* ∗ *length gs*,
lm @ $0^{pstr\ -\ n}$ @ *ys* @ $0^{a\text{-}md\ -\ pstr\ -\ length\ ys}$ @ *suf-lm*)
apply(*simp*)
apply(*rule-tac cn-merge-gs-ex*, *rule-tac ind*, *simp*, *simp*, *auto*)
apply(*drule-tac a* = *a* **and** *aa* = *aa* **and** *ba* = *ba* **in**
ci-cn-md-def, *simp*, *simp*)
apply(*rule min-max.le-supI2*, *auto*)
done
from *k1* **show** *?thesis*
proof(*erule-tac exE*, *erule-tac exE*, *simp*)
fix *ap bp*
from *k2* **show**
∃ *stp. abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*)
(*cn-merge-gs* (*map rec-ci gs*) *pstr* [+] *bp*) *stp* =
(*listsum* (*map* ((λ(*ap*, *pos*, *n*). *length ap*) ∘ *rec-ci*) *gs*) +
3 ∗ *length gs*,
lm @ $0^{pstr\ -\ n}$ @ *ys* @ $0^{a\text{-}md\ -\ (pstr\ +\ length\ ys)}$ @ *suf-lm*)

apply(*insert abc-append-exc1*[*of*
lm @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*
(*cn-merge-gs* (*map rec-ci gs*) *pstr*)
length (*cn-merge-gs* (*map rec-ci gs*) *pstr*)
lm @ $0^{pstr\ -\ n}$ @ *ys* @ $0^{a\text{-}md\ -\ pstr\ -\ length\ ys}$ @ *suf-lm 0*
[] *bp*], *simp add*: *cn-merge-gs-len*)
done
qed
qed

lemma *reset-new-paras'*:
⟦*length lm* = *n*;
pstr > *0*;
a-md ≥ *pstr* + *length ys* + *n*;
pstr > *length ys*⟧ ⟹
∃ *stp*. *abc-steps-l* (*0*, 0^{pstr} @ *ys* @ *0* # *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$
@
suf-lm) (*mv-boxes pstr 0* (*length ys*)) *stp* =
(*3* ∗ *length ys*, *ys* @ 0^{pstr} @ *0* # *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @
suf-lm)
thm *mv-boxes-ex2*
apply(*insert mv-boxes-ex2*[*of length ys pstr 0* []
$0^{pstr\ -\ length\ ys}$ *ys*
0 # *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*],
simp add: *exponent-add-iff*)
done

lemma [*simp*]:
⟦*rec-ci* (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*);
rec-calc-rel f ys rs; *rec-ci f* = (*a*, *aa*, *ba*);
pstr = *Max* (*set* (*Suc n* # *ba* # *map* (λ(*aprog*, *p*, *n*). *n*)
(*map rec-ci* (*f* # *gs*))))⟧
⟹ *length ys* < *pstr*
apply(*subgoal-tac length ys* = *aa*, *simp*)
apply(*subgoal-tac aa* < *ba* ∧ *ba* ≤ *pstr*,
rule basic-trans-rules(*22*), *auto*)
apply(*rule min-max.le-supI2*)
apply(*auto*)
apply(*erule-tac para-pattern*, *simp*)
done

lemma *reset-new-paras-prog-ex*:
⟦*rec-ci* (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*);
rec-ci f = (*a*, *aa*, *ba*);
Max (*set* (*Suc n* # *ba* # *map* (λ(*aprog*, *p*, *n*). *n*)
(*map rec-ci* (*f* # *gs*)))) = *pstr*⟧
⟹ ∃ *ap bp cp*. *aprog* = *ap* [+] *bp* [+] *cp* ∧
length ap = (∑(*ap*, *pos*, *n*)←*map rec-ci gs*. *length ap*) +

*3 *length gs + 3 * n ∧ bp = mv-boxes pstr 0 (length gs)*
apply(*simp add*: *rec-ci.simps*)
apply(*rule-tac x = cn-merge-gs (map rec-ci gs) (max (Suc n)*
(Max (insert ba (((λ(aprog, p, n). n) ∘ rec-ci) ' set gs)))) [+]
mv-boxes 0 (Suc (max (Suc n) (Max (insert ba
(((λ(aprog, p, n). n) ∘ rec-ci) ' set gs))) + length gs)) n **in** *exI*,
simp add: *cn-merge-gs-len*)
apply(*rule-tac x = a [+]*
recursive.empty aa (max (Suc n) (Max (insert ba
(((λ(aprog, p, n). n) ∘ rec-ci) ' set gs)))) [+]
empty-boxes (length gs) [+] recursive.empty
(max (Suc n) (Max (insert ba (((λ(aprog, p, n). n) ∘ rec-ci) ' set gs)))) n
[+] mv-boxes (Suc (max (Suc n) (Max (insert ba
(((λ(aprog, p, n). n) ∘ rec-ci) ' set gs))) + length gs)) 0 n **in** *exI*,
auto simp: *abc-append-commute*)
done

lemma *reset-new-paras*:
⟦*rec-ci (Cn n f gs) = (aprog, rs-pos, a-md)*;
rs-pos = n;
∀ k<length gs. rec-calc-rel (gs ! k) lm (ys ! k);
length ys = length gs;
length lm = n;
length ys = aa;
rec-ci f = (a, aa, ba);
pstr = Max (set (Suc n # ba # map (λ(aprog, p, n). n)
(map rec-ci (f # gs))))⟧
⟹ ∃ *stp. abc-steps-l ((∑ (ap, pos, n)←map rec-ci gs. length ap) +*
*3 * length gs + 3 * n,*
0^{pstr} *@ ys @ 0 # lm @* $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ *@ suf-lm) aprog*
stp =
*((∑ (ap, pos, n)←map rec-ci gs. length ap) + 6 * length gs + 3 * n,*
ys @ 0^{pstr} *@ 0 # lm @* $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ *@ suf-lm)*
proof −
assume *h*:
rec-ci (Cn n f gs) = (aprog, rs-pos, a-md)
rs-pos = n
length ys = aa
∀ k<length gs. rec-calc-rel (gs ! k) lm (ys ! k)
length ys = length gs length lm = n
rec-ci f = (a, aa, ba)
and *g*: *pstr = Max (set (Suc n # ba # map (λ(aprog, p, n). n)*
(map rec-ci (f # gs))))
thm *rec-ci.simps*
from *h* **and** *g* **have** *k1*:
∃ ap bp cp. aprog = ap [+] bp [+] cp ∧ length ap =
(∑ (ap, pos, n)←map rec-ci gs. length ap) +
*3 *length gs + 3 * n ∧ bp = mv-boxes pstr 0 (length ys)*

by(*drule-tac reset-new-paras-prog-ex*, *auto*)
from *h* **have** *k2*:
$\exists$ *stp*. *abc-steps-l* (*0*, 0^{pstr} @ *ys* @ *0* # *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @
suf-lm) (*mv-boxes pstr 0* (*length ys*)) *stp* =
(*3* $*$ (*length ys*),
ys @ 0^{pstr} @ *0* # *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
apply(*rule-tac reset-new-paras'*, *simp*)
apply(*simp add*: *g*)
apply(*drule-tac a* = *a* **and** *aa* = *aa* **and** *ba* = *ba* **in** *ci-cn-md-def*,
simp, *simp add*: *g*, *simp*)
apply(*subgoal-tac length gs* = *aa* $\wedge$ *aa* < *ba* $\wedge$ *ba* $\leq$ *pstr*, *arith*,
simp add: *para-pattern*)
apply(*insert g*, *auto intro*: *min-max.le-supI2*)
done
from *k1* **show**
$\exists$ *stp*. *abc-steps-l* (($\sum$ (*ap*, *pos*, *n*)$\leftarrow$*map rec-ci gs*. *length ap*) + *3*
$*$ *length gs* + *3* $*$ *n*, 0^{pstr} @ *ys* @ *0* # *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @
suf-lm) *aprog stp* =
(($\sum$ (*ap*, *pos*, *n*)$\leftarrow$*map rec-ci gs*. *length ap*) + *6* $*$ *length gs* +
3 $*$ *n*, *ys* @ 0^{pstr} @ *0* # *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
proof(*erule-tac exE*, *erule-tac exE*, *erule-tac exE*, *erule-tac exE*)
fix *ap bp apa cp*
assume *aprog* = *ap* [+] *bp* [+] *cp* $\wedge$ *length ap* =
($\sum$ (*ap*, *pos*, *n*)$\leftarrow$*map rec-ci gs*. *length ap*) + *3* $*$ *length gs* +
3 $*$ *n* $\wedge$ *bp* = *mv-boxes pstr 0* (*length ys*)
from *this* **and** *k2* **show** *?thesis*
apply(*simp*)
apply(*drule-tac as* =
($\sum$ (*ap*, *pos*, *n*)$\leftarrow$*map rec-ci gs*. *length ap*) + *3* $*$ *length gs* +
3 $*$ *n* **and** *ap* = *ap* **and** *cp* = *cp* **in** *abc-append-exc1*, *auto*)
apply(*rule-tac x* = *stp* **in** *exI*, *simp add*: *h*)
using *h*
apply(*simp*)
done
qed
qed

thm *rec-ci.simps*

lemma *calc-f-prog-ex*:
⟦*rec-ci* (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*);
rec-ci f = (*a*, *aa*, *ba*);
Max (*set* (*Suc n* # *ba* # *map* (λ(*aprog*, *p*, *n*). *n*)
(*map rec-ci* (*f* # *gs*)))) = *pstr*⟧
$\Longrightarrow$ $\exists$ *ap bp cp*. *aprog* = *ap* [+] *bp* [+] *cp* $\wedge$
length ap = ($\sum$ (*ap*, *pos*, *n*)$\leftarrow$*map rec-ci gs*. *length ap*) +
6 $*$*length gs* + *3* $*$ *n* $\wedge$ *bp* = *a*

apply(*simp add*: *rec-ci.simps*)
apply(*rule-tac x* = *cn-merge-gs* (*map rec-ci gs*) (*max* (*Suc n*) (*Max* (*insert ba*
  (((λ(*aprog*, *p*, *n*). *n*) $\circ$ *rec-ci*) ' *set gs*)))) [+]
  mv-boxes 0 (*Suc* (*max* (*Suc n*) (*Max* (*insert ba*
    (((λ(*aprog*, *p*, *n*). *n*) $\circ$ *rec-ci*) ' *set gs*))) + *length gs*)) *n* [+]
  mv-boxes (*max* (*Suc n*) (*Max* (*insert ba*
  (((λ(*aprog*, *p*, *n*). *n*) $\circ$ *rec-ci*) ' *set gs*)))) *0* (*length gs*) **in** *exI*,
  simp add: *cn-merge-gs-len*)
apply(*rule-tac x* = *recursive.empty aa* (*max* (*Suc n*) (*Max* (*insert ba*
  (((λ(*aprog*, *p*, *n*). *n*) $\circ$ *rec-ci*) ' *set gs*)))) [+]
  empty-boxes (*length gs*) [+] *recursive.empty* (*max* (*Suc n*) (
  Max (*insert ba* (((λ(*aprog*, *p*, *n*). *n*) $\circ$ *rec-ci*) ' *set gs*)))) *n* [+]
  mv-boxes (*Suc* (*max* (*Suc n*) (*Max* (*insert ba*
  (((λ(*aprog*, *p*, *n*). *n*) $\circ$ *rec-ci*) ' *set gs*))) + *length gs*)) *0 n* **in** *exI*,
 auto simp: *abc-append-commute*)
done

lemma *calc-cn-f*:
 assumes *ind*:
 $\bigwedge$*x aprog a-md rs-pos rs suf-lm lm*.
 ⟦*x* ∈ *set* (*f* # *gs*);
 rec-ci x = (*aprog*, *rs-pos*, *a-md*);
 rec-calc-rel x lm rs⟧
 ⟹ ∃ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog stp* =
 (*length aprog*, *lm* @ [*rs*] @ $0^{a\text{-}md\ -\ rs\text{-}pos\ -\ 1}$ @ *suf-lm*)
 and *h*: *rec-ci* (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*)
 rec-calc-rel (*Cn n f gs*) *lm rs*
 length ys = *length gs*
 rec-calc-rel f ys rs
 length lm = *n*
 rec-ci f = (*a*, *aa*, *ba*)
 and *p*: *pstr* = *Max* (*set* (*Suc n* # *ba* # *map* (λ(*aprog*, *p*, *n*). *n*)
          (*map rec-ci* (*f* # *gs*))))
 shows ∃ *stp*. *abc-steps-l*
 (($\sum$(*ap*, *pos*, *n*)←*map rec-ci gs*. *length ap*) + *6* ∗ *length gs* + *3* ∗ *n*,
 ys @ 0^{pstr} @ *0* # *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*) *aprog stp*
=
 (($\sum$(*ap*, *pos*, *n*)←*map rec-ci gs*. *length ap*) + *6* ∗ *length gs* +
    3 ∗ *n* + *length a*,
 ys @ *rs* # 0^{pstr} @ *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
proof −
 from *h* **have** *k1*:
  ∃ *ap bp cp*. *aprog* = *ap* [+] *bp* [+] *cp* ∧
  length ap = ($\sum$(*ap*, *pos*, *n*)←*map rec-ci gs*. *length ap*) +
  6 ∗*length gs* + *3* ∗ *n* ∧ *bp* = *a*
  by(*drule-tac calc-f-prog-ex*, *auto*)
 from *h* **and** *k1* **show** *?thesis*
 proof (*erule-tac exE*, *erule-tac exE*, *erule-tac exE*, *erule-tac exE*)
  fix *ap bp apa cp*

assume
aprog = *ap* [+] *bp* [+] *cp* ∧
length ap = ($\sum$(*ap*, *pos*, *n*)←*map rec-ci gs*. *length ap*) +
6 ∗ *length gs* + *3* ∗ *n* ∧ *bp* = *a*
from *h* **and** *this* **show** *?thesis*
apply(*simp*, *rule-tac abc-append-exc1*, *simp-all*)
apply(*insert ind*[*of f a aa ba ys rs*
$0^{pstr\ -\ ba\ +\ length\ gs}$ @ *0* # *lm* @
$0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ gs\ +\ n)}$ @ *suf-lm*], *simp*)
apply(*subgoal-tac ba* > *aa* ∧ *aa* = *length gs*∧ *pstr* ≥ *ba*, *simp*)
apply(*simp add*: *exponent-add-iff*)
apply(*case-tac pstr*, *simp add*: *p*)
apply(*simp only*: *exp-suc*, *simp*)
apply(*rule conjI*, *rule ci-ad-ge-paras*, *simp*, *rule conjI*)
apply(*subgoal-tac length ys* = *aa*, *simp*,
rule para-pattern, *simp*, *simp*)
apply(*insert p*, *simp*)
apply(*auto intro*: *min-max.le-supI2*)
done
qed
qed

lemma [*simp*]:
pstr > *length ys*
⟹ (*ys* @ *rs* # 0^{pstr} @ *lm* @
$0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*) ! *pstr* = (*0*::*nat*)
apply(*simp add*: *nth-append*)
done

lemma [*simp*]: *pstr* > *length ys* ⟹
(*ys* @ *rs* # 0^{pstr} @ *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
[*pstr* := *rs*, *length ys* := *0*] =
ys @ $0^{pstr\ -\ length\ ys}$ @ (*rs*::*nat*) # $0^{length\ ys}$ @ *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$
@ *suf-lm*
apply(*auto simp*: *list-update-append*)
apply(*case-tac pstr* − *length ys*,*simp-all*)
using *list-update-length*[*of*
$0^{pstr\ -\ Suc\ (length\ ys)}$ *0* $0^{length\ ys}$ @ *lm* @
$0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm rs*]
apply(*simp only*: *exponent-cons-iff exponent-add-iff*, *simp*)
apply(*subgoal-tac pstr* − *Suc* (*length ys*) = *nat*, *simp*, *simp*)
done

lemma *save-rs'*:
⟦*pstr* > *length ys*⟧
⟹ ∃ *stp*. *abc-steps-l* (*0*, *ys* @ *rs* # 0^{pstr} @ *lm* @
$0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)

(*recursive.empty* (*length ys*) *pstr*) *stp* =
(*3*, *ys* @ $0^{pstr - (length\ ys)}$ @ *rs* #
$0^{length\ ys}$ @ *lm* @ $0^{a\text{-}md - Suc\ (pstr + length\ ys + n)}$ @ *suf-lm*)
apply(*insert empty-ex*[*of length ys pstr*
ys @ *rs* # 0^{pstr} @ *lm* @ $0^{a\text{-}md - Suc(pstr + length\ ys + n)}$ @ *suf-lm*],
simp)
done

lemma *save-rs-prog-ex*:
⟦*rec-ci* (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*);
rec-ci f = (*a*, *aa*, *ba*);
Max (*set* (*Suc n* # *ba* # *map* (λ(*aprog*, *p*, *n*). *n*)
(*map rec-ci* (*f* # *gs*)))) = *pstr*⟧
⟹ ∃ *ap bp cp*. *aprog* = *ap* [+] *bp* [+] *cp* ∧
length ap = (∑ (*ap*, *pos*, *n*)←*map rec-ci gs*. *length ap*) +
6 ∗*length gs* + *3* ∗ *n* + *length a*
∧ *bp* = *empty aa pstr*
apply(*simp add*: *rec-ci.simps*)
apply(*rule-tac x* =
cn-merge-gs (*map rec-ci gs*) (*max* (*Suc n*) (*Max* (*insert ba*
(((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ‘ *set gs*))))
[+] *mv-boxes 0* (*Suc* (*max* (*Suc n*) (*Max* (*insert ba*
(((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ‘ *set gs*))) + *length gs*)) *n* [+]
mv-boxes (*max* (*Suc n*) (*Max* (*insert ba* (((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ‘ *set gs*))))
0 (*length gs*) [+] *a*
in *exI*, *simp add*: *cn-merge-gs-len*)
apply(*rule-tac x* =
empty-boxes (*length gs*) [+]
recursive.empty (*max* (*Suc n*) (*Max* (*insert ba*
(((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ‘ *set gs*)))) *n* [+]
mv-boxes (*Suc* (*max* (*Suc n*) (*Max* (*insert ba* (((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ‘ *set gs*)))
+ *length gs*)) *0 n* **in** *exI*,
auto simp: *abc-append-commute*)
done

lemma *save-rs*:
assumes *h*:
rec-ci (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*)
rec-calc-rel (*Cn n f gs*) *lm rs*
∀ *k*<*length gs*. *rec-calc-rel* (*gs* ! *k*) *lm* (*ys* ! *k*)
length ys = *length gs*
rec-calc-rel f ys rs
rec-ci f = (*a*, *aa*, *ba*)
length lm = *n*
and *pdef*: *pstr* = *Max* (*set* (*Suc n* # *ba* # *map* (λ(*aprog*, *p*, *n*). *n*)
(*map rec-ci* (*f* # *gs*))))

shows ∃ *stp. abc-steps-l*
((∑ (*ap*, *pos*, *n*)←*map rec-ci gs. length ap*) + *6* ∗ *length gs*
+ *3* ∗ *n* + *length a*, *ys* @ *rs* # 0^{pstr} @ *lm* @
$0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*) *aprog stp* =
((∑ (*ap*, *pos*, *n*)←*map rec-ci gs. length ap*) + *6* ∗ *length gs*
+ *3* ∗ *n* + *length a* + *3*,
ys @ $0^{pstr\ -\ length\ ys}$ @ *rs* # $0^{length\ ys}$ @ *lm* @
$0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
proof −
thm *rec-ci.simps*
from *h* **and** *pdef* **have** *k1*:
∃ *ap bp cp*. *aprog* = *ap* [+] *bp* [+] *cp* ∧
length ap = (∑ (*ap*, *pos*, *n*)←*map rec-ci gs. length ap*) +
6 ∗*length gs* + *3* ∗ *n* + *length a* ∧ *bp* = *empty* (*length ys*) *pstr*
apply(*subgoal-tac length ys* = *aa*)
apply(*drule-tac a* = *a* **and** *aa* = *aa* **and** *ba* = *ba* **in** *save-rs-prog-ex*,
simp, *simp*, *simp*)
by(*rule-tac para-pattern*, *simp*, *simp*)
from *k1* **show**
∃ *stp. abc-steps-l*
((∑ (*ap*, *pos*, *n*)←*map rec-ci gs. length ap*) + *6* ∗ *length gs* + *3* ∗ *n*
+ *length a*, *ys* @ *rs* # 0^{pstr} @ *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$
@ *suf-lm*) *aprog stp* =
((∑ (*ap*, *pos*, *n*)←*map rec-ci gs. length ap*) + *6* ∗ *length gs* + *3* ∗ *n*
+ *length a* + *3*, *ys* @ $0^{pstr\ -\ length\ ys}$ @ *rs* #
$0^{length\ ys}$ @ *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
proof (*erule-tac exE*, *erule-tac exE*, *erule-tac exE*, *erule-tac exE*)
fix *ap bp apa cp*
assume *aprog* = *ap* [+] *bp* [+] *cp* ∧ *length ap* =
(∑ (*ap*, *pos*, *n*)←*map rec-ci gs. length ap*) + *6* ∗ *length gs* +
3 ∗ *n* + *length a* ∧ *bp* = *recursive.empty* (*length ys*) *pstr*
thus *?thesis*
apply(*simp*, *rule-tac abc-append-exc1*, *simp-all*)
apply(*rule-tac save-rs′*, *insert h*)
apply(*subgoal-tac length gs* = *aa* ∧ *pstr* ≥ *ba* ∧ *ba* > *aa*,
arith)
apply(*simp add*: *para-pattern*, *insert pdef*, *auto*)
apply(*rule-tac min-max.le-supI2*, *simp*)
done
qed
qed

lemma [*simp*]: *length* (*empty-boxes n*) = *2*∗*n*
apply(*induct n*, *simp*, *simp*)
done

lemma *empty-step-ex*: *length lm* = *n* ⟹
∃ *stp. abc-steps-l* (*0*, *lm* @ *Suc x* # *suf-lm*) [*Dec n 2*, *Goto 0*] *stp*
= (*0*, *lm* @ *x* # *suf-lm*)

apply(*rule-tac x = Suc (Suc 0)* **in** *exI*,
simp add: *abc-steps-l.simps abc-step-l.simps abc-fetch.simps*
abc-lm-v.simps abc-lm-s.simps nth-append list-update-append)
done

lemma *empty-box-ex*:
⟦*length lm = n*⟧ ⟹
∃ *stp. abc-steps-l (0, lm @ x # suf-lm) [Dec n 2, Goto 0] stp =*
(Suc (Suc 0), lm @ 0 # suf-lm)
apply(*induct x*)
apply(*rule-tac x = Suc 0* **in** *exI*,
simp add: *abc-steps-l.simps abc-fetch.simps abc-step-l.simps*
abc-lm-v.simps nth-append abc-lm-s.simps, *simp*)
apply(*drule-tac x = x* **and** *suf-lm = suf-lm* **in** *empty-step-ex*,
erule-tac exE, *erule-tac exE*)
apply(*rule-tac x = stpa + stp* **in** *exI*, *simp add*: *abc-steps-add*)
done

lemma [*simp*]: *drop n lm = a # list* ⟹ *list = drop (Suc n) lm*
apply(*induct n arbitrary*: *lm a list*, *simp*)
apply(*case-tac lm*, *simp*, *simp*)
done

lemma *empty-boxes-ex*: ⟦*length lm* ≥ *n*⟧
⟹ ∃ *stp. abc-steps-l (0, lm) (empty-boxes n) stp =*
(2∗n, 0^n @ drop n lm)
apply(*induct n*, *simp*, *simp*)
apply(*rule-tac abc-append-exc2*, *auto*)
apply(*case-tac drop n lm*, *simp*, *simp*)
proof −
fix *n stp a list*
assume *h*: *Suc n ≤ length lm drop n lm = a # list*
thus ∃ *bstp. abc-steps-l (0, 0^n @ a # list) [Dec n 2, Goto 0] bstp =*
(Suc (Suc 0), 0 # 0^n @ drop (Suc n) lm)
apply(*insert empty-box-ex*[*of 0^n n a list*], *simp*, *erule-tac exE*)
apply(*rule-tac x = stp* **in** *exI*, *simp*, *simp only*: *exponent-cons-iff*)
apply(*simp add*: *exponent-def rep-ind del*: *replicate.simps*)
done
qed

lemma *empty-paras-prog-ex*:
⟦*rec-ci (Cn n f gs) = (aprog, rs-pos, a-md)*;
rec-ci f = (a, aa, ba);
Max (set (Suc n # ba # map (λ(aprog, p, n). n)
(map rec-ci (f # gs)))) = pstr⟧
⟹ ∃ *ap bp cp. aprog = ap [+] bp [+] cp* ∧
length ap = (∑(ap, pos, n)←map rec-ci gs. length ap) +
6 ∗length gs + 3 ∗ n + length a + 3 ∧ *bp = empty-boxes (length gs)*

apply(*simp add*: *rec-ci.simps*)
apply(*rule-tac x* = *cn-merge-gs* (*map rec-ci gs*) (*max* (*Suc n*)
(*Max* (*insert ba* (((λ(*aprog*, *p*, *n*). *n*) $\circ$ *rec-ci*) ' *set gs*)))) [+]
mv-boxes 0 (*Suc* (*max* (*Suc n*) (*Max*
(*insert ba* (((λ(*aprog*, *p*, *n*). *n*) $\circ$ *rec-ci*) ' *set gs*))) + *length gs*)) *n*
[+] *mv-boxes* (*max* (*Suc n*) (*Max* (*insert ba*
(((λ(*aprog*, *p*, *n*). *n*) $\circ$ *rec-ci*) ' *set gs*)))) *0* (*length gs*) [+]
a [+] *recursive.empty aa* (*max* (*Suc n*)
(*Max* (*insert ba* (((λ(*aprog*, *p*, *n*). *n*) $\circ$ *rec-ci*) ' *set gs*))))
in *exI*, *simp add*: *cn-merge-gs-len*)
apply(*rule-tac x* = *recursive.empty* (*max* (*Suc n*) (*Max* (*insert ba*
(((λ(*aprog*, *p*, *n*). *n*) $\circ$ *rec-ci*) ' *set gs*)))) *n* [+]
mv-boxes (*Suc* (*max* (*Suc n*) (*Max* (*insert ba*
(((λ(*aprog*, *p*, *n*). *n*) $\circ$ *rec-ci*) ' *set gs*))) + *length gs*)) *0 n* **in** *exI*,
auto simp: *abc-append-commute*)
done

lemma *empty-paras*:
assumes *h*:
rec-ci (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*)
rec-calc-rel (*Cn n f gs*) *lm rs*
$\forall$ *k*<*length gs*. *rec-calc-rel* (*gs* ! *k*) *lm* (*ys* ! *k*)
length ys = *length gs*
rec-calc-rel f ys rs
rec-ci f = (*a*, *aa*, *ba*)
and *pdef*: *pstr* = *Max* (*set* (*Suc n* # *ba* # *map* (λ(*aprog*, *p*, *n*). *n*)
(*map rec-ci* (*f* # *gs*))))
and *starts*: *ss* = ($\sum$(*ap*, *pos*, *n*)$\leftarrow$*map rec-ci gs*. *length ap*) +
6 $*$ *length gs* + *3* $*$ *n* + *length a* + *3*
shows $\exists$ *stp*. *abc-steps-l*
(*ss*, *ys* @ $0^{pstr\ -\ length\ ys}$ @ *rs* # $0^{length\ ys}$
@ *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*) *aprog stp* =
(*ss* + *2* $*$ *length gs*, 0^{pstr} @ *rs* # $0^{length\ ys}$ @ *lm* @
$0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
proof −
from *h* **and** *pdef* **and** *starts* **have** *k1*:
$\exists$ *ap bp cp*. *aprog* = *ap* [+] *bp* [+] *cp* $\wedge$
length ap = ($\sum$(*ap*, *pos*, *n*)$\leftarrow$*map rec-ci gs*. *length ap*) +
6 $*$*length gs* + *3* $*$ *n* + *length a* + *3*
$\wedge$ *bp* = *empty-boxes* (*length ys*)
by(*drule-tac empty-paras-prog-ex*, *auto*)
from *h* **have** *j1*: *aa* < *ba*
by(*simp add*: *ci-ad-ge-paras*)
from *h* **have** *j2*: *length gs* = *aa*
by(*drule-tac f* = *f* **in** *para-pattern*, *simp*, *simp*)
from *h* **and** *pdef* **have** *j3*: *ba* $\leq$ *pstr*
apply *simp*
apply(*rule-tac min-max.le-supI2*, *simp*)
done

from *k1* **show** *?thesis*
proof (*erule-tac exE*, *erule-tac exE*, *erule-tac exE*, *erule-tac exE*)
fix *ap bp apa cp*
assume *aprog* = *ap* [+] *bp* [+] *cp* ∧
length ap = ($\sum$(*ap*, *pos*, *n*)←*map rec-ci gs*. *length ap*) +
6 ∗ *length gs* + *3* ∗ *n* + *length a* + *3* ∧
bp = *empty-boxes* (*length ys*)
thus *?thesis*
apply(*simp*, *rule-tac abc-append-exc1*, *simp-all add*: *starts h*)
apply(*insert empty-boxes-ex*[*of*
length gs ys @ $0^{pstr\ -\ (length\ gs)}$ @ *rs* #
$0^{length\ gs}$ @ *lm* @ $0^{a\text{-}md\ -\ Suc\ (pstr\ +\ length\ gs\ +\ n)}$ @ *suf-lm*],
simp add: *h*)
apply(*erule-tac exE*, *rule-tac x* = *stp* **in** *exI*,
simp add: *exponent-def replicate.simps*[*THEN sym*]
replicate-add[*THEN sym*] *del*: *replicate.simps*)
apply(*subgoal-tac pstr* >(*length gs*), *simp*)
apply(*subgoal-tac ba* > *aa* ∧ *length gs* = *aa* ∧ *pstr* ≥ *ba*, *simp*)
apply(*simp add*: *j1 j2 j3*)
done
qed
qed

lemma *restore-rs-prog-ex*:
⟦*rec-ci* (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*);
rec-ci f = (*a*, *aa*, *ba*);
Max (*set* (*Suc n* # *ba* # *map* (λ(*aprog*, *p*, *n*). *n*)
(*map rec-ci* (*f* # *gs*)))) = *pstr*;
ss = ($\sum$(*ap*, *pos*, *n*)←*map rec-ci gs*. *length ap*) +
8 ∗ *length gs* + *3* ∗ *n* + *length a* + *3*⟧
⟹ ∃ *ap bp cp*. *aprog* = *ap* [+] *bp* [+] *cp* ∧ *length ap* = *ss* ∧
bp = *empty pstr n*
apply(*simp add*: *rec-ci.simps*)
apply(*rule-tac x* = *cn-merge-gs* (*map rec-ci gs*) (*max* (*Suc n*)
(*Max* (*insert ba* (((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ' *set gs*)))) [+]
mv-boxes 0 (*Suc* (*max* (*Suc n*) (*Max* (*insert ba* (((λ(*aprog*, *p*, *n*). *n*)
∘ *rec-ci*) ' *set gs*))) + *length gs*)) *n* [+]
mv-boxes (*max* (*Suc n*) (*Max* (*insert ba*
(((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ' *set gs*)))) *0* (*length gs*) [+]
a [+] *recursive.empty aa* (*max* (*Suc n*)
(*Max* (*insert ba* (((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ' *set gs*)))) [+]
empty-boxes (*length gs*) **in** *exI*, *simp add*: *cn-merge-gs-len*)
apply(*rule-tac x* = *mv-boxes* (*Suc* (*max* (*Suc n*)
(*Max* (*insert ba* (((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ' *set gs*)))
+ *length gs*)) *0 n*
in *exI*, *auto simp*: *abc-append-commute*)
done

```
lemma exp-add: a^(b+c) = a^b @ a^c
apply(simp add: exponent-def replicate-add)
done

lemma [simp]: n < pstr ⟹ (0^pstr)[n := rs] @ [0::nat] = 0^n @ rs # 0^(pstr − n)
using list-update-length[of 0^n 0::nat 0^(pstr − Suc n) rs]
apply(simp add: exp-ind-def[THEN sym] exp-add[THEN sym] exp-suc[THEN sym])
done

lemma restore-rs:
  assumes h: rec-ci (Cn n f gs) = (aprog, rs-pos, a-md)
  rec-calc-rel (Cn n f gs) lm rs
  ∀ k<length gs. rec-calc-rel (gs ! k) lm (ys ! k)
  length ys = length gs
  rec-calc-rel f ys rs
  rec-ci f = (a, aa, ba)
  and pdef: pstr = Max (set (Suc n # ba # map (λ(aprog, p, n). n)
                                  (map rec-ci (f # gs))))
  and starts: ss = (∑ (ap, pos, n)←map rec-ci gs. length ap) +
                              8 ∗ length gs + 3 ∗ n + length a + 3
  shows ∃ stp. abc-steps-l
          (ss, 0^pstr @ rs # 0^(length ys) @ lm @
                      0^(a-md − Suc (pstr + length ys + n)) @ suf-lm) aprog stp =
  (ss + 3, 0^n @ rs # 0^(pstr − n) @ 0^(length ys) @ lm @
                                0^(a-md − Suc (pstr + length ys + n)) @ suf-lm)
proof −
 from h and pdef and starts have k1:
   ∃ ap bp cp. aprog = ap [+] bp [+] cp ∧ length ap = ss ∧
                                         bp = empty pstr n
   by(drule-tac restore-rs-prog-ex, auto)
 from k1 show ?thesis
 proof (erule-tac exE, erule-tac exE, erule-tac exE, erule-tac exE)
   fix ap bp apa cp
   assume aprog = ap [+] bp [+] cp ∧ length ap = ss ∧
                              bp = recursive.empty pstr n
   thus ?thesis
     apply(simp, rule-tac abc-append-exc1, simp-all add: starts h)
     apply(insert empty-ex[of pstr n 0^pstr @ rs # 0^(length gs) @
                   lm @ 0^(a-md − Suc (pstr + length gs + n)) @ suf-lm], simp)
     apply(subgoal-tac pstr > n, simp)
     apply(erule-tac exE, rule-tac x = stp in exI,
                    simp add: nth-append list-update-append)
     apply(simp add: pdef)
     done
  qed
qed
```

```
lemma [simp]:xs ≠ [] ⟹ length xs ≥ Suc 0
by(case-tac xs, auto)

lemma  [simp]: n < max (Suc n) (max ba (Max (((λ(aprog, p, n). n) o
                                                         rec-ci) ‘ set gs)))
by(simp)

lemma restore-paras-prog-ex:
  ⟦rec-ci (Cn n f gs) = (aprog, rs-pos, a-md);
  rec-ci f = (a, aa, ba);
  Max (set (Suc n # ba # map (λ(aprog, p, n). n)
                        (map rec-ci (f # gs)))) = pstr;
  ss = (∑(ap, pos, n)←map rec-ci gs. length ap) +
                        8 ∗ length gs + 3 ∗ n + length a + 6⟧
  ⟹ ∃ ap bp cp. aprog = ap [+] bp [+] cp ∧ length ap = ss ∧
                    bp = mv-boxes (pstr + Suc (length gs)) (0::nat) n
apply(simp add: rec-ci.simps)
apply(rule-tac x = cn-merge-gs (map rec-ci gs) (max (Suc n)
      (Max (insert ba (((λ(aprog, p, n). n) ∘ rec-ci) ‘ set gs))))
      [+] mv-boxes 0 (Suc (max (Suc n)
       (Max (insert ba (((λ(aprog, p, n). n) ∘ rec-ci) ‘ set gs)))
     + length gs)) n [+] mv-boxes (max (Suc n)
    (Max (insert ba (((λ(aprog, p, n). n) ∘ rec-ci) ‘ set gs)))) 0 (length gs) [+]
     a [+] recursive.empty aa (max (Suc n)
      (Max (insert ba (((λ(aprog, p, n). n) ∘ rec-ci) ‘ set gs)))) [+]
     empty-boxes (length gs) [+]
     recursive.empty (max (Suc n) (Max (insert ba
    (((λ(aprog, p, n). n) ∘ rec-ci) ‘ set gs)))) n in exI, simp add: cn-merge-gs-len)
apply(rule-tac x = [] in exI, auto simp: abc-append-commute)
done

lemma restore-paras:
  assumes h: rec-ci (Cn n f gs) = (aprog, rs-pos, a-md)
  rec-calc-rel (Cn n f gs) lm rs
  ∀k<length gs. rec-calc-rel (gs ! k) lm (ys ! k)
  length ys = length gs
  rec-calc-rel f ys rs
  rec-ci f = (a, aa, ba)
  and pdef:
  pstr = Max (set (Suc n # ba # map (λ(aprog, p, n). n)
                        (map rec-ci (f # gs))))
  and starts: ss = (∑(ap, pos, n)←map rec-ci gs. length ap) +
                            8 ∗ length gs + 3 ∗ n + length a + 6
  shows ∃stp. abc-steps-l (ss, 0^n @ rs # 0^(pstr − n+ length ys) @
                        lm @ 0^(a-md − Suc (pstr + length ys + n)) @ suf-lm)
  aprog stp = (ss + 3 ∗ n, lm @ rs # 0^(a-md − Suc n) @ suf-lm)
proof −
  thm rec-ci.simps
  from h and pdef and starts have k1:
```

$\exists$ *ap bp cp. aprog* = *ap* [+] *bp* [+] *cp* $\wedge$ *length ap* = *ss* $\wedge$
bp = *mv-boxes* (*pstr* + *Suc* (*length gs*)) (*0::nat*) *n*
by(*drule-tac restore-paras-prog-ex*, *auto*)
from *k1* **show** *?thesis*
proof (*erule-tac exE*, *erule-tac exE*, *erule-tac exE*, *erule-tac exE*)
fix *ap bp apa cp*
assume *aprog* = *ap* [+] *bp* [+] *cp* $\wedge$ *length ap* = *ss* $\wedge$
bp = *mv-boxes* (*pstr* + *Suc* (*length gs*)) *0 n*
thus *?thesis*
apply(*simp*, *rule-tac abc-append-exc1*, *simp-all add*: *starts h*)
apply(*insert mv-boxes-ex2*[*of n pstr* + *Suc* (*length gs*) *0* []
rs # $0^{pstr - n + length\ gs}$ *lm*
$0^{a\text{-}md - Suc\ (pstr + length\ gs + n)}$ @ *suf-lm*], *simp*)
apply(*subgoal-tac pstr* > *n* $\wedge$
a-md > *pstr* + *length gs* + *n* $\wedge$ *length lm* = *n* , *simp add*: *exponent-add-iff*
h)
using *h pdef*
apply(*simp*)
apply(*frule-tac a* = *a* **and**
aa = *aa* **and** *ba* = *ba* **in** *ci-cn-md-def*, *simp*, *simp*)
apply(*subgoal-tac length lm* = *rs-pos*,
simp add: *ci-cn-para-eq*, *erule-tac para-pattern*, *simp*)
done
qed
qed

lemma *ci-cn-length*:
⟦*rec-ci* (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*);
rec-calc-rel (*Cn n f gs*) *lm rs*;
rec-ci f = (*a*, *aa*, *ba*)⟧
$\Longrightarrow$ *length aprog* = ($\sum$ (*ap*, *pos*, *n*)←*map rec-ci gs*. *length ap*) +
8 $*$ *length gs* + *6* $*$ *n* + *length a* + *6*
apply(*simp add*: *rec-ci.simps*, *auto simp*: *cn-merge-gs-len*)
done

lemma *cn-case*:
assumes *ind*:
$\bigwedge$*x aprog a-md rs-pos rs suf-lm lm*.
⟦*x* $\in$ *set* (*f* # *gs*);
rec-ci x = (*aprog*, *rs-pos*, *a-md*);
rec-calc-rel x lm rs⟧
$\Longrightarrow$ $\exists$ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md - rs\text{-}pos}$ @ *suf-lm*) *aprog stp* =
(*length aprog*, *lm* @ [*rs*] @ $0^{a\text{-}md - rs\text{-}pos - 1}$ @ *suf-lm*)
and *h*: *rec-ci* (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*)
rec-calc-rel (*Cn n f gs*) *lm rs*

shows $\exists$ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md - rs\text{-}pos}$ @ *suf-lm*) *aprog stp*
= (*length aprog*, *lm* @ [*rs*] @ $0^{a\text{-}md - rs\text{-}pos - 1}$ @ *suf-lm*)

apply(*insert h*, *case-tac rec-ci f*, *rule-tac calc-cn-reverse*, *simp*)
proof −
fix *a b c ys*
let *?pstr* = *Max* (*set* (*Suc n* # *c* # (*map* (λ(*aprog*, *p*, *n*). *n*)
(*map rec-ci* (*f* # *gs*)))))
let *?gs-len* = *listsum* (*map* (λ (*ap*, *pos*, *n*). *length ap*)
(*map rec-ci* (*gs*)))
assume *g*: *rec-ci* (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*)
rec-calc-rel (*Cn n f gs*) *lm rs*
∀ *k*<*length gs*. *rec-calc-rel* (*gs* ! *k*) *lm* (*ys* ! *k*)
length ys = *length gs*
rec-calc-rel f ys rs
n = *length lm*
rec-ci f = (*a*, *b*, *c*)
hence *k1*:
∃ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog stp* =
(*?gs-len* + *3* ∗ *length gs*, *lm* @ $0^{?pstr\ -\ n}$ @ *ys* @
$0^{a\text{-}md\ -\ ?pstr\ -\ length\ ys}$ @ *suf-lm*)
apply(*rule-tac a* = *a* **and** *aa* = *b* **and** *ba* = *c* **in** *cn-calc-gs*)
apply(*rule-tac ind*, *auto*)
done
thm *rec-ci.simps*
from *g* **have** *k2*:
∃ *stp*. *abc-steps-l* (*?gs-len* + *3* ∗ *length gs*, *lm* @
$0^{?pstr\ -\ n}$ @ *ys* @ $0^{a\text{-}md\ -\ ?pstr\ -\ length\ ys}$ @ *suf-lm*) *aprog stp* =
(*?gs-len* + *3* ∗ *length gs* + *3* ∗ *n*, $0^{?pstr}$ @ *ys* @ *0* # *lm* @
$0^{a\text{-}md\ -\ Suc\ (?pstr\ +\ length\ ys\ +\ n\ )}$ @ *suf-lm*)
thm *save-paras*
apply(*erule-tac ba* = *c* **in** *save-paras*, *auto intro*: *ci-cn-para-eq*)
done
from *g* **have** *k3*:
∃ *stp*. *abc-steps-l* (*?gs-len* + *3* ∗ *length gs* + *3* ∗ *n*,
$0^{?pstr}$ @ *ys* @ *0* # *lm* @ $0^{a\text{-}md\ -\ Suc\ (?pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*) *aprog*
stp =
(*?gs-len* + *6* ∗ *length gs* + *3* ∗ *n*,
ys @ $0^{?pstr}$ @ *0* # *lm* @ $0^{a\text{-}md\ -\ Suc\ (?pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
apply(*erule-tac ba* = *c* **in** *reset-new-paras*,
auto intro: *ci-cn-para-eq*)
using *para-pattern*[*of f a b c ys rs*]
apply(*simp*)
done
from *g* **have** *k4*:
∃ *stp*. *abc-steps-l* (*?gs-len* + *6* ∗ *length gs* + *3* ∗ *n*,
ys @ $0^{?pstr}$ @ *0* # *lm* @ $0^{a\text{-}md\ -\ Suc\ (?pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*) *aprog*
stp =
(*?gs-len* + *6* ∗ *length gs* + *3* ∗ *n* + *length a*,
ys @ *rs* # $0^{?pstr}$ @ *lm* @ $0^{a\text{-}md\ -\ Suc\ (?pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
apply(*rule-tac ba* = *c* **in** *calc-cn-f*, *rule-tac ind*, *auto*)

done
thm *rec-ci.simps*
from *g h* **have** *k5*:
$\exists$ *stp. abc-steps-l* (*?gs-len* + *6* ∗ *length gs* + *3* ∗ *n* + *length a*,
ys @ *rs* # *0* $^{?pstr}$ @ *lm* @ *0* $^{a\text{-}md\ -\ Suc\ (?pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
aprog stp =
(*?gs-len* + *6* ∗ *length gs* + *3* ∗ *n* + *length a* + *3*,
ys @ *0* $^{?pstr\ -\ length\ ys}$ @ *rs* # *0* $^{length\ ys}$ @ *lm* @
0 $^{a\text{-}md\ -\ Suc\ (?pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
apply(*rule-tac save-rs*, *auto simp*: *h*)
done
thm *rec-ci.simps*
thm *empty-boxes.simps*
from *g* **have** *k6*:
$\exists$ *stp. abc-steps-l* (*?gs-len* + *6* ∗ *length gs* + *3* ∗ *n* +
length a + *3*, *ys* @ *0* $^{?pstr\ -\ length\ ys}$ @ *rs* # *0* $^{length\ ys}$ @ *lm* @
0 $^{a\text{-}md\ -\ Suc\ (?pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
aprog stp =
(*?gs-len* + *8* ∗ *length gs* + *3* ∗*n* + *length a* + *3*,
0 $^{?pstr}$ @ *rs* # *0* $^{length\ ys}$ @ *lm* @
0 $^{a\text{-}md\ -Suc\ (?pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
apply(*drule-tac suf-lm* = *suf-lm* **in** *empty-paras*, *auto*)
apply(*rule-tac x* = *stp* **in** *exI*, *simp*)
done
from *g* **have** *k7*:
$\exists$ *stp. abc-steps-l* (*?gs-len* + *8* ∗ *length gs* + *3* ∗*n* +
length a + *3*, *0* $^{?pstr}$ @ *rs* # *0* $^{length\ ys}$ @ *lm* @
0 $^{a\text{-}md\ -Suc\ (?pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*) *aprog stp* =
(*?gs-len* + *8* ∗ *length gs* + *3* ∗ *n* + *length a* + *6*,
0 ^{n} @ *rs* # *0* $^{?pstr\ -\ n}$ @ *0* $^{length\ ys}$ @ *lm* @
0 $^{a\text{-}md\ -Suc\ (?pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*)
apply(*drule-tac suf-lm* = *suf-lm* **in** *restore-rs*, *auto*)
apply(*rule-tac x* = *stp* **in** *exI*, *simp*)
done
from *g* **have** *k8*: $\exists$ *stp. abc-steps-l* (*?gs-len* + *8* ∗ *length gs* +
3 ∗ *n* + *length a* + *6*,
0 ^{n} @ *rs* # *0* $^{?pstr\ -\ n}$ @ *0* $^{length\ ys}$ @ *lm* @
0 $^{a\text{-}md\ -Suc\ (?pstr\ +\ length\ ys\ +\ n)}$ @ *suf-lm*) *aprog stp* =
(*?gs-len* + *8* ∗ *length gs* + *6* ∗ *n* + *length a* + *6*,
lm @ *rs* # *0* $^{a\text{-}md\ -\ Suc\ n}$ @ *suf-lm*)
apply(*drule-tac suf-lm* = *suf-lm* **in** *restore-paras*, *auto*)
apply(*simp add*: *exponent-add-iff*)
apply(*rule-tac x* = *stp* **in** *exI*, *simp*)
done
from *g* **have** *j1*:
length aprog = *?gs-len* + *8* ∗ *length gs* + *6* ∗ *n* + *length a* + *6*
by(*drule-tac a* = *a* **and** *aa* = *b* **and** *ba* = *c* **in** *ci-cn-length*,
simp, *simp*, *simp*)

from *g* **have** *j2*: *rs-pos* = *n*
by(*simp add*: *ci-cn-para-eq*)
from *k1* **and** *k2* **and** *k3* **and** *k4* **and** *k5* **and** *k6* **and** *k7* **and** *k8*
and *j1* **and** *j2* **show**
$\exists$ *stp*. *abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog stp* =
(*length aprog*, *lm* @ [*rs*] @ $0^{a\text{-}md\ -\ rs\text{-}pos\ -\ 1}$ @ *suf-lm*)
apply(*auto*)
apply(*rule-tac x* = *stp* + *stpa* + *stpb* + *stpc* +
stpd + *stpe* + *stpf* + *stpg* **in** *exI*, *simp add*: *abc-steps-add*)
done
qed

Correctness of the complier (terminate case), which says if the execution of a recursive function *recf* terminates and gives result, then the Abacus program compiled from *recf* termintes and gives the same result. Additionally, to facilitate induction proof, we append *anything* to the end of Abacus memory.

lemma *aba-rec-equality*:
⟦*rec-ci recf* = (*ap*, *arity*, *fp*);
rec-calc-rel recf args r⟧
$\Longrightarrow$ ($\exists$ *stp*. (*abc-steps-l* (*0*, *args* @ $0^{fp\ -\ arity}$ @ *anything*) *ap stp*) =
(*length ap*, *args*@[*r*]@$0^{fp\ -\ arity\ -\ 1}$ @ *anything*))
apply(*induct arbitrary*: *ap fp arity r anything args*
rule: *rec-ci.induct*)
prefer *5*
proof(*case-tac rec-ci g*, *case-tac rec-ci f*, *simp*)
fix *n f g ap fp arity r anything args a b c aa ba ca*
assume *f-ind*:
$\bigwedge$*ap fp arity r anything args*.
⟦*aa* = *ap* $\wedge$ *ba* = *arity* $\wedge$ *ca* = *fp*; *rec-calc-rel f args r*⟧ $\Longrightarrow$
$\exists$ *stp*. *abc-steps-l* (*0*, *args* @ $0^{fp\ -\ arity}$ @ *anything*) *ap stp* =
(*length ap*, *args* @ *r* # $0^{fp\ -\ Suc\ arity}$ @ *anything*)
and *g-ind*:
$\bigwedge$*x xa y xb ya ap fp arity r anything args*.
⟦*x* = (*aa*, *ba*, *ca*); *xa* = *aa* $\wedge$ *y* = (*ba*, *ca*); *xb* = *ba* $\wedge$ *ya* = *ca*;
a = *ap* $\wedge$ *b* = *arity* $\wedge$ *c* = *fp*; *rec-calc-rel g args r*⟧
$\Longrightarrow$ $\exists$ *stp*. *abc-steps-l* (*0*, *args* @ $0^{fp\ -\ arity}$ @ *anything*) *ap stp* =
(*length ap*, *args* @ *r* # $0^{fp\ -\ Suc\ arity}$ @ *anything*)
and *h*: *rec-ci* (*Pr n f g*) = (*ap*, *arity*, *fp*)
rec-calc-rel (*Pr n f g*) *args r*
rec-ci g = (*a*, *b*, *c*)
rec-ci f = (*aa*, *ba*, *ca*)
from *h* **have** *nf-ind*:
$\bigwedge$ *args r anything*. *rec-calc-rel f args r* $\Longrightarrow$
$\exists$ *stp*. *abc-steps-l* (*0*, *args* @ $0^{ca\ -\ ba}$ @ *anything*) *aa stp* =
(*length aa*, *args* @ *r* # $0^{ca\ -\ Suc\ ba}$ @ *anything*)
and *ng-ind*:
$\bigwedge$ *args r anything*. *rec-calc-rel g args r* $\Longrightarrow$
$\exists$ *stp*. *abc-steps-l* (*0*, *args* @ $0^{c\ -\ b}$ @ *anything*) *a stp* =

(*length a*, *args* @ *r* # $0^{c - Suc\ b}$ @ *anything*)
apply(*insert f-ind*[*of aa ba ca*], *simp*)
apply(*insert g-ind*[*of* (*aa*, *ba*, *ca*) *aa* (*ba*, *ca*) *ba ca a b c*],
simp)
done
from *nf-ind* **and** *ng-ind* **and** *h* **show**
∃ *stp*. *abc-steps-l* (*0*, *args* @ $0^{fp - arity}$ @ *anything*) *ap stp* =
(*length ap*, *args* @ *r* # $0^{fp - Suc\ arity}$ @ *anything*)
apply(*auto intro*: *nf-ind ng-ind pr-case*)
done
next
fix *ap fp arity r anything args*
assume *h*:
rec-ci z = (*ap*, *arity*, *fp*) *rec-calc-rel z args r*
thus ∃ *stp*. *abc-steps-l* (*0*, *args* @ $0^{fp - arity}$ @ *anything*) *ap stp* =
(*length ap*, *args* @ [*r*] @ $0^{fp - arity - 1}$ @ *anything*)
by (*rule-tac z-case*)
next
fix *ap fp arity r anything args*
assume *h*:
rec-ci s = (*ap*, *arity*, *fp*)
rec-calc-rel s args r
thus
∃ *stp*. *abc-steps-l* (*0*, *args* @ $0^{fp - arity}$ @ *anything*) *ap stp* =
(*length ap*, *args* @ [*r*] @ $0^{fp - arity - 1}$ @ *anything*)
by(*erule-tac s-case*, *simp*)
next
fix *m n ap fp arity r anything args*
assume *h*: *rec-ci* (*id m n*) = (*ap*, *arity*, *fp*)
rec-calc-rel (*id m n*) *args r*
thus ∃ *stp*. *abc-steps-l* (*0*, *args* @ $0^{fp - arity}$ @ *anything*) *ap stp*
= (*length ap*, *args* @ [*r*] @ $0^{fp - arity - 1}$ @ *anything*)
by(*erule-tac id-case*)
next
fix *n f gs ap fp arity r anything args*
assume *ind*: ⋀*x ap fp arity r anything args*.
⟦*x* ∈ *set* (*f* # *gs*);
rec-ci x = (*ap*, *arity*, *fp*);
rec-calc-rel x args r⟧
⟹ ∃ *stp*. *abc-steps-l* (*0*, *args* @ $0^{fp - arity}$ @ *anything*) *ap stp* =
(*length ap*, *args* @ [*r*] @ $0^{fp - arity - 1}$ @ *anything*)
and *h*: *rec-ci* (*Cn n f gs*) = (*ap*, *arity*, *fp*)
rec-calc-rel (*Cn n f gs*) *args r*
from *h* **show**
∃ *stp*. *abc-steps-l* (*0*, *args* @ $0^{fp - arity}$ @ *anything*)
ap stp = (*length ap*, *args* @ [*r*] @ $0^{fp - arity - 1}$ @ *anything*)
apply(*rule-tac cn-case*, *rule-tac ind*, *auto*)
done

next
  fix *n f ap fp arity r anything args*
  assume *ind*:
    $\bigwedge$*ap fp arity r anything args*.
    ⟦*rec-ci f* = (*ap*, *arity*, *fp*); *rec-calc-rel f args r*⟧ $\Longrightarrow$
    ∃ *stp*. *abc-steps-l* (*0*, *args* @ $0^{fp - arity}$ @ *anything*) *ap stp* =
    (*length ap*, *args* @ [*r*] @ $0^{fp - arity - 1}$ @ *anything*)
  and *h*: *rec-ci* (*Mn n f*) = (*ap*, *arity*, *fp*)
    rec-calc-rel (*Mn n f*) *args r*
  from *h* **show**
    ∃ *stp*. *abc-steps-l* (*0*, *args* @ $0^{fp - arity}$ @ *anything*) *ap stp* =
        (*length ap*, *args* @ [*r*] @ $0^{fp - arity - 1}$ @ *anything*)
    apply(*rule-tac mn-case*, *rule-tac ind*, *auto*)
    done
qed

thm *abc-append-state-in-exc*
lemma *abc-append-uhalt1*:
  ⟦∀ *stp*. (λ (*ss*, *e*). *ss* < *length bp*) (*abc-steps-l* (*0*, *lm*) *bp stp*);
    p = *ap* [+] *bp* [+] *cp*⟧
  $\Longrightarrow$ ∀ *stp*. (λ (*ss*, *e*). *ss* < *length p*)
          (*abc-steps-l* (*length ap*, *lm*) *p stp*)
apply(*auto*)
apply(*erule-tac x* = *stp* **in** *allE*, *auto*)
apply(*frule-tac ap* = *ap* **and** *cp* = *cp* **in** *abc-append-state-in-exc*, *auto*)
done

lemma *abc-append-unhalt2*:
  ⟦*abc-steps-l* (*0*, *am*) *ap stp* = (*length ap*, *lm*); *bp* ≠ [];
  ∀ *stp*. (λ (*ss*, *e*). *ss* < *length bp*) (*abc-steps-l* (*0*, *lm*) *bp stp*);
  p = *ap* [+] *bp* [+] *cp*⟧
  $\Longrightarrow$ ∀ *stp*. (λ (*ss*, *e*). *ss* < *length p*) (*abc-steps-l* (*0*, *am*) *p stp*)
proof −
  assume *h*:
    abc-steps-l (*0*, *am*) *ap stp* = (*length ap*, *lm*)
    bp ≠ []
    ∀ *stp*. (λ (*ss*, *e*). *ss* < *length bp*) (*abc-steps-l* (*0*, *lm*) *bp stp*)
    p = *ap* [+] *bp* [+] *cp*
  have ∃ *stp*. (*abc-steps-l* (*0*, *am*) *p stp*) = (*length ap*, *lm*)
    using *h*
    thm *abc-add-exc1*
    apply(*simp add*: *abc-append.simps*)
    apply(*rule-tac abc-add-exc1*, *auto*)
    done
  from *this* **obtain** *stpa* **where** *g1*:
    (*abc-steps-l* (*0*, *am*) *p stpa*) = (*length ap*, *lm*) **..**
  moreover have *g2*: ∀ *stp*. (λ (*ss*, *e*). *ss* < *length p*)

(*abc-steps-l* (*length ap*, *lm*) *p stp*)
using *h*
apply(*erule-tac abc-append-uhalt1*, *simp*)
done
moreover from *g1* **and** *g2* **have**
∀ *stp*. (λ (*ss*, *e*). *ss* < *length p*)
(*abc-steps-l* (*0*, *am*) *p* (*stpa* + *stp*))
apply(*simp add*: *abc-steps-add*)
done
thus ∀ *stp*. (λ (*ss*, *e*). *ss* < *length p*)
(*abc-steps-l* (*0*, *am*) *p stp*)
apply(*rule-tac allI*, *auto*)
apply(*case-tac stp* ≥ *stpa*)
apply(*erule-tac x* = *stp* − *stpa* **in** *allE*, *simp*)
proof −
fix *stp a b*
assume *g3*: *abc-steps-l* (*0*, *am*) *p stp* = (*a*, *b*)
¬ *stpa* ≤ *stp*
thus *a* < *length p*
using *g1 h*
apply(*case-tac a* < *length p*, *simp*, *simp*)
apply(*subgoal-tac* ∃ *d*. *stpa* = *stp* + *d*)
using *abc-state-keep*[*of p a b stpa* − *stp*]
apply(*erule-tac exE*, *simp add*: *abc-steps-add*)
apply(*rule-tac x* = *stpa* − *stp* **in** *exI*, *simp*)
done
qed
qed

Correctness of the complier (non-terminating case for Mn). There are many cases when a recursive function does not terminate. For the purpose of Uiversal Turing Machine, we only need to prove the case for *Mn* and *Cn*. This lemma is for *Mn*. For *Mn n f*, this lemma describes what happens when *f* always terminates but always does not return zero, so that *Mn* has to loop forever.

lemma *Mn-unhalt*:
assumes *mn-rf*: *rf* = *Mn n f*
and *compiled-mnrf*: *rec-ci rf* = (*aprog*, *rs-pos*, *a-md*)
and *compiled-f*: *rec-ci f* = (*aprog'*, *rs-pos'*, *a-md'*)
and *args*: *length lm* = *n*
and *unhalt-condition*: ∀ *y*. (∃ *rs*. *rec-calc-rel f* (*lm* @ [*y*]) *rs* ∧ *rs* ≠ *0*)
shows ∀ *stp*. *case abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*)
aprog stp of (*ss*, *e*) ⇒ *ss* < *length aprog*
using *mn-rf compiled-mnrf compiled-f args unhalt-condition*
proof(*rule-tac allI*)
fix *stp*
assume *h*: *rf* = *Mn n f*
rec-ci rf = (*aprog*, *rs-pos*, *a-md*)

rec-ci f = (*aprog*′, *rs-pos*′, *a-md*′)
∀ *y*. ∃ *rs*. *rec-calc-rel f* (*lm* @ [*y*]) *rs* ∧ *rs* ≠ *0 length lm* = *n*

thm *mn-ind-step*
have ∃ *stpa* ≥ *stp*. *abc-steps-l* (*0*, *lm* @ *0* # *0*$^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*) *aprog* *stpa*
= (*0*, *lm* @ *stp* # *0*$^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
proof(*induct stp*, *auto*)
show ∃ *stpa*. *abc-steps-l* (*0*, *lm* @ *0* # *0*$^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
aprog stpa = (*0*, *lm* @ *0* # *0*$^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
apply(*rule-tac x* = *0* **in** *exI*, *simp add*: *abc-steps-l.simps*)
done
next
fix *stp stpa*
assume *g1*: *stp* ≤ *stpa*
and *g2*: *abc-steps-l* (*0*, *lm* @ *0* # *0*$^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
aprog stpa
= (*0*, *lm* @ *stp* # *0*$^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
have ∃ *rs*. *rec-calc-rel f* (*lm* @ [*stp*]) *rs* ∧ *rs* ≠ *0*
using *h*
apply(*erule-tac x* = *stp* **in** *allE*, *simp*)
done
from *this* **obtain** *rs* **where** *g3*:
rec-calc-rel f (*lm* @ [*stp*]) *rs* ∧ *rs* ≠ *0* **..**
hence ∃ *stpb*. *abc-steps-l* (*0*, *lm* @ *stp* # *0*$^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @
suf-lm) *aprog stpb*
= (*0*, *lm* @ *Suc stp* # *0*$^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
using *h*
apply(*rule-tac mn-ind-step*)
apply(*rule-tac aba-rec-equality*, *simp*, *simp*)
proof −
show *rec-ci f* = ((*aprog*′, *rs-pos*′, *a-md*′)) **using** *h* **by** *simp*
next
show *rec-ci* (*Mn n f*) = (*aprog*, *rs-pos*, *a-md*) **using** *h* **by** *simp*
next
show *rec-calc-rel f* (*lm* @ [*stp*]) *rs* **using** *g3* **by** *simp*
next
show *0* < *rs* **using** *g3* **by** *simp*
next
show *Suc rs-pos* < *a-md*
using *g3 h*
apply(*auto*)
apply(*frule-tac f* = *f* **in** *para-pattern*, *simp*, *simp*)
apply(*simp add*: *rec-ci.simps*, *auto*)
apply(*subgoal-tac Suc* (*length lm*) < *a-md*′)
apply(*arith*)
apply(*simp add*: *ci-ad-ge-paras*)
done
next
show *rs-pos*′ = *Suc rs-pos*

using *g3 h*
apply(*auto*)
apply(*frule-tac f* = *f* **in** *para-pattern*, *simp*, *simp*)
apply(*simp add*: *rec-ci.simps*)
done
qed
thus ∃ *stpa*≥*Suc stp*. *abc-steps-l* (*0*, *lm* @ *0* # $0^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*) *aprog stpa*
= (*0*, *lm* @ *Suc stp* # $0^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
using *g2*
apply(*erule-tac exE*)
apply(*case-tac stpb* = *0*, *simp add*: *abc-steps-l.simps*)
apply(*rule-tac x* = *stpa* + *stpb* **in** *exI*, *simp add*:
abc-steps-add)
using *g1*
apply(*arith*)
done
qed
from *this* **obtain** *stpa* **where**
stp ≤ *stpa* ∧ *abc-steps-l* (*0*, *lm* @ *0* # $0^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*)
aprog stpa = (*0*, *lm* @ *stp* # $0^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*) **..**
thus *case abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog stp*
of (*ss*, *e*) ⇒ *ss* < *length aprog*
apply(*case-tac abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm*) *aprog*
stp, *simp*, *case-tac a* ≥ *length aprog*,
simp, *simp*)
apply(*subgoal-tac* ∃ *d*. *stpa* = *stp* + *d*, *erule-tac exE*)
apply(*subgoal-tac lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suf-lm* = *lm* @ *0* #
$0^{a\text{-}md\ -\ Suc\ rs\text{-}pos}$ @ *suf-lm*, *simp add*: *abc-steps-add*)
apply(*frule-tac as* = *a* **and** *lm* = *b* **and** *stp* = *d* **in** *abc-state-keep*,
simp)
using *h*
apply(*simp add*: *rec-ci.simps*, *simp*,
simp only: *exp-ind-def*[*THEN sym*])
apply(*case-tac rs-pos*, *simp*, *simp*)
apply(*rule-tac x* = *stpa* − *stp* **in** *exI*, *simp*, *simp*)
done
qed

lemma *abc-append-cons-eq*[*intro*!]:
⟦*ap* = *bp*; *cp* = *dp*⟧ ⟹ *ap* [+] *cp* = *bp* [+] *dp*
by *simp*

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* [+]
empty gb (*p* + *i*) [+]

cn-merge-gs (*map rec-ci* (*drop* (*Suc i*) *gs*)) (*p* + *Suc i*)
apply(*induct i arbitrary*: *gs p*, *case-tac gs*, *simp*, *simp*)
apply(*case-tac gs*, *simp*, *case-tac rec-ci a*,
simp add: *abc-append-commute*[*THEN sym*])
done

Correctness of the complier (non-terminating case for Mn). There are many cases when a recursive function does not terminate. For the purpose of Uiversal Turing Machine, we only need to prove the case for *Mn* and *Cn*. This lemma is for *Cn*. For *Cn f g1 g2 . . . gi*, *gi+1*, . . . *gn*, this lemma describes what happens when every one of *g1*, *g2*, . . . *gi* terminates, but *gi+1* does not terminate, so that whole function *Cn f g1 g2 . . . gi*, *gi+1*, . . . *gn* does not terminate.

lemma *cn-gi-uhalt*:
assumes *cn-recf*: *rf* = *Cn n f gs*
and *compiled-cn-recf*: *rec-ci rf* = (*aprog*, *rs-pos*, *a-md*)
and *args-length*: *length lm* = *n*
and *exist-unhalt-recf*: *i* < *length gs gi* = *gs* ! *i*
and *complied-unhalt-recf*: *rec-ci gi* = (*ga*, *gb*, *gc*) *gb* = *n*
and *all-halt-before-gi*: ∀ *j* < *i*. (∃ *rs*. *rec-calc-rel* (*gs*!*j*) *lm rs*)
and *unhalt-condition*: ⋀ *slm*. ∀ *stp*. *case abc-steps-l* (*0*, *lm* @ $0^{gc - gb}$ @ *slm*) *ga stp of* (*se*, *e*) ⇒ *se* < *length ga*
shows ∀ *stp*. *case abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md - rs\text{-}pos}$ @ *suflm*) *aprog stp of* (*ss*, *e*) ⇒ *ss* < *length aprog*
using *cn-recf compiled-cn-recf args-length exist-unhalt-recf complied-unhalt-recf all-halt-before-gi unhalt-condition*
proof(*case-tac rec-ci f*, *simp*)
fix *a b c*
assume *h1*: *rf* = *Cn n f gs*
rec-ci (*Cn n f gs*) = (*aprog*, *rs-pos*, *a-md*)
length lm = *n*
gi = *gs* ! *i*
rec-ci (*gs*!*i*) = (*ga*, *n*, *gc*)
gb = *n rec-ci f* = (*a*, *b*, *c*)
and *h2*: ∀*j*<*i*. ∃*rs*. *rec-calc-rel* (*gs* ! *j*) *lm rs*
i < *length gs*
and *ind*:
⋀ *slm*. ∀ *stp*. *case abc-steps-l* (*0*, *lm* @ $0^{gc - n}$ @ *slm*) *ga stp of* (*se*, *e*) ⇒ *se* < *length ga*
have *h3*: *rs-pos* = *n*
using *h1*
by(*rule-tac ci-cn-para-eq*, *simp*)
let *?ggs* = *take i gs*
have ∃ *ys*. (*length ys* = *i* ∧
(∀ *k* < *i*. *rec-calc-rel* (*?ggs* ! *k*) *lm* (*ys* ! *k*)))
using *h2*
apply(*induct i*, *simp*, *simp*)
apply(*erule-tac exE*)

apply(*erule-tac x = ia* **in** *allE*, *simp*)
apply(*erule-tac exE*)
apply(*rule-tac x = ys @ [x]* **in** *exI*, *simp add*: *nth-append*, *auto*)
apply(*subgoal-tac k = length ys*, *simp*, *simp*)
done
from *this* **obtain** *ys* **where** *g1*:
(*length ys = i* ∧ (∀ *k < i*. *rec-calc-rel* (*?ggs ! k*)
lm (*ys ! k*))) **..**
let *?pstr = Max* (*set* (*Suc n # c # map* (λ(*aprog*, *p*, *n*). *n*)
(*map rec-ci* (*f # gs*))))
have ∃ *stp*. *abc-steps-l* (*0*, *lm @ $0^{a\text{-}md - n}$ @ suflm*)
(*cn-merge-gs* (*map rec-ci ?ggs*) *?pstr*) *stp* =
(*listsum* (*map* ((λ(*ap*, *pos*, *n*). *length ap*) ∘ *rec-ci*) *?ggs*) +
3 ∗ length ?ggs, *lm @ $0^{?pstr - n}$ @ ys @ $0^{a\text{-}md - (?pstr + length\ ?ggs)}$ @*
suflm)
apply(*rule-tac cn-merge-gs-ex*)
apply(*rule-tac aba-rec-equality*, *simp*, *simp*)
using *h1*
apply(*simp add*: *rec-ci.simps*, *auto*)
using *g1*
apply(*simp*)
using *h2 g1*
apply(*simp*)
apply(*rule-tac min-max.le-supI2*)
apply(*rule-tac Max-ge*, *simp*, *simp*, *rule-tac disjI2*)
apply(*subgoal-tac aa* ∈ *set gs*, *simp*)
using *h2*
apply(*rule-tac A = set* (*take i gs*) **in** *subsetD*,
simp add: *set-take-subset*, *simp*)
done
thm *cn-merge-gs.simps*
from *this* **obtain** *stpa* **where** *g2*:
abc-steps-l (*0*, *lm @ $0^{a\text{-}md - n}$ @ suflm*)
(*cn-merge-gs* (*map rec-ci ?ggs*) *?pstr*) *stpa* =
(*listsum* (*map* ((λ(*ap*, *pos*, *n*). *length ap*) ∘ *rec-ci*) *?ggs*) +
3 ∗ length ?ggs, *lm @ $0^{?pstr - n}$ @ ys @ $0^{a\text{-}md - (?pstr + length\ ?ggs)}$ @*
suflm) **..**
moreover have
∃ *cp*. *aprog* = (*cn-merge-gs*
(*map rec-ci ?ggs*) *?pstr*) [+] *ga* [+] *cp*
using *h1*
apply(*simp add*: *rec-ci.simps*)
apply(*rule-tac x = empty n* (*?pstr + i*) [+]
(*cn-merge-gs* (*map rec-ci* (*drop* (*Suc i*) *gs*)) (*?pstr + Suc i*))
[+]*mv-boxes 0* (*Suc* (*max* (*Suc n*) (*Max* (*insert c*
(((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ‘ *set gs*))) +
length gs)) *n* [+] *mv-boxes* (*max* (*Suc n*) (*Max* (*insert c*
(((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ‘ *set gs*)))) *0* (*length gs*) [+]
a [+] *recursive.empty b* (*max* (*Suc n*)

(*Max* (*insert c* (((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ' *set gs*)))) [+]
empty-boxes (*length gs*) [+] *recursive.empty* (*max* (*Suc n*)
(*Max* (*insert c* (((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ' *set gs*)))) *n* [+]
mv-boxes (*Suc* (*max* (*Suc n*) (*Max* (*insert c*
(((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ' *set gs*))) + *length gs*)) *0 n* **in** *exI*)
apply(*simp add*: *abc-append-commute* [*THEN sym*])
apply(*auto*)
using *cn-merge-gs-split*[*of i gs ga length lm gc*
(*max* (*Suc* (*length lm*))
(*Max* (*insert c* (((λ(*aprog*, *p*, *n*). *n*) ∘ *rec-ci*) ' *set gs*))))]
h2
apply(*simp*)
done
from *this* **obtain** *cp* **where** *g3*:
aprog = (*cn-merge-gs* (*map rec-ci ?ggs*) *?pstr*) [+] *ga* [+] *cp* **..**
show ∀ *stp*. *case abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suflm*)
aprog stp of (*ss*, *e*) ⇒ *ss* < *length aprog*
proof(*rule-tac abc-append-unhalt2*)
show *abc-steps-l* (*0*, *lm* @ $0^{a\text{-}md\ -\ rs\text{-}pos}$ @ *suflm*) (
cn-merge-gs (*map rec-ci ?ggs*) *?pstr*) *stpa* =
(*length* ((*cn-merge-gs* (*map rec-ci ?ggs*) *?pstr*)),
lm @ $0^{?pstr\ -\ n}$ @ *ys* @ $0^{a\text{-}md\ -(?pstr\ +\ length\ ?ggs)}$ @ *suflm*)
using *h3 g2*
apply(*simp add*: *cn-merge-gs-length*)
done
next
show *ga* ≠ []
using *h1*
apply(*simp add*: *rec-ci-not-null*)
done
next
show ∀*stp*. *case abc-steps-l* (*0*, *lm* @ $0^{?pstr\ -\ n}$ @ *ys*
@ $0^{a\text{-}md\ -\ (?pstr\ +\ length\ (take\ i\ gs))}$ @ *suflm*) *ga stp of*
(*ss*, *e*) ⇒ *ss* < *length ga*
using *ind*[*of* $0^{?pstr\ -gc}$ @ *ys* @ $0^{a\text{-}md\ -\ (?pstr\ +\ length\ (take\ i\ gs))}$
@ *suflm*]
apply(*subgoal-tac lm* @ $0^{?pstr\ -\ n}$ @ *ys*
@ $0^{a\text{-}md\ -\ (?pstr\ +\ length\ (take\ i\ gs))}$ @ *suflm*
= *lm* @ $0^{gc\ -\ n}$ @
$0^{?pstr\ -gc}$ @ *ys* @ $0^{a\text{-}md\ -\ (?pstr\ +\ length\ (take\ i\ gs))}$ @ *suflm*, *simp*)
apply(*simp add*: *exponent-def replicate-add*[*THEN sym*])
apply(*subgoal-tac gc* > *n* ∧ *?pstr* ≥ *gc*)
apply(*erule-tac conjE*)
apply(*simp add*: *h1*)
using *h1*
apply(*auto*)
apply(*rule-tac min-max.le-supI2*)
apply(*rule-tac Max-ge*, *simp*, *simp*)

apply(*rule-tac disjI2*)
using *h2*
thm *rev-image-eqI*
apply(*rule-tac x = gs!i* **in** *rev-image-eqI*, *simp*, *simp*)
done
next
show *aprog = cn-merge-gs (map rec-ci (take i gs)) ?pstr [+] ga [+] cp*
using *g3* **by** *simp*
qed
qed

lemma *abc-rec-halt-eq'*:
⟦*rec-ci re = (ap, ary, fp)*;
rec-calc-rel re args r⟧
⟹ (∃ *stp*. (*abc-steps-l* (*0*, *args* @ $0^{fp - ary}$) *ap stp*) =
(*length ap*, *args*@[*r*]@$0^{fp - ary - 1}$))
using *aba-rec-equality*[*of re ap ary fp args r* []]
by *simp*

thm *abc-step-l.simps*
definition *dummy-abc* :: *nat* ⇒ *abc-inst list*
where
dummy-abc k = [*Inc k*, *Dec k 0*, *Goto 3*]

lemma *abc-rec-halt-eq''*:
⟦*rec-ci re = (aprog, rs-pos, a-md)*;
rec-calc-rel re lm rs⟧
⟹ (∃ *stp lm' m*. (*abc-steps-l* (*0*, *lm*) *aprog stp*) =
(*length aprog*, *lm'*) ∧ *abc-list-crsp lm'* (*lm* @ *rs* # 0^m))
apply(*frule-tac abc-rec-halt-eq'*, *auto*)
apply(*drule-tac abc-list-crsp-steps*)
apply(*rule-tac rec-ci-not-null*, *simp*)
apply(*erule-tac exE*, *rule-tac x = stp* **in** *exI*,
auto simp: *abc-list-crsp-def*)
done

lemma [*simp*]: *length* (*dummy-abc* (*length lm*)) = *3*
apply(*simp add*: *dummy-abc-def*)
done

lemma [*simp*]: *dummy-abc* (*length lm*) ≠ []
apply(*simp add*: *dummy-abc-def*)
done

lemma *dummy-abc-steps-ex*:
∃ *bstp*. *abc-steps-l* (*0*, *lm'*) (*dummy-abc* (*length lm*)) *bstp* =
((*Suc* (*Suc* (*Suc 0*))), *abc-lm-s lm'* (*length lm*) (*abc-lm-v lm'* (*length lm*)))

```
apply(rule-tac x = Suc (Suc (Suc 0)) in exI)
apply(auto simp: abc-steps-l.simps abc-step-l.simps
  dummy-abc-def abc-fetch.simps)
apply(auto simp: abc-lm-s.simps abc-lm-v.simps nth-append)
apply(simp add: butlast-append)
done

lemma [elim]:
  lm @ rs # 0^m = lm' @ 0^n ⟹
  ∃ m. abc-lm-s lm' (length lm) (abc-lm-v lm' (length lm)) =
                          lm @ rs # 0^m
proof(cases length lm' > length lm)
  case True
  assume h: lm @ rs # 0^m = lm' @ 0^n length lm < length lm'
  hence m ≥ n
    apply(drule-tac list-length)
    apply(simp)
    done
  hence ∃ d. m = d + n
    apply(rule-tac x = m − n in exI, simp)
    done
  from this obtain d where m = d + n ..
  from h and this show ?thesis
    apply(auto simp: abc-lm-s.simps abc-lm-v.simps
                 exponent-def replicate-add)
    done
next
  case False
  assume h:lm @ rs # 0^m = lm' @ 0^n
    and    g: ¬ length lm < length lm'
  have take (Suc (length lm)) (lm @ rs # 0^m) =
                      take (Suc (length lm)) (lm' @ 0^n)
    using h by simp
  moreover have n ≥ (Suc (length lm) − length lm')
    using h g
    apply(drule-tac list-length)
    apply(simp)
    done
  ultimately show
    ∃ m. abc-lm-s lm' (length lm) (abc-lm-v lm' (length lm)) =
                                                  lm @ rs # 0^m
    using g h
    apply(simp add: abc-lm-s.simps abc-lm-v.simps
                                  exponent-def min-def)
    apply(rule-tac x = 0 in exI,
      simp add:replicate-append-same replicate-Suc[THEN sym]
                                del:replicate-Suc)
    done
qed
```

lemma [*elim*]:
  abc-list-crsp lm' (*lm* @ *rs* # *0*^{m})
  ⟹ ∃ *m*. *abc-lm-s lm'* (*length lm*) (*abc-lm-v lm'* (*length lm*))
          = *lm* @ *rs* # *0*^{m}
apply(*auto simp*: *abc-list-crsp-def*)
apply(*simp add*: *abc-lm-v.simps abc-lm-s.simps*)
apply(*rule-tac x* = *m* + *n* **in** *exI*,
    simp add: *exponent-def replicate-add*)
done

lemma *abc-append-dummy-complie*:
  ⟦*rec-ci recf* = (*ap*, *ary*, *fp*);
   rec-calc-rel recf args r;
   length args = *k*⟧
  ⟹ (∃ *stp m*. (*abc-steps-l* (*0*, *args*) (*ap* [+] *dummy-abc k*) *stp*) =
          (*length ap* + *3*, *args* @ *r* # *0*^{m}))
apply(*drule-tac abc-rec-halt-eq''*, *auto simp*: *numeral-3-eq-3*)
proof −
  fix *stp lm' m*
  assume *h*: *rec-calc-rel recf args r*
   abc-steps-l (*0*, *args*) *ap stp* = (*length ap*, *lm'*)
   abc-list-crsp lm' (*args* @ *r* # *0*^{m})
  thm *abc-append-exc2*
  thm *abc-lm-s.simps*
  have ∃ *stp*. *abc-steps-l* (*0*, *args*) (*ap* [+]
   (*dummy-abc* (*length args*))) *stp* = (*length ap* + *3*,
   abc-lm-s lm' (*length args*) (*abc-lm-v lm'* (*length args*)))
   using *h*
   apply(*rule-tac bm* = *lm'* **in** *abc-append-exc2*,
      auto intro: *dummy-abc-steps-ex simp*: *numeral-3-eq-3*)
   done
  thus ∃ *stp m*. *abc-steps-l* (*0*, *args*) (*ap* [+]
   dummy-abc (*length args*)) *stp* = (*Suc* (*Suc* (*Suc* (*length ap*))), *args* @ *r* # *0*^{m})
   using *h*
   apply(*erule-tac exE*)
   apply(*rule-tac x* = *stpa* **in** *exI*, *auto*)
   done
qed

lemma [*simp*]: *length* (*dummy-abc k*) = *3*
apply(*simp add*: *dummy-abc-def*)
done

lemma [*simp*]: *length args* = *k* ⟹ *abc-lm-v* (*args* @ *r* # *0*^{m}) *k* = *r*
apply(*simp add*: *abc-lm-v.simps nth-append*)
done

lemma *t-compiled-by-rec*:
⟦*rec-ci recf* = (*ap*, *ary*, *fp*);
rec-calc-rel recf args r;
length args = *k*;
ly = *layout-of* (*ap* [+] *dummy-abc k*);
mop-ss = *start-of ly* (*length* (*ap* [+] *dummy-abc k*));
tp = *tm-of* (*ap* [+] *dummy-abc k*)⟧
⟹ ∃ *stp m l*. *steps* (*Suc 0*, *Bk* # *Bk* # *ires*, <*args*> @ Bk^{rn}) (*tp* @ (*tMp k* (*mop-ss* − *1*))) *stp*
= (*0*, Bk^{m} @ *Bk* # *Bk* # *ires*, $Oc^{Suc\ r}$ @ Bk^{l})
using *abc-append-dummy-complie*[*of recf ap ary fp args r k*]
apply(*simp*)
apply(*erule-tac exE*)+
apply(*frule-tac tprog* = *tp* **and** *as* = *length ap* + *3* **and** *n* = *k*
and *ires* = *ires* **and** *rn* = *rn* **in** *abacus-turing-eq-halt*, *simp-all*, *simp*)
apply(*erule-tac exE*)+
apply(*simp*)
apply(*rule-tac x* = *stpa* **in** *exI*, *rule-tac x* = *ma* **in** *exI*, *rule-tac x* = *l* **in** *exI*, *simp*)
done

thm *tms-of.simps*

lemma [*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*
apply(*induct xs*, *simp*, *simp*)
apply(*case-tac a*, *simp*)
done

lemma *tshift-append*: *tshift* (*xs* @ *ys*) *n* = *tshift xs n* @ *tshift ys n*
apply(*simp add*: *tshift.simps*)
done

lemma [*simp*]: *length* (*tMp n ss*) = *4* ∗ *n* + *12*
apply(*auto simp*: *tMp.simps tshift-append shift-length mp-up-def*)
done

lemma *length-tm-even*[*intro*]: ∃ *x*. *length* (*tm-of ap*) = *2*∗*x*
apply(*subgoal-tac t-ncorrect* (*tm-of ap*))
apply(*simp add*: *t-ncorrect.simps*, *auto*)
done

lemma [*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 [*elim*]: ⟦k < *length ap*; *ap* ! k = *Inc n*;
(*a*, *b*) ∈ *set* (*abacus.tshift* (*abacus.tshift 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(*case-tac Suc k* = *length ap*, *simp*)
apply(*rule-tac start-of-le*, *simp*)
apply(*auto simp*: *tinc-b-def tshift.simps start-of.simps*
layout-of.simps length-of.simps startof-not0)
done

lemma *findnth-le*[*elim*]: (*a*, *b*) ∈ *set* (*abacus.tshift* (*findnth n*) (*start-of* (*layout-of ap*) *k* − *Suc 0*))
⟹ *b* ≤ *Suc* (*start-of* (*layout-of ap*) *k* + *2* ∗ *n*)
apply(*induct n*, *simp add*: *findnth.simps tshift.simps*)
apply(*simp add*: *findnth.simps tshift-append*, *auto*)
apply(*auto simp*: *tshift.simps*)
done

lemma [*elim*]: ⟦*k* < *length ap*; *ap* ! *k* = *Inc n*; (*a*, *b*) ∈
set (*abacus.tshift* (*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(*case-tac Suc k* = *length ap*, *simp*)
apply(*rule-tac start-of-le*, *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 tshift.simps start-of.simps*
layout-of.simps length-of.simps startof-not0)
done

lemma *start-of-eq*: *length ap* < *as* ⟹ *start-of* (*layout-of ap*) *as* = *start-of* (*layout-of ap*) (*length ap*)
apply(*induct as*, *simp*)
apply(*case-tac length ap* < *as*, *simp add*: *start-of.simps*)
apply(*subgoal-tac as* = *length ap*)
apply(*simp add*: *start-of.simps*)
apply *arith*
done

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-le*)
done

lemma [*elim*]: ⟦*k* < *length ap*;
ap ! *k* = *Dec n e*;
(*a*, *b*) ∈ *set* (*abacus.tshift* (*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(*simp add*: *tshift.simps start-of.simps*
layout-of.simps length-of.simps startof-not0)
apply(*rule-tac start-of-all-le*)
done

thm *length-of.simps*
lemma [*elim*]: ⟦*k* < *length ap*; *ap* ! *k* = *Dec n e*; (*a*, *b*) ∈ *set* (*abacus.tshift* (*abacus.tshift 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 2* ∗ *n* + *start-of* (*layout-of ap*) *k* + *16* = *start-of* (*layout-of ap*) (*Suc k*)
∧ *start-of* (*layout-of ap*) (*Suc k*) ≤ *start-of* (*layout-of ap*) (*length ap*))
apply(*simp add*: *startof-not0*, *rule-tac conjI*)
apply(*simp add*: *start-of.simps layout-of.simps length-of.simps*)
apply(*rule-tac start-of-all-le*)
apply(*auto simp*: *tdec-b-def tshift.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(*simp*)
apply(*case-tac ap*!*k*, *simp-all add*: *ci.simps tshift-append*, *auto intro*: *start-of-all-le*)
done
lemma *concat-in*: *i* < *length* (*concat xs*) ⟹ ∃ *k* < *length xs*. *concat xs* ! *i* ∈ *set* (*xs* ! *k*)
apply(*induct xs rule*: *list-tl-induct*, *simp*, *simp*)
apply(*case-tac i* < *length* (*concat list*), *simp*)
apply(*erule-tac exE*, *rule-tac x* = *k* **in** *exI*)
apply(*simp add*: *nth-append*)
apply(*rule-tac x* = *length list* **in** *exI*, *simp*)
apply(*simp add*: *nth-append*)

done

lemma [*simp*]: *length* (*tms-of ap*) = *length ap*
apply(*simp add*: *tms-of.simps tpairs-of.simps*)
done

lemma *in-tms*: $i < length\ (tm\text{-}of\ ap) \Longrightarrow \exists\ k < length\ ap.\ (tm\text{-}of\ ap\ !\ i) \in set\ (tms\text{-}of\ ap\ !\ k)$
apply(*simp add*: *tm-of.simps*)
using *concat-in*[*of i tms-of ap*]
by *simp*

lemma *all-le-start-of*: *list-all* $(\lambda(acn, s).\ s \leq start\text{-}of\ (layout\text{-}of\ ap)\ (length\ ap))$ (*tm-of ap*)
apply(*simp add*: *list-all-length*)
apply(*rule-tac allI*, *rule-tac impI*)
apply(*drule-tac in-tms*, *auto elim*: *tms-any-less*)
done

lemma *length-ci*: $[\![k < length\ ap;\ length\ (ci\ ly\ y\ (ap\ !\ k)) = 2 * qa]\!]$
$\Longrightarrow layout\text{-}of\ ap\ !\ k = qa$
apply(*case-tac ap* ! *k*)
apply(*auto simp*: *layout-of.simps ci.simps*
length-of.simps shift-length tinc-b-def tdec-b-def)
done

lemma [*intro*]: *length* (*ci ly y i*) *mod* 2 = 0
apply(*auto simp*: *ci.simps shift-length tinc-b-def tdec-b-def*
split: *abc-inst.splits*)
done

lemma [*intro*]: *listsum* (*map* $(length \circ (\lambda(x, y).\ ci\ ly\ x\ y))$ *zs*) *mod* 2 = 0
apply(*induct zs rule*: *list-tl-induct*, *simp*)
apply(*case-tac a*, *simp*)
apply(*subgoal-tac length* (*ci ly aa b*) *mod* 2 = 0)
apply(*auto*)
done

lemma *zip-pre*:
(*length ys*) ≤ *length ap* ⟹
zip ys ap = *zip ys* (*take* (*length ys*) (*ap*::'*a list*))
proof(*induct ys arbitrary*: *ap*, *simp*, *case-tac ap*, *simp*)
fix *a ys ap aa list*
assume *ind*: ⋀(*ap*::'*a list*). *length ys* ≤ *length ap* ⟹
zip ys ap = *zip ys* (*take* (*length ys*) *ap*)
and *h*: *length* (*a* # *ys*) ≤ *length ap* (*ap*::'*a list*) = *aa* # (*list*::'*a list*)
from *h* **show** *zip* (*a* # *ys*) *ap* = *zip* (*a* # *ys*) (*take* (*length* (*a* # *ys*)) *ap*)
using *ind*[*of list*]
apply(*simp*)

```
    done
qed

lemma start-of-listsum:
  ⟦k ≤ length ap; length ss = k⟧ ⟹ start-of (layout-of ap) k =
        Suc (listsum (map (length ∘ (λ(x, y). ci ly x y)) (zip ss ap)) div 2)
proof(induct k arbitrary: ss, simp add: start-of.simps, simp)
  fix k ss
  assume ind: ⋀ss. length ss = k ⟹ start-of (layout-of ap) k =
            Suc (listsum (map (length ∘ (λ(x, y). ci ly x y)) (zip ss ap)) div 2)
  and h: Suc k ≤  length ap length (ss::nat list) = Suc k
  have ∃ ys y. ss = ys @ [y]
    using h
    apply(rule-tac x = butlast ss in exI,
          rule-tac x = last ss in exI)
    apply(case-tac ss = [], auto)
    done
  from this obtain ys y where k1: ss = (ys::nat list) @ [y]
    by blast
  from h and this have k2:
    start-of (layout-of ap) k =
    Suc (listsum (map (length ∘ (λ(x, y). ci ly x y)) (zip ys ap)) div 2)
    apply(rule-tac ind, simp)
    done
  have k3: zip ys ap = zip ys (take k ap)
    using zip-pre[of ys ap] k1 h
    apply(simp)
    done
  have k4: (zip [y] (drop (length ys) ap)) = [(y, ap ! length ys)]
    using k1 h
    apply(case-tac drop (length ys) ap, simp)
    apply(subgoal-tac hd (drop (length ys) ap) = ap ! length ys)
    apply(simp)
    apply(rule-tac hd-drop-conv-nth, simp)
    done
  from k1 and h k2 k3 k4 show start-of (layout-of ap) (Suc k) =
    Suc (listsum (map (length ∘ (λ(x, y). ci ly x y)) (zip ss ap)) div 2)
    apply(simp add: zip-append1 start-of.simps)
    apply(subgoal-tac
      listsum (map (length ∘ (λ(x, y). ci ly x y)) (zip ys (take k ap))) mod 2 = 0
∧
      length (ci ly y (ap!k)) mod 2 = 0)
    apply(auto)
    apply(rule-tac length-ci, simp, simp)
    done
qed

lemma length-start-of-tm: start-of (layout-of ap) (length ap) = Suc (length (tm-of
ap)  div 2)
```

apply(*simp add*: *tm-of.simps length-concat tms-of.simps tpairs-of.simps*)
apply(*rule-tac start-of-listsum*, *simp*, *simp*)
done

lemma *tm-even*: *length* (*tm-of ap*) *mod 2* = *0*
apply(*subgoal-tac t-ncorrect* (*tm-of ap*), *auto*)
apply(*simp add*: *t-ncorrect.simps*)
done

lemma [*elim*]: *list-all* (λ(*acn*, *s*). $s \leq$ *Suc q*) *xs*
$\Longrightarrow$ *list-all* (λ(*acn*, *s*). $s \leq q + (2 * n + 6)$) *xs*
apply(*simp add*: *list-all-length*)
apply(*auto*)
done

lemma [*simp*]: *length mp-up* = *12*
apply(*simp add*: *mp-up-def*)
done

lemma [*elim*]: ⟦$na < 4 * n$; *tshift* (*mop-bef n*) *q* ! *na* = (*a*, *b*)⟧ $\Longrightarrow b \leq q + (2 * n + 6)$
apply(*induct n*, *simp*, *simp add*: *mop-bef.simps nth-append tshift-append shift-length*)
apply(*case-tac na* $< 4*n$, *simp*, *simp*)
apply(*subgoal-tac na* $= 4*n \vee na = 1 + 4*n \vee na = 2 + 4*n \vee na = 3 + 4*n$,
auto simp: *shift-length*)
apply(*simp-all add*: *tshift.simps*)
done

lemma *mp-up-all-le*: *list-all* (λ(*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*, $6 + 2 * n + q$)]
apply(*auto*)
done

lemma [*intro*]: *list-all* (λ(*acn*, *s*). $s \leq q + (2 * n + 6)$) (*tMp n q*)
apply(*auto simp*: *list-all-length tMp.simps tshift-append nth-append shift-length*)
apply(*auto simp*: *tshift.simps mp-up-def*)
apply(*subgoal-tac na* $- 4*n \geq 0 \wedge na - 4*n < 12$, *auto split*: *nat.splits*)
apply(*insert mp-up-all-le*[*of q n*])
apply(*simp add*: *list-all-length*)
apply(*erule-tac x* = *na* $- 4 * n$ **in** *allE*, *simp add*: *numeral-3-eq-3*)
done

lemma *t-compiled-correct*:
⟦*tp* = *tm-of ap*; *ly* = *layout-of ap*; *mop-ss* = *start-of ly* (*length ap*)⟧ $\Longrightarrow$

t-correct (*tp* @ *tMp n* (*mop-ss* − *Suc 0*))
using *tm-even*[*of ap*] *length-start-of-tm*[*of ap*] *all-le-start-of*[*of ap*]
apply(*auto simp*: *t-correct.simps iseven-def*)
apply(*rule-tac x* = *q* + *2*∗*n* + *6* **in** *exI*, *simp*)
done

end

theory *UF*
imports *Main rec-def turing-basic 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.

11 Univeral Function

11.1 The construction of component functions

This section constructs a set of component functions used to construct *rec-F*.

The recursive function used to do arithmatic addition.

definition *rec-add* :: *recf*
where
rec-add ≡ *Pr 1* (*id 1 0*) (*Cn 3 s* [*id 3 2*])

The recursive function used to do arithmatic 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 arithmatic 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 arithmatic 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*]

Signal function, which returns 1 when the input argument is greater than *0*.

definition *rec-sg* :: *recf*
where
rec-sg = *Cn 1 rec-minus* [*constn 1*,
Cn 1 rec-minus [*constn 1*, *id 1 0*]]

rec-less compares its two arguments, returns *1* if the first is less than the second; otherwise returns *0*.

definition *rec-less* :: *recf*
where
rec-less = *Cn 2 rec-sg* [*Cn 2 rec-minus* [*id 2 1*, *id 2 0*]]

rec-not inverse its argument: returns *1* when the argument is *0*; returns *0* otherwise.

definition *rec-not* :: *recf*
where
rec-not = *Cn 1 rec-minus* [*constn 1*, *id 1 0*]

rec-eq compares its two arguments: returns *1* if they are equal; return *0* otherwise.

definition *rec-eq* :: *recf*
where
rec-eq = *Cn 2 rec-minus* [*Cn 2* (*constn 1*) [*id 2 0*],
Cn 2 rec-add [*Cn 2 rec-minus* [*id 2 0*, *id 2 1*],
Cn 2 rec-minus [*id 2 1*, *id 2 0*]]]

rec-conj computes the conjunction of its two arguments, returns *1* if both of them are non-zero; returns *0* otherwise.

definition *rec-conj* :: *recf*
where
rec-conj = *Cn 2 rec-sg* [*Cn 2 rec-mult* [*id 2 0*, *id 2 1*]]

rec-disj computes the disjunction of its two arguments, returns *0* if both of them are zero; returns *0* otherwise.

definition *rec-disj* :: *recf*
where
rec-disj = *Cn 2 rec-sg* [*Cn 2 rec-add* [*id 2 0*, *id 2 1*]]

Computes the arity of recursive function.

```
fun arity :: recf ⇒ nat
  where
  arity z = 1
| arity s = 1
| arity (id m n) = m
| arity (Cn n f gs) = n
| arity (Pr n f g) = Suc n
| arity (Mn n f) = n
```

get-fstn-args n (*Suc k*) returns [*id n 0*, *id n 1*, *id n 2*, . . . , *id n k*], the effect of which is to take out the first *Suc k* arguments out of the *n* input arguments.

```
fun get-fstn-args :: nat ⇒  nat ⇒ recf list
  where
  get-fstn-args n 0 = []
| get-fstn-args n (Suc y) = get-fstn-args n y @ [id n y]
```

rec-sigma f returns the recursive functions which sums up the results of *f*:

$$(rec\_sigma f)(x, y) = f(x, 0) + f(x, 1) + \cdots + f(x, y)$$

```
fun rec-sigma :: recf ⇒ recf
  where
  rec-sigma rf =
      (let vl = arity rf in
         Pr (vl − 1) (Cn (vl − 1) rf (get-fstn-args (vl − 1) (vl − 1) @
                  [Cn (vl − 1) (constn 0) [id (vl − 1) 0]]))
           (Cn (Suc vl) rec-add [id (Suc vl) vl,
                  Cn (Suc vl) rf (get-fstn-args (Suc vl) (vl − 1)
                     @ [Cn (Suc vl) s [id (Suc vl) (vl − 1)]])]))
```

rec-exec is the interpreter function for reursive functions. The function is defined such that it always returns meaningful results for primitive recursive functions.

```
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 =
            (let ys = (map (λ a. rec-exec a xs) gs) in
                                  rec-exec f ys) |
  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
proof
show *wf* (*measures* [λ (*r*, *xs*). *size r*, (λ (*r*, *xs*). *last xs*)])
by *auto*
next
fix *n f gs xs x*
assume (*x*::*recf*) ∈ *set gs*
thus ((*x*, *xs*), *Cn n f gs*, *xs*) ∈
measures [λ(*r*, *xs*). *size r*, λ(*r*, *xs*). *last xs*]
by(*induct gs*, *auto*)
next
fix *n f gs xs x*
assume *x* = *map* (λ*a*. *rec-exec a xs*) *gs*
⋀*x*. *x* ∈ *set gs* ⟹ *rec-exec-dom* (*x*, *xs*)
thus ((*f*, *x*), *Cn n f gs*, *xs*) ∈
measures [λ(*r*, *xs*). *size r*, λ(*r*, *xs*). *last xs*]
by(*auto*)
next
fix *n f g xs*
show ((*f*, *butlast xs*), *Pr n f g*, *xs*) ∈
measures [λ(*r*, *xs*). *size r*, λ(*r*, *xs*). *last xs*]
by *auto*
next
fix *n f g xs*
assume *last xs* ≠ (*0*::*nat*) **thus**
((*Pr n f g*, *butlast xs* @ [*last xs* − *1*]), *Pr n f g*, *xs*)
∈ *measures* [λ(*r*, *xs*). *size r*, λ(*r*, *xs*). *last xs*]
by *auto*
next
fix *n f g xs*
show ((*g*, *butlast xs* @ [*last xs* − *1*] @ [*rec-exec* (*Pr n f g*) (*butlast xs* @ [*last xs*
− *1*])]),
Pr n f g, *xs*) ∈ *measures* [λ(*r*, *xs*). *size r*, λ(*r*, *xs*). *last xs*]
by *auto*
next
fix *n f xs x*
show ((*f*, *xs* @ [*x*]), *Mn n f*, *xs*) ∈
measures [λ(*r*, *xs*). *size r*, λ(*r*, *xs*). *last xs*]
by *auto*
qed

declare *rec-exec.simps*[*simp del*] *constn.simps*[*simp del*]

Correctness of *rec-add*.

lemma *add-lemma*: ⋀ *x y*. *rec-exec rec-add* [*x*, *y*] = *x* + *y*
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*] = (*if x* = *0 then 0 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*] =
(*if x* < *y then 1 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*] = (*if x* = *0 then 1 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*] = (*if x* = *y then 1 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*] = (*if x* = *0* ∨ *y* = *0 then 0 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*: $\bigwedge$ *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 [*simp*]: *arity z* = *1*
 by(*simp add*: *arity.simps*)

lemma *rec-pr-0-simp-rewrite*:
rec-exec (*Pr n f g*) (*xs* @ [*0*]) = *rec-exec f xs*
by(*simp add*: *rec-exec.simps*)

lemma *rec-pr-0-simp-rewrite-single-param*:
rec-exec (*Pr n f g*) [*0*] = *rec-exec f* []
by(*simp add*: *rec-exec.simps*)

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 *rec-pr-Suc-simp-rewrite-single-param*:
rec-exec (*Pr n f g*) ([*Suc x*]) =
rec-exec g ([*x*] @ [*rec-exec* (*Pr n f g*) ([*x*])])
by(*simp add*: *rec-exec.simps*)

lemma *Sigma-0-simp-rewrite-single-param*:
Sigma f [*0*] = *f* [*0*]
by(*simp add*: *Sigma.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 *Sigma-Suc-simp-rewrite-single*:
Sigma f ([*Suc x*]) = *Sigma f* ([*x*]) + *f* ([*Suc x*])
by(*simp add*: *Sigma.simps*)

lemma [*simp*]: (*xs* @ *ys*) ! (*Suc* (*length xs*)) = *ys* ! *1*
by(*simp add*: *nth-append*)

lemma *get-fstn-args-take*: ⟦*length xs* = *m*; *n* ≤ *m*⟧ ⟹
map (λ *f*. *rec-exec f xs*) (*get-fstn-args m n*)= *take n xs*
proof(*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 [*simp*]: *primerec f n* $\Longrightarrow$ *arity f* = *n*
  apply(*case-tac 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*))
    $\Longrightarrow$ *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*, ..., *xn*, *k*) = *f*(*x1*, *x2*, ..., *xn*, *0*) ∗ *f*(*x1*, *x2*, ..., *xn*, *1*) ∗ ... *f*(*x1*, *x2*, ..., *xn*, *k*)

fun *rec-accum* :: *recf* ⇒ *recf*
  where
  rec-accum rf =
    (*let vl* = *arity rf in*
      Pr (*vl* − *1*) (*Cn* (*vl* − *1*) *rf* (*get-fstn-args* (*vl* − *1*) (*vl* − *1*) @
          [*Cn* (*vl* − *1*) (*constn 0*) [*id* (*vl* − *1*) *0*]]))
       (*Cn* (*Suc vl*) *rec-mult* [*id* (*Suc vl*) (*vl*),
          Cn (*Suc vl*) *rf* (*get-fstn-args* (*Suc vl*) (*vl* − *1*)
           @ [*Cn* (*Suc vl*) *s* [*id* (*Suc vl*) (*vl* − *1*)]])]))

Accum is the formal specification of *rec-accum*.

function *Accum* :: (*nat list* ⇒ *nat*) ⇒ *nat list* ⇒ *nat*
  where
  Accum f xs = (*if last xs* = *0 then f xs*
          else (*Accum f* (*butlast xs* @ [*last xs* − *1*]) ∗
           f xs))
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, ..., 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: primerec rf (Suc (length xs)) ⟹
     (rec-exec (rec-accum rf) (xs @ [x]) = 0) =
      (∃ t ≤ x. rec-exec rf (xs @ [t]) = 0)
apply(induct x, simp-all add: rec-accum-Suc-simp-rewrite)
apply(simp add: rec-exec.simps rec-accum.simps get-fstn-args-take,
      auto)
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
```

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(*case-tac rec-exec* (*rec-accum rf*) (*xs* @ [*rec-exec rt xs*]) = *0*, *simp-all*)
apply(*erule-tac exE*, *erule-tac x* = *t* **in** *allE*, *simp*)
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(*case-tac rec-exec* (*rec-accum rf*) (*xs* @ [*rec-exec rt xs*]) = *0*, *simp*, *simp*)
apply(*erule-tac x* = *x* **in** *allE*, *simp*)
done

rec-ex t f (*x1*, *x2*, ..., *xn*) computes the charactrization function of the following FOL formula: (∃ *x* ≤ *t*(*x1*, *x2*, ..., *xn*). (*f*(*x1*, *x2*, ..., *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*: *primerec rf* (*Suc* (*length xs*))
          ⟹ (*rec-exec* (*rec-sigma rf*) (*xs* @ [*x*]) = *0*) =
                          (∀ *t* ≤ *x*. *rec-exec rf* (*xs* @ [*t*]) = *0*)
apply(*induct x*, *simp-all add*: *rec-sigma-Suc-simp-rewrite*)
apply(*simp add*: *rec-exec.simps rec-sigma.simps*
              get-fstn-args-take, *auto*)
apply(*case-tac t* = *Suc x*, *simp-all*)
done

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-ex.simps rec-exec.simps map-append 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

Defintiion 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 [*simp*]: {*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*, *case-tac x* = *Suc w*, *auto*)

lemma [*simp*]: *Minr Rr xs w* ≤ *w* ⟹ *Minr Rr xs* (*Suc w*) = *Minr Rr xs w*
apply(*simp add*: *Minr.simps*, *auto*)
apply(*case-tac* ∀ *x*≤*w*. ¬ *Rr* (*xs* @ [*x*]), *auto*)
done

lemma [*simp*]: ∀ *x*≤*w*. ¬ *Rr* (*xs* @ [*x*]) ⟹
    {*x*. *x* ≤ *w* ∧ *Rr* (*xs* @ [*x*])} = {}
by *auto*

lemma [*simp*]: ⟦*Minr Rr xs w* = *Suc w*; *Rr* (*xs* @ [*Suc w*])⟧ ⟹
    Minr Rr xs (*Suc w*) = *Suc w*
apply(*simp add*: *Minr.simps*)
apply(*case-tac* ∀ *x*≤*w*. ¬ *Rr* (*xs* @ [*x*]), *auto*)
done

lemma [*simp*]: ⟦*Minr Rr xs w* = *Suc w*; ¬ *Rr* (*xs* @ [*Suc w*])⟧ ⟹
    Minr Rr xs (*Suc w*) = *Suc* (*Suc w*)
apply(*simp add*: *Minr.simps*)
apply(*case-tac* ∀ *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 ⟹ 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 ⟹
                   primerec rt n ∧ 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

lemma min-Suc-Suc[simp]: min (Suc (Suc x)) x = x
 by auto

declare numeral-3-eq-3[simp]
```

lemma [*intro*]: *primerec rec-pred* (*Suc 0*)
apply(*simp add*: *rec-pred-def*)
apply(*rule-tac prime-cn*, *auto*)
apply(*case-tac i*, *auto intro*: *prime-id*)
done

lemma [*intro*]: *primerec rec-minus* (*Suc* (*Suc 0*))
apply(*auto simp*: *rec-minus-def*)
done

lemma [*intro*]: *primerec* (*constn n*) (*Suc 0*)
apply(*induct n*)
apply(*auto simp*: *constn.simps intro*: *prime-z prime-cn prime-s*)
done

lemma [*intro*]: *primerec rec-sg* (*Suc 0*)
apply(*simp add*: *rec-sg-def*)
apply(*rule-tac k* = *Suc* (*Suc 0*) **in** *prime-cn*, *auto*)
apply(*case-tac i*, *auto*)
apply(*case-tac ia*, *auto intro*: *prime-id*)
done

lemma [*simp*]: *length* (*get-fstn-args m n*) = *n*
apply(*induct n*)
apply(*auto simp*: *get-fstn-args.simps*)
done

lemma *primerec-getpren*[*elim*]: ⟦*i* < *n*; *n* ≤ *m*⟧ ⟹ *primerec* (*get-fstn-args m n* ! *i*) *m*
apply(*induct n*, *auto simp*: *get-fstn-args.simps*)
apply(*case-tac i* = *n*, *auto simp*: *nth-append intro*: *prime-id*)
done

lemma [*intro*]: *primerec rec-add* (*Suc* (*Suc 0*))
apply(*simp add*: *rec-add-def*)
apply(*rule-tac prime-pr*, *auto*)
done

lemma [*intro*]:*primerec rec-mult* (*Suc* (*Suc 0*))
apply(*simp add*: *rec-mult-def*)
apply(*rule-tac prime-pr*, *auto intro*: *prime-z*)
apply(*case-tac i*, *auto intro*: *prime-id*)
done

lemma [*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*)

apply(*case-tac i*, *auto intro*: *prime-id*)
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 [*simp*]: *Rr* (*xs* @ [*0*]) ⟹
 Min {*x*. *x* = (*0*::*nat*) ∧ *Rr* (*xs* @ [*x*])} = *0*
by(*rule-tac Min-eqI*, *simp*, *simp*, *simp*)

lemma [*intro*]: *primerec rec-not* (*Suc 0*)
apply(*simp add*: *rec-not-def*)
apply(*rule prime-cn*, *auto*)
apply(*case-tac i*, *auto intro*: *prime-id*)
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 (λ*args*. *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(case-tac x = Suc w, simp, simp)
    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))⟧
     ⟹ 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*: *rec-exec.simps rec-Minr.simps 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*)

Defintiion 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 rq = rec-all rt rf  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]
lemma [simp]: (min (Suc (Suc (Suc (x)))) (x)) = x
by simp

declare numeral-2-eq-2[simp]

lemma [intro]: primerec rec-disj (Suc (Suc 0))
  apply(simp add: rec-disj-def, auto)
  apply(auto)
  apply(case-tac ia, auto intro: prime-id)
  done

lemma [intro]: primerec rec-less (Suc (Suc 0))
  apply(simp add: rec-less-def, auto)
  apply(auto)
  apply(case-tac ia , auto intro: prime-id)
  done

lemma [intro]: primerec rec-eq (Suc (Suc 0))
  apply(simp add: rec-eq-def)
  apply(rule-tac prime-cn, auto)
  apply(case-tac i, auto)
  apply(case-tac ia, auto)
  apply(case-tac [!] i, auto intro: prime-id)
  done

lemma [intro]: primerec rec-le (Suc (Suc 0))
  apply(simp add: rec-le-def)
  apply(rule-tac prime-cn, auto)
  apply(case-tac i, auto)
  done

lemma [simp]:
  length ys = Suc n ⟹ (take n ys @ [ys ! n, ys ! n]) =
                                  ys @ [ys ! n]
apply(simp)
apply(subgoal-tac ∃ xs y. ys = xs @ [y], auto)
apply(rule-tac x = butlast ys in exI, rule-tac x = last ys in exI)
apply(case-tac ys = [], simp-all)
```

done

lemma *Maxr-Suc-simp*:
Maxr Rr xs (*Suc w*) =(*if Rr* (*xs* @ [*Suc w*]) *then Suc w*
else Maxr Rr xs w)
apply(*auto simp*: *Maxr.simps*)
apply(*rule-tac max-absorb1*)
apply(*subgoal-tac* (*Max* {*y*. *y* ≤ *w* ∧ *Rr* (*xs* @ [*y*])} ≤ (*Suc w*)) =
(∀ *a*∈{*y*. *y* ≤ *w* ∧ *Rr* (*xs* @ [*y*])}. *a* ≤ (*Suc w*)), *simp*)
apply(*rule-tac Max-le-iff*, *auto*)
done

lemma [*simp*]: *min* (*Suc n*) *n* = *n* **by** *simp*

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 [*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*)
apply(*case-tac ma* = *Suc w*, *auto*)
done

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*)
apply(*case-tac i*, *auto*)
apply(*case-tac ia*, *auto simp*: *h nth-append*)
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)
      using prrf
      apply(case-tac i, auto)
      apply(case-tac ia, auto simp: h nth-append)
      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)
      using prrf
      apply(case-tac i, auto)
      apply(case-tac ia, 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*)
apply(*case-tac i*, *auto*)
apply(*case-tac ia*, *auto intro*: *prime-id*)
apply(*simp add*: *h*)
apply(*simp add*: *nth-append*, *auto intro*: *prime-id*)
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* (λ*args*. *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 (λ*args*. *0* < *rec-exec rf args*) *xs w*
apply(*simp*)
done
qed
qed

quo is the formal specification of division.

fun *quo* :: *nat list* ⇒ *nat*
where
quo [*x*, *y*] = (*let Rr* =
(λ *zs*. ((*zs* ! (*Suc 0*) ∗ *zs* ! (*Suc* (*Suc 0*))
≤ *zs* ! *0*) ∧ *zs* ! *Suc 0* ≠ (*0*::*nat*)))
in Maxr Rr [*x*, *y*] *x*)

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

The following lemmas shows more directly the menaing of *quo*:

lemma [*elim*!]: *y* > *0* ⟹ *quo* [*x*, *y*] = *x div y*
proof(*simp add*: *quo.simps Maxr.simps*, *auto*,

```
    rule-tac Max-eqI, simp, auto)
  fix xa ya
  assume h: y * ya ≤ x  y > 0
  hence (y * ya) div y ≤ x div y
    by(insert div-le-mono[of y * ya x y], simp)
  from this and h show ya ≤ x div y by simp
next
  fix xa
  show y * (x div y) ≤ x
    apply(subgoal-tac y * (x div y) + x mod y = x)
    apply(rule-tac k = x mod y in add-leD1, simp)
    apply(simp)
    done
qed

lemma [intro]: quo [x, 0] = 0
by(simp add: quo.simps Maxr.simps)

lemma quo-div: quo [x, y] = x div y
by(case-tac y=0, auto)
```

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:
  ⋀ x y. rec-exec rec-noteq [x, y] =
              (if x ≠ 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 [*intro*]: *primerec rec-conj* (*Suc* (*Suc 0*))
  apply(*simp add*: *rec-conj-def*)
  apply(*rule-tac prime-cn*, *auto*)+
  apply(*case-tac i*, *auto intro*: *prime-id*)
  done

lemma [*intro*]: *primerec rec-noteq* (*Suc* (*Suc 0*))
apply(*simp add*: *rec-noteq-def*)
apply(*rule-tac prime-cn*, *auto*)+
apply(*case-tac i*, *auto intro*: *prime-id*)
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*)
      apply(*case-tac i*, *auto*)
      apply(*case-tac* [!] *ia*, *auto*)
      apply(*case-tac i*, *auto*)
      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)*
proof(*simp add*: *rec-exec.simps rec-mod-def quo-lemma2*)
fix *x y*
show *x − x div y ∗ y = x mod (y::nat)*
using *mod-div-equality2[of y x]*
apply(*subgoal-tac y ∗ (x div y) = (x div y) ∗ y, arith, simp*)
done
qed

lemmas for embranch function

type-synonym *ftype = nat list ⇒ nat*
type-synonym *rtype = nat list ⇒ bool*

The specfation 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, simp, case-tac 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(case-tac rcs = [], simp)
    show rec-exec (rec-embranch (zip rgs [a])) xs = 0
      using h
      apply(simp add: rec-embranch.simps rec-exec.simps)
      apply(erule-tac x = 0 in allE, simp)
      done
  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
      apply(simp add: rec-embranch.simps rec-exec.simps)
      apply(case-tac rcs,
        auto simp: rec-exec.simps rec-embranch.simps)
      apply(case-tac list,
        auto simp: rec-exec.simps rec-embranch.simps)
      done
  qed
qed
```

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(*simp add*: *rec-exec.simps rec-embranch.simps*)
apply(*case-tac zip rgs list*, *simp*, *case-tac ab*, *simp add*: *rec-embranch.simps rec-exec.simps*)
apply(*subgoal-tac arity a* = *length xs*, *auto*)
apply(*subgoal-tac arity aaa* = *length xs*, *auto*)
apply(*case-tac rgs*, *simp*, *case-tac list*, *simp*, *simp*)
done

lemma *zip-null-iff*: ⟦*length xs* = *k*; *length ys* = *k*; *zip xs ys* = []⟧ ⟹ *xs* = [] ∧ *ys* = []
apply(*case-tac xs*, *simp*, *simp*)
apply(*case-tac ys*, *simp*, *simp*)
done

lemma *zip-null-gr*: ⟦*length xs* = *k*; *length ys* = *k*; *zip xs ys* ≠ []⟧ ⟹ *0* < *k*
apply(*case-tac xs*, *simp*, *simp*)
done

lemma *Embranch-0*:
⟦*length rgs* = *k*; *length rcs* = *k*; *k* > *0*;
∀ *j* < *k*. *rec-exec* (*rcs* ! *j*) *xs* = *0*⟧ ⟹
Embranch (*zip* (*map rec-exec rgs*) (*map* (λ*r args*. *0* < *rec-exec r args*) *rcs*)) *xs* = *0*
proof(*induct rgs arbitrary*: *rcs k*, *simp*, *simp*)
fix *a rgs rcs k*
assume *ind*:
⋀*rcs k*. ⟦*length rgs* = *k*; *length rcs* = *k*; *0* < *k*; ∀*j*<*k*. *rec-exec* (*rcs* ! *j*) *xs* = *0*⟧
⟹ *Embranch* (*zip* (*map rec-exec rgs*) (*map* (λ*r args*. *0* < *rec-exec r args*) *rcs*)) *xs* = *0*
and *h*: *Suc* (*length rgs*) = *k length rcs* = *k*
∀*j*<*k*. *rec-exec* (*rcs* ! *j*) *xs* = *0*
from *h* **show**
Embranch (*zip* (*rec-exec a* # *map rec-exec rgs*)
(*map* (λ*r args*. *0* < *rec-exec r args*) *rcs*)) *xs* = *0*
apply(*case-tac rcs*, *simp*, *case-tac rgs* = [], *simp*)
apply(*simp add*: *Embranch.simps*)
apply(*erule-tac x* = *0* **in** *allE*, *simp*)
apply(*simp add*: *Embranch.simps*)
apply(*erule-tac x* = *0* **in** *all-dupE*, *simp*)
apply(*rule-tac ind*, *simp*, *simp*, *simp*, *auto*)
apply(*erule-tac x* = *Suc j* **in** *allE*, *simp*)
done

qed

The correctness of *rec-embranch*.

lemma *embranch-lemma*:
  assumes *branch-num*:
  length rgs = n length rcs = n n > 0
  and *partition*:
  (∃ i < n. (rec-exec (rcs ! i) xs = 1 ∧ (∀ j < n. j ≠ i ⟶
                                                    rec-exec (rcs ! j) xs = 0)))
  and *prime-all*: *list-all (λ rf. primerec rf (length xs)) (rgs @ rcs)*
  shows *rec-exec (rec-embranch (zip rgs rcs)) xs =*
                  Embranch (zip (map rec-exec rgs)
                     (map (λ r args. 0 < rec-exec r args) rcs)) xs
  using *branch-num partition prime-all*
proof(*induct rgs arbitrary*: *rcs n*, *simp*)
  fix *a rgs rcs n*
  assume *ind*:
    ⋀rcs n. ⟦length rgs = n; length rcs = n; 0 < n;
    ∃ i<n. rec-exec (rcs ! i) xs = 1 ∧ (∀ j<n. j ≠ i ⟶ rec-exec (rcs ! j) xs = 0);
    list-all (λrf. primerec rf (length xs)) (rgs @ rcs)⟧
    ⟹ rec-exec (rec-embranch (zip rgs rcs)) xs =
    Embranch (zip (map rec-exec rgs) (map (λr args. 0 < rec-exec r args) rcs)) xs
  and *h*: *length (a # rgs) = n length (rcs::recf list) = n 0 < n*
         ∃ i<n. rec-exec (rcs ! i) xs = 1 ∧
        (∀ j<n. j ≠ i ⟶ rec-exec (rcs ! j) xs = 0)
    list-all (λrf. 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 (λr args.
              0 < rec-exec r args) rcs)) xs
    apply(*case-tac rcs*, *simp*, *simp*)
    apply(*case-tac rec-exec aa xs = 0*)
    apply(*case-tac* [!] *zip rgs list = []*, *simp*)
   apply(*subgoal-tac rgs = [] ∧ list = []*, *simp add*: *Embranch.simps rec-exec.simps*
rec-embranch.simps)
    apply(*rule-tac zip-null-iff*, *simp*, *simp*, *simp*)
  proof −
    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
      rec-exec aa xs = 0 rcs = aa # list zip rgs list ≠ []
    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*, λ*args*. *0* < *rec-exec aa args*) #
zip (*map rec-exec rgs*) (*map* (λ*r args*. *0* < *rec-exec r args*) *list*)) *xs*
apply(*simp add*: *Embranch.simps*)
apply(*rule-tac n* = *n* − *Suc 0* **in** *ind*)
apply(*case-tac n*, *simp*, *simp*)
apply(*case-tac n*, *simp*, *simp*)
apply(*case-tac n*, *simp*, *simp add*: *zip-null-gr*)
apply(*auto*)
apply(*case-tac i*, *simp*, *simp*)
apply(*rule-tac x* = *nat* **in** *exI*, *simp*)
apply(*rule-tac allI*, *erule-tac x* = *Suc j* **in** *allE*, *simp*)
done
next
fix *aa list*
assume *g*: *Suc* (*length rgs*) = *n Suc* (*length list*) = *n*
$\exists$ *i*<*n*. *rec-exec* ((*aa* # *list*) ! *i*) *xs* = *Suc 0* $\wedge$
($\forall$ *j*<*n*. *j* $\neq$ *i* $\longrightarrow$ *rec-exec* ((*aa* # *list*) ! *j*) *xs* = *0*)
primerec a (*length xs*) $\wedge$ *list-all* (λ*rf*. *primerec rf* (*length xs*)) *rgs* $\wedge$
primerec aa (*length xs*) $\wedge$ *list-all* (λ*rf*. *primerec rf* (*length xs*)) *list*
rcs = *aa* # *list rec-exec aa xs* $\neq$ *0 zip rgs list* = []
thus *rec-exec* (*rec-embranch* ((*a*, *aa*) # *zip rgs list*)) *xs* =
Embranch ((*rec-exec a*, λ*args*. *0* < *rec-exec aa args*) #
zip (*map rec-exec rgs*) (*map* (λ*r args*. *0* < *rec-exec r args*) *list*)) *xs*
apply(*subgoal-tac rgs* = [] $\wedge$ *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*
$\exists$ *i*<*n*. *rec-exec* ((*aa* # *list*) ! *i*) *xs* = *Suc 0* $\wedge$
($\forall$ *j*<*n*. *j* $\neq$ *i* $\longrightarrow$ *rec-exec* ((*aa* # *list*) ! *j*) *xs* = *0*)
primerec a (*length xs*) $\wedge$ *list-all* (λ*rf*. *primerec rf* (*length xs*)) *rgs*
$\wedge$ *primerec aa* (*length xs*) $\wedge$ *list-all* (λ*rf*. *primerec rf* (*length xs*)) *list*
rcs = *aa* # *list rec-exec aa xs* $\neq$ *0 zip rgs list* $\neq$ []
have *rec-exec aa xs* = *Suc 0*
using *g*
apply(*case-tac 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(*case-tac zip rgs list*, *simp*, *case-tac ab*)
apply(*simp add*: *rec-embranch.simps*)
apply(*subgoal-tac arity aaa* = *length xs*, *simp*, *auto*)

```
    apply(case-tac rgs, simp, simp, case-tac list, simp, simp)
    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(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
      apply(case-tac n, simp, simp)
      done
  next
    show list ≠ []
      using g
      apply(case-tac list, simp, simp)
      done
  next
    show list-all (λrf. primerec rf (length xs)) (rgs @ list)
      using g
      apply auto
      done
  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, case-tac n, simp, simp)
  apply(case-tac rgs, simp, simp)
  apply(auto)
  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*)
apply(*case-tac* [!] *i*, *auto*)
apply(*case-tac ia*, *auto simp*: *numeral-3-eq-3 numeral-2-eq-2*)
done

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)
      apply(case-tac [!] i, auto simp: numeral-2-eq-2)
      apply(case-tac ia, auto simp: numeral-3-eq-3)
      done
  next
    show primerec (Cn (Suc 0) rec-minus
             [recf.id (Suc 0) 0, constn (Suc 0)]) (Suc 0)
      apply(auto)
      apply(case-tac i, auto)
      done
  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(erule-tac x = u in allE, auto)
      apply(case-tac u, simp, case-tac nat, simp, simp)
      apply(case-tac v, simp, case-tac nat, 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 [*simp*]: *rec-exec rec-dummyfac* [*0*, *0*] = *Suc 0*
by(*simp add*: *rec-dummyfac-def rec-exec.simps*)

lemma *rec-cn-simp*: *rec-exec* (*Cn n f gs*) *xs* =
(*let rgs* = *map* (λ *g*. *rec-exec g xs*) *gs in*
rec-exec f rgs)
by(*simp add*: *rec-exec.simps*)

lemma *rec-id-simp*: *rec-exec* (*id m n*) *xs* = *xs* ! *n*
by(*simp add*: *rec-exec.simps*)

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 [*simp*]: *n* < *Suc* (*n*!)
apply(*induct n*, *simp*)
apply(*simp add*: *fac.simps*)
apply(*case-tac n*, *auto simp*: *fac.simps*)
done

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(*erule-tac x* = *u* **in** *allE*, *simp*)
apply(*case-tac Prime u*, *simp*)
apply(*rule-tac x* = *u* **in** *exI*, *simp*, *auto*)
done

lemma [*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*!
apply(*induct n*, *simp*)
apply(*case-tac q* ≤ *n*, *simp add*: *fac-Suc*)

```
apply(subgoal-tac q = Suc n, simp only: fac-Suc)
apply(rule-tac dvd-mult2, simp, simp)
done

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
      apply(rule-tac x = q in exI, auto)
      apply(rule-tac dvd-imp-le, auto)
      done
  next
    case False thus ?thesis
    proof −
      assume g: ¬ n < q
      have j: q > Suc 0
        using k by(case-tac 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(case-tac i, auto)

lemma [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)
    apply(erule-tac x = p in allE, simp add: prime-lemma)
    apply(simp add: prime-lemma split: if-splits)
    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] = xˆy
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* (yˆu) = *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 logarithmatic operation on that page. The other is *lg*.

fun *lo* :: *nat* ⇒ *nat* ⇒ *nat*
where
lo x y = (*if* $x > 1 \wedge y > 1 \wedge$ {*u*. *loR* [x, y, u]} ≠ {} *then Max* {*u*. *loR* [x, y, u]}
else 0)

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

lemma [*elim*]: *primerec rf n* ⟹ $n > 0$
apply(*induct rule*: *primerec.induct*, *auto*)
done

```
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 [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(case-tac i, auto, case-tac nat, simp, simp add: numeral-2-eq-2)
done

lemma [intro]: primerec rec-quo (Suc (Suc 0))
apply(simp add: rec-quo-def)
apply(tactic ⟨⟨ resolve-tac [@{thm prime-cn},
    @{thm prime-id}] 1⟩⟩, auto+)+
done

lemma [intro]: primerec rec-mod (Suc (Suc 0))
apply(simp add: rec-mod-def)
apply(tactic ⟨⟨ resolve-tac [@{thm prime-cn},
    @{thm prime-id}] 1⟩⟩, auto+)+
done

lemma [intro]: primerec rec-power (Suc (Suc 0))
apply(simp add: rec-power-def  numeral-2-eq-2 numeral-3-eq-3)
apply(tactic ⟨⟨ resolve-tac [@{thm prime-cn},
    @{thm prime-id}, @{thm prime-pr}] 1⟩⟩, auto+)+
done
```

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 [simp]: Max {ya. ya = 0 ∧ loR [0, y, ya]} = 0
apply(rule-tac Max-eqI, auto simp: loR.simps)
done

lemma [simp]: Suc 0 < y ⟹ Suc (Suc 0) < y * y
apply(induct y, simp)
apply(case-tac y, simp, simp)
done

lemma less-mult: ⟦x > 0; y > Suc 0⟧ ⟹ x < y * x
apply(case-tac y, simp, simp)
done

lemma x-less-exp: ⟦y > Suc 0⟧ ⟹ x < yˆx
apply(induct x, simp, simp)
apply(case-tac x, simp, auto)
apply(rule-tac y = y* yˆnat in le-less-trans, simp)
apply(rule-tac less-mult, auto)
done
```

lemma *le-mult*: $y \neq (0::nat) \Longrightarrow x \leq x * y$
by(*induct y*, *simp*, *simp*)

lemma *uplimit-loR*: $[\![Suc\ 0 < x;\ Suc\ 0 < y;\ loR\ [x, y, xa]]\!] \Longrightarrow$
$xa \leq x$
apply(*simp add*: *loR.simps*)
apply(*rule-tac classical*, *auto*)
apply(*subgoal-tac* $xa < y\hat{\ }xa$)
apply(*subgoal-tac* $y\hat{\ }xa \leq y\hat{\ }xa * q$, *simp*)
apply(*rule-tac le-mult*, *case-tac q*, *simp*, *simp*)
apply(*rule-tac x-less-exp*, *simp*)
done

lemma [*simp*]: $[\![xa \leq x;\ loR\ [x, y, xa];\ Suc\ 0 < x;\ Suc\ 0 < y]\!] \Longrightarrow$
$\{u.\ loR\ [x, y, u]\} = \{ya.\ ya \leq x \wedge loR\ [x, y, ya]\}$
apply(*rule-tac Collect-cong*, *auto*)
apply(*erule-tac uplimit-loR*, *simp*, *simp*)
done

lemma *Maxr-lo*: $[\![Suc\ 0 < x;\ Suc\ 0 < y]\!] \Longrightarrow$
Maxr loR $[x, y]\ x = lo\ x\ y$
apply(*simp add*: *Maxr.simps lo.simps*, *auto*)
apply(*erule-tac* $x = xa$ **in** *allE*, *simp*, *simp add*: *uplimit-loR*)
done

lemma *lo-lemma'*: $[\![Suc\ 0 < x;\ Suc\ 0 < y]\!] \Longrightarrow$
rec-exec rec-lo $[x, y] = lo\ x\ y$
by(*simp add*: *Maxr-lo rec-lo-Maxr-lor*)

lemma *lo-lemma''*: $[\![\neg\ Suc\ 0 < x]\!] \Longrightarrow$ *rec-exec rec-lo* $[x, y] = lo\ x\ y$
apply(*case-tac x*, *auto simp*: *rec-exec.simps rec-lo-def*
Let-def lo.simps)
done

lemma *lo-lemma'''*: $[\![\neg\ Suc\ 0 < y]\!] \Longrightarrow$ *rec-exec rec-lo* $[x, y] = lo\ x\ y$
apply(*case-tac 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(*case-tac* $Suc\ 0 < x \wedge Suc\ 0 < y$)
apply(*auto simp*: *lo-lemma' lo-lemma'' lo-lemma'''*)
done

fun *lgR* :: *nat list* $\Rightarrow$ *bool*
where
lgR $[x, y, u] = (y\hat{\ }u \leq x)$

lg specifies the *lg* function given on page 79 of Boolos's book. It is one of the two notions of integeral logarithmatic 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 ((λ 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 [*simp*]: $[\![Suc\ 0 < y;\ lgR\ [x, y, xa]]\!] \Longrightarrow xa \leq x$
apply(*simp add*: *lgR.simps*)
apply(*subgoal-tac* $y \hat{} xa > xa$, *simp*)
apply(*erule x-less-exp*)
done

lemma [*simp*]: $[\![Suc\ 0 < x;\ Suc\ 0 < y;\ lgR\ [x, y, xa]]\!] \Longrightarrow$
$\{u.\ lgR\ [x, y, u]\} = \{ya.\ ya \leq x \wedge lgR\ [x, y, ya]\}$
apply(*rule-tac Collect-cong*, *auto*)
done

lemma *maxr-lg*: $[\![Suc\ 0 < x;\ Suc\ 0 < y]\!] \Longrightarrow Maxr\ lgR\ [x, y]\ x = lg\ x\ y$
apply(*simp add*: *lg.simps Maxr.simps*, *auto*)
apply(*erule-tac* $x = xa$ **in** *allE*, *simp*)
done

lemma *lg-lemma'*: $[\![Suc\ 0 < x;\ Suc\ 0 < y]\!] \Longrightarrow rec\text{-}exec\ rec\text{-}lg\ [x, y] = lg\ x\ y$
apply(*simp add*: *maxr-lg lg-maxr*)
done

lemma *lg-lemma''*: $\neg\ Suc\ 0 < x \Longrightarrow rec\text{-}exec\ rec\text{-}lg\ [x, y] = lg\ x\ y$
apply(*simp add*: *rec-exec.simps rec-lg-def Let-def lg.simps*)
done

lemma *lg-lemma'''*: $\neg\ Suc\ 0 < y \Longrightarrow rec\text{-}exec\ rec\text{-}lg\ [x, y] = lg\ x\ y$
apply(*simp add*: *rec-exec.simps rec-lg-def Let-def lg.simps*)
done

The correctness of *rec-lg*.

lemma *lg-lemma*: $rec\text{-}exec\ rec\text{-}lg\ [x, y] = lg\ x\ y$
apply(*case-tac* $Suc\ 0 < x \wedge Suc\ 0 < y$, *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* :: $nat \Rightarrow nat \Rightarrow nat$
where
$Entry\ sr\ i = lo\ sr\ (Pi\ (Suc\ i))$

rec-entry is the recursive function used to implement *Entry*.

definition *rec-entry*:: *recf*
where
$rec\text{-}entry = Cn\ 2\ rec\text{-}lo\ [id\ 2\ 0,\ Cn\ 2\ rec\text{-}pi\ [Cn\ 2\ s\ [id\ 2\ 1]]]$

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

The correctness of *rec-entry*.

lemma *entry-lemma*: $rec\text{-}exec\ rec\text{-}entry\ [str, i] = Entry\ str\ i$

by(*simp add*: *rec-entry-def rec-exec.simps lo-lemma pi-lemma*)

11.2 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^(xs ! n + dbound) − 2^dbound))

fun rec-strt' :: nat ⇒ nat ⇒ recf
  where
  rec-strt' vl 0 = Cn vl z [id vl 0]
| rec-strt' vl (Suc n) = (let rec-dbound =
  Cn vl rec-add [rec-listsum2 vl n, Cn vl (constn n) [id vl 0]]
  in Cn vl rec-add [rec-strt' vl n, Cn vl rec-minus
  [Cn vl rec-power [Cn vl (constn 2) [id vl 0], Cn vl rec-add
  [id vl (n), rec-dbound]],
  Cn vl rec-power [Cn vl (constn 2) [id vl 0], rec-dbound]]])

declare strt'.simps[simp del] rec-strt'.simps[simp del]

lemma strt'-lemma: ⟦length xs = vl; n ≤ vl⟧ ⟹
  rec-exec (rec-strt' vl n) xs = strt' xs n
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 ((λ*a*. *rec-exec a xs*) ∘ (λ*i*. *Cn vl s* [*recf.id vl i*]))
  [*0*..<*vl*]
    = *map Suc xs*
apply(*induct vl arbitrary*: *xs*, *simp*, *auto simp*: *rec-exec.simps*)
apply(*subgoal-tac* ∃ *ys y*. *xs* = *ys* @ [*y*], *auto*)
proof −
  fix *ys y*
  assume *ind*: ⋀*xs*. *length xs* = *length* (*ys*::*nat list*) ⟹
    map ((λ*a*. *rec-exec a xs*) ∘ (λ*i*. *Cn* (*length ys*) *s*
     [*recf.id* (*length ys*) (*i*)])) [*0*..<*length ys*] = *map Suc xs*
  show
   map ((λ*a*. *rec-exec a* (*ys* @ [*y*])) ∘ (λ*i*. *Cn* (*Suc* (*length ys*)) *s*
  [*recf.id* (*Suc* (*length ys*)) (*i*)])) [*0*..<*length ys*] = *map Suc ys*
  proof −
   have *map* ((λ*a*. *rec-exec a ys*) ∘ (λ*i*. *Cn* (*length ys*) *s*
     [*recf.id* (*length ys*) (*i*)])) [*0*..<*length ys*] = *map Suc ys*
    apply(*rule-tac ind*, *simp*)
    done
   moreover have
    map ((λ*a*. *rec-exec a* (*ys* @ [*y*])) ∘ (λ*i*. *Cn* (*Suc* (*length ys*)) *s*
      [*recf.id* (*Suc* (*length ys*)) (*i*)])) [*0*..<*length ys*]
     = *map* ((λ*a*. *rec-exec a ys*) ∘ (λ*i*. *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* $\neq$ [], *auto*)
done
qed

The correctness of *rec-strt*.

lemma *strt-lemma*: *length xs = vl* $\Longrightarrow$
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*) $\circ$ (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* $\Rightarrow$ *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* $\Rightarrow$ *nat*
where
newleft0 [*p*, *r*] = *p*

definition *rec-newleft0* :: *recf*
where
rec-newleft0 = id 2 0

fun *newrgt0* :: *nat list* $\Rightarrow$ *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* $\Rightarrow$ *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(*case-tac i*, *simp*, *case-tac nat*, *simp*,
case-tac nata, *simp*, *case-tac natb*, *simp*, *simp*)
done

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*]]
thm *embranch-lemma*
have *k1*: *rec-exec* (*rec-embranch* (*zip ?rgs ?rrs*)) [*p*, *r*, *a*]
= *Embranch* (*zip* (*map rec-exec ?rgs*) (*map* (λ*r 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(*case-tac a* = *0* ∨ *a* = *1*, *rule-tac x* = *0* **in** *exI*)
prefer *2*
apply(*case-tac a* = *2*, *rule-tac x* = *Suc 0* **in** *exI*)
prefer *2*
apply(*case-tac a* = *3*, *rule-tac x* = *2* **in** *exI*)
prefer *2*
apply(*case-tac 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* (λ*r 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 ⟹ P 0;*
i = 1 ⟹ P 1; i = 2 ⟹ P 2; i = 3 ⟹ P 3; i = 4 ⟹ P 4⟧ ⟹ *P i*
apply(*case-tac i, auto*)
apply(*case-tac nat, auto*)
apply(*case-tac nata, auto simp: numeral-2-eq-2*)
apply(*case-tac nat, auto simp: numeral-3-eq-3 numeral-4-eq-4*)
done

lemma [*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(case-tac a = 0, rule-tac x = 0 in exI)
    prefer 2
    apply(case-tac a = 1, rule-tac x = Suc 0 in exI)
    prefer 2
    apply(case-tac a = 2, rule-tac x = 2 in exI)
    prefer 2
    apply(case-tac a = 3, rule-tac x = 3 in exI)
    prefer 2
    apply(case-tac 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)

thm Prime.simps

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
apply(induct vl arbitrary: xs, simp)
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(case-tac xs ≠ [], auto)
    done
qed

lemma [elim]:
  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, case-tac 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 stat-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
apply(induct t)
apply(simp add: rec-conf-def rec-exec.simps conf.simps inpt-lemma trpl-lemma)
apply(simp add: conf-step conf.simps)
done
```

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 eq-lemma)
apply(auto)
apply(case-tac 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*]

lemma *primerec-not0*: *primerec f n* ⟹ *n* > *0*
by(*induct f n rule*: *primerec.induct*, *auto*)

lemma [*elim*]: *primerec f 0* ⟹ *RR*
apply(*drule-tac primerec-not0*, *simp*)
done

lemma [*simp*]: *length xs* = *Suc n* ⟹ *length* (*butlast xs*) = *n*
apply(*subgoal-tac* ∃ *y ys*. *xs* = *ys* @ [*y*], *auto*)
apply(*rule-tac x* = *last xs* **in** *exI*)
apply(*rule-tac x* = *butlast xs* **in** *exI*)

```
apply(case-tac xs = [], auto)
done
```

The lemma relates the interpreter of primitive fucntions with the calculation relation of general recursive functions.

```
lemma prime-rel-exec-eq: primerec r (length xs)
        ⟹ rec-calc-rel r xs rs = (rec-exec r xs = rs)
proof(induct r xs arbitrary: rs rule: rec-exec.induct, simp-all)
  fix xs rs
  assume primerec z (length (xs::nat list))
  hence length xs = Suc 0 by(erule-tac prime-z-reverse, simp)
  thus rec-calc-rel z xs rs = (rec-exec z xs = rs)
    apply(case-tac xs, simp, auto)
    apply(erule-tac calc-z-reverse, simp add: rec-exec.simps)
    apply(simp add: rec-exec.simps, rule-tac calc-z)
    done
next
  fix xs rs
  assume primerec s (length (xs::nat list))
  hence length xs = Suc 0 ..
  thus rec-calc-rel s xs rs = (rec-exec s xs = rs)
    by(case-tac xs, auto simp: rec-exec.simps intro: calc-s
                    elim: calc-s-reverse)
next
  fix m n xs rs
  assume primerec (recf.id m n) (length (xs::nat list))
  thus
    rec-calc-rel (recf.id m n) xs rs =
              (rec-exec (recf.id m n) xs = rs)
    apply(erule-tac prime-id-reverse)
    apply(simp add: rec-exec.simps, auto)
    apply(erule-tac calc-id-reverse, simp)
    apply(rule-tac calc-id, auto)
    done
next
  fix n f gs xs rs
  assume ind1:
    ⋀x rs. ⟦x ∈ set gs; primerec x (length xs)⟧ ⟹
            rec-calc-rel x xs rs = (rec-exec x xs = rs)
    and ind2:
    ⋀x rs. ⟦x = map (λa. rec-exec a xs) gs;
            primerec f (length gs)⟧ ⟹
           rec-calc-rel f (map (λa. rec-exec a xs) gs) rs =
          (rec-exec f (map (λa. rec-exec a xs) gs) = rs)
    and h: primerec (Cn n f gs) (length xs)
  show rec-calc-rel (Cn n f gs) xs rs =
              (rec-exec (Cn n f gs) xs = rs)
  proof(auto simp: rec-exec.simps, erule-tac calc-cn-reverse, auto)
    fix ys
```

```
    assume g1:∀ k<length gs. rec-calc-rel (gs ! k) xs (ys ! k)
      and g2: length ys = length gs
      and g3: rec-calc-rel f ys rs
    have rec-calc-rel f (map (λa. rec-exec a xs) gs) rs =
              (rec-exec f (map (λa. rec-exec a xs) gs) = rs)
      apply(rule-tac ind2, auto)
      using h
      apply(erule-tac prime-cn-reverse, simp)
      done
    moreover have ys = (map (λa. rec-exec a xs) gs)
    proof(rule-tac nth-equalityI, auto simp: g2)
      fix i
      assume i < length gs thus ys ! i = rec-exec (gs!i) xs
        using ind1[of gs ! i ys ! i] g1 h
        apply(erule-tac prime-cn-reverse, simp)
        done
    qed
    ultimately show rec-exec f (map (λa. rec-exec a xs) gs) = rs
      using g3
      by(simp)
  next
    from h show
      rec-calc-rel (Cn n f gs) xs
             (rec-exec f (map (λa. rec-exec a xs) gs))
      apply(rule-tac rs = (map (λa. rec-exec a xs) gs) in calc-cn,
           auto)
      apply(erule-tac [!] prime-cn-reverse, auto)
    proof −
      fix k
      assume k < length gs primerec f (length gs)
           ∀ i<length gs. primerec (gs ! i) (length xs)
      thus rec-calc-rel (gs ! k) xs (rec-exec (gs ! k) xs)
        using ind1[of gs!k (rec-exec (gs ! k) xs)]
        by(simp)
    next
      assume primerec f (length gs)
           ∀ i<length gs. primerec (gs ! i) (length xs)
      thus rec-calc-rel f (map (λa. rec-exec a xs) gs)
        (rec-exec f (map (λa. rec-exec a xs) gs))
        using ind2[of (map (λa. rec-exec a xs) gs)
                 (rec-exec f (map (λa. rec-exec a xs) gs))]
        by simp
    qed
  qed
next
  fix n f g xs rs
  assume ind1:
    ⋀rs. ⟦last xs = 0; primerec f (length xs − Suc 0)⟧
    ⟹ rec-calc-rel f (butlast xs) rs =
```

```
                  (rec-exec f (butlast xs) = rs)
and ind2 :
  ⋀rs. ⟦0 < last xs;
        primerec (Pr n f g) (Suc (length xs − Suc 0))⟧ ⟹
        rec-calc-rel (Pr n f g) (butlast xs @ [last xs − Suc 0]) rs
      = (rec-exec (Pr n f g) (butlast xs @ [last xs − Suc 0]) = rs)
and ind3:
  ⋀rs. ⟦0 < last xs; primerec g (Suc (Suc (length xs − Suc 0)))⟧
     ⟹ rec-calc-rel g (butlast xs @
             [last xs − Suc 0, rec-exec (Pr n f g)
              (butlast xs @ [last xs − Suc 0])]) rs =
           (rec-exec g (butlast xs @ [last xs − Suc 0,
              rec-exec (Pr n f g)
               (butlast xs @ [last xs − Suc 0])]) = rs)
and h: primerec (Pr n f g) (length (xs::nat list))
show rec-calc-rel (Pr n f g) xs rs = (rec-exec (Pr n f g) xs = rs)
proof(auto)
  assume rec-calc-rel (Pr n f g) xs rs
  thus rec-exec (Pr n f g) xs = rs
  proof(erule-tac calc-pr-reverse)
    fix l
    assume g: xs = l @ [0]
             rec-calc-rel f l rs
             n = length l
    thus rec-exec (Pr n f g) xs = rs
      using ind1[of rs] h
      apply(simp add: rec-exec.simps,
              erule-tac prime-pr-reverse, simp)
      done
  next
    fix l y ry
    assume d:xs = l @ [Suc y]
            rec-calc-rel (Pr (length l) f g) (l @ [y]) ry
            n = length l
            rec-calc-rel g (l @ [y, ry]) rs
    moreover hence primerec g (Suc (Suc n)) using h
    proof(erule-tac prime-pr-reverse)
      assume primerec g (Suc (Suc n)) length xs = Suc n
      thus ?thesis by simp
    qed
    ultimately show rec-exec (Pr n f g) xs = rs
      apply(simp)
      using ind3[of rs]
      apply(simp add: rec-pr-Suc-simp-rewrite)
      using ind2[of ry] h
      apply(simp)
      done
  qed
next
```

```
    show rec-calc-rel (Pr n f g) xs (rec-exec (Pr n f g) xs)
    proof -
      have rec-calc-rel (Pr n f g) (butlast xs @ [last xs])
                (rec-exec (Pr n f g) (butlast xs @ [last xs]))
        using h
        apply(erule-tac prime-pr-reverse, simp)
        apply(case-tac last xs, simp)
        apply(rule-tac calc-pr-zero, simp)
        using ind1[of rec-exec (Pr n f g) (butlast xs @ [0])]
        apply(simp add: rec-exec.simps, simp, simp, simp)
        thm calc-pr-ind
        apply(rule-tac rk = rec-exec (Pr n f g)
              (butlast xs@[last xs - Suc 0]) in calc-pr-ind)
        using ind2[of rec-exec (Pr n f g)
                (butlast xs @ [last xs - Suc 0])] h
        apply(simp, simp, simp)
      proof -
        fix nat
        assume length xs = Suc n
              primerec g (Suc (Suc n))
              last xs = Suc nat
        thus
          rec-calc-rel g (butlast xs @ [nat, rec-exec (Pr n f g)
            (butlast xs @ [nat])]) (rec-exec (Pr n f g) (butlast xs @ [Suc nat]))
          using ind3[of rec-exec (Pr n f g)
                              (butlast xs @ [Suc nat])]
          apply(simp add: rec-exec.simps)
          done
      qed
      thus rec-calc-rel (Pr n f g) xs (rec-exec (Pr n f g) xs)
        using h
        apply(erule-tac prime-pr-reverse, simp)
        apply(subgoal-tac butlast xs @ [last xs] = xs, simp)
        apply(case-tac xs, simp, simp)
        done
    qed
  qed
next
  fix n f xs rs
  assume primerec (Mn n f) (length (xs::nat list))
  thus rec-calc-rel (Mn n f) xs rs = (rec-exec (Mn n f) xs = rs)
    by(erule-tac prime-mn-reverse)
qed

declare numeral-2-eq-2[simp] numeral-3-eq-3[simp]

lemma [intro]: primerec rec-right (Suc 0)
apply(simp add: rec-right-def rec-lo-def Let-def)
apply(tactic ⟪ resolve-tac [@{thm prime-cn},
```

@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
done

lemma [*simp*]:
rec-calc-rel rec-right [*r*] *rs* = (*rec-exec rec-right* [*r*] = *rs*)
apply(*rule-tac prime-rel-exec-eq*, *auto*)
done

lemma [*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* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
apply(*simp add*: *rec-dummyfac-def*)
apply(*tactic* ⟪ *resolve-tac* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
done

lemma [*intro*]: *primerec rec-trpl* (*Suc* (*Suc* (*Suc 0*)))
apply(*simp add*: *rec-trpl-def*)
apply(*tactic* ⟪ *resolve-tac* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
done

lemma [*intro!*]: ⟦*0* < *vl*; *n* ≤ *vl*⟧ ⟹ *primerec* (*rec-listsum2 vl n*) *vl*
apply(*induct n*)
apply(*simp-all add*: *rec-strt'.simps Let-def rec-listsum2.simps*)
apply(*tactic* ⟪ *resolve-tac* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
done

lemma [*elim*]: ⟦*0* < *vl*; *n* ≤ *vl*⟧ ⟹ *primerec* (*rec-strt' vl n*) *vl*
apply(*induct n*)
apply(*simp-all add*: *rec-strt'.simps Let-def*)
apply(*tactic* ⟪ *resolve-tac* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)
done

lemma [*elim*]: *vl* > *0* ⟹ *primerec* (*rec-strt vl*) *vl*
apply(*simp add*: *rec-strt.simps rec-strt'.simps*)
apply(*tactic* ⟪ *resolve-tac* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
done

lemma [*elim*]:
i < *vl* ⟹ *primerec* ((*map* (λ*i*. *recf.id* (*Suc vl*) (*i*))

```
    [Suc 0..<vl] @ [recf.id (Suc vl) (vl)]) ! i) (Suc vl)
apply(induct i, auto simp: nth-append)
done

lemma [intro]: primerec rec-newleft0 ((Suc (Suc 0)))
apply(simp add: rec-newleft-def rec-embranch.simps
             Let-def arity.simps rec-newleft0-def
             rec-newleft1-def rec-newleft2-def rec-newleft3-def)
apply(tactic ⟪ resolve-tac [@{thm prime-cn},
    @{thm prime-id}, @{thm prime-pr}] 1⟫, auto+)+
done

lemma [intro]: primerec rec-newleft1 ((Suc (Suc 0)))
apply(simp add: rec-newleft-def rec-embranch.simps
             Let-def arity.simps rec-newleft0-def
             rec-newleft1-def rec-newleft2-def rec-newleft3-def)
apply(tactic ⟪ resolve-tac [@{thm prime-cn},
    @{thm prime-id}, @{thm prime-pr}] 1⟫, auto+)+
done

lemma [intro]: primerec rec-newleft2 ((Suc (Suc 0)))
apply(simp add: rec-newleft-def rec-embranch.simps
             Let-def arity.simps rec-newleft0-def
             rec-newleft1-def rec-newleft2-def rec-newleft3-def)
apply(tactic ⟪ resolve-tac [@{thm prime-cn},
    @{thm prime-id}, @{thm prime-pr}] 1⟫, auto+)+
done

lemma [intro]: primerec rec-newleft3 ((Suc (Suc 0)))
apply(simp add: rec-newleft-def rec-embranch.simps
             Let-def arity.simps rec-newleft0-def
             rec-newleft1-def rec-newleft2-def rec-newleft3-def)
apply(tactic ⟪ resolve-tac [@{thm prime-cn},
    @{thm prime-id}, @{thm prime-pr}] 1⟫, auto+)+
done

lemma [intro]: primerec rec-newleft (Suc (Suc (Suc 0)))
apply(simp add: rec-newleft-def rec-embranch.simps
             Let-def arity.simps)
apply(rule-tac prime-cn, auto+)
done

lemma [intro]: primerec rec-left (Suc 0)
apply(simp add: rec-left-def rec-lo-def rec-entry-def Let-def)
apply(tactic ⟪ resolve-tac [@{thm prime-cn},
    @{thm prime-id}, @{thm prime-pr}] 1⟫, auto+)+
done

lemma [intro]: primerec rec-actn (Suc (Suc (Suc 0)))
```

apply(*simp add*: *rec-left-def rec-lo-def rec-entry-def*
Let-def rec-actn-def)
apply(*tactic* ⟪ *resolve-tac* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
done

lemma [*intro*]: *primerec rec-stat* (*Suc 0*)
apply(*simp add*: *rec-left-def rec-lo-def rec-entry-def Let-def*
rec-actn-def rec-stat-def)
apply(*tactic* ⟪ *resolve-tac* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
done

lemma [*intro*]: *primerec rec-newstat* (*Suc* (*Suc* (*Suc 0*)))
apply(*simp add*: *rec-left-def rec-lo-def rec-entry-def*
Let-def rec-actn-def rec-stat-def rec-newstat-def)
apply(*tactic* ⟪ *resolve-tac* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
done

lemma [*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* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
done

lemma [*intro*]: *primerec rec-newconf* (*Suc* (*Suc 0*))
apply(*simp add*: *rec-newconf-def*)
apply(*tactic* ⟪ *resolve-tac* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
done

lemma [*intro*]: $0 < vl \Longrightarrow$ *primerec* (*rec-inpt* (*Suc vl*)) (*Suc vl*)
apply(*simp add*: *rec-inpt-def*)
apply(*tactic* ⟪ *resolve-tac* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
done

lemma [*intro*]: *primerec rec-conf* (*Suc* (*Suc* (*Suc 0*)))
apply(*simp add*: *rec-conf-def*)
apply(*tactic* ⟪ *resolve-tac* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
apply(*auto simp*: *numeral-4-eq-4*)
done

lemma [*simp*]:
rec-calc-rel rec-conf [*m*, *r*, *t*] *rs* =

(*rec-exec rec-conf* [*m*, *r*, *t*] = *rs*)
apply(*rule-tac prime-rel-exec-eq*, *auto*)
done

lemma [*intro*]: *primerec rec-lg* (*Suc* (*Suc 0*))
apply(*simp add*: *rec-lg-def Let-def*)
apply(*tactic* ⟪ *resolve-tac* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
done

lemma [*intro*]: *primerec rec-nonstop* (*Suc* (*Suc* (*Suc 0*)))
apply(*simp add*: *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*)
apply(*tactic* ⟪ *resolve-tac* [@{*thm prime-cn*},
@{*thm prime-id*}, @{*thm prime-pr*}] *1*⟫, *auto*+)+
done

lemma *nonstop-eq*[*simp*]:
rec-calc-rel rec-nonstop [*m*, *r*, *t*] *rs* =
(*rec-exec rec-nonstop* [*m*, *r*, *t*] = *rs*)
apply(*rule prime-rel-exec-eq*, *auto*)
done

lemma *halt-lemma′*:
rec-calc-rel rec-halt [*m*, *r*] *t* =
(*rec-calc-rel rec-nonstop* [*m*, *r*, *t*] *0* ∧
(∀ *t′*< *t*.
(∃ *y*. *rec-calc-rel rec-nonstop* [*m*, *r*, *t′*] *y* ∧
y ≠ *0*)))
apply(*auto simp*: *rec-halt-def*)
apply(*erule calc-mn-reverse*, *simp*)
apply(*erule-tac calc-mn-reverse*)
apply(*erule-tac x* = *t′* **in** *allE*, *simp*)
apply(*rule-tac calc-mn*, *simp-all*)
done

The following lemma gives the correctness of *rec-halt*. It says: if *rec-halt* calculates that the TM coded by *m* will reach a standard final configuration after *t* steps of execution, then it is indeed so.

lemma *halt-lemma*:
rec-calc-rel (*rec-halt*) [*m*, *r*] *t* =
(*rec-exec rec-nonstop* [*m*, *r*, *t*] = *0* ∧
(∀ *t′*< *t*. (∃ *y*. *rec-exec rec-nonstop* [*m*, *r*, *t′*] = *y*
∧ *y* ≠ *0*)))
using *halt-lemma′*[*of m r t*]
by *simp*

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
apply(simp add: rec-exec.simps rec-valu-def lg-lemma)
done

lemma [intro]: primerec rec-valu (Suc 0)
apply(simp add: rec-valu-def)
apply(rule-tac k = Suc (Suc 0) in prime-cn)
apply(auto simp: prime-s)
proof −
  show primerec rec-lg (Suc (Suc 0)) by auto
next
  show Suc (Suc 0) = Suc (Suc 0) by simp
next
  show primerec (constn (Suc (Suc 0))) (Suc 0) by auto
qed

lemma [simp]: rec-calc-rel rec-valu [r] rs =
                       (rec-exec rec-valu [r] = rs)
apply(rule-tac prime-rel-exec-eq, auto)
done

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 get-fstn-args-nth:
  k < n ⟹ (get-fstn-args m n ! k) = id m (k)
apply(induct n, simp)
apply(case-tac k = n, simp-all add: get-fstn-args.simps
                                   nth-append)
done
```

lemma [*simp*]:
  $[\![ys \neq [];\ \ k < length\ ys]\!] \Longrightarrow$
  (*get-fstn-args* (*length ys*) (*length ys*) ! *k*) =
  id (*length ys*) (*k*)
by(*erule-tac get-fstn-args-nth*)

lemma *calc-rel-get-pren*:
  $[\![ys \neq [];\ \ k < length\ ys]\!] \Longrightarrow$
  rec-calc-rel (*get-fstn-args* (*length ys*) (*length ys*) ! *k*) *ys*
  (*ys* ! *k*)
apply(*simp*)
apply(*rule-tac calc-id*, *auto*)
done

lemma [*elim*]:
  $[\![xs \neq [];\ k < Suc\ (length\ xs)]\!] \Longrightarrow$
  rec-calc-rel (*get-fstn-args* (*Suc* (*length xs*))
  (*Suc* (*length xs*)) ! *k*) (*m* # *xs*) ((*m* # *xs*) ! *k*)
using *calc-rel-get-pren*[*of m#xs k*]
apply(*simp*)
done

The correctness of *rec-F*, halt case.

lemma *F-lemma*:
  rec-calc-rel rec-halt [*m*, *r*] *t* $\Longrightarrow$
  rec-calc-rel rec-F [*m*, *r*] (*valu* (*rght* (*conf m r t*)))
apply(*simp add*: *rec-F-def*)
apply(*rule-tac rs* = [*rght* (*conf m r t*)] **in** *calc-cn*,
  auto simp: *value-lemma*)
apply(*rule-tac rs* = [*conf m r t*] **in** *calc-cn*,
  auto simp: *right-lemma*)
apply(*rule-tac rs* = [*m*, *r*, *t*] **in** *calc-cn*, *auto*)
apply(*subgoal-tac* $k = 0 \lor k = Suc\ 0 \lor k = Suc\ (Suc\ 0)$,
  auto simp:*nth-append*)
apply(*rule-tac* [*1*−*2*] *calc-id*, *simp-all add*: *conf-lemma*)
done

The correctness of *rec-F*, nonhalt case.

lemma *F-lemma2*:
  $\forall\ t.\ \neg$ *rec-calc-rel rec-halt* [*m*, *r*] *t* $\Longrightarrow$
  $\forall\ rs.\ \neg$ *rec-calc-rel rec-F* [*m*, *r*] *rs*
apply(*auto simp*: *rec-F-def*)
apply(*erule-tac calc-cn-reverse*, *simp* (*no-asm-use*))+
proof −
  fix *rs rsa rsb rsc*
  assume *h*:
    $\forall t.\ \neg$ *rec-calc-rel rec-halt* [*m*, *r*] *t*
    length rsa = *Suc 0*
    rec-calc-rel rec-valu rsa rs

```
    length rsb = Suc 0
    rec-calc-rel rec-right rsb (rsa ! 0)
    length rsc = (Suc (Suc (Suc 0)))
    rec-calc-rel rec-conf rsc (rsb ! 0)
    and g: ∀ k<Suc (Suc (Suc 0)). rec-calc-rel ([recf.id (Suc (Suc 0)) 0,
          recf.id (Suc (Suc 0)) (Suc 0), rec-halt] ! k) [m, r] (rsc ! k)
  have rec-calc-rel (rec-halt ) [m, r]
                              (rsc ! (Suc (Suc 0)))
    using g
    apply(erule-tac x = (Suc (Suc 0)) in allE)
    apply(simp add:nth-append)
    done
  thus False
    using h
    apply(erule-tac x = ysb ! (Suc (Suc 0)) in allE, simp)
    done
qed
```

11.3 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 :: block 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 :: block list ⇒ nat
  where
  bl2wc xs = bl2nat xs 0

fun trpl-code :: t-conf ⇒ 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 :: taction ⇒ 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 ⇒ taction
  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 :: block ⇒ 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 :: tprog ⇒ 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 :: tprog ⇒ nat
  where
  code tp = (let nl = modify-tprog tp in
             godel-code nl)
```

11.4 Relating interperter functions to the execution of TMs

```
lemma [simp]: bl2wc [] = 0 by(simp add: bl2wc.simps bl2nat.simps)
term trpl

lemma [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 [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 [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

term set-of
lemma prime-coprime: ⟦Prime x; Prime y; x≠y⟧ ⟹ coprime x y
proof(simp only: Prime.simps coprime-nat, 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
    apply(auto)
    apply(case-tac ka, simp, simp)
    apply(case-tac k, simp, simp)
    done
  from this show False
  proof(erule-tac disjE)
    assume (Suc 0::nat) < k
```

hence $k < d*k \wedge d < d*k$
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 $ka < d * ka \wedge d < d*ka$
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* $= \{y.\ y \leq$ *Suc* (*Pi i!*) $\wedge$ *Pi i* $< y \wedge$ *Prime y*$\}$
have *finite ?setx* **by** *simp*
moreover have *?setx* $\neq \{\}$
using *prime-ex*[*of Pi i*]
apply(*auto*)
done
ultimately show *Pi i* $<$ *Min ?setx*
apply(*simp add*: *Min-gr-iff*)
done
qed

lemma *Pi-inc-gr*: $i < j \Longrightarrow$ *Pi i* $<$ *Pi j*
proof(*induct j*, *simp*)
fix j
assume *ind*: $i < j \Longrightarrow$ *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(case-tac i < j)
using Pi-inc-gr[of i j]
apply(simp)
using Pi-inc-gr[of j i]
apply(simp)
done

lemma [intro]: Prime (Suc (Suc 0))
apply(auto simp: Prime.simps)
apply(case-tac u, simp, case-tac nat, simp, simp)
done

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)
by(rule-tac coprime-exp2-nat, erule-tac Pi-coprime)

lemma coprime-dvd-mult-nat2: ⟦coprime (k::nat) n; k dvd n * m⟧ ⟹ k dvd m
apply(erule-tac coprime-dvd-mult-nat)
apply(simp add: dvd-def, auto)
apply(rule-tac x = ka in exI)
apply(subgoal-tac n * m = m * n, simp)
apply(simp add: nat-mult-commute)
done

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(case-tac xs = [], simp, simp)
  done
moreover have butlast xs @ last xs # y # ys = xs @ y # ys
  using h
  apply(case-tac 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, simp add: godel-code'.simps)
  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(case-tac ps, simp, simp)
  from g have
    coprime (Pi (Suc i)) (godel-code' (butlast ps) (Suc 0) *
                                  Pi (length ps) ^(last ps))
  proof(rule-tac coprime-mult-nat, simp)
    show coprime (Pi (Suc i)) (Pi (length ps) ^ last ps)
      apply(rule-tac coprime-exp-nat, rule prime-coprime, auto)
      using Pi-notEq[of Suc i length ps] h by simp
  qed
```

```
  from this and k show coprime (Pi (Suc i)) (godel-code' ps (Suc 0))
    by simp
qed

lemma Pi-coprime-suf: i < j ⟹ coprime (Pi i) (godel-code' ps j)
proof(induct length ps arbitrary: ps, simp add: godel-code'.simps)
  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(case-tac ps = [], simp, simp)
    done
  from g have
    coprime (Pi i) (godel-code' (butlast ps) j *
                      Pi (length ps + j − 1)^last ps)
    apply(rule-tac coprime-mult-nat, simp)
    using Pi-power-coprime[of i length ps + j − 1 1 last ps] h
    apply(auto)
    done
  from k and this show coprime (Pi i) (godel-code' ps j)
    by auto
qed

lemma godel-finite:
  finite {u. Pi (Suc i) ^ u dvd godel-code' nl (Suc 0)}
proof(rule-tac n = godel-code' nl (Suc 0) in
                      bounded-nat-set-is-finite, auto,
      case-tac ia < godel-code' nl (Suc 0), auto)
  fix ia
  assume g1: Pi (Suc i) ^ ia dvd godel-code' nl (Suc 0)
    and g2: ¬ ia < godel-code' nl (Suc 0)
  from g1 have Pi (Suc i)^ia ≤ godel-code' nl (Suc 0)
    apply(erule-tac dvd-imp-le)
    using godel-code'-not0[of nl Suc 0] by simp
  moreover have ia < Pi (Suc i)^ia
    apply(rule x-less-exp)
    using Pi-gr-1 by auto
  ultimately show False
    using g2
    by(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*]
apply(*simp*)
done
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(*rule-tac coprime-dvd-mult-nat*)
show *coprime* (*Pi* (*Suc i*)^*y*) *?suf′*
proof −
have *coprime* (*Pi* (*Suc i*) ^ *y*) (*?suf′*^(*Suc 0*))
apply(*rule-tac coprime-exp2-nat*)

```
        apply(rule-tac  Pi-coprime-suf, simp)
        done
      thus ?thesis by simp
    qed
  qed
  hence Pi (Suc i) ^ y dvd Pi (Suc i) ^ nl ! i
  proof(rule-tac coprime-dvd-mult-nat2)
    show coprime (Pi (Suc i) ^ y) ?pref
    proof −
      have coprime (Pi (Suc i)^y) (?pref^Suc 0)
        apply(rule-tac coprime-exp2-nat)
        apply(rule-tac Pi-coprime-pre, simp)
        done
      thus ?thesis by simp
    qed
  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 [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 −
 fix u
 show coprime (Pi (Suc i) ^ u) ((Suc (Suc 0)) ^ length nl)
 proof(rule-tac coprime-exp2-nat)
   have Pi 0 = (2::nat)
     apply(simp add: Pi.simps)
     done
   moreover have coprime (Pi (Suc i)) (Pi 0)
     apply(rule-tac Pi-coprime, simp)
     done
   ultimately show coprime (Pi (Suc i)) (Suc (Suc 0)) by simp
 qed
qed
```

lemma *godel-code-get-nth*:
  $i < length\ nl \Longrightarrow$
    $Max\ \{u.\ Pi\ (Suc\ i)\ \hat{}\ u\ dvd\ godel\text{-}code\ nl\} = nl\ !\ i$
by(*simp add*: *godel-code.simps godel-code′-get-nth*)

lemma $trpl\ l\ st\ r = godel\text{-}code'\ [l,\ st,\ r]\ 0$
apply(*simp add*: *trpl.simps godel-code′.simps*)
done

lemma *mod-dvd-simp*: $(x\ mod\ y = (0::nat)) = (y\ dvd\ x)$
by(*simp add*: *dvd-def*, *auto*)

lemma *dvd-power-le*: $[\![a > Suc\ 0;\ a\ \hat{}\ y\ dvd\ a\ \hat{}\ l]\!] \Longrightarrow y \leq l$
apply(*case-tac* $y \leq l$, *simp*, *simp*)
apply(*subgoal-tac* $\exists\ d.\ y = l + d$, *auto simp*: *power-add*)
apply(*rule-tac* $x = y - l$ **in** *exI*, *simp*)
done

lemma [*elim*]: $Pi\ n = 0 \Longrightarrow RR$
  using *Pi-not-0*[*of n*] **by** *simp*

lemma [*elim*]: $Pi\ n = Suc\ 0 \Longrightarrow RR$
  using *Pi-gr-1*[*of n*] **by** *simp*

lemma *finite-power-dvd*:
  $[\![(a::nat) > Suc\ 0;\ y \neq 0]\!] \Longrightarrow finite\ \{u.\ a\hat{}u\ dvd\ y\}$
apply(*auto simp*: *dvd-def*)
apply(*rule-tac* $n = y$ **in** *bounded-nat-set-is-finite*, *auto*)
apply(*case-tac k*, *simp*,*simp*)
apply(*rule-tac trans-less-add1*)
apply(*erule-tac x-less-exp*)
done

lemma *conf-decode1*: $[\![m \neq n;\ m \neq k;\ k \neq n]\!] \Longrightarrow$
  $Max\ \{u.\ Pi\ m\ \hat{}\ u\ dvd\ Pi\ m\ \hat{}\ l * Pi\ n\ \hat{}\ st * Pi\ k\ \hat{}\ r\} = l$
proof −
  let $?setx = \{u.\ Pi\ m\ \hat{}\ u\ dvd\ Pi\ m\ \hat{}\ l * Pi\ n\ \hat{}\ st * Pi\ k\ \hat{}\ r\}$
  assume g: $m \neq n\ m \neq k\ k \neq n$
  show $Max\ ?setx = l$
  proof(*rule-tac Max-eqI*)
    show *finite ?setx*
      apply(*rule-tac finite-power-dvd*, *auto simp*: *Pi-gr-1*)
      done
  next
    fix y
    assume h: $y \in ?setx$
    have $Pi\ m\ \hat{}\ y\ dvd\ Pi\ m\ \hat{}\ l$

```
    proof -
      have Pi m ^ y dvd Pi m ^ l * Pi n ^ st
        using h g
        apply(rule-tac n = Pi k^r in coprime-dvd-mult-nat)
        apply(rule Pi-power-coprime, simp, simp)
        done
      thus Pi m^y dvd Pi m^l
        apply(rule-tac n =  Pi n ^ st in coprime-dvd-mult-nat)
        using g
        apply(rule-tac Pi-power-coprime, simp, simp)
        done
    qed
    thus y ≤ (l::nat)
      apply(rule-tac a = Pi m in power-le-imp-le-exp)
      apply(simp-all add: Pi-gr-1)
      apply(rule-tac dvd-power-le, auto)
      done
  next
    show l ∈ ?setx by simp
  qed
qed

lemma conf-decode2:
  ⟦m ≠ n; m ≠ k; n ≠ k;
  ¬ Suc 0 < Pi m ^ l * Pi n ^ st * Pi k ^ r⟧ ⟹ l = 0
apply(case-tac Pi m ^ l * Pi n ^ st * Pi k ^ r, auto)
done

lemma [simp]: left (trpl l st r) = l
apply(simp add: left.simps trpl.simps lo.simps
              loR.simps mod-dvd-simp, auto simp: conf-decode1)
apply(case-tac Pi 0 ^ l * Pi (Suc 0) ^ st * Pi (Suc (Suc 0)) ^ r,
      auto)
apply(erule-tac x = l in allE, auto)
done

lemma [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(case-tac Pi 0 ^ l * Pi (Suc 0) ^ st *
                             Pi (Suc (Suc 0)) ^ r, auto)
apply(erule-tac x = st in allE, auto)
done

lemma [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(case-tac 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::tprog) ! (2 * (st − Suc 0))))
and h: Suc st ≤ length (tp::tprog) 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
apply(auto)

```
    apply(cases tp, simp, case-tac a, simp add: modify-tprog.simps)
    done
next
  case False
  assume g: st ≠ 0
  hence ∃ aa ab ba bb tp'. tp = (aa, ab) # (ba, bb) # tp'
    using h
    apply(case-tac tp, simp, case-tac list, 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' ! (4 * (st − Suc 0)) =
    action-map (fst ((tp'::tprog) ! (2 * (st − Suc 0))))
    apply(rule-tac ind)
    using h g by auto
  thus ?thesis
    using g1 g
    apply(case-tac st, simp, simp add: Four-Suc)
    done
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::tprog) 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(auto)
      apply(cases tp, simp, case-tac a, simp add: modify-tprog.simps)
      apply(case-tac list, simp, case-tac ab,
            simp add: modify-tprog.simps)
      done
  next
    case False
    assume g: st ≠ 0
    hence ∃ aa ab ba bb tp'. tp = (aa, ab) # (ba, bb) # tp'
      using h
```

apply(*case-tac tp*, *simp*, *case-tac list*, *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(*case-tac 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(simp add: fetch.simps nth-of.simps)
    apply(case-tac b, auto simp: block-map.simps nth-of.simps split: if-splits)
    done
  ultimately show
    Entry (godel-code (modify-tprog tp))
                    (4 * (st − Suc 0) + 2 * (r mod 2))
          = action-map nact
    by simp
qed

lemma [simp]: fetch tp 0 b = (nact, ns) ⟹ ns = 0
by(simp add: fetch.simps)

lemma Five-Suc: 5 = Suc 4 by simp

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::tprog) 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, simp, case-tac a, simp add: modify-tprog.simps)
      apply(case-tac list, simp, case-tac ab,
                      simp add: modify-tprog.simps, auto)
      done
  next
    case False
    assume g: st ≠ 0
```

```
    hence ∃ aa ab ba bb tp'. tp = (aa, ab) # (ba, bb) # tp'
      using h
      apply(case-tac tp, simp, case-tac list, 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 (4 * (st − Suc 0) + 2 * b) =
                            snd (tp' ! (2 * (st − Suc 0) + b))
      apply(rule-tac ind)
      using h g by auto
    thus ?thesis
      using g1 g
      apply(case-tac st, simp, simp)
      done
  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
      apply(simp add: length-modify)
      done
    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(simp add: fetch.simps nth-of.simps)
    apply(case-tac b, auto simp: block-map.simps nth-of.simps
                             split: if-splits)
```

```
    done
  ultimately show Entry (godel-code (modify-tprog tp)) (?i)
        = ns
    by simp
qed


lemma [intro!]:
  ⟦a = a'; b = b'; c = c'⟧ ⟹ trpl a b c = trpl a' b' c'
by simp

lemma [simp]: bl2wc [Bk] = 0
by(simp add: bl2wc.simps bl2nat.simps)

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 [simp]: c ≠ [] ⟹ 2 * bl2wc (tl c) = bl2wc c − bl2wc c mod 2
apply(case-tac c, simp, case-tac a)
apply(auto simp: bl2wc.simps bl2nat.simps bl2nat-double)
done

lemma [simp]:
  c ≠ [] ⟹ bl2wc (Oc # tl c) = Suc (bl2wc c) − bl2wc c mod 2
apply(case-tac c, simp, case-tac a)
apply(auto simp: bl2wc.simps bl2nat.simps bl2nat-double)
done

lemma [simp]: bl2wc (Bk # c) = 2*bl2wc (c)
apply(simp add: bl2wc.simps bl2nat.simps bl2nat-double)
```

done

lemma [*simp*]: *bl2wc* [*Oc*] = *Suc 0*
by(*simp add*: *bl2wc.simps bl2nat.simps*)

lemma [*simp*]: $b \neq [] \Longrightarrow$ *bl2wc* (*tl b*) = *bl2wc b div 2*
apply(*case-tac b*, *simp*, *case-tac a*)
apply(*auto simp*: *bl2wc.simps bl2nat.simps bl2nat-double*)
done

lemma [*simp*]: $b \neq [] \Longrightarrow$ *bl2wc* ([*hd b*]) = *bl2wc b mod 2*
apply(*case-tac b*, *simp*, *case-tac a*)
apply(*auto simp*: *bl2wc.simps bl2nat.simps bl2nat-double*)
done

lemma [*simp*]: $[\![b \neq []; c \neq []]\!] \Longrightarrow$ *bl2wc* (*hd b* # *c*) = *2* ∗ *bl2wc c* + *bl2wc b mod 2*
apply(*case-tac b*, *simp*, *case-tac a*)
apply(*auto simp*: *bl2wc.simps bl2nat.simps bl2nat-double*)
done

lemma [*simp*]: *2* ∗ (*bl2wc c div 2*) = *bl2wc c* − *bl2wc c mod 2*
by(*simp add*: *mult-div-cancel*)

lemma [*simp*]: *bl2wc* (*Oc* # *list*) *mod 2* = *Suc 0*
by(*simp add*: *bl2wc.simps bl2nat.simps bl2nat-double*)

declare *code.simps*[*simp del*]
declare *nth-of.simps*[*simp del*]
declare *new-tape.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 (*tstep c tp*) =
rec-exec rec-newconf [*code tp*, *trpl-code c*]
apply(*cases c*, *auto simp*: *tstep.simps*)
proof(*case-tac fetch tp a* (*case c of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*),
simp add: *newconf.simps trpl-code.simps*)
fix *a b c aa ba*
assume *h*: (*a*::*nat*) ≤ *length tp div 2*
fetch tp a (*case c of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*) = (*aa*, *ba*)
moreover hence *actn* (*code tp*) *a* (*bl2wc c*) = *action-map aa*
apply(*rule-tac b* = (*case c of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*)
in *fetch-action-eq*, *auto*)
apply(*auto split*: *list.splits*)
apply(*case-tac ab*, *auto*)

```
    done
  moreover from h have (newstat (code tp) a (bl2wc c)) = ba
    apply(rule-tac b = (case c of [] ⇒ Bk | x # xs ⇒ x)
          in fetch-state-eq, auto split: list.splits)
    apply(case-tac ab, auto)
    done
  ultimately show
    trpl-code (ba, new-tape aa (b, c)) =
    trpl (newleft (bl2wc b) (bl2wc c) (actn (code tp) a (bl2wc c)))
    (newstat (code tp) a (bl2wc c)) (newrght (bl2wc b) (bl2wc c)
     (actn  (code tp) a (bl2wc c)))
    by(auto simp: new-tape.simps trpl-code.simps
         newleft.simps newrght.simps split: taction.splits)
qed

lemma [simp]: a^0 = []
apply(simp add: exp-zero)
done
lemma [simp]: bl2nat (Oc # Oc^x) 0 = (2 * 2 ^ x - Suc 0)
apply(induct x)
apply(simp add: bl2nat.simps)
apply(simp add: bl2nat.simps bl2nat-double exp-ind-def)
done

lemma [simp]: bl2nat (Oc^y) 0 = 2^y - Suc 0
apply(induct y, auto simp: bl2nat.simps exp-ind-def bl2nat-double)
apply(case-tac (2::nat)^y, auto)
done

lemma [simp]: bl2nat (Bk^l) n = 0
apply(induct l, auto simp: bl2nat.simps bl2nat-double exp-ind-def)
done

lemma bl2nat-cons-bk: bl2nat (ks @ [Bk]) 0 = bl2nat ks 0
apply(induct ks, auto simp: bl2nat.simps split: block.splits)
apply(case-tac a, auto simp: bl2nat.simps bl2nat-double)
done

lemma bl2nat-cons-oc:
  bl2nat (ks @ [Oc]) 0 =  bl2nat ks 0 + 2 ^ length ks
apply(induct ks, auto simp: bl2nat.simps split: block.splits)
apply(case-tac a, auto simp: bl2nat.simps bl2nat-double)
done

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::block 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(case-tac 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(case-tac k, simp-all add: bl2nat.simps)
    apply(simp-all only: bl2nat-cons-bk bl2nat-cons-oc)
    done
qed

lemma bl2nat-exp:  n ≠ 0 ⟹ bl2nat bl n = 2^n * bl2nat bl 0
apply(induct bl)
apply(auto simp: bl2nat.simps)
apply(case-tac a, auto simp: bl2nat.simps bl2nat-double)
done

lemma nat-minus-eq: ⟦a = b; c = d⟧ ⟹ a − c = b − d
by auto

lemma tape-of-nat-list-butlast-last:
  ys ≠ [] ⟹ <ys @ [y]> = <ys> @ Bk # Oc^(Suc y)
apply(induct ys, simp, simp)
apply(case-tac ys = [], simp add: tape-of-nl-abv
                              tape-of-nat-list.simps)
apply(simp)
done

lemma listsum2-append:
  ⟦n ≤ length xs⟧ ⟹ listsum2 (xs @ ys) n = listsum2 xs n
apply(induct n)
apply(auto simp: listsum2.simps nth-append)
done

lemma strt'-append:
  ⟦n ≤ length xs⟧ ⟹ strt' xs n = strt' (xs @ ys) n
proof(induct n arbitrary: xs ys)
  fix xs ys
  show strt' xs 0 = strt' (xs @ ys) 0 by(simp add: strt'.simps)
```

```
next
  fix n xs ys
  assume ind:
    ⋀ xs ys. n ≤ length xs ⟹ strt' xs n = strt' (xs @ ys) n
    and h: Suc n ≤ length (xs::nat list)
  show strt' xs (Suc n) = strt' (xs @ ys) (Suc n)
    using ind[of xs ys] h
    apply(simp add: strt'.simps nth-append listsum2-append)
    done
qed

lemma length-listsum2-eq:
  ⟦length (ys::nat list) = k⟧
       ⟹ length (<ys>) = listsum2 (map Suc ys) k + k − 1
apply(induct k arbitrary: ys, simp-all add: listsum2.simps)
apply(subgoal-tac ∃ xs x. ys = xs @ [x], auto)
proof −
  fix xs x
  assume ind: ⋀ys. length ys = length xs ⟹ length (<ys>)
    = listsum2 (map Suc ys) (length xs) +
      length (xs::nat list) − Suc 0
  have length (<xs>)
    = listsum2 (map Suc xs) (length xs) + length xs − Suc 0
    apply(rule-tac ind, simp)
    done
  thus length (<xs @ [x]>) =
    Suc (listsum2 (map Suc xs @ [Suc x]) (length xs) + x + length xs)
    apply(case-tac xs = [])
    apply(simp add: tape-of-nl-abv listsum2.simps
      tape-of-nat-list.simps)
    apply(simp add: tape-of-nat-list-butlast-last)
    using listsum2-append[of length xs map Suc xs [Suc x]]
    apply(simp)
    done
next
  fix k ys
  assume length ys = Suc k
  thus ∃ xs x. ys = xs @ [x]
    apply(rule-tac x = butlast ys in exI,
          rule-tac x = last ys in exI)
    apply(case-tac ys, auto)
    done
qed

lemma tape-of-nat-list-length:
      length (<(ys::nat list)>) =
              listsum2 (map Suc ys) (length ys) + length ys − 1
  using length-listsum2-eq[of ys length ys]
  apply(simp)
```

```
  done


lemma [simp]:
 trpl-code (steps (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 rec-t-eq-steps:
  turing-basic.t-correct tp ⟹
  trpl-code (steps (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:
      t-correct tp ⟹ trpl-code (steps (Suc 0, Bk^l, <lm>) tp n)
      = rec-exec rec-conf [code tp, bl2wc (<lm>), n]
      and h: t-correct tp
    show
      trpl-code (steps (Suc 0, Bk^l, <lm>) tp (Suc n)) =
      rec-exec rec-conf [code tp, bl2wc (<lm>), Suc n]
    proof(case-tac steps (Suc 0, Bk^l, <lm>) tp  n,
        simp only: tstep-red conf-lemma conf.simps)
      fix a b c
      assume g: steps (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 (tstep (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: s-keep)
        done
      ultimately show
        trpl-code (tstep (a, b, c) tp) =
            newconf (code tp) (conf (code tp) (bl2wc (<lm>)) n)
        by(simp add: newconf-lemma)
    qed
```

```
  qed
qed

lemma [simp]: bl2wc (Bk^m) = 0
apply(induct m)
apply(simp, simp)
done

lemma [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(case-tac 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*.

```
lemma nonstop-t-eq:
  ⟦steps (Suc 0, Bk^l, <lm>) tp stp = (0, Bk^m, Oc^rs @ Bk^n);
  turing-basic.t-correct tp;
  rs > 0⟧
  ⟹ rec-exec rec-nonstop [code tp, bl2wc (<lm>), stp] = 0
proof(simp add: nonstop-lemma nonstop.simps nstd.simps)
  assume h: steps (Suc 0, Bk^l, <lm>) tp stp = (0, Bk^m, Oc^rs @ Bk^n)
  and tc-t: turing-basic.t-correct tp 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(case-tac rs, simp, simp add: bl2nat.simps)
      done
  qed
qed

lemma [simp]: actn m 0 r = 4
by(simp add: actn.simps)

lemma [simp]: newstat m 0 r = 0
by(simp add: newstat.simps)

declare exp-def[simp del]

lemma halt-least-step:
  ⟦steps (Suc 0, Bk^l, <lm>) tp stp = (0, Bk^m, Oc^rs @ Bk^n);
    turing-basic.t-correct tp;
    0<rs⟧ ⟹
    ∃ stp. (nonstop (code tp) (bl2wc (<lm>)) stp = 0 ∧
       (∀ stp'. nonstop (code tp) (bl2wc (<lm>)) stp' = 0 ⟶ stp ≤ stp'))
proof(induct stp, simp add: steps.simps, simp)
  fix stp
  assume ind:
```

```
  steps (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:
  steps (Suc 0, Bk^l, <lm>) tp (Suc stp) = (0, Bk^m, Oc^rs @ Bk^n)
  turing-basic.t-correct tp
  0 < rs
from h show
  ∃ stp. nonstop (code tp) (bl2wc (<lm>)) stp = 0
  ∧ (∀ stp'. nonstop (code tp) (bl2wc (<lm>)) stp' = 0 ⟶ stp ≤ stp')
proof(simp add: tstep-red,
    case-tac steps (Suc 0, Bk^l, <lm>) tp stp, simp,
     case-tac a, simp add: tstep-0)
  assume steps (Suc 0, Bk^l, <lm>) tp stp = (0, Bk^m, Oc^rs @ Bk^n)
  thus ∃ stp. nonstop (code tp) (bl2wc (<lm>)) stp = 0 ∧
    (∀ stp'. nonstop (code tp) (bl2wc (<lm>)) stp' = 0 ⟶ stp ≤ stp')
    apply(erule-tac ind)
    done
next
  fix a b c nat
  assume steps (Suc 0, Bk^l, <lm>) tp stp = (a, b, c)
    a = Suc nat
  thus ∃ 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:steps (Suc 0, Bk^l, <lm>) tp stp = (Suc nat, b, c)
      nonstop (code tp) (bl2wc (<lm>)) stp' = 0
    thus  Suc stp ≤ stp'
    proof(case-tac Suc stp ≤ stp', simp, simp)
      assume ¬ Suc stp ≤ stp'
      hence stp' ≤ stp by simp
      hence ¬ isS0 (steps (Suc 0, Bk^l, <lm>) tp stp')
        using g
        apply(case-tac steps (Suc 0, Bk^l, <lm>) tp stp',auto,
          simp add: isS0-def)
        apply(subgoal-tac ∃ n. stp = stp' + n,
          auto simp: steps-add steps-0)
        apply(rule-tac x = stp − stp'  in exI, simp)
        done
      hence nonstop (code tp) (bl2wc (<lm>)) stp' = 1
      proof(case-tac steps (Suc 0, Bk^l, <lm>) tp stp',
          simp add: isS0-def nonstop.simps)
        fix a b c
        assume k:
          0 < a steps (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
  qed
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 [elim]: 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(case-tac 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*: *x* > *Suc 0* ⟹ *lo* (*x* ^ *r*) *x* = *r*
apply(*auto simp*: *lo.simps loR.simps mod-dvd-simp*)
apply(*case-tac x*^*r*, *simp-all*)
done

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(*case-tac 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 *F-t-halt-eq*:
⟦*steps* (*Suc 0*, Bk^l, <*lm*>) *tp stp* = (*0*, Bk^m, Oc^{rs}@Bk^n);
turing-basic.t-correct tp;
0<*rs*⟧
⟹ *rec-calc-rel 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
    steps (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-calc-rel rec-halt [code tp, (bl2wc (<lm>))] stpa
    using h
    apply(simp add: halt-lemma nonstop-lemma, auto)
    done
  show
    rec-calc-rel rec-F [code tp, (bl2wc (<lm>))] (rs − Suc 0)
  proof −
    have
      rec-calc-rel rec-F [code tp, (bl2wc (<lm>))]
                     (valu (rght (conf (code tp) (bl2wc (<lm>)) stpa)))
      apply(rule F-lemma) using g2 h by auto
    moreover 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
    ultimately show ?thesis by simp
  qed
qed


end
theory UTM
imports Main uncomputable recursive abacus UF GCD
begin
```

12 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 resurve 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, start state of ever TM is fixed to state 1 while the final state is fixed to 0.

The input and output of *prepare* are illustrated respectively by Figure 1 and 2.

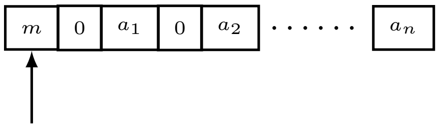

Figure 1: The input of TM *prepare*

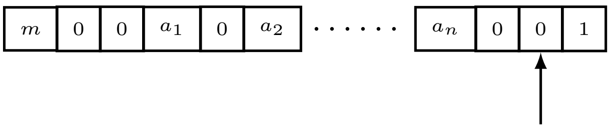

Figure 2: The output of TM *prepare*

As shown in Figure 1, the input of *prepare* is the same as the the input of UTM, where m is the Godel coding of the TM being interpreted and a_1 through a_n are the n input arguments of the TM under interpretation. The purpose of *purpose* is to transform this initial tape layout to the one shown in Figure 2, which is convenient for the generation of Wang's codding of $a_1, \ldots, a_n$. The coding procedure starts from a_n and ends after a_1 is encoded. The coding result is stored in an accumulator at the end of the tape (initially represented by the 1 two blanks right to a_n in Figure 2). In Figure 2, arguments $a_1, \ldots, a_n$ are separated by two blanks on both ends with the rest so that movement conditions can be implemented conveniently in subsequent TMs, because, by convention, two consecutive blanks are usually used to signal the end or start of a large chunk of data. The diagram of *prepare* is given in Figure 3.

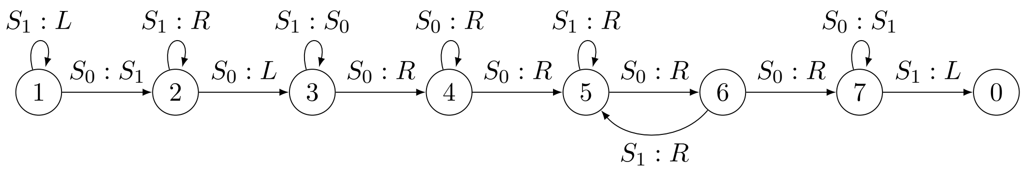

Figure 3: The diagram of TM *prepare*

The purpose of TM *mainwork* is to compute the Wang's encoding of $a_1, \ldots, a_n$. Every bit of $a_1, \ldots, a_n$, including the separating bits, is processed from left to right. In order to detect the termination condition when the left most bit of a_1 is reached, TM *mainwork* needs to look ahead and consider three different situations at the start of every iteration:

1. The TM configuration for the first situation is shown in Figure 4, where the accumulator is stored in r, both of the next two bits to be encoded are 1. The configuration at the end of the iteration is shown in Figure 5, where the first 1-bit has been encoded and cleared. Notice that the accumulator has been changed to $(r + 1) \times 2$ to reflect the encoded bit.

2. The TM configuration for the second situation is shown in Figure 6, where the accumulator is stored in r, the next two bits to be encoded are 1 and 0. After the first 1-bit was encoded and cleared, the second 0-bit is difficult to detect and process. To solve this problem, these two consecutive bits are encoded in one iteration. In this situation, only the first 1-bit needs to be cleared since the second one is cleared by definition. The configuration at the end of the iteration is shown in Figure 7. Notice that the accumulator has been changed to $(r+1) \times 4$ to reflect the two encoded bits.

3. The third situation corresponds to the case when the last bit of a_1 is reached. The TM configurations at the start and end of the iteration are shown in Figure 8 and 9 respectively. For this situation, only the read write head needs to be moved to the left to prepare a initial configuration for TM *adjust* to start with.

The diagram of *mainwork* is given in Figure 10. The two rectangular nodes labeled with $2 \times x$ and $4 \times x$ are two TMs compiling from recursive functions so that we do not have to design and verify two quite complicated TMs.

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.

definition *rec-twice* :: *recf*

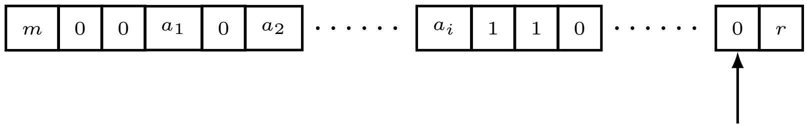

Figure 4: The first situation for TM *mainwork* to consider

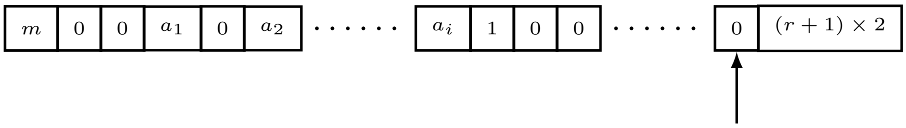

Figure 5: The output for the first case of TM *mainwork*'s processing

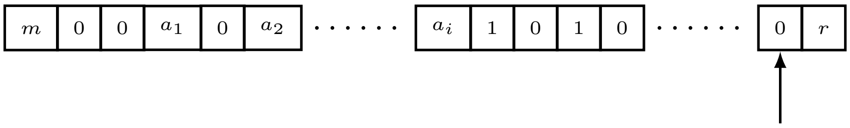

Figure 6: The second situation for TM *mainwork* to consider

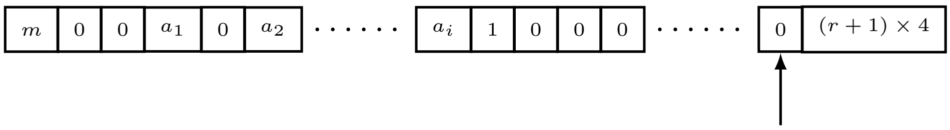

Figure 7: The output for the second case of TM *mainwork*'s processing

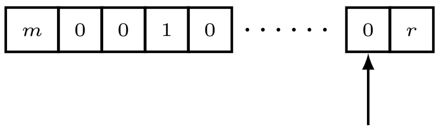

Figure 8: The third situation for TM *mainwork* to consider

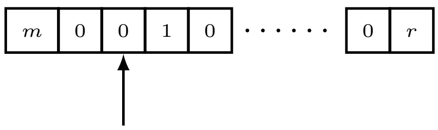

Figure 9: The output for the third case of TM *mainwork*'s processing

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*]

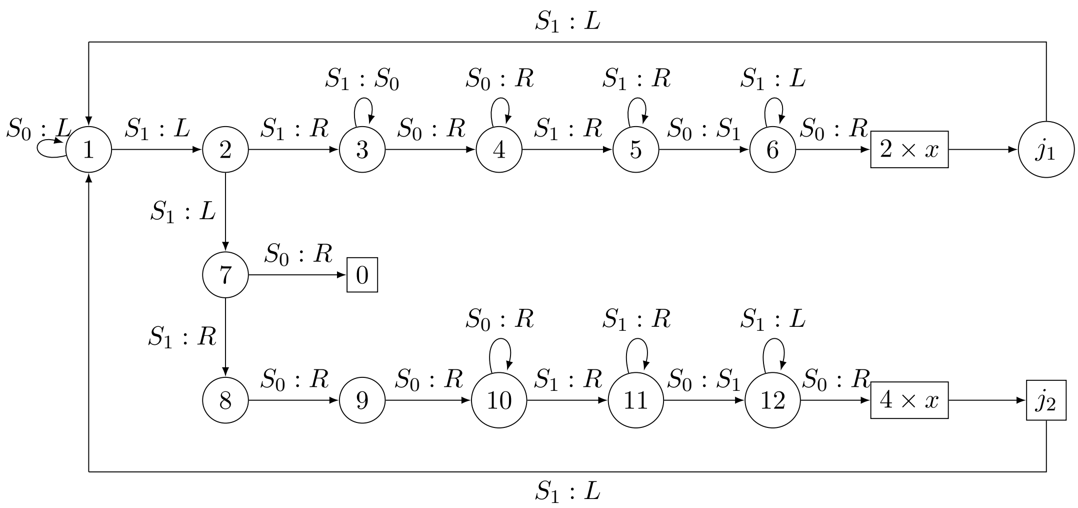

Figure 10: The diagram of TM *mainwork*

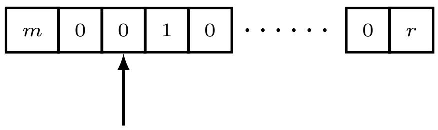

Figure 11: Initial configuration of TM *adjust*

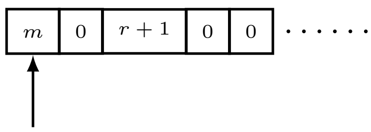

Figure 12: Final configuration of TM *adjust*

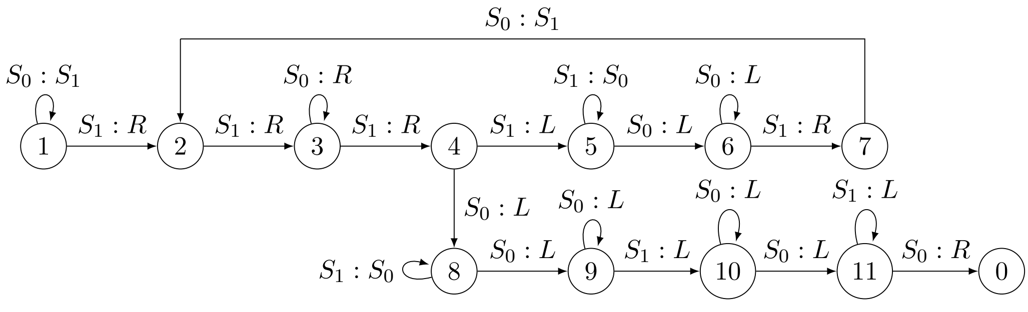

Figure 13: Diagram of TM *adjust*

definition *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* :: *tprog*
where
t-twice = *change-termi-state* (*tm-of* (*abc-twice*) @ (*tMp 1* (*start-of twice-ly* (*length abc-twice*) − *Suc 0*)))

definition *t-fourtimes* :: *tprog*
where
t-fourtimes = *change-termi-state* (*tm-of* (*abc-fourtimes*) @
(*tMp 1* (*start-of fourtimes-ly* (*length abc-fourtimes*) − *Suc 0*)))

definition *t-twice-len* :: *nat*
where
t-twice-len = *length t-twice div 2*

definition *t-wcode-main-first-part*:: *tprog*
where
t-wcode-main-first-part ≡
[(*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* :: *tprog*
where
t-wcode-main = (*t-wcode-main-first-part* @ *tshift t-twice 12* @ [(*L*, *1*), (*L*, *1*)]
@ *tshift t-fourtimes* (*t-twice-len* + *13*) @ [(*L*, *1*), (*L*, *1*)])

fun *bl-bin* :: *block 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* = *block list* ⇒ *nat* ⇒ *tape* ⇒ *bool*

fun *wcode-before-double* :: *bin-inv-t*
  where
  wcode-before-double ires rs (*l*, *r*) =
    (∃ *ln rn*. *l* = *Bk* # *Bk* # Bk^{ln} @ *Oc* # *ires* ∧
        r = $Oc^{(Suc\ (Suc\ rs))}$ @ Bk^{rn})

declare *wcode-before-double.simps*[*simp del*]

fun *wcode-after-double* :: *bin-inv-t*
  where
  wcode-after-double ires rs (*l*, *r*) =
    (∃ *ln rn*. *l* = *Bk* # *Bk* # Bk^{ln} @ *Oc* # *ires* ∧
      r = $Oc^{Suc\ (Suc\ (Suc\ 2*rs))}$ @ Bk^{rn})

declare *wcode-after-double.simps*[*simp del*]

fun *wcode-on-left-moving-1-B* :: *bin-inv-t*
  where
  wcode-on-left-moving-1-B ires rs (*l*, *r*) =
    (∃ *ml mr rn*. *l* = Bk^{ml} @ *Oc* # *Oc* # *ires* ∧
        r = Bk^{mr} @ $Oc^{Suc\ rs}$ @ Bk^{rn} ∧
        ml + *mr* > *Suc 0* ∧ *mr* > *0*)

declare *wcode-on-left-moving-1-B.simps*[*simp del*]

fun *wcode-on-left-moving-1-O* :: *bin-inv-t*
  where
  wcode-on-left-moving-1-O ires rs (*l*, *r*) =
    (∃ *ln rn*.
        l = *Oc* # *ires* ∧
        r = *Oc* # Bk^{ln} @ *Bk* # *Bk* # $Oc^{Suc\ rs}$ @ Bk^{rn})

declare *wcode-on-left-moving-1-O.simps*[*simp del*]

fun *wcode-on-left-moving-1* :: *bin-inv-t*
  where
  wcode-on-left-moving-1 ires rs (*l*, *r*) =
      (*wcode-on-left-moving-1-B ires rs* (*l*, *r*) ∨ *wcode-on-left-moving-1-O ires rs* (*l*, *r*))

declare *wcode-on-left-moving-1.simps*[*simp del*]

fun *wcode-on-checking-1* :: *bin-inv-t*
where
wcode-on-checking-1 ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l = ires \wedge$
$r = Oc \# Oc \# Bk^{ln} @ Bk \# Bk \# Oc^{Suc\ rs} @ Bk^{rn})$

fun *wcode-erase1* :: *bin-inv-t*
where
wcode-erase1 ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l = Oc \# ires \wedge$
$tl\ r = Bk^{ln} @ Bk \# Bk \# Oc^{Suc\ rs} @ Bk^{rn})$

declare *wcode-erase1.simps* [*simp del*]

fun *wcode-on-right-moving-1* :: *bin-inv-t*
where
wcode-on-right-moving-1 ires rs $(l, r) =$
$(\exists\ ml\ mr\ rn.$
$l = Bk^{ml} @ Oc \# ires \wedge$
$r = Bk^{mr} @ Oc^{Suc\ rs} @ Bk^{rn} \wedge$
$ml + mr > Suc\ 0)$

declare *wcode-on-right-moving-1.simps* [*simp del*]

declare *wcode-on-right-moving-1.simps*[*simp del*]

fun *wcode-goon-right-moving-1* :: *bin-inv-t*
where
wcode-goon-right-moving-1 ires rs $(l, r) =$
$(\exists\ ml\ mr\ ln\ rn.$
$l = Oc^{ml} @ Bk \# Bk \# Bk^{ln} @ Oc \# ires \wedge$
$r = Oc^{mr} @ Bk^{rn} \wedge$
$ml + mr = Suc\ rs)$

declare *wcode-goon-right-moving-1.simps*[*simp del*]

fun *wcode-backto-standard-pos-B* :: *bin-inv-t*
where
wcode-backto-standard-pos-B ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l = Bk \# Bk^{ln} @ Oc \# ires \wedge$
$r = Bk \# Oc^{(Suc\ (Suc\ rs))} @ Bk^{rn})$

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 $(l, r) =$
$(\exists\ ml\ mr\ ln\ rn.$

$l = Oc^{ml}$ @ 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 [*simp*]: <*0*::*nat*> = [*Oc*]
apply(*simp add*: *tape-of-nat-abv exponent-def tape-of-nat-list.simps*)
done

lemma *tape-of-Suc-nat*: <*Suc* (*a* ::*nat*)> = *replicate a Oc* @ [*Oc*, *Oc*]
apply(*simp add*: *tape-of-nat-abv exp-ind tape-of-nat-list.simps*)
apply(*simp only*: *exp-ind-def*[*THEN sym*])
apply(*simp only*: *exp-ind*, *simp*, *simp add*: *exponent-def*)
done

lemma [*simp*]: *length* (<*a*::*nat*>) = *Suc a*
apply(*simp add*: *tape-of-nat-abv tape-of-nat-list.simps*)
done

lemma [*simp*]: <[*a*::*nat*]> = <*a*>
apply(*simp add*: *tape-of-nat-abv tape-of-nl-abv exponent-def*
tape-of-nat-list.simps)
done

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

declare *exp-def*[*simp del*]

```
lemma bl-bin-nat-Suc:
  bl-bin (<Suc a>) = bl-bin (<a>) + 2^(Suc a)
apply(simp add: tape-of-nat-abv bin-wc-eq)
apply(simp add: bl2wc.simps)
done
lemma [simp]:  rev (a^aa) = a^aa
apply(simp add: exponent-def)
done

declare tape-of-nl-abv-cons[simp del]

lemma tape-of-nl-rev: rev (<lm::nat list>) = (<rev lm>)
apply(induct lm rule: list-tl-induct, simp)
apply(case-tac list = [], simp add: tape-of-nl-abv tape-of-nat-list.simps)
apply(simp add: tape-of-nat-list-butlast-last tape-of-nl-abv-cons)
done
lemma [simp]: a^(Suc 0) = [a]
by(simp add: exp-def)
lemma tape-of-nl-cons-app1: (<a # xs @ [b]>) = (Oc^(Suc a) @ Bk # (<xs@ [b]>))
apply(case-tac xs, simp add: tape-of-nl-abv tape-of-nat-list.simps)
apply(simp add: tape-of-nl-abv  tape-of-nat-list.simps)
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-abv)
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-nl-abv  tape-of-nat-list.simps)
  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(case-tac xs, simp add: tape-of-nl-abv  tape-of-nat-list.simps)
    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-nl-abv tape-of-nat-list.simps)
done
qed
qed

lemma [simp]: length (<aa # a # list>) = Suc (Suc aa) + length (<a # list>)
apply(simp add: tape-of-nl-abv tape-of-nat-list.simps)
done

lemma [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

lemma [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, simp)
apply(simp add: tape-of-nl-cons-app2 add-mult-distrib)
apply(simp add: tape-of-nl-abv tape-of-nat-list.simps)
done

lemma tape-of-nl-app-Suc: ((<list @ [Suc ab]>)) = (<list @ [ab]>) @ [Oc]
apply(induct list)
apply(simp add: tape-of-nl-abv tape-of-nat-list.simps exp-ind)
apply(case-tac list)
apply(simp-all add:tape-of-nl-abv tape-of-nat-list.simps exp-ind)
done

lemma [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 [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* :: *t-conf* ⇒ *nat*
  where
  wcode-double-case-state (*st*, *l*, *r*) =
   13 − *st*

fun *wcode-double-case-step* :: *t-conf* ⇒ *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* :: *t-conf* ⇒ *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* :: (*t-conf* × *t-conf*) *set*
  where *wcode-double-case-le* ≡ (*inv-image lex-pair wcode-double-case-measure*)

lemma [*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*)

term *fetch*

lemma [*simp*]: *fetch t-wcode-main* (*Suc 0*) *Bk* = (*L*, *Suc 0*)
apply(*simp add*: *t-wcode-main-def t-wcode-main-first-part-def*
fetch.simps nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main* (*Suc 0*) *Oc* = (*L*, *Suc* (*Suc 0*))
apply(*simp add*: *t-wcode-main-def t-wcode-main-first-part-def*
fetch.simps nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main* (*Suc* (*Suc 0*)) *Oc* = (*R*, *3*)
apply(*simp add*: *t-wcode-main-def t-wcode-main-first-part-def*
fetch.simps nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main* (*Suc* (*Suc* (*Suc 0*))) *Bk* = (*R*, *4*)
apply(*simp add*: *t-wcode-main-def t-wcode-main-first-part-def*
fetch.simps nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main* (*Suc* (*Suc* (*Suc 0*))) *Oc* = (*W0*, *3*)
apply(*simp add*: *t-wcode-main-def t-wcode-main-first-part-def*
fetch.simps nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 4 Bk* = (*R*, *4*)
apply(*simp add*: *t-wcode-main-def t-wcode-main-first-part-def*
fetch.simps nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 4 Oc* = (*R*, *5*)
apply(*simp add*: *t-wcode-main-def t-wcode-main-first-part-def*
fetch.simps nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 5 Oc* = (*R*, *5*)
apply(*simp add*: *t-wcode-main-def t-wcode-main-first-part-def*
fetch.simps nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 5 Bk* = (*W1*, *6*)
apply(*simp add*: *t-wcode-main-def t-wcode-main-first-part-def*
fetch.simps nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 6 Bk* = (*R*, *13*)
apply(*simp add*: *t-wcode-main-def t-wcode-main-first-part-def*

fetch.simps nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 6 Oc* = (*L*, *6*)
apply(*simp add*: *t-wcode-main-def t-wcode-main-first-part-def*
fetch.simps nth-of.simps)
done
lemma [*elim*]: Bk^{mr} = [] ⟹ *mr* = *0*
apply(*case-tac mr*, *auto simp*: *exponent-def*)
done

lemma [*simp*]: *wcode-on-left-moving-1 ires rs* (*b*, []) = *False*
apply(*simp add*: *wcode-on-left-moving-1.simps wcode-on-left-moving-1-B.simps*
wcode-on-left-moving-1-O.simps, *auto*)
done

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 [*simp*]: *wcode-on-left-moving-1 ires rs* (*b*, *r*) ⟹ *b* ≠ []
apply(*simp add*: *wcode-double-case-inv-simps*, *auto*)
done

lemma [*elim*]: ⟦*wcode-on-left-moving-1 ires rs* (*b*, *Bk* # *list*);
tl b = *aa* ∧ *hd b* # *Bk* # *list* = *ba*⟧ ⟹
wcode-on-left-moving-1 ires rs (*aa*, *ba*)
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(*case-tac ml*, *simp*)
apply(*rule-tac x* = *mr* − *Suc* (*Suc 0*) **in** *exI*, *rule-tac x* = *rn* **in** *exI*)
apply(*case-tac mr*, *simp*, *case-tac nat*, *simp*, *simp add*: *exp-ind*)
apply(*rule-tac disjI1*)
apply(*rule-tac x* = *nat* **in** *exI*, *rule-tac x* = *Suc mr* **in** *exI*, *rule-tac x* = *rn* **in** *exI*,
simp add: *exp-ind-def*)
apply(*erule-tac exE*)+
apply(*simp*)
done

lemma [*elim*]:
⟦*wcode-on-left-moving-1 ires rs* (*b*, *Oc* # *list*); *tl b* = *aa* ∧ *hd b* # *Oc* # *list* = *ba*⟧
⟹ *wcode-on-checking-1 ires rs* (*aa*, *ba*)
apply(*simp only*: *wcode-double-case-inv-simps*)
apply(*erule-tac disjE*)
apply(*erule-tac* [!] *exE*)+
apply(*case-tac mr*, *simp*, *simp add*: *exp-ind-def*)
apply(*rule-tac x* = *ln* **in** *exI*, *rule-tac x* = *rn* **in** *exI*, *simp*)
done

lemma [*simp*]: *wcode-on-checking-1 ires rs* (*b*, []) = *False*
apply(*auto simp*: *wcode-double-case-inv-simps*)
done

lemma [*simp*]: *wcode-on-checking-1 ires rs* (*b*, *Bk* # *list*) = *False*
apply(*auto simp*: *wcode-double-case-inv-simps*)
done

lemma [*elim*]: ⟦*wcode-on-checking-1 ires rs* (*b*, *Oc* # *ba*);*Oc* # *b* = *aa* ∧ *list* = *ba*⟧
⟹ *wcode-erase1 ires rs* (*aa*, *ba*)
apply(*simp only*: *wcode-double-case-inv-simps*)
apply(*erule-tac exE*)+
apply(*rule-tac x* = *ln* **in** *exI*, *rule-tac x* = *rn* **in** *exI*, *simp*)
done

lemma [*simp*]: *wcode-on-checking-1 ires rs* (*b*, []) = *False*
apply(*simp add*: *wcode-double-case-inv-simps*)
done

lemma [*simp*]: *wcode-on-checking-1 ires rs* ([], *Bk* # *list*) = *False*
apply(*simp add*: *wcode-double-case-inv-simps*)
done

lemma [*simp*]: *wcode-erase1 ires rs* (*b*, []) = *False*
apply(*simp add*: *wcode-double-case-inv-simps*)
done

lemma [*simp*]: *wcode-on-right-moving-1 ires rs* (*b*, []) = *False*
apply(*simp add*: *wcode-double-case-inv-simps exp-ind-def*)
done

lemma [*simp*]: *wcode-on-right-moving-1 ires rs* (*b*, []) = *False*
apply(*simp add*: *wcode-double-case-inv-simps exp-ind-def*)
done

lemma [*elim*]: ⟦*wcode-on-right-moving-1 ires rs* (*b*, *Bk* # *ba*); *Bk* # *b* = *aa* ∧ *list* = *b*⟧ ⟹
wcode-on-right-moving-1 ires rs (*aa*, *ba*)
apply(*simp only*: *wcode-double-case-inv-simps*)
apply(*erule-tac exE*)+
apply(*rule-tac x* = *Suc ml* **in** *exI*, *rule-tac x* = *mr* − *Suc 0* **in** *exI*,
rule-tac x = *rn* **in** *exI*)
apply(*simp add*: *exp-ind-def*)
apply(*case-tac mr*, *simp*, *simp add*: *exp-ind-def*)
done

lemma [*elim*]:
⟦*wcode-on-right-moving-1 ires rs* (*b*, *Oc* # *ba*); *Oc* # *b* = *aa* ∧ *list* = *ba*⟧
⟹ *wcode-goon-right-moving-1 ires rs* (*aa*, *ba*)
apply(*simp only*: *wcode-double-case-inv-simps*)
apply(*erule-tac exE*)+
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 add*: *exp-ind-def*)
apply(*case-tac ml*, *simp*, *case-tac nat*, *simp*, *simp*)
apply(*simp add*: *exp-ind-def*)
done

lemma [*simp*]:
wcode-on-right-moving-1 ires rs (*b*, []) ⟹ *False*
apply(*simp add*: *wcode-double-case-inv-simps exponent-def*)
done

lemma [*elim*]: ⟦*wcode-erase1 ires rs* (*b*, *Bk* # *ba*); *Bk* # *b* = *aa* ∧ *list* = *ba*; *c* = *Bk* # *ba*⟧
⟹ *wcode-on-right-moving-1 ires rs* (*aa*, *ba*)
apply(*simp only*: *wcode-double-case-inv-simps*)
apply(*erule-tac exE*)+
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*)
done

lemma [*elim*]: ⟦*wcode-erase1 ires rs* (*aa*, *Oc* # *list*); *b* = *aa* ∧ *Bk* # *list* = *ba*⟧ ⟹
wcode-erase1 ires rs (*aa*, *ba*)
apply(*simp only*: *wcode-double-case-inv-simps*)
apply(*erule-tac exE*)+
apply(*rule-tac x* = *ln* **in** *exI*, *rule-tac x* = *rn* **in** *exI*, *auto*)
done

lemma [*elim*]: ⟦*wcode-goon-right-moving-1 ires rs* (*aa*, []); *b* = *aa* ∧ [*Oc*] = *ba*⟧
⟹ *wcode-backto-standard-pos ires rs* (*aa*, *ba*)
apply(*simp only*: *wcode-double-case-inv-simps*)

apply(*erule-tac exE*)+
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*)
apply(*case-tac mr*, *simp-all add*: *exponent-def*)
done

lemma [*elim*]:
⟦*wcode-goon-right-moving-1 ires rs* (*aa*, *Bk* # *list*); *b* = *aa* ∧ *Oc* # *list* = *ba*⟧
⟹ *wcode-backto-standard-pos ires rs* (*aa*, *ba*)
apply(*simp only*: *wcode-double-case-inv-simps*)
apply(*erule-tac exE*)+
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* − *Suc 0* **in** *exI*, *simp*)
apply(*case-tac mr*, *simp*, *case-tac rn*, *simp*, *simp-all add*: *exp-ind-def*)
done

lemma [*elim*]: ⟦*wcode-goon-right-moving-1 ires rs* (*b*, *Oc* # *ba*); *Oc* # *b* = *aa* ∧ *list* = *ba*⟧
⟹ *wcode-goon-right-moving-1 ires rs* (*aa*, *ba*)
apply(*simp only*: *wcode-double-case-inv-simps*)
apply(*erule-tac exE*)+
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 add*: *exp-ind-def*)
apply(*case-tac mr*, *simp*, *case-tac rn*, *simp-all add*: *exp-ind-def*)
done

lemma [*elim*]: ⟦*wcode-backto-standard-pos ires rs* (*b*, []); *Bk* # *b* = *aa*⟧ ⟹ *False*
apply(*auto simp*: *wcode-double-case-inv-simps wcode-backto-standard-pos-O.simps wcode-backto-standard-pos-B.simps*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*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*)+
apply(*rule-tac x* = *ln* **in** *exI*, *rule-tac x* = *rn* **in** *exI*, *simp*)
apply(*auto*)
apply(*case-tac* [!] *mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*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 [*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)
apply(*case-tac mr*, *simp*, *simp add*: *exp-ind-def*)
done

lemma [*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(*case-tac ml*, *simp*)
apply(*rule-tac disjI1*, *rule-tac conjI*)
apply(*rule-tac x* = *ln* **in** *exI*, *simp*, *rule-tac x* = *rn* **in** *exI*, *simp*)
apply(*rule-tac disjI2*)
apply(*rule-tac x* = *nat* **in** *exI*, *rule-tac x* = *Suc mr* **in** *exI*, *rule-tac x* = *ln* **in** *exI*,
rule-tac x = *rn* **in** *exI*, *simp*)
apply(*simp add*: *exp-ind-def*)
done

declare *new-tape.simps*[*simp del*] *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*. *steps* (*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*. *steps* (*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 add*: *tstep-red*)

fix *na a b c*
show $a \neq 13 \wedge$ *wcode-double-case-inv a ires rs* $(b, c) \longrightarrow$
(*case tstep* (a, b, c) *t-wcode-main of* $(st, x) \Rightarrow$
wcode-double-case-inv st ires rs x$) \wedge$
(*tstep* (a, b, c) *t-wcode-main*, $a, b, c) \in$ *wcode-double-case-le*
apply(*rule-tac impI*, *simp add*: *wcode-double-case-inv.simps*)
apply(*auto split*: *if-splits simp*: *tstep.simps*,
case-tac [!] *c*, *simp-all*, *case-tac* [!] (*c*::*block list*)!*0*)
apply(*simp-all add*: *new-tape.simps 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 add*: *exp-ind-def*)
apply(*rule-tac* x = *Suc 0* **in** *exI*, *simp add*: *exp-ind-def*)
apply(*auto*)
done
next
show $\neg$ *?P* (*?f 0*)
apply(*simp add*: *steps.simps*)
done
qed
thus *let* $P = \lambda(st, l, r).\ st = 13$;
$Q = \lambda(st, l, r)$. *wcode-double-case-inv st ires rs* (l, r);
f = *steps* (*Suc 0*, *Bk* # Bk^{m} @ *Oc* # *Oc* # *ires*, *Bk* # $Oc^{Suc\ rs}$ @ Bk^{n})
t-wcode-main
in $\exists n.\ P\ (f\ n) \wedge Q\ (f\ n)$
apply(*simp add*: *Let-def*)
done
qed

lemma [*elim*]: *t-ncorrect tp*
$\Longrightarrow$ *t-ncorrect* (*abacus.tshift tp a*)
apply(*simp add*: *t-ncorrect.simps shift-length*)
done

lemma *tshift-fetch*: ⟦ *fetch tp a b* = (*aa*, *st'*); $0 < st'$⟧
$\Longrightarrow$ *fetch* (*abacus.tshift tp* (*length tp1 div 2*)) *a b*
= (*aa*, *st'* + *length tp1 div 2*)
apply(*subgoal-tac* $a > 0$)
apply(*auto simp*: *fetch.simps nth-of.simps shift-length nth-map*
tshift.simps split: *block.splits if-splits*)
done

lemma *t-steps-steps-eq*: ⟦*steps* (*st*, *l*, *r*) *tp stp* = (*st*′, *l*′, *r*′);
0 < *st*′;
0 < *st* ∧ *st* ≤ *length tp div 2*;
t-ncorrect tp1;
t-ncorrect tp⟧
⟹ *t-steps* (*st* + *length tp1 div 2*, *l*, *r*) (*tshift tp* (*length tp1 div 2*),
length tp1 div 2) *stp*
= (*st*′ + *length tp1 div 2*, *l*′, *r*′)
apply(*induct stp arbitrary*: *st*′ *l*′ *r*′, *simp add*: *steps.simps t-steps.simps*,
simp add: *tstep-red stepn*)
apply(*case-tac* (*steps* (*st*, *l*, *r*) *tp stp*), *simp*)
proof −
fix *stp st*′ *l*′ *r*′ *a b c*
assume *ind*: ⋀*st*′ *l*′ *r*′.
⟦*a* = *st*′ ∧ *b* = *l*′ ∧ *c* = *r*′; *0* < *st*′⟧
⟹ *t-steps* (*st* + *length tp1 div 2*, *l*, *r*)
(*abacus.tshift tp* (*length tp1 div 2*), *length tp1 div 2*) *stp* =
(*st*′ + *length tp1 div 2*, *l*′, *r*′)
and *h*: *tstep* (*a*, *b*, *c*) *tp* = (*st*′, *l*′, *r*′) *0* < *st*′ *t-ncorrect tp1* *t-ncorrect tp*
have *k*: *t-steps* (*st* + *length tp1 div 2*, *l*, *r*) (*abacus.tshift tp* (*length tp1 div 2*),
length tp1 div 2) *stp* = (*a* + *length tp1 div 2*, *b*, *c*)
apply(*rule-tac ind*, *simp*)
using *h*
apply(*case-tac a*, *simp-all add*: *tstep.simps fetch.simps*)
done
from *h* **and** *this* **show** *t-step* (*t-steps* (*st* + *length tp1 div 2*, *l*, *r*) (*abacus.tshift tp* (*length tp1 div 2*), *length tp1 div 2*) *stp*)
(*abacus.tshift tp* (*length tp1 div 2*), *length tp1 div 2*) =
(*st*′ + *length tp1 div 2*, *l*′, *r*′)
apply(*simp add*: *k*)
apply(*simp add*: *tstep.simps t-step.simps*)
apply(*case-tac fetch tp a* (*case c of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*), *simp*)
apply(*subgoal-tac fetch* (*abacus.tshift tp* (*length tp1 div 2*)) *a*
(*case c of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*) = (*aa*, *st*′ + *length tp1 div 2*), *simp*)
apply(*simp add*: *tshift-fetch*)
done
qed

lemma *t-tshift-lemma*: ⟦ *steps* (*st*, *l*, *r*) *tp stp* = (*st*′, *l*′, *r*′);
st′ ≠ *0*;
stp > *0*;
0 < *st* ∧ *st* ≤ *length tp div 2*;
t-ncorrect tp1;
t-ncorrect tp;
t-ncorrect tp2
⟧
⟹ ∃ *stp*>*0*. *steps* (*st* + *length tp1 div 2*, *l*, *r*) (*tp1* @ *tshift tp* (*length tp1*

```
div 2) @ tp2) stp
                  = (st' + length tp1 div 2, l', r')
proof -
  assume h: steps (st, l, r) tp stp = (st', l', r')
    st' ≠ 0 stp > 0
    0 < st ∧ st ≤ length tp div 2
    t-ncorrect tp1
    t-ncorrect tp
    t-ncorrect tp2
  from h have
    ∃ stp>0. t-steps (st + length tp1 div 2, l, r) (tp1 @ abacus.tshift tp (length tp1
div 2) @ tp2, 0) stp =
                            (st' + length tp1 div 2, l', r')
    apply(rule-tac stp = stp in turing-shift, simp-all add: shift-length)
    apply(simp add: t-steps-steps-eq)
    apply(simp add: t-ncorrect.simps shift-length)
    done
  thus ∃ stp>0. steps (st + length tp1 div 2, l, r) (tp1 @ tshift tp (length tp1 div
2) @ tp2) stp
                  = (st' + length tp1 div 2, l', r')
    apply(erule-tac exE)
    apply(rule-tac x = stp in exI, simp)
    apply(subgoal-tac length (tp1 @ abacus.tshift tp (length tp1 div 2) @ tp2) mod
2 = 0)
    apply(simp only: steps-eq)
    using h
    apply(auto simp: t-ncorrect.simps shift-length)
    apply arith
    done
qed


lemma t-twice-len-ge: Suc 0 ≤ length t-twice div 2
apply(simp add: t-twice-def tMp.simps shift-length)
done

lemma [intro]: rec-calc-rel (recf.id (Suc 0) 0) [rs] rs
  apply(rule-tac calc-id, simp-all)
  done

lemma [intro]: rec-calc-rel (constn 2) [rs] 2
using prime-rel-exec-eq[of constn 2 [rs] 2]
apply(subgoal-tac primerec (constn 2) 1, auto)
done

lemma  [intro]: rec-calc-rel rec-mult [rs, 2] (2 * rs)
using prime-rel-exec-eq[of rec-mult [rs, 2]  2*rs]
apply(subgoal-tac primerec rec-mult (Suc (Suc 0)), auto)
done
```

lemma *t-twice-correct*: $\exists$ *stp ln rn. steps* (*Suc 0*, *Bk* # *Bk* # *ires*, $Oc^{Suc\ rs}$ @ Bk^{n})
(*tm-of abc-twice* @ *tMp* (*Suc 0*) (*start-of twice-ly* (*length abc-twice*) − *Suc 0*)) *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 $\exists$ *stp m l. steps* (*Suc 0*, *Bk* # *Bk* # *ires*, <[*rs*]> @ Bk^{n}) (*tm-of abc-twice* @ *tMp* (*Suc 0*)
(*start-of twice-ly* (*length abc-twice*) − *1*)) *stp* = (*0*, Bk^{m} @ *Bk* # *Bk* # *ires*, $Oc^{Suc\ (2*rs)}$ @ Bk^{l})
proof(*rule-tac t-compiled-by-rec*)
show *rec-ci rec-twice* = (*a*, *b*, *c*) **by** (*simp add*: *h*)
next
show *rec-calc-rel rec-twice* [*rs*] (*2* ∗ *rs*)
apply(*simp add*: *rec-twice-def*)
apply(*rule-tac rs* = [*rs*, *2*] **in** *calc-cn*, *simp-all*)
apply(*rule-tac allI*, *case-tac k*, *auto*)
done
next
show *length* [*rs*] = *Suc 0* **by** *simp*
next
show *layout-of* (*a* [+] *dummy-abc* (*Suc 0*)) = *layout-of* (*a* [+] *dummy-abc* (*Suc 0*))
by *simp*
next
show *start-of twice-ly* (*length abc-twice*) =
start-of (*layout-of* (*a* [+] *dummy-abc* (*Suc 0*))) (*length* (*a* [+] *dummy-abc* (*Suc 0*)))
using *h*
apply(*simp add*: *twice-ly-def abc-twice-def*)
done
next
show *tm-of abc-twice* = *tm-of* (*a* [+] *dummy-abc* (*Suc 0*))
using *h*
apply(*simp add*: *abc-twice-def*)
done
qed
thus $\exists$ *stp ln rn. steps* (*Suc 0*, *Bk* # *Bk* # *ires*, $Oc^{Suc\ rs}$ @ Bk^{n})
(*tm-of abc-twice* @ *tMp* (*Suc 0*) (*start-of twice-ly* (*length abc-twice*) − *Suc 0*)) *stp* =
(*0*, Bk^{ln} @ *Bk* # *Bk* # *ires*, $Oc^{Suc\ (2\ *\ rs)}$ @ Bk^{rn})
apply(*simp add*: *tape-of-nl-abv tape-of-nat-list.simps*)
done
qed

lemma *change-termi-state-fetch*: ⟦*fetch ap a b* = (*aa*, *st*); *st* > *0*⟧
$\Longrightarrow$ *fetch* (*change-termi-state ap*) *a b* = (*aa*, *st*)

apply(*case-tac b*, *auto simp*: *fetch.simps nth-of.simps change-termi-state.simps nth-map*
split: *if-splits block.splits*)
done

lemma *change-termi-state-exec-in-range*:
⟦*steps* (*st*, *l*, *r*) *ap stp* = (*st'*, *l'*, *r'*); *st'* ≠ *0*⟧
⟹ *steps* (*st*, *l*, *r*) (*change-termi-state ap*) *stp* = (*st'*, *l'*, *r'*)
proof(*induct stp arbitrary*: *st l r st' l' r'*, *simp add*: *steps.simps*)
fix *stp st l r st' l' r'*
assume *ind*: ⋀*st l r st' l' r'*.
⟦*steps* (*st*, *l*, *r*) *ap stp* = (*st'*, *l'*, *r'*); *st'* ≠ *0*⟧ ⟹
steps (*st*, *l*, *r*) (*change-termi-state ap*) *stp* = (*st'*, *l'*, *r'*)
and *h*: *steps* (*st*, *l*, *r*) *ap* (*Suc stp*) = (*st'*, *l'*, *r'*) *st'* ≠ *0*
from *h* **show** *steps* (*st*, *l*, *r*) (*change-termi-state ap*) (*Suc stp*) = (*st'*, *l'*, *r'*)
proof(*simp add*: *tstep-red*, *case-tac steps* (*st*, *l*, *r*) *ap stp*, *simp*)
fix *a b c*
assume *g*: *steps* (*st*, *l*, *r*) *ap stp* = (*a*, *b*, *c*)
tstep (*a*, *b*, *c*) *ap* = (*st'*, *l'*, *r'*) *0* < *st'*
hence *steps* (*st*, *l*, *r*) (*change-termi-state ap*) *stp* = (*a*, *b*, *c*)
apply(*rule-tac ind*, *simp*)
apply(*case-tac a*, *simp-all add*: *tstep-0*)
done
from *g* **and** *this* **show** *tstep* (*steps* (*st*, *l*, *r*) (*change-termi-state ap*) *stp*)
(*change-termi-state ap*) = (*st'*, *l'*, *r'*)
apply(*simp add*: *tstep.simps*)
apply(*case-tac fetch ap a* (*case c of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*), *simp*)
apply(*subgoal-tac fetch* (*change-termi-state ap*) *a* (*case c of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*)
= (*aa*, *st'*), *simp*)
apply(*simp add*: *change-termi-state-fetch*)
done
qed
qed

lemma *change-termi-state-fetch0*:
⟦*0* < *a*; *a* ≤ *length ap div 2*; *t-correct ap*; *fetch ap a b* = (*aa*, *0*)⟧
⟹ *fetch* (*change-termi-state ap*) *a b* = (*aa*, *Suc* (*length ap div 2*))
apply(*case-tac b*, *auto simp*: *fetch.simps nth-of.simps change-termi-state.simps nth-map*
split: *if-splits block.splits*)
done

lemma *turing-change-termi-state*:
⟦*steps* (*Suc 0*, *l*, *r*) *ap stp* = (*0*, *l'*, *r'*); *t-correct ap*⟧
⟹ ∃ *stp*. *steps* (*Suc 0*, *l*, *r*) (*change-termi-state ap*) *stp* =
(*Suc* (*length ap div 2*), *l'*, *r'*)
apply(*drule first-halt-point*)
apply(*erule-tac exE*)

apply(*rule-tac x = Suc stp* **in** *exI*, *simp add*: *tstep-red*)
apply(*case-tac steps* (*Suc 0*, *l*, *r*) *ap stp*)
apply(*simp add*: *isS0-def change-termi-state-exec-in-range*)
apply(*subgoal-tac steps* (*Suc 0*, *l*, *r*) (*change-termi-state ap*) *stp* = (*a*, *b*, *c*), *simp*)
apply(*simp add*: *tstep.simps*)
apply(*case-tac fetch ap a* (*case c of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*), *simp*)
apply(*subgoal-tac fetch* (*change-termi-state ap*) *a*
(*case c of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*) = (*aa*, *Suc* (*length ap div 2*)), *simp*)
apply(*rule-tac ap* = *ap* **in** *change-termi-state-fetch0*, *simp-all*)
apply(*rule-tac tp* = (*l*, *r*) **and** *l* = *b* **and** *r* = *c* **and** *stp* = *stp* **and** *A* = *ap* **in** *s-keep*, *simp-all*)
apply(*simp add*: *change-termi-state-exec-in-range*)
done

lemma *t-twice-change-term-state*:
$\exists$ *stp ln rn*. *steps* (*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})
using *t-twice-correct*[*of ires rs n*]
apply(*erule-tac exE*)
apply(*erule-tac exE*)
apply(*erule-tac exE*)
proof(*drule-tac turing-change-termi-state*)
fix *stp ln rn*
show *t-correct* (*tm-of abc-twice* @ *tMp* (*Suc 0*) (*start-of twice-ly* (*length abc-twice*) − *Suc 0*))
apply(*rule-tac t-compiled-correct*, *simp-all*)
apply(*simp add*: *twice-ly-def*)
done
next
fix *stp ln rn*
show $\exists$*stp*. *steps* (*Suc 0*, *Bk* # *Bk* # *ires*, $Oc^{Suc\ rs}$ @ Bk^{n})
(*change-termi-state* (*tm-of abc-twice* @ *tMp* (*Suc 0*)
(*start-of twice-ly* (*length abc-twice*) − *Suc 0*))) *stp* =
(*Suc* (*length* (*tm-of abc-twice* @ *tMp* (*Suc 0*) (*start-of twice-ly* (*length abc-twice*) − *Suc 0*)) *div 2*),
Bk^{ln} @ *Bk* # *Bk* # *ires*, $Oc^{Suc\ (2\ *\ rs)}$ @ Bk^{rn}) ⟹
$\exists$*stp ln rn*. *steps* (*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})
apply(*erule-tac exE*)
apply(*simp add*: *t-twice-len-def t-twice-def*)
apply(*rule-tac x* = *stp* **in** *exI*, *rule-tac x* = *ln* **in** *exI*, *rule-tac x* = *rn* **in** *exI*, *simp*)
done
qed

lemma *t-twice-append-pre*:
steps (*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})
⟹ $\exists$ *stp*>*0*. *steps* (*Suc 0* + *length t-wcode-main-first-part div 2*, *Bk* # *Bk* #

ires, $Oc^{Suc\ rs}$ @ Bk^n)
(*t-wcode-main-first-part* @ *tshift t-twice* (*length t-wcode-main-first-part div 2*) @
([(*L*, *1*), (*L*, *1*)] @ *tshift t-fourtimes* (*t-twice-len* + *13*) @ [(*L*, *1*), (*L*, *1*)]))
stp
= (*Suc* (*t-twice-len*) + *length t-wcode-main-first-part div 2*, Bk^{ln} @ *Bk* # *Bk* # *ires*, $Oc^{Suc\ (2\ *\ rs)}$ @ Bk^{rn})
proof(*rule-tac t-tshift-lemma*, *simp-all add*: *t-twice-len-ge*)
assume *steps* (*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})
thus *0* < *stp*
apply(*case-tac stp*, *simp add*: *steps.simps t-twice-len-ge t-twice-len-def*)
using *t-twice-len-ge*
apply(*simp*, *simp*)
done
next
show *t-ncorrect t-wcode-main-first-part*
apply(*simp add*: *t-ncorrect.simps t-wcode-main-first-part-def*)
done
next
show *t-ncorrect t-twice*
using *length-tm-even*[*of abc-twice*]
apply(*auto simp*: *t-ncorrect.simps t-twice-def*)
apply(*arith*)
done
next
show *t-ncorrect* ((*L*, *Suc 0*) # (*L*, *Suc 0*) #
abacus.tshift t-fourtimes (*t-twice-len* + *13*) @ [(*L*, *Suc 0*), (*L*, *Suc 0*)])
using *length-tm-even*[*of abc-fourtimes*]
apply(*simp add*: *t-ncorrect.simps shift-length t-fourtimes-def*)
apply *arith*
done
qed

lemma *t-twice-append*:
∃ *stp ln rn*. *steps* (*Suc 0* + *length t-wcode-main-first-part div 2*, *Bk* # *Bk* # *ires*, $Oc^{Suc\ rs}$ @ Bk^n)
(*t-wcode-main-first-part* @ *tshift t-twice* (*length t-wcode-main-first-part div 2*) @
([(*L*, *1*), (*L*, *1*)] @ *tshift 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})
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(*erule-tac exE*)

apply(*rule-tac x = stpa* **in** *exI*, *rule-tac x = ln* **in** *exI*, *rule-tac x = rn* **in** *exI*)
apply(*simp*)
done

lemma [*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 shift-length t-twice-len-def, *auto*)
apply(*simp add*: *t-twice-def*)
apply(*subgoal-tac length* (*tm-of abc-twice*) *mod 2* = *0*)
apply *arith*
apply(*rule-tac tm-even*)
done

lemma *wcode-jump1*:
∃ *stp ln rn*. *steps* (*Suc* (*t-twice-len*) + *length t-wcode-main-first-part div 2*,
Bk^m @ *Bk* # *Bk* # *ires*, $Oc^{Suc\ (2\ *\ rs)}$ @ Bk^n)
t-wcode-main stp
= (*Suc 0*, Bk^{ln} @ *Bk* # *ires*, *Bk* # $Oc^{Suc\ (2\ *\ rs)}$ @ Bk^{rn})
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 tstep.simps exp-ind-def new-tape.simps*)
apply(*case-tac m*, *simp*, *simp add*: *exp-ind-def*)
apply(*simp add*: *exp-ind-def*[*THEN sym*] *exp-ind*[*THEN sym*])
done

lemma *wcode-main-first-part-len*:
length t-wcode-main-first-part = *24*
apply(*simp add*: *t-wcode-main-first-part-def*)
done

lemma *wcode-double-case*:
shows ∃ *stp ln rn*. *steps* (*Suc 0*, *Bk* # Bk^m @ *Oc* # *Oc* # *ires*, *Bk* # $Oc^{Suc\ rs}$ @ Bk^n) *t-wcode-main stp* =
(*Suc 0*, *Bk* # Bk^{ln} @ *Oc* # *ires*, *Bk* # $Oc^{Suc\ (2\ *\ rs\ +\ 2)}$ @ Bk^{rn})
proof −
have ∃ *stp ln rn*. *steps* (*Suc 0*, *Bk* # Bk^m @ *Oc* # *Oc* # *ires*, *Bk* # $Oc^{Suc\ rs}$ @ Bk^n) *t-wcode-main stp* =
(*13*, *Bk* # *Bk* # Bk^{ln} @ *Oc* # *ires*, $Oc^{Suc\ (Suc\ rs)}$ @ Bk^{rn})
using *wcode-double-case-first-correctness*[*of ires rs m n*]
apply(*simp*)
apply(*erule-tac exE*)
apply(*case-tac steps* (*Suc 0*, *Bk* # Bk^m @ *Oc* # *Oc* # *ires*,
Bk # $Oc^{Suc\ rs}$ @ Bk^n) *t-wcode-main na*,
auto simp: *wcode-double-case-inv.simps*
wcode-before-double.simps)
apply(*rule-tac x = na* **in** *exI*, *rule-tac x = ln* **in** *exI*, *rule-tac x = rn* **in** *exI*)

apply(*simp*)
done
from *this* **obtain** *stpa lna rna* **where** *stp1*:
steps $(Suc\ 0,\ Bk\ \#\ Bk^{m}\ @\ Oc\ \#\ Oc\ \#\ ires,\ Bk\ \#\ Oc^{Suc\ rs}\ @\ Bk^{n})$ *t-wcode-main stpa* =
$(13,\ Bk\ \#\ Bk\ \#\ Bk^{lna}\ @\ Oc\ \#\ ires,\ Oc^{Suc\ (Suc\ rs)}\ @\ Bk^{rna})$ **by** *blast*
have $\exists$ *stp ln rn. steps* $(13,\ Bk\ \#\ Bk\ \#\ Bk^{lna}\ @\ Oc\ \#\ ires,\ Oc^{Suc\ (Suc\ rs)}\ @\ Bk^{rna})$ *t-wcode-main stp* =
$(13 + t\text{-}twice\text{-}len,\ Bk\ \#\ Bk\ \#\ Bk^{ln}\ @\ Oc\ \#\ ires,\ Oc^{Suc\ (Suc\ (Suc\ (2\ *rs)))}\ @\ Bk^{rn})$
using *t-twice-append*[*of* Bk^{lna} @ *Oc* # *ires Suc rs rna*]
apply(*erule-tac exE*)
apply(*erule-tac exE*)
apply(*erule-tac exE*)
apply(*simp add*: *wcode-main-first-part-len*)
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*: *exp-ind-def*[*THEN sym*] *exp-add*[*THEN sym*])
done
from *this* **obtain** *stpb lnb rnb* **where** *stp2*:
steps $(13,\ Bk\ \#\ Bk\ \#\ Bk^{lna}\ @\ Oc\ \#\ ires,\ Oc^{Suc\ (Suc\ rs)}\ @\ Bk^{rna})$ *t-wcode-main stpb* =
$(13 + t\text{-}twice\text{-}len,\ Bk\ \#\ Bk\ \#\ Bk^{lnb}\ @\ Oc\ \#\ ires,\ Oc^{Suc\ (Suc\ (Suc\ (2\ *rs)))}\ @\ Bk^{rnb})$ **by** *blast*
have $\exists$ *stp ln rn. steps* $(13 + t\text{-}twice\text{-}len,\ Bk\ \#\ Bk\ \#\ Bk^{lnb}\ @\ Oc\ \#\ ires,$
$Oc^{Suc\ (Suc\ (Suc\ (2\ *rs)))}\ @\ Bk^{rnb})$ *t-wcode-main stp* =
$(Suc\ 0,\ Bk\ \#\ Bk^{ln}\ @\ Oc\ \#\ ires,\ Bk\ \#\ Oc^{Suc\ (Suc\ (Suc\ (2\ *rs)))}\ @\ Bk^{rn})$
using *wcode-jump1*[*of lnb Oc* # *ires Suc rs rnb*]
apply(*erule-tac exE*)
apply(*erule-tac exE*)
apply(*erule-tac exE*)
apply(*rule-tac x* = *stp* **in** *exI*,
rule-tac x = *ln* **in** *exI*,
rule-tac x = *rn* **in** *exI*, *simp add*:*wcode-main-first-part-len t-wcode-main-def*)
apply(*subgoal-tac* Bk^{lnb} @ *Bk* # *Bk* # *Oc* # *ires* = *Bk* # *Bk* # Bk^{lnb} @ *Oc* # *ires*, *simp*)
apply(*simp add*: *exp-ind-def*[*THEN sym*] *exp-ind*[*THEN sym*])
apply(*simp*)
apply(*case-tac lnb*, *simp*, *simp add*: *exp-ind-def*[*THEN sym*] *exp-ind*)
done
from *this* **obtain** *stpc lnc rnc* **where** *stp3*:
steps $(13 + t\text{-}twice\text{-}len,\ Bk\ \#\ Bk\ \#\ Bk^{lnb}\ @\ Oc\ \#\ ires,$
$Oc^{Suc\ (Suc\ (Suc\ (2\ *rs)))}\ @\ Bk^{rnb})$ *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* **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 add*: *steps-add*)
done
qed

fun *wcode-on-left-moving-2-B* :: *bin-inv-t*
where
wcode-on-left-moving-2-B ires rs $(l, r) =$
$(\exists\ ml\ mr\ rn.\ l = Bk^{ml}\ @\ Oc\ \#\ Bk\ \#\ Oc\ \#\ ires\ \wedge$
$r = Bk^{mr}\ @\ Oc^{Suc\ rs}\ @\ Bk^{rn}\ \wedge$
$ml + mr > Suc\ 0 \wedge mr > 0)$

fun *wcode-on-left-moving-2-O* :: *bin-inv-t*
where
wcode-on-left-moving-2-O ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l = Bk\ \#\ Oc\ \#\ ires\ \wedge$
$r = Oc\ \#\ Bk^{ln}\ @\ Bk\ \#\ Bk\ \#\ Oc^{Suc\ rs}\ @\ Bk^{rn})$

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* $(l, r) \vee$
wcode-on-left-moving-2-O ires rs (l, r))

fun *wcode-on-checking-2* :: *bin-inv-t*
where
wcode-on-checking-2 ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l = Oc\#ires\ \wedge$
$r = Bk\ \#\ Oc\ \#\ Bk^{ln}\ @\ Bk\ \#\ Bk\ \#\ Oc^{Suc\ rs}\ @\ Bk^{rn})$

fun *wcode-goon-checking* :: *bin-inv-t*
where
wcode-goon-checking ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l = ires\ \wedge$
$r = Oc\ \#\ Bk\ \#\ Oc\ \#\ Bk^{ln}\ @\ Bk\ \#\ Bk\ \#\ Oc^{Suc\ rs}\ @\ Bk^{rn})$

fun *wcode-right-move* :: *bin-inv-t*
where
wcode-right-move ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l = Oc\ \#\ ires\ \wedge$
$r = Bk\ \#\ Oc\ \#\ Bk^{ln}\ @\ Bk\ \#\ Bk\ \#\ Oc^{Suc\ rs}\ @\ Bk^{rn})$

fun *wcode-erase2* :: *bin-inv-t*
where
wcode-erase2 ires rs $(l, r) =$
$(\exists\ ln\ rn.\ l = Bk\ \#\ Oc\ \#\ ires\ \wedge$

$tl\ r = Bk^{ln}$ @ Bk # Bk # $Oc^{Suc\ rs}$ @ Bk^{rn})

fun *wcode-on-right-moving-2* :: *bin-inv-t*
where
wcode-on-right-moving-2 ires rs $(l, r) =$
($\exists$ *ml mr rn*. $l = Bk^{ml}$ @ *Oc* # *ires* $\wedge$
$r = Bk^{mr}$ @ $Oc^{Suc\ rs}$ @ Bk^{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*. $l = Oc^{ml}$ @ *Bk* # *Bk* # Bk^{ln} @ *Oc* # *ires* $\wedge$
$r = Oc^{mr}$ @ Bk^{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^{ln} @ *Oc* # *ires* $\wedge$
$r = Bk$ # $Oc^{Suc\ (Suc\ rs)}$ @ Bk^{rn})

fun *wcode-backto-standard-pos-2-O* :: *bin-inv-t*
where
wcode-backto-standard-pos-2-O ires rs $(l, r) =$
($\exists$ *ml mr ln rn*. $l = Oc^{ml}$ @ *Bk* # *Bk* # Bk^{ln} @ *Oc* # *ires* $\wedge$
$r = Oc^{mr}$ @ Bk^{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^{ln} @ *Oc* # *ires* $\wedge$
$r = Oc^{Suc\ (Suc\ rs)}$ @ Bk^{rn})

declare *wcode-on-left-moving-2-B.simps*[*simp del*] *wcode-on-left-moving-2.simps*[*simp del*]
wcode-on-left-moving-2-O.simps[*simp del*] *wcode-on-checking-2.simps*[*simp del*]
wcode-goon-checking.simps[*simp del*] *wcode-right-move.simps*[*simp del*]
wcode-erase2.simps[*simp del*]
wcode-on-right-moving-2.simps[*simp del*] *wcode-goon-right-moving-2.simps*[*simp del*]
wcode-backto-standard-pos-2-B.simps[*simp del*] *wcode-backto-standard-pos-2-O.simps*[*simp*

del]
wcode-backto-standard-pos-2.simps[*simp del*]

lemmas *wcode-fourtimes-invs* =
wcode-on-left-moving-2-B.simps wcode-on-left-moving-2.simps
wcode-on-left-moving-2-O.simps wcode-on-checking-2.simps
wcode-goon-checking.simps wcode-right-move.simps
wcode-erase2.simps
wcode-on-right-moving-2.simps wcode-goon-right-moving-2.simps
wcode-backto-standard-pos-2-B.simps wcode-backto-standard-pos-2-O.simps
wcode-backto-standard-pos-2.simps

fun *wcode-fourtimes-case-inv* :: *nat* ⇒ *bin-inv-t*
where
wcode-fourtimes-case-inv st ires rs (*l*, *r*) =
(*if st* = *Suc 0 then wcode-on-left-moving-2 ires rs* (*l*, *r*)
else if st = *Suc* (*Suc 0*) *then wcode-on-checking-2 ires rs* (*l*, *r*)
else if st = *7 then wcode-goon-checking ires rs* (*l*, *r*)
else if st = *8 then wcode-right-move ires rs* (*l*, *r*)
else if st = *9 then wcode-erase2 ires rs* (*l*, *r*)
else if st = *10 then wcode-on-right-moving-2 ires rs* (*l*, *r*)
else if st = *11 then wcode-goon-right-moving-2 ires rs* (*l*, *r*)
else if st = *12 then wcode-backto-standard-pos-2 ires rs* (*l*, *r*)
else if st = *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* :: *t-conf* ⇒ *nat*
where
wcode-fourtimes-case-state (*st*, *l*, *r*) = *13* − *st*

fun *wcode-fourtimes-case-step* :: *t-conf* ⇒ *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* :: *t-conf* ⇒ *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* :: (*t-conf* × *t-conf*) *set*
where *wcode-fourtimes-case-le* ≡ (*inv-image lex-pair wcode-fourtimes-case-measure*)

lemma *wf-wcode-fourtimes-case-le*[*intro*]: *wf wcode-fourtimes-case-le*
by(*auto intro*:*wf-inv-image simp*: *wcode-fourtimes-case-le-def*)

lemma [*simp*]: *fetch t-wcode-main* (*Suc* (*Suc 0*)) *Bk* = (*L*, *7*)
apply(*simp add*: *t-wcode-main-def fetch.simps*
t-wcode-main-first-part-def nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 7 Oc* = (*R*, *8*)
apply(*simp add*: *t-wcode-main-def fetch.simps*
t-wcode-main-first-part-def nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 8 Bk* = (*R*, *9*)
apply(*simp add*: *t-wcode-main-def fetch.simps*
t-wcode-main-first-part-def nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 9 Bk* = (*R*, *10*)
apply(*simp add*: *t-wcode-main-def fetch.simps*
t-wcode-main-first-part-def nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 9 Oc* = (*W0*, *9*)
apply(*simp add*: *t-wcode-main-def fetch.simps*
t-wcode-main-first-part-def nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 10 Bk* = (*R*, *10*)
apply(*simp add*: *t-wcode-main-def fetch.simps*
t-wcode-main-first-part-def nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 10 Oc* = (*R*, *11*)
apply(*simp add*: *t-wcode-main-def fetch.simps*
t-wcode-main-first-part-def nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 11 Bk* = (*W1*, *12*)
apply(*simp add*: *t-wcode-main-def fetch.simps*
t-wcode-main-first-part-def nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 11 Oc* = (*R*, *11*)
apply(*simp add*: *t-wcode-main-def fetch.simps*

t-wcode-main-first-part-def nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 12 Oc* = (*L*, *12*)
apply(*simp add*: *t-wcode-main-def fetch.simps*
t-wcode-main-first-part-def nth-of.simps)
done

lemma [*simp*]: *fetch t-wcode-main 12 Bk* = (*R*, *t-twice-len* + *14*)
apply(*simp add*: *t-wcode-main-def fetch.simps*
t-wcode-main-first-part-def nth-of.simps)
done

lemma [*simp*]: *wcode-on-left-moving-2 ires rs* (*b*, []) = *False*
apply(*auto simp*: *wcode-fourtimes-invs*)
done

lemma [*simp*]: *wcode-on-checking-2 ires rs* (*b*, []) = *False*
apply(*auto simp*: *wcode-fourtimes-invs*)
done

lemma [*simp*]: *wcode-goon-checking ires rs* (*b*, []) = *False*
apply(*auto simp*: *wcode-fourtimes-invs*)
done

lemma [*simp*]: *wcode-right-move ires rs* (*b*, []) = *False*
apply(*auto simp*: *wcode-fourtimes-invs*)
done

lemma [*simp*]: *wcode-erase2 ires rs* (*b*, []) = *False*
apply(*auto simp*: *wcode-fourtimes-invs*)
done

lemma [*simp*]: *wcode-on-right-moving-2 ires rs* (*b*, []) = *False*
apply(*auto simp*: *wcode-fourtimes-invs exponent-def*)
done

lemma [*simp*]: *wcode-backto-standard-pos-2 ires rs* (*b*, []) = *False*
apply(*auto simp*: *wcode-fourtimes-invs exponent-def*)
done

lemma [*simp*]: *wcode-on-left-moving-2 ires rs* (*b*, *Bk* # *list*) $\Longrightarrow$ *b* $\neq$ []
apply(*simp add*: *wcode-fourtimes-invs*, *auto*)
done

lemma [*simp*]: *wcode-on-left-moving-2 ires rs* (*b*, *Bk* # *list*) $\Longrightarrow$ *wcode-on-left-moving-2 ires rs* (*tl b*, *hd b* # *Bk* # *list*)
apply(*simp only*: *wcode-fourtimes-invs*)

apply(*erule-tac disjE*)
apply(*erule-tac exE*)+
apply(*case-tac ml*, *simp*)
apply(*rule-tac x* = *mr* − (*Suc* (*Suc 0*)) **in** *exI*, *rule-tac x* = *rn* **in** *exI*, *simp*)
apply(*case-tac mr*, *simp*, *case-tac nat*, *simp*, *simp add*: *exp-ind*)
apply(*rule-tac disjI1*)
apply(*rule-tac x* = *nat* **in** *exI*, *rule-tac x* = *Suc mr* **in** *exI*, *rule-tac x* = *rn* **in** *exI*,
simp add: *exp-ind-def*)
apply(*simp*)
done

lemma [*simp*]: *wcode-on-checking-2 ires rs* (*b*, *Bk* # *list*) ⟹ *b* ≠ []
apply(*auto simp*: *wcode-fourtimes-invs*)
done

lemma [*simp*]: *wcode-on-checking-2 ires rs* (*b*, *Bk* # *list*)
⟹ *wcode-goon-checking ires rs* (*tl b*, *hd b* # *Bk* # *list*)
apply(*simp only*: *wcode-fourtimes-invs*)
apply(*auto*)
done

lemma [*simp*]: *wcode-goon-checking ires rs* (*b*, *Bk* # *list*) = *False*
apply(*simp add*: *wcode-fourtimes-invs*)
done

lemma [*simp*]: *wcode-right-move ires rs* (*b*, *Bk* # *list*) ⟹ *b*≠ []
apply(*simp add*: *wcode-fourtimes-invs*)
done

lemma [*simp*]: *wcode-right-move ires rs* (*b*, *Bk* # *list*) ⟹ *wcode-erase2 ires rs* (*Bk* # *b*, *list*)
apply(*auto simp*:*wcode-fourtimes-invs*)
apply(*rule-tac x* = *ln* **in** *exI*, *rule-tac x* = *rn* **in** *exI*, *simp*)
done

lemma [*simp*]: *wcode-erase2 ires rs* (*b*, *Bk* # *list*) ⟹ *b* ≠ []
apply(*auto simp*: *wcode-fourtimes-invs*)
done

lemma [*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*)
apply(*rule-tac x* = *Suc* (*Suc ln*) **in** *exI*, *simp add*: *exp-ind*, *auto*)
done

lemma [*simp*]: *wcode-on-right-moving-2 ires rs* (*b*, *Bk* # *list*) ⟹ *b* ≠ []
apply(*auto simp*:*wcode-fourtimes-invs*)

done

lemma [*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(*rule-tac x* = *Suc ml* **in** *exI*, *simp add*: *exp-ind-def*)
apply(*rule-tac x* = *mr* − *1* **in** *exI*, *case-tac mr*, *auto simp*: *exp-ind-def*)
done

lemma [*simp*]: *wcode-goon-right-moving-2 ires rs* (*b*, *Bk* # *list*) ⟹ *b* ≠ []
apply(*auto simp*: *wcode-fourtimes-invs*)
done

lemma [*simp*]: *wcode-goon-right-moving-2 ires rs* (*b*, *Bk* # *list*) ⟹
wcode-backto-standard-pos-2 ires rs (*b*, *Oc* # *list*)
apply(*simp add*: *wcode-fourtimes-invs*, *auto*)
apply(*rule-tac x* = *ml* **in** *exI*, *auto*)
apply(*rule-tac x* = *Suc 0* **in** *exI*, *simp*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*rule-tac x* = *rn* − *1* **in** *exI*, *simp*)
apply(*case-tac rn*, *simp*, *simp add*: *exp-ind-def*)
done

lemma [*simp*]: *wcode-backto-standard-pos-2 ires rs* (*b*, *Bk* # *list*) ⟹ *b* ≠ []
apply(*simp add*: *wcode-fourtimes-invs*, *auto*)
done

lemma [*simp*]: *wcode-on-left-moving-2 ires rs* (*b*, *Oc* # *list*) ⟹ *b* ≠ []
apply(*simp add*: *wcode-fourtimes-invs*, *auto*)
done

lemma [*simp*]: *wcode-on-left-moving-2 ires rs* (*b*, *Oc* # *list*) ⟹
wcode-on-checking-2 ires rs (*tl b*, *hd b* # *Oc* # *list*)
apply(*auto simp*: *wcode-fourtimes-invs*)
apply(*case-tac* [!] *mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: *wcode-goon-right-moving-2 ires rs* (*b*, []) ⟹ *b* ≠ []
apply(*auto simp*: *wcode-fourtimes-invs*)
done

lemma [*simp*]: *wcode-goon-right-moving-2 ires rs* (*b*, []) ⟹
wcode-backto-standard-pos-2 ires rs (*b*, [*Oc*])
apply(*simp only*: *wcode-fourtimes-invs*)
apply(*erule-tac exE*)+
apply(*rule-tac disjI1*)
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*)
apply(*case-tac mr*, *simp*, *simp add*: *exp-ind-def*)

done

lemma *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})
apply(*auto simp*: *wcode-fourtimes-invs*)
apply(*case-tac* [!] *mr*, *auto simp*: *exp-ind-def*)
done

lemma [*simp*]: *wcode-on-checking-2 ires rs* (*b*, *Oc* # *list*) $\Longrightarrow$ *False*
apply(*simp add*: *wcode-fourtimes-invs*)
done

lemma [*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 [*simp*]: *wcode-right-move ires rs* (*b*, *Oc* # *list*) = *False*
apply(*auto simp*: *wcode-fourtimes-invs*)
done

lemma [*simp*]: *wcode-erase2 ires rs* (*b*, *Oc* # *list*) $\Longrightarrow$ *b* $\neq$ []
apply(*simp add*: *wcode-fourtimes-invs*)
done

lemma [*simp*]: *wcode-erase2 ires rs* (*b*, *Oc* # *list*)
$\Longrightarrow$ *wcode-erase2 ires rs* (*b*, *Bk* # *list*)
apply(*auto simp*: *wcode-fourtimes-invs*)
done

lemma [*simp*]: *wcode-on-right-moving-2 ires rs* (*b*, *Oc* # *list*) $\Longrightarrow$ *b* $\neq$ []
apply(*simp only*: *wcode-fourtimes-invs*)
apply(*auto*)
done

lemma [*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(*case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*rule-tac x* = *Suc 0* **in** *exI*, *auto*)
apply(*rule-tac x* = *ml* − *2* **in** *exI*)
apply(*case-tac ml*, *simp*, *case-tac nat*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: *wcode-goon-right-moving-2 ires rs* (*b*, *Oc* # *list*) $\Longrightarrow$ *b* $\neq$ []
apply(*simp only*:*wcode-fourtimes-invs*, *auto*)
done

lemma [*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})
apply(*simp add*: *wcode-fourtimes-invs*, *auto*)
apply(*case-tac* [!] *mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: *wcode-on-checking-2 ires rs* (*b*, *Oc* # *list*) = *False*
apply(*simp add*: *wcode-fourtimes-invs*)
done

lemma [*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(*rule-tac x* = *Suc ml* **in** *exI*, *auto simp*: *exp-ind-def*)
apply(*rule-tac x* = *mr* − *1* **in** *exI*)
apply(*case-tac mr*, *case-tac rn*, *auto simp*: *exp-ind-def*)
done

lemma [*simp*]: *wcode-backto-standard-pos-2 ires rs* (*b*, *Oc* # *list*) $\Longrightarrow$ *b* $\neq$ []
apply(*simp only*: *wcode-fourtimes-invs*, *auto*)
done

lemma [*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(*case-tac ml*, *simp*)
apply(*rule-tac disjI2*)
apply(*rule-tac conjI*, *rule-tac x* = *ln* **in** *exI*, *simp*)
apply(*rule-tac x* = *rn* **in** *exI*, *simp*)
apply(*rule-tac disjI1*)
apply(*rule-tac x* = *nat* **in** *exI*, *rule-tac x* = *Suc mr* **in** *exI*,
rule-tac x = *ln* **in** *exI*, *rule-tac x* = *rn* **in** *exI*, *simp add*: *exp-ind-def*)
apply(*simp*)
done

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*. *steps* (*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. steps (Suc 0, Bk # Bk^m @ Oc # Bk # 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-fourtimes-case-le
    by auto
next
  show ∀ na. ¬ ?P (?f na) ∧ ?Q (?f na) ⟶
           ?Q (?f (Suc na)) ∧ (?f (Suc na), ?f na) ∈ wcode-fourtimes-case-le
  apply(rule-tac allI,
   case-tac steps (Suc 0, Bk # Bk^m @ Oc # Bk # Oc # ires, Bk # Oc^(Suc rs)
@ Bk^n) t-wcode-main na, simp,
   rule-tac impI)
  apply(simp add: tstep-red tstep.simps, case-tac c, simp, case-tac [2] aa, simp-all)

  apply(simp-all add: wcode-fourtimes-case-inv.simps new-tape.simps
           wcode-fourtimes-case-le-def lex-pair-def split: if-splits)
  done
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 add: exp-ind-def)
    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 t-fourtimes-len-gr: t-fourtimes-len > 0
apply(simp add: t-fourtimes-len-def t-fourtimes-def)
done

lemma t-fourtimes-correct:
  ∃ stp ln rn. steps (Suc 0, Bk # Bk # ires, Oc^(Suc rs) @ Bk^n)
```

(*tm-of abc-fourtimes* @ *tMp* (*Suc 0*) (*start-of fourtimes-ly* (*length abc-fourtimes*) − *Suc 0*)) *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*. *steps* (*Suc 0*, *Bk* # *Bk* # *ires*, <[*rs*]> @ Bk^n) (*tm-of abc-fourtimes* @ *tMp* (*Suc 0*)
(*start-of fourtimes-ly* (*length abc-fourtimes*) − *1*)) *stp* = (*0*, Bk^m @ *Bk* # *Bk* # *ires*, $Oc^{Suc\ (4*rs)}$ @ Bk^l)
proof(*rule-tac t-compiled-by-rec*)
show *rec-ci rec-fourtimes* = (*a*, *b*, *c*) **by** (*simp add*: *h*)
next
show *rec-calc-rel rec-fourtimes* [*rs*] (*4* ∗ *rs*)
using *prime-rel-exec-eq* [*of rec-fourtimes* [*rs*] *4* ∗ *rs*]
apply(*subgoal-tac primerec rec-fourtimes* (*length* [*rs*]))
apply(*simp-all add*: *rec-fourtimes-def rec-exec.simps*)
apply(*auto*)
apply(*simp only*: *Nat.One-nat-def*[*THEN sym*], *auto*)
done
next
show *length* [*rs*] = *Suc 0* **by** *simp*
next
show *layout-of* (*a* [+] *dummy-abc* (*Suc 0*)) = *layout-of* (*a* [+] *dummy-abc* (*Suc 0*))
by *simp*
next
show *start-of fourtimes-ly* (*length abc-fourtimes*) =
start-of (*layout-of* (*a* [+] *dummy-abc* (*Suc 0*))) (*length* (*a* [+] *dummy-abc* (*Suc 0*)))
using *h*
apply(*simp add*: *fourtimes-ly-def abc-fourtimes-def*)
done
next
show *tm-of abc-fourtimes* = *tm-of* (*a* [+] *dummy-abc* (*Suc 0*))
using *h*
apply(*simp add*: *abc-fourtimes-def*)
done
qed
thus ∃ *stp ln rn*. *steps* (*Suc 0*, *Bk* # *Bk* # *ires*, $Oc^{Suc\ rs}$ @ Bk^n)
(*tm-of abc-fourtimes* @ *tMp* (*Suc 0*) (*start-of fourtimes-ly* (*length abc-fourtimes*) − *Suc 0*)) *stp* =
(*0*, Bk^{ln} @ *Bk* # *Bk* # *ires*, $Oc^{Suc\ (4\ *\ rs)}$ @ Bk^{rn})
apply(*simp add*: *tape-of-nl-abv tape-of-nat-list.simps*)
done
qed

lemma *t-fourtimes-change-term-state*:
∃ *stp ln rn*. *steps* (*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})
using *t-fourtimes-correct*[*of ires rs n*]
apply(*erule-tac exE*)
apply(*erule-tac exE*)
apply(*erule-tac exE*)
proof(*drule-tac turing-change-termi-state*)
fix *stp ln rn*
show *t-correct* (*tm-of abc-fourtimes* @ *tMp* (*Suc 0*)
(*start-of fourtimes-ly* (*length abc-fourtimes*) − *Suc 0*))
apply(*rule-tac t-compiled-correct*, *auto simp*: *fourtimes-ly-def*)
done
next
fix *stp ln rn*
show ∃ *stp*. *steps* (*Suc 0*, *Bk* # *Bk* # *ires*, $Oc^{Suc\ rs}$ @ Bk^n)
(*change-termi-state* (*tm-of abc-fourtimes* @ *tMp* (*Suc 0*)
(*start-of fourtimes-ly* (*length abc-fourtimes*) − *Suc 0*))) *stp* =
(*Suc* (*length* (*tm-of abc-fourtimes* @ *tMp* (*Suc 0*) (*start-of fourtimes-ly*
(*length abc-fourtimes*) − *Suc 0*)) *div 2*), Bk^{ln} @ *Bk* # *Bk* # *ires*, $Oc^{Suc\ (4\ *\ rs)}$
@ Bk^{rn}) ⟹
∃ *stp ln rn*. *steps* (*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})
apply(*erule-tac exE*)
apply(*simp add*: *t-fourtimes-len-def t-fourtimes-def*)
apply(*rule-tac x* = *stp* **in** *exI*, *rule-tac x* = *ln* **in** *exI*, *rule-tac x* = *rn* **in** *exI*,
simp)
done
qed

lemma *t-fourtimes-append-pre*:
steps (*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})
⟹ ∃ *stp*>*0*. *steps* (*Suc 0* + *length* (*t-wcode-main-first-part* @
tshift 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* @
tshift t-twice (*length t-wcode-main-first-part div 2*) @ [(*L*, *1*), (*L*, *1*)]) @
tshift t-fourtimes (*length* (*t-wcode-main-first-part* @
tshift 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* @
tshift t-twice (*length t-wcode-main-first-part div 2*) @ [(*L*, *1*), (*L*, *1*)]) *div 2*,
Bk^{ln} @ *Bk* # *Bk* # *ires*, $Oc^{Suc\ (4\ *\ rs)}$ @ Bk^{rn})
proof(*rule-tac t-tshift-lemma*, *auto*)
assume *steps* (*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})
thus *0* < *stp*
using *t-fourtimes-len-gr*

```
    apply(case-tac stp, simp-all add: steps.simps)
    done
next
  show Suc 0 ≤ length t-fourtimes div 2
    apply(simp add: t-fourtimes-def shift-length tMp.simps)
    done
next
  show t-ncorrect (t-wcode-main-first-part @
    abacus.tshift t-twice (length t-wcode-main-first-part div 2) @
    [(L, Suc 0), (L, Suc 0)])
    apply(simp add: t-ncorrect.simps t-wcode-main-first-part-def shift-length
                t-twice-def)
    using tm-even[of abc-twice]
    by arith
next
  show t-ncorrect t-fourtimes
    apply(simp add: t-fourtimes-def steps.simps t-ncorrect.simps)
    using tm-even[of abc-fourtimes]
    by arith
next
  show t-ncorrect [(L, Suc 0), (L, Suc 0)]
    apply(simp add: t-ncorrect.simps)
    done
qed

lemma [simp]: length t-wcode-main-first-part = 24
apply(simp add: t-wcode-main-first-part-def)
done

lemma [simp]: (26 + length t-twice) div 2 = (length t-twice) div 2 + 13
using tm-even[of abc-twice]
apply(simp add: t-twice-def)
done

lemma [simp]: ((26 + length (abacus.tshift t-twice 12)) div 2)
            = (length (abacus.tshift t-twice 12) div 2 + 13)
using tm-even[of abc-twice]
apply(simp add: t-twice-def)
done

lemma [simp]: t-twice-len + 14 =  14 + length (abacus.tshift t-twice 12) div 2
using tm-even[of abc-twice]
apply(simp add: t-twice-def t-twice-len-def shift-length)
done

lemma t-fourtimes-append:
  ∃ stp ln rn.
  steps (Suc 0 + length (t-wcode-main-first-part @ tshift 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* @ *tshift t-twice* (*length t-wcode-main-first-part div 2*) @ [(*L*, *1*), (*L*, *1*)]) @ *tshift t-fourtimes* (*t-twice-len* + *13*) @ [(*L*, *1*), (*L*, *1*)]) *stp*
= (*Suc t-fourtimes-len* + *length* (*t-wcode-main-first-part* @ *tshift 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(*erule-tac exE*)
apply(*rule-tac x* = *stpa* **in** *exI*, *rule-tac x* = *ln* **in** *exI*, *rule-tac x* = *rn* **in** *exI*)
apply(*simp add*: *t-twice-len-def shift-length*)
done

lemma *t-wcode-main-len*: *length t-wcode-main* = *length t-twice* + *length t-fourtimes* + *28*
apply(*simp add*: *t-wcode-main-def shift-length*)
done

lemma [*simp*]: *fetch t-wcode-main* (*14* + *length t-twice div 2* + *t-fourtimes-len*) *b*
= (*L*, *Suc 0*)
using *tm-even*[*of abc-twice*] *tm-even*[*of abc-fourtimes*]
apply(*case-tac b*)
apply(*simp-all only*: *fetch.simps*)
apply(*auto simp*: *nth-of.simps t-wcode-main-len t-twice-len-def*
t-fourtimes-def t-twice-def t-fourtimes-def t-fourtimes-len-def)
apply(*auto simp*: *t-wcode-main-def t-wcode-main-first-part-def shift-length t-twice-def nth-append*
t-fourtimes-def)
done

lemma *wcode-jump2*:
∃ *stp ln rn*. *steps* (*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-tac x* = *Suc 0* **in** *exI*)
apply(*simp add*: *steps.simps shift-length*)
apply(*rule-tac x* = *lnb* **in** *exI*, *rule-tac x* = *rnb* **in** *exI*)
apply(*simp add*: *tstep.simps new-tape.simps*)
done

lemma *wcode-fourtimes-case*:
shows ∃ *stp ln rn*.
steps (*Suc 0*, *Bk* # Bk^m @ *Oc* # *Bk* # *Oc* # *ires*, *Bk* # $Oc^{Suc\ rs}$ @ Bk^n) *t-wcode-main stp* =

(*Suc 0*, *Bk* # Bk^{ln} @ *Oc* # *ires*, *Bk* # $Oc^{Suc\ (4*rs\ +\ 4)}$ @ Bk^{rn})
proof −
have ∃ *stp ln rn*.
steps (*Suc 0*, *Bk* # Bk^{m} @ *Oc* # *Bk* # *Oc* # *ires*, *Bk* # $Oc^{Suc\ rs}$ @ Bk^{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*]
apply(*simp add*: *wcode-fourtimes-case-inv.simps*, *auto*)
apply(*rule-tac x* = *na* **in** *exI*, *rule-tac x* = *ln* **in** *exI*,
rule-tac x = *rn* **in** *exI*)
apply(*simp*)
done
from *this* **obtain** *stpa lna rna* **where** *stp1*:
steps (*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*. *steps* (*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(*rule-tac x* = *stp* **in** *exI*,
rule-tac x = *ln* + *lna* **in** *exI*,
rule-tac x = *rn* **in** *exI*, *simp*)
apply(*simp add*: *exp-ind-def*[*THEN sym*] *exp-add*[*THEN sym*])
done
from *this* **obtain** *stpb lnb rnb* **where** *stp2*:
steps (*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*. *steps* (*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*:
steps (*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 add: steps-add)
    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* :: *t-conf* ⇒ *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* :: *t-conf* ⇒ *nat*
where
wcode-halt-case-step (*st*, *l*, *r*) =
(*if st* = *1 then length l*
else 0)

fun *wcode-halt-case-measure* :: *t-conf* ⇒ *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* :: (*t-conf* × *t-conf*) *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 intro*:*wf-inv-image 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 [*simp*]: *fetch t-wcode-main 7 Bk* = (*R*, *0*)
apply(*simp add*: *fetch.simps t-wcode-main-def nth-append nth-of.simps*

t-wcode-main-first-part-def)
done

lemma [*simp*]: *wcode-on-left-moving-3 ires rs* (*b*, []) = *False*
apply(*simp only*: *wcode-halt-invs*)
apply(*simp add*: *exp-ind-def*)
done

lemma [*simp*]: *wcode-on-checking-3 ires rs* (*b*, []) = *False*
apply(*simp add*: *wcode-halt-invs*)
done

lemma [*simp*]: *wcode-goon-checking-3 ires rs* (*b*, []) = *False*
apply(*simp add*: *wcode-halt-invs*)
done

lemma [*simp*]: *wcode-on-left-moving-3 ires rs* (*b*, *Bk* # *list*)
$\Longrightarrow$ *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(*case-tac ml*, *simp*)
apply(*rule-tac x* = *mr* − *2* **in** *exI*, *rule-tac x* = *rn* **in** *exI*)
apply(*case-tac mr*, *simp*, *simp add*: *exp-ind*, *simp add*: *exp-ind*[*THEN sym*])
apply(*rule-tac disjI1*)
apply(*rule-tac x* = *nat* **in** *exI*, *rule-tac x* = *Suc mr* **in** *exI*,
rule-tac x = *rn* **in** *exI*, *simp add*: *exp-ind-def*)
apply(*simp*)
done

lemma [*simp*]: *wcode-goon-checking-3 ires rs* (*b*, *Bk* # *list*) $\Longrightarrow$
(*b* = [] $\longrightarrow$ *wcode-stop ires rs* ([*Bk*], *list*)) $\wedge$
(*b* $\neq$ [] $\longrightarrow$ *wcode-stop ires rs* (*Bk* # *b*, *list*))
apply(*auto simp*: *wcode-halt-invs*)
done

lemma [*simp*]: *wcode-on-left-moving-3 ires rs* (*b*, *Oc* # *list*) $\Longrightarrow$ *b* $\neq$ []
apply(*auto simp*: *wcode-halt-invs*)
done

lemma [*simp*]: *wcode-on-left-moving-3 ires rs* (*b*, *Oc* # *list*) $\Longrightarrow$
wcode-on-checking-3 ires rs (*tl b*, *hd b* # *Oc* # *list*)
apply(*simp add*:*wcode-halt-invs*, *auto*)
apply(*case-tac* [!] *mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: *wcode-on-checking-3 ires rs* (*b*, *Oc* # *list*) = *False*
apply(*auto simp*: *wcode-halt-invs*)
done

lemma [*simp*]: *wcode-on-left-moving-3 ires rs* (*b*, *Bk* # *list*) $\Longrightarrow$ *b* $\neq$ []
apply(*simp add*: *wcode-halt-invs*, *auto*)
done

lemma [*simp*]: *wcode-on-checking-3 ires rs* (*b*, *Bk* # *list*) $\Longrightarrow$ *b* $\neq$ []
apply(*auto simp*: *wcode-halt-invs*)
done

lemma [*simp*]: *wcode-on-checking-3 ires rs* (*b*, *Bk* # *list*) $\Longrightarrow$
wcode-goon-checking-3 ires rs (*tl b*, *hd b* # *Bk* # *list*)
apply(*auto simp*: *wcode-halt-invs*)
done

lemma [*simp*]: *wcode-goon-checking-3 ires rs* (*b*, *Oc* # *list*) = *False*
apply(*simp add*: *wcode-goon-checking-3.simps*)
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*. *steps* (*Suc 0*, *Bk* # Bk^m @ *Oc* # *Bk* # *Bk* # *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* = *0*)
let *?Q* = (λ (*st*, *l*, *r*). *wcode-halt-case-inv st ires rs* (*l*, *r*))
let *?f* = (λ *stp*. *steps* (*Suc 0*, *Bk* # Bk^m @ *Oc* # *Bk* # *Bk* # *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-halt-case-le* **by** *auto*
next
show $\forall$ *na*. $\neg$ *?P* (*?f na*) $\wedge$ *?Q* (*?f na*) $\longrightarrow$
?Q (*?f* (*Suc na*)) $\wedge$ (*?f* (*Suc na*), *?f na*) $\in$ *wcode-halt-case-le*
apply(*rule-tac allI*, *rule-tac impI*, *case-tac ?f na*)
apply(*simp add*: *tstep-red tstep.simps*)
apply(*case-tac c*, *simp*, *case-tac* [*2*] *aa*)
apply(*simp-all split*: *if-splits add*: *new-tape.simps wcode-halt-case-le-def lex-pair-def*)
done
next
show *?Q* (*?f 0*)
apply(*simp add*: *steps.simps wcode-halt-invs*)
apply(*rule-tac x* = *Suc m* **in** *exI*, *simp add*: *exp-ind-def*)
apply(*rule-tac x* = *Suc 0* **in** *exI*, *auto*)
done
next
show $\neg$ *?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 [intro]: ∃ xs. (<rev list @ [aa::nat]> :: block list) = Oc # xs
apply(case-tac rev list, simp)
apply(simp add: tape-of-nat-abv tape-of-nat-list.simps exp-ind-def)
apply(case-tac list, simp, simp)
done

lemma wcode-halt-case:
  ∃stp ln rn. steps (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)
  using t-halt-case-correctness[of ires rs m n]
apply(simp)
apply(erule-tac exE)
apply(case-tac steps (Suc 0, Bk # Bk^m @ Oc # Bk # Bk # ires,
               Bk # Oc^(Suc rs) @ Bk^n) t-wcode-main na)
apply(auto simp: wcode-halt-case-inv.simps wcode-stop.simps)
apply(rule-tac x = na in exI, rule-tac x = ln in exI,
      rule-tac x = rn in exI, simp)
done

lemma bl-bin-one: bl-bin [Oc] =  Suc 0
apply(simp add: bl-bin.simps)
done

lemma t-wcode-main-lemma-pre:
  ⟦args ≠ []; lm = <args::nat list>⟧ ⟹
       ∃ stp ln rn. steps (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 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.
   steps (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 # $Oc^{bl\text{-}bin\ lm\ +\ rs\ *\ 2\ \hat{}\ (length\ lm\ -\ 1)}$ @ 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*.
steps (*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 # $Oc^{bl\text{-}bin\ lm\ +\ rs\ *\ 2\ \hat{}\ (length\ lm\ -\ 1)}$ @ Bk^{rn})
proof(*case-tac xs*::*nat list*, *simp*)
show ∃ *stp ln rn*.
steps (*Suc 0*, Bk # Bk^m @ *rev* (<*a*>) @ Bk # Bk # *ires*, Bk # $Oc^{Suc\ rs}$ @ Bk^n) *t-wcode-main stp* =
(0, Bk # *ires*, Bk # Oc # Bk^{ln} @ Bk # Bk # $Oc^{bl\text{-}bin\ (<a>)\ +\ rs\ *\ 2\ \hat{}\ a}$ @ Bk^{rn})
proof(*induct a arbitrary*: *m n rs ires*, *simp*)
fix *m n rs ires*
show ∃ *stp ln rn*. *steps* (*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 # $Oc^{bl\text{-}bin\ [Oc]\ +\ rs}$ @ Bk^{rn})
apply(*simp add*: *bl-bin-one*)
apply(*rule-tac wcode-halt-case*)
done
next
fix *a m n rs ires*
assume *ind2*:
⋀*m n rs ires*.
∃ *stp ln rn*.
steps (*Suc 0*, Bk # Bk^m @ *rev* (<*a*>) @ Bk # Bk # *ires*, Bk # $Oc^{Suc\ rs}$ @ Bk^n) *t-wcode-main stp* =
(0, Bk # *ires*, Bk # Oc # Bk^{ln} @ Bk # Bk # $Oc^{bl\text{-}bin\ (<a>)\ +\ rs\ *\ 2\ \hat{}\ a}$ @ Bk^{rn})
show ∃ *stp ln rn*.
steps (*Suc 0*, Bk # Bk^m @ *rev* (<*Suc a*>) @ Bk # Bk # *ires*, Bk # $Oc^{Suc\ rs}$ @ Bk^n) *t-wcode-main stp* =
(0, Bk # *ires*, Bk # Oc # Bk^{ln} @ Bk # Bk # $Oc^{bl\text{-}bin\ (<Suc\ a>)\ +\ rs\ *\ 2\ \hat{}\ Suc\ a}$ @ Bk^{rn})
proof −
have ∃ *stp ln rn*.
steps (*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*: *exp-ind-def*)
done
from *this* **obtain** *stpa lna rna* **where** *stp1*:
steps (*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*.
steps (*Suc 0*, *Bk* # Bk^{lna} @ *rev* (<*a*>) @ *Bk* # *Bk* # *ires*, *Bk* # $Oc^{Suc\ (2\ *\ rs\ +\ 2)}$ @ Bk^{rna}) *t-wcode-main stp* =
(*0*, *Bk* # *ires*, *Bk* # *Oc* # Bk^{ln} @ *Bk* # *Bk* # $Oc^{bl\text{-}bin\ (<a>)\ +\ (2*rs\ +\ 2)\ *\ 2\ \hat{}\ a}$ @ Bk^{rn})
using *ind2*[*of lna ires 2*∗*rs* + *2 rna*] **by** *simp*
from *this* **obtain** *stpb lnb rnb* **where** *stp2*:
steps (*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* # $Oc^{bl\text{-}bin\ (<a>)\ +\ (2*rs\ +\ 2)\ *\ 2\ \hat{}\ a}$ @ 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*)
apply(*simp add*: *steps-add bl-bin-nat-Suc exponent-def*)
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*. *steps* (*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* # $Oc^{bl\text{-}bin\ lm\ +\ rs\ *\ 2\ \hat{}\ (length\ lm\ -\ 1)}$ @ 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*.
steps (*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*.
steps (*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*:
steps (*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*. *steps* (*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*$^{bl\text{-}bin\ (<aa\#list>)+\ (4*rs\ +\ 4)\ *\ 2\hat{}(length\ (<aa\#list>)\ -\ 1)}$ @ *Bk*^{rn})
apply(*rule-tac args* = (*aa*::*nat*)#*list* **in** *ind*, *simp-all*)
done
from *this* **obtain** *stpb lnb rnb* **where** *stp2*:
steps (*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*$^{bl\text{-}bin\ (<aa\#list>)+\ (4*rs\ +\ 4)\ *\ 2\hat{}(length\ (<aa\#list>)\ -\ 1)}$ @ *Bk*^{rnb})
by *blast*
from *stp1* **and** *stp2* **and** *h*
show ∃ *stp ln rn*.
steps (*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*$^{bl\text{-}bin\ (Oc^{Suc\ aa}\ @\ Bk\ \#\ <list\ @\ [0]>)\ +\ rs\ *\ (2\ *\ 2\ \hat{}\ (aa\ +\ length\ (<list\ @\ [0]>)))}$ @ *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*:

$\bigwedge$ *m n rs args lm.*
⟦*lm* = <*aa* # *list* @ [*ab*]>; *args* = *aa* # *list* @ [*ab*]⟧
⟹ ∃ *stp ln rn.*
steps (*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 # $Oc^{bl\text{-}bin\ (<aa\ \#\ list\ @\ [ab]>)\ +\ rs\ *\ 2\ \hat{}\ (length\ (<aa\ \#\ list\ @\ [ab]>)\ -\ Suc\ 0)}$
@ Bk^{rn})
and *k*: *args* = *aa* # *list* @ [*Suc ab*] *lm* = <*aa* # *list* @ [*Suc ab*]>
show ∃ *stp ln rn.*
steps (*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 # $Oc^{bl\text{-}bin\ (<aa\ \#\ list\ @\ [Suc\ ab]>)\ +\ rs\ *\ 2\ \hat{}\ (length\ (<aa\ \#\ list\ @\ [Suc\ ab]>)\ -\ Suc\ 0)}$
@ Bk^{rn})
proof(*simp add*: *tape-of-nl-cons-app1*)
have ∃ *stp ln rn.*
steps (*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*: *exp-ind-def*)
done
from *this* **obtain** *stpa lna rna* **where** *stp1*:
steps (*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. steps* (*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 # $Oc^{bl\text{-}bin\ (<aa\ \#\ list\ @\ [ab]>\ )\ +\ (2*rs\ +\ 2)*\ 2\hat{}(length\ (<aa\ \#\ list\ @\ [ab]>)\ -\ Suc\ 0)}$
@ Bk^{rn})
apply(*rule-tac ind2*, *simp-all*)
done
from *this* **obtain** *stpb lnb rnb* **where** *stp2*:
steps (*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 # $Oc^{bl\text{-}bin}$ (<*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*.
steps (*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* #
$Oc^{bl\text{-}bin}$ ($Oc^{Suc\ aa}$ @ *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*
exp-ind-def)
done
qed
qed
qed
qed

term *t-wcode-main*
definition *t-wcode-prepare* :: *tprog*
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* :: *t-conf* ⇒ *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* :: *t-conf* ⇒ *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 :: t-conf ⇒ 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 :: t-conf ⇒ 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 :: (t-conf × t-conf) set
  where wcode-prepare-le ≡ (inv-image lex-triple wcode-prepare-measure)

lemma [intro]: wf lex-pair
by(auto intro:wf-lex-prod simp:lex-pair-def)

lemma wf-wcode-prepare-le[intro]: wf wcode-prepare-le
by(auto intro:wf-inv-image simp: wcode-prepare-le-def
          recursive.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 [*simp*]: *fetch t-wcode-prepare* (*Suc 0*) *Bk* = (*W1*, *2*)
apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-prepare* (*Suc 0*) *Oc* = (*L*, *1*)
apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-prepare* (*Suc* (*Suc 0*)) *Bk* = (*L*, *3*)
apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-prepare* (*Suc* (*Suc 0*)) *Oc* = (*R*, *2*)
apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-prepare* (*Suc* (*Suc* (*Suc 0*))) *Bk* = (*R*, *4*)
apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-prepare* (*Suc* (*Suc* (*Suc 0*))) *Oc* = (*W0*, *3*)
apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-prepare 4 Bk* = (*R*, *4*)
apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-prepare 4 Oc* = (*R*, *5*)
apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-prepare 5 Oc* = (*R*, *5*)
apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-prepare 5 Bk* = (*R*, *6*)
apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-prepare 6 Oc* = (*R*, *5*)
apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-prepare 6 Bk* = (*R*, *7*)
apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-prepare 7 Oc* = (*L*, *0*)

apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-prepare 7 Bk* = (*W1*, *7*)
apply(*simp add*: *fetch.simps t-wcode-prepare-def nth-of.simps*)
done

lemma *tape-of-nl-not-null*: *lm* ≠ [] ⟹ <*lm*::*nat list*> ≠ []
apply(*case-tac lm*, *auto*)
apply(*case-tac list*, *auto simp*: *tape-of-nl-abv tape-of-nat-list.simps exp-ind-def*)
done

lemma [*simp*]: *lm* ≠ [] ⟹ *wprepare-add-one m lm* (*b*, []) = *False*
apply(*simp add*: *wprepare-invs*)
apply(*simp add*: *tape-of-nl-not-null*)
done

lemma [*simp*]: *lm* ≠ [] ⟹ *wprepare-goto-first-end m lm* (*b*, []) = *False*
apply(*simp add*: *wprepare-invs*)
done

lemma [*simp*]: *lm* ≠ [] ⟹ *wprepare-erase m lm* (*b*, []) = *False*
apply(*simp add*: *wprepare-invs*)
done

lemma [*simp*]: *lm* ≠ [] ⟹ *wprepare-goto-start-pos m lm* (*b*, []) = *False*
apply(*simp add*: *wprepare-invs tape-of-nl-not-null*)
done

lemma [*simp*]: ⟦*lm* ≠ []; *wprepare-loop-start m lm* (*b*, [])⟧ ⟹ *b* ≠ []
apply(*simp add*: *wprepare-invs tape-of-nl-not-null*, *auto*)
done

lemma [*simp*]: ⟦*lm* ≠ []; *wprepare-loop-start m lm* (*b*, [])⟧ ⟹
wprepare-loop-goon m lm (*Bk* # *b*, [])
apply(*simp only*: *wprepare-invs tape-of-nl-not-null*)
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 [*simp*]: ⟦*lm* ≠ []; *wprepare-loop-goon m lm* (*b*, [])⟧ ⟹ *b* ≠ []
apply(*simp only*: *wprepare-invs tape-of-nl-not-null*, *auto*)
done

lemma [*simp*]:⟦*lm* ≠ []; *wprepare-loop-goon m lm* (*b*, [])⟧ ⟹
wprepare-add-one2 m lm (*Bk* # *b*, [])
apply(*simp only*: *wprepare-invs tape-of-nl-not-null*, *auto split*: *if-splits*)
apply(*case-tac mr*, *simp*, *simp add*: *exp-ind-def*)
done

lemma [*simp*]: *wprepare-add-one2 m lm* (*b*, []) ⟹ *b* ≠ []
apply(*simp only*: *wprepare-invs tape-of-nl-not-null*, *auto*)
done

lemma [*simp*]: *wprepare-add-one2 m lm* (*b*, []) ⟹ *wprepare-add-one2 m lm* (*b*, [*Oc*])
apply(*simp only*: *wprepare-invs tape-of-nl-not-null*, *auto*)
done

lemma [*simp*]: *Bk* # *list* = <(*m*::*nat*) # *lm*> @ *ys* = *False*
apply(*case-tac lm*, *auto simp*: *tape-of-nl-abv tape-of-nat-list.simps exp-ind-def*)
done

lemma [*simp*]: ⟦*lm* ≠ []; *wprepare-add-one m lm* (*b*, *Bk* # *list*)⟧
⟹ (*b* = [] ⟶ *wprepare-goto-first-end m lm* ([], *Oc* # *list*)) ∧
(*b* ≠ [] ⟶ *wprepare-goto-first-end m lm* (*b*, *Oc* # *list*))
apply(*simp only*: *wprepare-invs*, *auto*)
apply(*rule-tac x* = *0* **in** *exI*, *simp add*: *exp-ind-def*)
apply(*case-tac lm*, *simp*, *simp add*: *tape-of-nl-abv tape-of-nat-list.simps exp-ind-def*)
apply(*rule-tac x* = *rn* **in** *exI*, *simp*)
done

lemma [*simp*]: *wprepare-goto-first-end m lm* (*b*, *Bk* # *list*) ⟹ *b* ≠ []
apply(*simp only*: *wprepare-invs tape-of-nl-not-null*, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: *wprepare-goto-first-end m lm* (*b*, *Bk* # *list*) ⟹
wprepare-erase m lm (*tl b*, *hd b* # *Bk* # *list*)
apply(*simp only*: *wprepare-invs tape-of-nl-not-null*, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*case-tac mr*, *auto simp*: *exp-ind-def*)
done

lemma [*simp*]: *wprepare-erase m lm* (*b*, *Bk* # *list*) ⟹ *b* ≠ []
apply(*simp only*: *wprepare-invs exp-ind-def*, *auto*)
done

lemma [*simp*]: *wprepare-erase m lm* (*b*, *Bk* # *list*) ⟹
wprepare-goto-start-pos m lm (*Bk* # *b*, *list*)
apply(*simp only*: *wprepare-invs*, *auto*)
done

lemma [*simp*]: $[\![$*wprepare-add-one m lm* (*b*, *Bk* # *list*)$]\!]$ $\Longrightarrow$ *list* $\neq$ []
apply(*simp only*: *wprepare-invs*)
apply(*case-tac lm*, *simp-all add*: *tape-of-nl-abv*
tape-of-nat-list.simps exp-ind-def , *auto*)
done

lemma [*simp*]: $[\![$*lm* $\neq$ []; *wprepare-goto-first-end m lm* (*b*, *Bk* # *list*)$]\!]$ $\Longrightarrow$ *list* $\neq$ []
apply(*simp only*: *wprepare-invs*, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*simp add*: *tape-of-nl-not-null*)
done

lemma [*simp*]: $[\![$*lm* $\neq$ []; *wprepare-goto-first-end m lm* (*b*, *Bk* # *list*)$]\!]$ $\Longrightarrow$ *b* $\neq$ []
apply(*simp only*: *wprepare-invs*, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def tape-of-nl-not-null*)
done

lemma [*simp*]: $[\![$*lm* $\neq$ []; *wprepare-erase m lm* (*b*, *Bk* # *list*)$]\!]$ $\Longrightarrow$ *list* $\neq$ []
apply(*simp only*: *wprepare-invs*, *auto*)
done

lemma [*simp*]: $[\![$*lm* $\neq$ []; *wprepare-erase m lm* (*b*, *Bk* # *list*)$]\!]$ $\Longrightarrow$ *b* $\neq$ []
apply(*simp only*: *wprepare-invs*, *auto simp*: *exp-ind-def*)
done

lemma [*simp*]: $[\![$*lm* $\neq$ []; *wprepare-goto-start-pos m lm* (*b*, *Bk* # *list*)$]\!]$ $\Longrightarrow$ *list* $\neq$ []
apply(*simp only*: *wprepare-invs*, *auto*)
apply(*simp add*: *tape-of-nl-not-null*)
apply(*case-tac lm*, *simp*, *case-tac list*)
apply(*simp-all add*: *tape-of-nl-abv tape-of-nat-list.simps exp-ind-def*)
done

lemma [*simp*]: $[\![$*lm* $\neq$ []; *wprepare-goto-start-pos m lm* (*b*, *Bk* # *list*)$]\!]$ $\Longrightarrow$ *b* $\neq$ []
apply(*simp only*: *wprepare-invs*)
apply(*auto*)
done

lemma [*simp*]: $[\![$*lm* $\neq$ []; *wprepare-loop-goon m lm* (*b*, *Bk* # *list*)$]\!]$ $\Longrightarrow$ *b* $\neq$ []
apply(*simp only*: *wprepare-invs*, *auto*)
done

lemma [*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*))
apply(*simp only*: *wprepare-invs*, *simp*)
apply(*case-tac list*, *simp-all split*: *if-splits*, *auto*)
apply(*case-tac* [*1*−*3*] *mr*, *simp-all add*: *exp-ind-def*)

apply(*case-tac mr*, *simp-all add*: *exp-ind-def tape-of-nl-not-null*)
apply(*case-tac* [*1*−*2*] *mr*, *simp-all add*: *exp-ind-def*)
apply(*case-tac rn*, *simp*, *case-tac nat*, *auto simp*: *exp-ind-def*)
done

lemma [*simp*]: *wprepare-add-one2 m lm* (*b*, *Bk* # *list*) ⟹ *b* ≠ []
apply(*simp only*: *wprepare-invs*, *simp*)
done

lemma [*simp*]: *wprepare-add-one2 m lm* (*b*, *Bk* # *list*) ⟹
  (*list* = [] ⟶ *wprepare-add-one2 m lm* (*b*, [*Oc*])) ∧
  (*list* ≠ [] ⟶ *wprepare-add-one2 m lm* (*b*, *Oc* # *list*))
apply(*simp only*: *wprepare-invs*, *auto*)
done

lemma [*simp*]: *wprepare-goto-first-end m lm* (*b*, *Oc* # *list*)
  ⟹ (*b* = [] ⟶ *wprepare-goto-first-end m lm* ([*Oc*], *list*)) ∧
   (*b* ≠ [] ⟶ *wprepare-goto-first-end m lm* (*Oc* # *b*, *list*))
apply(*simp only*: *wprepare-invs*, *auto*)
apply(*rule-tac x* = *1* **in** *exI*, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*case-tac ml*, *simp-all add*: *exp-ind-def*)
apply(*rule-tac x* = *rn* **in** *exI*, *simp*)
apply(*rule-tac x* = *Suc ml* **in** *exI*, *simp-all add*: *exp-ind-def*)
apply(*rule-tac x* = *mr* − *1* **in** *exI*, *simp*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*, *auto*)
done

lemma [*simp*]: *wprepare-erase m lm* (*b*, *Oc* # *list*) ⟹ *b* ≠ []
apply(*simp only*: *wprepare-invs*, *auto simp*: *exp-ind-def*)
done

lemma [*simp*]: *wprepare-erase m lm* (*b*, *Oc* # *list*)
 ⟹ *wprepare-erase m lm* (*b*, *Bk* # *list*)
apply(*simp only*:*wprepare-invs*, *auto simp*: *exp-ind-def*)
done

lemma [*simp*]: ⟦*lm* ≠ []; *wprepare-goto-start-pos m lm* (*b*, *Bk* # *list*)⟧
  ⟹ *wprepare-goto-start-pos m lm* (*Bk* # *b*, *list*)
apply(*simp only*:*wprepare-invs*, *auto*)
apply(*case-tac* [!] *lm*, *simp*, *simp-all*)
done

lemma [*simp*]: *wprepare-loop-start m lm* (*b*, *aa*) ⟹ *b* ≠ []
apply(*simp only*:*wprepare-invs*, *auto*)
done
lemma [*elim*]: *Bk* # *list* = Oc^{mr} @ Bk^{rn} ⟹ ∃ *rn*. *list* = Bk^{rn}
apply(*case-tac mr*, *simp-all*)
apply(*case-tac rn*, *simp-all add*: *exp-ind-def*, *auto*)

done

lemma *rev-equal-iff*: $x = y \Longrightarrow rev\ x = rev\ y$
by *simp*

lemma *tape-of-nl-false1*:
  $lm \neq [] \Longrightarrow rev\ b\ @\ [Bk] \neq Bk^{ln}\ @\ Oc\ \#\ Oc^{m}\ @\ Bk\ \#\ Bk\ \#$ <*lm*::*nat list*>
apply(*auto*)
apply(*drule-tac rev-equal-iff*, *simp add*: *tape-of-nl-rev*)
apply(*case-tac rev lm*)
apply(*case-tac* [*2*] *list*, *auto simp*: *tape-of-nl-abv tape-of-nat-list.simps exp-ind-def*)
done

lemma [*simp*]: *wprepare-loop-start-in-middle m lm* (*b*, [*Bk*]) = *False*
apply(*simp add*: *wprepare-loop-start-in-middle.simps*, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*case-tac lm1*, *simp*, *simp add*: *tape-of-nl-not-null*)
done

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 [*simp*]: *wprepare-loop-goon-in-middle m lm* (*Bk* # *b*, []) = *False*
apply(*simp add*: *wprepare-loop-goon-in-middle.simps*, *auto*)
done

lemma [*simp*]: ⟦*lm* ≠ []; *wprepare-loop-start m lm* (*b*, [*Bk*])⟧ ⟹
  wprepare-loop-goon m lm (*Bk* # *b*, [])
apply(*simp only*: *wprepare-invs*, *simp*)
apply(*simp add*: *wprepare-loop-goon-on-rightmost.simps*
  wprepare-loop-start-on-rightmost.simps, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*rule-tac rev-eq*)
apply(*simp add*: *tape-of-nl-rev*)
apply(*simp add*: *exp-ind-def*[*THEN sym*] *exp-ind*)
done

lemma [*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)
apply(*case-tac* [!] *mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*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*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*simp add*: *tape-of-nl-rev*)
apply(*simp add*: *exp-ind-def*[*THEN sym*] *exp-ind*)
done

lemma [*simp*]: ⟦*lm* ≠ []; *wprepare-loop-start-in-middle m lm* (*b*, *Bk* # *a* # *lista*)⟧
⟹ *wprepare-loop-goon-on-rightmost m lm* (*Bk* # *b*, *a* # *lista*) = *False*
apply(*simp add*: *wprepare-loop-start-in-middle.simps*
wprepare-loop-goon-on-rightmost.simps, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*case-tac lm1*::*nat list*, *simp-all*, *case-tac list*, *simp*)
apply(*simp add*: *tape-of-nl-abv tape-of-nat-list.simps tape-of-nat-abv exp-ind-def*)
apply(*case-tac* [!] *rna*, *simp-all add*: *exp-ind-def*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*case-tac lm1*, *simp*, *case-tac list*, *simp*)
apply(*simp-all add*: *tape-of-nl-abv tape-of-nat-list.simps exp-ind-def tape-of-nat-abv*)
done

lemma [*simp*]:
⟦*lm* ≠ []; *wprepare-loop-start-in-middle m lm* (*b*, *Bk* # *a* # *lista*)⟧
⟹ *wprepare-loop-goon-in-middle m lm* (*Bk* # *b*, *a* # *lista*)
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*: *exp-ind-def*)
apply(*case-tac lm1*, *simp*)
apply(*rule-tac x* = *Suc aa* **in** *exI*, *simp*)
apply(*rule-tac x* = *list* **in** *exI*)
apply(*case-tac list*, *simp-all add*: *tape-of-nl-abv tape-of-nat-list.simps*)
done

lemma [*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*)))
apply(*simp only*: *wprepare-add-one.simps*, *auto*)
done

lemma [*simp*]: *wprepare-loop-start m lm* (*b*, *Oc* # *list*) ⟹ *b* ≠ []
apply(*simp*)
done

lemma [*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*, *auto*)
apply(*rule-tac x* = *rn* **in** *exI*, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*case-tac rn*, *auto simp*: *exp-ind-def*)
done

lemma [*simp*]: *wprepare-loop-start-in-middle m lm* (*b*, *Oc* # *list*) ⟹
wprepare-loop-start-in-middle m lm (*Oc* # *b*, *list*)
apply(*simp add*: *wprepare-loop-start-in-middle.simps*, *auto*)
apply(*rule-tac x* = *rn* **in** *exI*, *auto*)
apply(*case-tac mr*, *simp*, *simp add*: *exp-ind-def*)
apply(*rule-tac x* = *nat* **in** *exI*, *simp*)
apply(*rule-tac x* = *lm1* **in** *exI*, *simp*)
done

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 [*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 [*simp*]: *wprepare-goto-start-pos m lm* (*b*, *Oc* # *list*) ⟹ *b* ≠ []
apply(*simp add*: *wprepare-goto-start-pos.simps*)
done

lemma [*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 add*: *exp-ind-def*, *auto*)
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*: *exp-ind-def*)
apply(*rule-tac x* = *nat* **in** *exI*, *simp*)
apply(*rule-tac x* = *a#lista* **in** *exI*, *simp*)
done

lemma [*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(*case-tac lm1*, *simp-all add*: *wprepare-loop1 wprepare-loop2*)
done

lemma [*simp*]: *wprepare-loop-goon m lm* (*b*, *Oc # list*)
⟹ *wprepare-loop-start m lm* (*Oc # b*, *list*)
apply(*simp add*: *wprepare-loop-goon.simps*
wprepare-loop-start.simps)
done

lemma [*simp*]: *wprepare-add-one m lm* (*b*, *Oc # list*)
⟹ *b* = [] ⟶ *wprepare-add-one m lm* ([], *Bk # Oc # list*)
apply(*auto*)
apply(*simp add*: *wprepare-add-one.simps*)
done

lemma [*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(*rule-tac x* = *rn* **in** *exI*, *simp*)
apply(*simp add*: *tape-of-nat-abv tape-of-nat-list.simps exp-ind-def*, *auto*)
done

lemma [*simp*]: *wprepare-goto-start-pos m* (*a # aa # listaa*) (*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(rule-tac x = rn in exI, simp)
apply(simp add: tape-of-nl-abv tape-of-nat-list.simps exp-ind-def)
apply(rule-tac x = a in exI, rule-tac x = aa#listaa in exI, simp)
done

lemma [simp]: ⟦lm ≠ []; wprepare-goto-start-pos m lm (b, Oc # list)⟧
    ⟹ wprepare-loop-start m lm (Oc # b, list)
apply(case-tac lm, simp-all)
apply(case-tac lista, simp-all add: wprepare-loop-start.simps)
done

lemma [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-stop.simps 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. steps (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. steps (Suc 0, [], (<m # lm>)) t-wcode-prepare stp)
  have ∃ n. ?P (?f n) ∧ ?Q (?f n)
  proof(rule-tac halt-lemma2)
    show wf wcode-prepare-le by auto
  next
    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, case-tac ?f n,
            simp add: tstep-red tstep.simps)
      apply(case-tac c, simp, case-tac [2] aa)
      apply(simp-all add: wprepare-inv.simps wcode-prepare-le-def new-tape.simps
                      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
  next
    show ?Q (?f 0)
      apply(simp add: steps.simps wprepare-inv.simps wprepare-invs)
      done
  next
    show ¬ ?P (?f 0)
      apply(simp add: steps.simps)
      done
  qed
  thus ?thesis
    apply(auto)
    done
qed

lemma [intro]: t-correct t-wcode-prepare
apply(simp add: t-correct.simps t-wcode-prepare-def iseven-def)
apply(rule-tac x = 7 in exI, simp)
done

lemma twice-len-even: length (tm-of abc-twice) mod 2 = 0
apply(simp add: tm-even)
done

lemma fourtimes-len-even: length (tm-of abc-fourtimes) mod 2 = 0
apply(simp add: tm-even)
done

lemma t-correct-termi: t-correct tp ⟹
      list-all (λ(acn, st). (st ≤ Suc (length tp div 2))) (change-termi-state tp)
apply(auto simp: t-correct.simps List.list-all-length)
apply(erule-tac x = n in allE, simp)
apply(case-tac tp!n, auto simp: change-termi-state.simps split: if-splits)
done


lemma t-correct-shift:
         list-all (λ(acn, st). (st ≤ y)) tp ⟹
          list-all (λ(acn, st). (st ≤ y + off)) (tshift tp off)
apply(auto simp: t-correct.simps List.list-all-length)
apply(erule-tac x = n in allE, simp add: shift-length)
apply(case-tac tp!n, auto simp: tshift.simps)
done

lemma [intro]:
  t-correct (tm-of abc-twice @ tMp (Suc 0)
        (start-of twice-ly (length abc-twice) − Suc 0))
apply(rule-tac t-compiled-correct, simp-all)
```

```
apply(simp add: twice-ly-def)
done

lemma [intro]: t-correct (tm-of abc-fourtimes @ tMp (Suc 0)
  (start-of fourtimes-ly (length abc-fourtimes) − Suc 0))
apply(rule-tac t-compiled-correct, simp-all)
apply(simp add: fourtimes-ly-def)
done


lemma [intro]: t-correct t-wcode-main
apply(auto simp: t-wcode-main-def t-correct.simps shift-length
          t-twice-def t-fourtimes-def)
proof −
 show iseven (60 + (length (tm-of abc-twice) +
          length (tm-of abc-fourtimes)))
  using twice-len-even fourtimes-len-even
  apply(auto simp: iseven-def)
  apply(rule-tac x = 30 + q + qa in exI, simp)
  done
next
 show  list-all (λ(acn, s). s ≤ (60 + (length (tm-of abc-twice) +
       length (tm-of abc-fourtimes))) div 2) t-wcode-main-first-part
  apply(auto simp: t-wcode-main-first-part-def shift-length t-twice-def)
  done
next
 have list-all (λ(acn, s). s ≤ Suc (length (tm-of abc-twice @ tMp (Suc 0)
  (start-of twice-ly (length abc-twice) − Suc 0)) div 2))
  (change-termi-state (tm-of abc-twice @ tMp (Suc 0)
  (start-of twice-ly (length abc-twice) − Suc 0)))
  apply(rule-tac t-correct-termi, auto)
  done
 hence list-all (λ(acn, s). s ≤  Suc (length (tm-of abc-twice @ tMp (Suc 0)
  (start-of twice-ly (length abc-twice) − Suc 0)) div 2) + 12)
   (abacus.tshift (change-termi-state (tm-of abc-twice @ tMp (Suc 0)
       (start-of twice-ly (length abc-twice) − Suc 0))) 12)
  apply(rule-tac t-correct-shift, simp)
  done
 thus  list-all (λ(acn, s). s ≤
       (60 + (length (tm-of abc-twice) + length (tm-of abc-fourtimes))) div 2)
   (abacus.tshift (change-termi-state (tm-of abc-twice @ tMp (Suc 0)
          (start-of twice-ly (length abc-twice) − Suc 0))) 12)
  apply(simp)
  apply(simp add: list-all-length, auto)
  done
next
 have list-all (λ(acn, s). s ≤ Suc (length (tm-of abc-fourtimes @ tMp (Suc 0)
  (start-of fourtimes-ly (length abc-fourtimes) − Suc 0)) div 2))
   (change-termi-state (tm-of abc-fourtimes @ tMp (Suc 0)
```

(*start-of fourtimes-ly* (*length abc-fourtimes*) − *Suc 0*)))
apply(*rule-tac t-correct-termi*, *auto*)
done
hence *list-all* (λ(*acn*, *s*). *s* ≤ *Suc* (*length* (*tm-of abc-fourtimes* @ *tMp* (*Suc 0*)
(*start-of fourtimes-ly* (*length abc-fourtimes*) − *Suc 0*)) *div 2*) + (*t-twice-len* + *13*))
(*abacus.tshift* (*change-termi-state* (*tm-of abc-fourtimes* @ *tMp* (*Suc 0*)
(*start-of fourtimes-ly* (*length abc-fourtimes*) − *Suc 0*))) (*t-twice-len* + *13*))
apply(*rule-tac t-correct-shift*, *simp*)
done
thus *list-all* (λ(*acn*, *s*). *s* ≤ (*60* + (*length* (*tm-of abc-twice*) + *length* (*tm-of abc-fourtimes*))) *div 2*)
(*abacus.tshift* (*change-termi-state* (*tm-of abc-fourtimes* @ *tMp* (*Suc 0*)
(*start-of fourtimes-ly* (*length abc-fourtimes*) − *Suc 0*))) (*t-twice-len* + *13*))
apply(*simp add*: *t-twice-len-def t-twice-def*)
using *twice-len-even fourtimes-len-even*
apply(*auto simp*: *list-all-length*)
done
qed

lemma [*intro*]: *t-correct* (*t-wcode-prepare* |+| *t-wcode-main*)
apply(*auto intro*: *t-correct-add*)
done

lemma *prepare-mainpart-lemma*:
args ≠ [] ⟹
∃ *stp ln rn*. *steps* (*Suc 0*, [], <*m* # *args*>) (*t-wcode-prepare* |+| *t-wcode-main*) *stp*
= (*0*, *Bk* # $Oc^{Suc\ m}$, *Bk* # *Oc* # Bk^{ln} @ *Bk* # *Bk* # $Oc^{bl\text{-}bin\ (<args>)}$ @ Bk^{rn})
proof −
let *?P1* = λ (*l*, *r*). *l* = [] ∧ *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* # $Oc^{bl\text{-}bin\ (<args>)}$ @ Bk^{rn})
let *?P3* = λ *tp*. *False*
assume *h*: *args* ≠ []
have *?P1* ⊢−> λ *tp*. (∃ *stp tp*′. *steps* (*Suc 0*, *tp*)
(*t-wcode-prepare* |+| *t-wcode-main*) *stp* = (*0*, *tp*′) ∧ *?Q2 tp*′)
proof(*rule-tac turing-merge.t-merge-halt*[*of t-wcode-prepare t-wcode-main ?P1 ?P2 ?P3 ?P3 ?Q1 ?Q2*],
auto simp: *turing-merge-def*)
show ∃ *stp*. *case steps* (*Suc 0*, [], <*m* # *args*>) *t-wcode-prepare stp of* (*st*, *tp*′)
⇒ *st* = *0* ∧ *wprepare-stop m args tp*′
using *wprepare-correctness*[*of args m*] *h*
apply(*simp*, *auto*)
apply(*rule-tac x* = *n* **in** *exI*, *simp add*: *wprepare-inv.simps*)

done
next
fix *a b*
assume *wprepare-stop m args* (*a*, *b*)
thus ∃ *stp*. *case steps* (*Suc 0*, *a*, *b*) *t-wcode-main stp of*
(*st*, *tp′*) ⇒ (*st* = *0*) ∧ (*case tp′ of* (*l*, *r*) ⇒ *l* = *Bk* # $Oc^{Suc\ m}$ ∧
(∃ *ln rn*. *r* = *Bk* # *Oc* # Bk^{ln} @ *Bk* # *Bk* # $Oc^{bl\text{-}bin\ (<args>)}$ @ Bk^{rn}))
proof(*simp only*: *wprepare-stop.simps*, *erule-tac exE*)
fix *rn*
assume *a* = *Bk* # <*rev args*> @ *Bk* # *Bk* # $Oc^{Suc\ m}$ ∧
b = *Bk* # *Oc* # Bk^{rn}
thus *?thesis*
using *t-wcode-main-lemma-pre*[*of args* <*args*> *0* $Oc^{Suc\ m}$ *0 rn*] *h*
apply(*simp*)
apply(*erule-tac exE*)+
apply(*rule-tac x* = *stp* **in** *exI*, *simp add*: *tape-of-nl-rev*, *auto*)
done
qed
next
show *wprepare-stop m args* ⊢−> *wprepare-stop m args*
by(*simp add*: *t-imply-def*)
qed
thus ∃ *stp ln rn*. *steps* (*Suc 0*, [], <*m* # *args*>) (*t-wcode-prepare* |+| *t-wcode-main*) *stp*
= (*0*, *Bk* # $Oc^{Suc\ m}$, *Bk* # *Oc* # Bk^{ln} @ *Bk* # *Bk* # $Oc^{bl\text{-}bin\ (<args>)}$ @ Bk^{rn})
apply(*simp add*: *t-imply-def*)
apply(*erule-tac exE*)+
apply(*auto*)
done
qed

lemma [*simp*]: *tinres r r′* ⟹
fetch t ss (*case r of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*) =
fetch t ss (*case r′ of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*)
apply(*simp add*: *fetch.simps*, *auto split*: *if-splits simp*: *tinres-def*)
apply(*case-tac* [!] *r′*, *simp-all*)
apply(*case-tac* [!] *n*, *simp-all add*: *exp-ind-def*)
apply(*case-tac* [!] *r*, *simp-all*)
done

lemma [*intro*]: ∃ *n*. $(a{::}block)^n$ = []
by *auto*

lemma [*simp*]: ⟦*tinres r r′*; *r* ≠ []; *r′* ≠ []⟧ ⟹ *hd r* = *hd r′*
apply(*auto simp*: *tinres-def*)
done

lemma [*intro*]: *hd* $(Bk^{Suc\ n}) = Bk$
apply(*simp add*: *exp-ind-def*)
done

lemma [*simp*]: $[\![$*tinres r* []; $r \neq []]\!] \Longrightarrow$ *hd r* = *Bk*
apply(*auto simp*: *tinres-def*)
apply(*case-tac n*, *auto*)
done

lemma [*simp*]: $[\![$*tinres* [] r'; $r' \neq []]\!] \Longrightarrow$ *hd* $r' = Bk$
apply(*auto simp*: *tinres-def*)
done

lemma [*intro*]: $\exists\, na.\ tl\ r = tl\ (r\ @\ Bk^{n})\ @\ Bk^{na} \vee tl\ (r\ @\ Bk^{n}) = tl\ r\ @\ Bk^{na}$
apply(*case-tac r*, *simp*)
apply(*case-tac n*, *simp*)
apply(*rule-tac x* = *0* **in** *exI*, *simp*)
apply(*rule-tac x* = *nat* **in** *exI*, *simp add*: *exp-ind-def*)
apply(*simp*)
apply(*rule-tac x* = *n* **in** *exI*, *simp*)
done

lemma [*simp*]: *tinres r* $r' \Longrightarrow$ *tinres* (*tl r*) (*tl* r')
apply(*auto simp*: *tinres-def*)
apply(*case-tac* r', *simp-all*)
apply(*case-tac n*, *simp-all add*: *exp-ind-def*)
apply(*rule-tac x* = *0* **in** *exI*, *simp*)
apply(*rule-tac x* = *nat* **in** *exI*, *simp-all*)
apply(*rule-tac x* = *n* **in** *exI*, *simp*)
done

lemma [*simp*]: $[\![$*tinres r* []; $r \neq []]\!] \Longrightarrow$ *tinres* (*tl r*) []
apply(*case-tac r*, *auto simp*: *tinres-def*)
apply(*case-tac n*, *simp-all add*: *exp-ind-def*)
apply(*rule-tac x* = *nat* **in** *exI*, *simp*)
done

lemma [*simp*]: $[\![$*tinres* [] $r']\!] \Longrightarrow$ *tinres* [] (*tl* r')
apply(*case-tac* r', *auto simp*: *tinres-def*)
apply(*case-tac n*, *simp-all add*: *exp-ind-def*)
apply(*rule-tac x* = *nat* **in** *exI*, *simp*)
done

lemma [*simp*]: *tinres r* $r' \Longrightarrow$ *tinres* (*b* # *r*) (*b* # r')
apply(*auto simp*: *tinres-def*)
done

lemma *tinres-step2*:
$[\![$*tinres r* r'; *tstep* (*ss*, *l*, *r*) *t* = (*sa*, *la*, *ra*); *tstep* (*ss*, *l*, r') *t* = (*sb*, *lb*, *rb*)$]\!]$

$\Longrightarrow$ *la* = *lb* ∧ *tinres ra rb* ∧ *sa* = *sb*
apply(*case-tac ss* = *0*, *simp add*: *tstep-0*)
apply(*simp add*: *tstep.simps* [*simp del*])
apply(*case-tac fetch t ss* (*case r of* [] ⇒ *Bk* | *x* # *xs* ⇒ *x*), *simp*)
apply(*auto simp*: *new-tape.simps*)
apply(*simp-all split*: *taction.splits if-splits*)
apply(*auto*)
done

lemma *tinres-steps2*:
⟦*tinres r r′*; *steps* (*ss*, *l*, *r*) *t stp* = (*sa*, *la*, *ra*); *steps* (*ss*, *l*, *r′*) *t stp* = (*sb*, *lb*, *rb*)⟧
$\Longrightarrow$ *la* = *lb* ∧ *tinres ra rb* ∧ *sa* = *sb*
apply(*induct stp arbitrary*: *sa la ra sb lb rb*, *simp add*: *steps.simps*)
apply(*simp add*: *tstep-red*)
apply(*case-tac* (*steps* (*ss*, *l*, *r*) *t stp*))
apply(*case-tac* (*steps* (*ss*, *l*, *r′*) *t 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 stp* = (*sa*, *la*, *ra*);
steps (*ss*, *l*, *r′*) *t stp* = (*sb*, *lb*, *rb*)⟧ $\Longrightarrow$ *la* = *lb* ∧ *tinres ra rb* ∧ *sa* = *sb*
and *h*: *tinres r r′ tstep* (*steps* (*ss*, *l*, *r*) *t stp*) *t* = (*sa*, *la*, *ra*)
tstep (*steps* (*ss*, *l*, *r′*) *t stp*) *t* = (*sb*, *lb*, *rb*) *steps* (*ss*, *l*, *r*) *t stp* = (*a*, *b*, *c*)
steps (*ss*, *l*, *r′*) *t stp* = (*aa*, *ba*, *ca*)
have *b* = *ba* ∧ *tinres c ca* ∧ *a* = *aa*
apply(*rule-tac ind*, *simp-all add*: *h*)
done
thus *la* = *lb* ∧ *tinres ra rb* ∧ *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

definition *t-wcode-adjust* :: *tprog*
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 [*simp*]: *fetch t-wcode-adjust* (*Suc 0*) *Bk* = (*W1*, *1*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust* (*Suc 0*) *Oc* = (*R*, *2*)

apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust* (*Suc* (*Suc* *0*)) *Oc* = (*R*, *3*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust* (*Suc* (*Suc* (*Suc* *0*))) *Oc* = (*R*, *4*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust* (*Suc* (*Suc* (*Suc* *0*))) *Bk* = (*R*, *3*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 4 Bk* = (*L*, *8*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 4 Oc* = (*L*, *5*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 5 Oc* = (*W0*, *5*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 5 Bk* = (*L*, *6*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 6 Oc* = (*R*, *7*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 6 Bk* = (*L*, *6*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 7 Bk* = (*W1*, *2*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 8 Bk* = (*L*, *9*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 8 Oc* = (*W0*, *8*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)

done

lemma [*simp*]: *fetch t-wcode-adjust 9 Oc* = (*L*, *10*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 9 Bk* = (*L*, *9*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 10 Bk* = (*L*, *11*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 10 Oc* = (*L*, *10*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 11 Oc* = (*L*, *11*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

lemma [*simp*]: *fetch t-wcode-adjust 11 Bk* = (*R*, *0*)
apply(*simp add*: *fetch.simps t-wcode-adjust-def nth-of.simps*)
done

fun *wadjust-start* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
  where
  wadjust-start m rs (*l*, *r*) =
    $(\exists\ ln\ rn.\ l = Bk\ \#\ Oc^{Suc\ m} \wedge$
      $tl\ r = Oc\ \#\ Bk^{ln}\ @\ Bk\ \#\ Oc^{Suc\ rs}\ @\ Bk^{rn})$

fun *wadjust-loop-start* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
  where
  wadjust-loop-start m rs (*l*, *r*) =
    $(\exists\ ln\ rn\ ml\ mr.\ l = Oc^{ml}\ @\ Bk\ \#\ Oc^{Suc\ m} \wedge$
      $r = Oc\ \#\ Bk^{ln}\ @\ Bk\ \#\ Oc^{mr}\ @\ Bk^{rn} \wedge$
      $ml + mr = Suc\ (Suc\ rs) \wedge mr > 0)$

fun *wadjust-loop-right-move* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
  where
  wadjust-loop-right-move m rs (*l*, *r*) =
  $(\exists\ ml\ mr\ nl\ nr\ rn.\ l = Bk^{nl}\ @\ Oc\ \#\ Oc^{ml}\ @\ Bk\ \#\ Oc^{Suc\ m} \wedge$
    $r = Bk^{nr}\ @\ Oc^{mr}\ @\ Bk^{rn} \wedge$
    $ml + mr = Suc\ (Suc\ rs) \wedge mr > 0 \wedge$
    $nl + nr > 0)$

fun *wadjust-loop-check* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
  where

wadjust-loop-check m rs (*l*, *r*) =
(∃ *ml mr ln rn*. $l = Oc \# Bk^{ln} @ Bk \# Oc \# Oc^{ml} @ Bk \# Oc^{Suc\ m} \wedge$
$r = Oc^{mr} @ Bk^{rn} \wedge ml + mr = (Suc\ rs)$)

fun *wadjust-loop-erase* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-loop-erase m rs (*l*, *r*) =
(∃ *ml mr ln rn*. $l = Bk^{ln} @ Bk \# Oc \# Oc^{ml} @ Bk \# Oc^{Suc\ m} \wedge$
$tl\ r = Oc^{mr} @ Bk^{rn} \wedge ml + mr = (Suc\ rs) \wedge mr > 0$)

fun *wadjust-loop-on-left-moving-O* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-loop-on-left-moving-O m rs (*l*, *r*) =
(∃ *ml mr ln rn*. $l = Oc^{ml} @ Bk \# Oc^{Suc\ m} \wedge$
$r = Oc \# Bk^{ln} @ Bk \# Bk \# Oc^{mr} @ Bk^{rn} \wedge$
$ml + mr = Suc\ rs \wedge mr > 0$)

fun *wadjust-loop-on-left-moving-B* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-loop-on-left-moving-B m rs (*l*, *r*) =
(∃ *ml mr nl nr rn*. $l = Bk^{nl} @ Oc \# Oc^{ml} @ Bk \# Oc^{Suc\ m} \wedge$
$r = Bk^{nr} @ Bk \# Bk \# Oc^{mr} @ Bk^{rn} \wedge$
$ml + mr = Suc\ rs \wedge mr > 0$)

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*) ∨
wadjust-loop-on-left-moving-B m rs (*l*, *r*))

fun *wadjust-loop-right-move2* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-loop-right-move2 m rs (*l*, *r*) =
(∃ *ml mr ln rn*. $l = Oc \# Oc^{ml} @ Bk \# Oc^{Suc\ m} \wedge$
$r = Bk^{ln} @ Bk \# Bk \# Oc^{mr} @ Bk^{rn} \wedge$
$ml + mr = Suc\ rs \wedge mr > 0$)

fun *wadjust-erase2* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-erase2 m rs (*l*, *r*) =
(∃ *ln rn*. $l = Bk^{ln} @ Bk \# Oc \# Oc^{Suc\ rs} @ Bk \# Oc^{Suc\ m} \wedge$
$tl\ r = Bk^{rn}$)

fun *wadjust-on-left-moving-O* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-on-left-moving-O m rs (*l*, *r*) =
(∃ *rn*. $l = Oc^{Suc\ rs} @ Bk \# Oc^{Suc\ m} \wedge$
$r = Oc \# Bk^{rn}$)

fun *wadjust-on-left-moving-B* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-on-left-moving-B m rs (*l*, *r*) =
$(\exists\ ln\ rn.\ l = Bk^{ln} @ Oc \# Oc^{Suc\ rs} @ Bk \# Oc^{Suc\ m} \wedge$
$r = Bk^{rn})$

fun *wadjust-on-left-moving* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-on-left-moving m rs (*l*, *r*) =
(*wadjust-on-left-moving-O m rs* (*l*, *r*) ∨
wadjust-on-left-moving-B m rs (*l*, *r*))

fun *wadjust-goon-left-moving-B* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-goon-left-moving-B m rs (*l*, *r*) =
$(\exists\ rn.\ l = Oc^{Suc\ m} \wedge$
$r = Bk \# Oc^{Suc\ (Suc\ rs)} @ Bk^{rn})$

fun *wadjust-goon-left-moving-O* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-goon-left-moving-O m rs (*l*, *r*) =
$(\exists\ ml\ mr\ rn.\ l = Oc^{ml} @ Bk \# Oc^{Suc\ m} \wedge$
$r = Oc^{mr} @ Bk^{rn} \wedge$
$ml + mr = Suc\ (Suc\ rs) \wedge mr > 0)$

fun *wadjust-goon-left-moving* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-goon-left-moving m rs (*l*, *r*) =
(*wadjust-goon-left-moving-B m rs* (*l*, *r*) ∨
wadjust-goon-left-moving-O m rs (*l*, *r*))

fun *wadjust-backto-standard-pos-B* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-backto-standard-pos-B m rs (*l*, *r*) =
$(\exists\ rn.\ l = [] \wedge$
$r = Bk \# Oc^{Suc\ m} @ Bk \# Oc^{Suc\ (Suc\ rs)} @ Bk^{rn})$

fun *wadjust-backto-standard-pos-O* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-backto-standard-pos-O m rs (*l*, *r*) =
$(\exists\ ml\ mr\ rn.\ l = Oc^{ml} \wedge$
$r = Oc^{mr} @ Bk \# Oc^{Suc\ (Suc\ rs)} @ Bk^{rn} \wedge$
$ml + mr = Suc\ m \wedge mr > 0)$

fun *wadjust-backto-standard-pos* :: *nat* ⇒ *nat* ⇒ *tape* ⇒ *bool*
where
wadjust-backto-standard-pos m rs (*l*, *r*) =

(*wadjust-backto-standard-pos-B m rs* (*l*, *r*) ∨
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* ⇒ *t-conf* ⇒ *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)

thm dropWhile.simps

fun wadjust-stage :: nat ⇒ t-conf ⇒ 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 ⇒ t-conf ⇒ 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 ⇒ t-conf ⇒ 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 × t-conf) ⇒ 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 × t-conf) × nat × t-conf) set
  where wadjust-le ≡ (inv-image lex-square wadjust-measure)

lemma [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 [*simp*]: *wadjust-start m rs* (*c*, []) = *False*
apply(*auto simp*: *wadjust-start.simps*)
done

lemma [*simp*]: *wadjust-loop-right-move m rs* (*c*, []) $\Longrightarrow$ *c* $\neq$ []
apply(*auto simp*: *wadjust-loop-right-move.simps*)
done

lemma [*simp*]: *wadjust-loop-right-move m rs* (*c*, [])
$\Longrightarrow$ *wadjust-loop-check m rs* (*Bk* # *c*, [])
apply(*simp only*: *wadjust-loop-right-move.simps wadjust-loop-check.simps*)
apply(*auto*)
apply(*case-tac* [!] *mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: *wadjust-loop-check m rs* (*c*, []) $\Longrightarrow$ *c* $\neq$ []
apply(*simp only*: *wadjust-loop-check.simps*, *auto*)
done

lemma [*simp*]: *wadjust-loop-start m rs* (*c*, []) = *False*
apply(*simp add*: *wadjust-loop-start.simps*)
done

lemma [*simp*]: *wadjust-loop-right-move m rs* (*c*, []) $\Longrightarrow$
wadjust-loop-right-move m rs (*Bk* # *c*, [])
apply(*simp only*: *wadjust-loop-right-move.simps*)
apply(*erule-tac exE*)+
apply(*auto*)
apply(*case-tac* [!] *mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: *wadjust-loop-check m rs* (*c*, []) $\Longrightarrow$ *wadjust-erase2 m rs* (*tl c*, [*hd c*])
apply(*simp only*: *wadjust-loop-check.simps wadjust-erase2.simps*, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*, *auto*)
done

lemma [*simp*]: *wadjust-loop-erase m rs* (*c*, [])
$\Longrightarrow$ (*c* = [] $\longrightarrow$ *wadjust-loop-on-left-moving m rs* ([], [*Bk*])) $\wedge$
(*c* $\neq$ [] $\longrightarrow$ *wadjust-loop-on-left-moving m rs* (*tl c*, [*hd c*]))
apply(*simp add*: *wadjust-loop-erase.simps*, *auto*)
apply(*case-tac* [!] *mr*, *simp-all add*: *exp-ind-def*)

done

lemma [*simp*]: *wadjust-loop-on-left-moving m rs* (*c*, []) = *False*
apply(*auto simp*: *wadjust-loop-on-left-moving.simps*)
done

lemma [*simp*]: *wadjust-loop-right-move2 m rs* (*c*, []) = *False*
apply(*auto simp*: *wadjust-loop-right-move2.simps*)
done

lemma [*simp*]: *wadjust-erase2 m rs* ([], []) = *False*
apply(*auto simp*: *wadjust-erase2.simps*)
done

lemma [*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*)
apply(*rule-tac x* = *0* **in** *exI*, *simp add*: *exp-ind-def*)
done

lemma [*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 exp-ind-def*, *auto*)
apply(*rule-tac x* = *Suc n* **in** *exI*, *simp add*: *exp-ind*)
done

lemma [*simp*]: ⟦*wadjust-erase2 m rs* (*c*, []); *c* ≠ []⟧ ⟹
wadjust-on-left-moving m rs (*tl c*, [*hd c*])
apply(*simp only*: *wadjust-erase2.simps*)
apply(*erule-tac exE*)+
apply(*case-tac ln*, *simp-all add*: *exp-ind-def wadjust-on-left-moving.simps*)
done

lemma [*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 [*simp*]: *wadjust-on-left-moving m rs* ([], []) = *False*
apply(*simp add*: *wadjust-on-left-moving.simps*
wadjust-on-left-moving-O.simps wadjust-on-left-moving-B.simps)
done

lemma [*simp*]: *wadjust-on-left-moving-O m rs* (*c*, []) = *False*
apply(*simp add*: *wadjust-on-left-moving-O.simps*)
done

lemma [*simp*]: ⟦*wadjust-on-left-moving-B m rs* (*c*, []); *c* ≠ []; *hd c* = *Bk*⟧ ⟹
wadjust-on-left-moving-B m rs (*tl c*, [*Bk*])
apply(*simp add*: *wadjust-on-left-moving-B.simps*, *auto*)
apply(*case-tac* [!] *ln*, *simp-all add*: *exp-ind-def*, *auto*)
done

lemma [*simp*]: ⟦*wadjust-on-left-moving-B m rs* (*c*, []); *c* ≠ []; *hd c* = *Oc*⟧ ⟹
wadjust-on-left-moving-O m rs (*tl c*, [*Oc*])
apply(*simp add*: *wadjust-on-left-moving-B.simps wadjust-on-left-moving-O.simps*, *auto*)
apply(*case-tac* [!] *ln*, *simp-all add*: *exp-ind-def*)
done

lemma [*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 [*simp*]: *wadjust-on-left-moving 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 [*simp*]: *wadjust-goon-left-moving m rs* (*c*, []) = *False*
apply(*auto simp*: *wadjust-goon-left-moving.simps wadjust-goon-left-moving-B.simps*
wadjust-goon-left-moving-O.simps)
done

lemma [*simp*]: *wadjust-backto-standard-pos m rs* (*c*, []) = *False*
apply(*auto simp*: *wadjust-backto-standard-pos.simps*
wadjust-backto-standard-pos-B.simps wadjust-backto-standard-pos-O.simps)
done

lemma [*simp*]:
wadjust-start m rs (*c*, *Bk* # *list*) ⟹
(*c* = [] ⟶ *wadjust-start m rs* ([], *Oc* # *list*)) ∧
(*c* ≠ [] ⟶ *wadjust-start m rs* (*c*, *Oc* # *list*))
apply(*auto simp*: *wadjust-start.simps*)
done

lemma [*simp*]: *wadjust-loop-start m rs* (*c*, *Bk* # *list*) = *False*
apply(*auto simp*: *wadjust-loop-start.simps*)
done

lemma [*simp*]: *wadjust-loop-right-move m rs* (*c*, *b*) ⟹ *c* ≠ []
apply(*simp only*: *wadjust-loop-right-move.simps*, *auto*)
done

lemma [*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(*rule-tac x* = *ml* **in** *exI*, *simp*)
apply(*rule-tac x* = *mr* **in** *exI*, *simp*)
apply(*rule-tac x* = *Suc nl* **in** *exI*, *simp add*: *exp-ind-def*)
apply(*case-tac nr*, *simp*, *case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*rule-tac x* = *nat* **in** *exI*, *auto*)
done

lemma [*simp*]: *wadjust-loop-check m rs* (*c*, *b*) ⟹ *c* ≠ []
apply(*simp only*: *wadjust-loop-check.simps*, *auto*)
done

lemma [*simp*]: *wadjust-loop-check m rs* (*c*, *Bk* # *list*)
  ⟹ *wadjust-erase2 m rs* (*tl c*, *hd c* # *Bk* # *list*)
apply(*auto simp*: *wadjust-loop-check.simps wadjust-erase2.simps*)
apply(*case-tac* [!] *mr*, *simp-all add*: *exp-ind-def*, *auto*)
done

lemma [*simp*]: *wadjust-loop-erase m rs* (*c*, *b*) ⟹ *c* ≠ []
apply(*simp only*: *wadjust-loop-erase.simps*, *auto*)
done

declare *wadjust-loop-on-left-moving-O.simps*[*simp del*]
  wadjust-loop-on-left-moving-B.simps[*simp del*]

lemma [*simp*]: ⟦*wadjust-loop-erase m rs* (*c*, *Bk* # *list*); *hd c* = *Bk*⟧
  ⟹ *wadjust-loop-on-left-moving-B m rs* (*tl c*, *Bk* # *Bk* # *list*)
apply(*simp only*: *wadjust-loop-erase.simps*
  wadjust-loop-on-left-moving-B.simps)
apply(*erule-tac exE*)+
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*, *simp*)
apply(*case-tac ln*, *simp-all add*: *exp-ind-def*, *auto*)
apply(*simp add*: *exp-ind exp-ind-def*[*THEN sym*])
done

lemma [*simp*]: ⟦*wadjust-loop-erase m rs* (*c*, *Bk* # *list*); *c* ≠ []; *hd c* = *Oc*⟧ ⟹
  wadjust-loop-on-left-moving-O m rs (*tl c*, *Oc* # *Bk* # *list*)
apply(*simp only*: *wadjust-loop-erase.simps wadjust-loop-on-left-moving-O.simps*,
  auto)
apply(*case-tac* [!] *ln*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: ⟦*wadjust-loop-erase m rs* (*c*, *Bk* # *list*); *c* ≠ []⟧ ⟹
  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 [*simp*]: *wadjust-loop-on-left-moving m rs* (*c*, *b*) $\Longrightarrow$ *c* $\neq$ []
apply(*simp add*: *wadjust-loop-on-left-moving.simps*
wadjust-loop-on-left-moving-O.simps wadjust-loop-on-left-moving-B.simps, *auto*)
done

lemma [*simp*]: *wadjust-loop-on-left-moving-O m rs* (*c*, *Bk* # *list*) = *False*
apply(*simp add*: *wadjust-loop-on-left-moving-O.simps*)
done

lemma [*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*)+
apply(*rule-tac x* = *ml* **in** *exI*, *rule-tac x* = *mr* **in** *exI*)
apply(*case-tac nl*, *simp-all add*: *exp-ind-def*, *auto*)
apply(*rule-tac x* = *Suc nr* **in** *exI*, *auto simp*: *exp-ind-def*)
done

lemma [*simp*]: ⟦*wadjust-loop-on-left-moving-B m rs* (*c*, *Bk* # *list*); *hd c* = *Oc*⟧
$\Longrightarrow$ *wadjust-loop-on-left-moving-O m rs* (*tl c*, *Oc* # *Bk* # *list*)
apply(*simp only*: *wadjust-loop-on-left-moving-O.simps*
wadjust-loop-on-left-moving-B.simps)
apply(*erule-tac exE*)+
apply(*rule-tac x* = *ml* **in** *exI*, *rule-tac x* = *mr* **in** *exI*)
apply(*case-tac nl*, *simp-all add*: *exp-ind-def*, *auto*)
done

lemma [*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 [*simp*]: *wadjust-loop-right-move2 m rs* (*c*, *b*) $\Longrightarrow$ *c* $\neq$ []
apply(*simp only*: *wadjust-loop-right-move2.simps*, *auto*)
done

lemma [*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*)
apply(*case-tac ln*, *simp-all add*: *exp-ind-def*)
apply(*rule-tac x* = *0* **in** *exI*, *simp*)
apply(*rule-tac x* = *rn* **in** *exI*, *simp*)
apply(*rule-tac x* = *Suc ml* **in** *exI*, *simp add*: *exp-ind-def*, *auto*)
apply(*rule-tac x* = *Suc nat* **in** *exI*, *simp add*: *exp-ind*)
apply(*rule-tac x* = *rn* **in** *exI*, *auto*)

apply(*rule-tac x* = *Suc ml* **in** *exI*, *auto simp*: *exp-ind-def*)
done

lemma [*simp*]: *wadjust-erase2 m rs* (*c*, *Bk* # *list*) $\Longrightarrow$ *c* $\neq$ []
apply(*auto simp*:*wadjust-erase2.simps*)
done

lemma [*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*)
apply(*case-tac ln*, *simp-all add*: *exp-ind-def wadjust-on-left-moving.simps*
wadjust-on-left-moving-O.simps wadjust-on-left-moving-B.simps)
apply(*auto*)
apply(*rule-tac x* = (*Suc* (*Suc rn*)) **in** *exI*, *simp add*: *exp-ind-def*)
apply(*rule-tac x* = *Suc nat* **in** *exI*, *simp add*: *exp-ind*)
apply(*rule-tac x* = (*Suc* (*Suc rn*)) **in** *exI*, *simp add*: *exp-ind-def*)
done

lemma [*simp*]: *wadjust-on-left-moving m rs* (*c*,*b*) $\Longrightarrow$ *c* $\neq$ []
apply(*simp only*:*wadjust-on-left-moving.simps*
wadjust-on-left-moving-O.simps
wadjust-on-left-moving-B.simps
, *auto*)
done

lemma [*simp*]: *wadjust-on-left-moving-O m rs* (*c*, *Bk* # *list*) = *False*
apply(*simp add*: *wadjust-on-left-moving-O.simps*)
done

lemma [*simp*]: ⟦*wadjust-on-left-moving-B m rs* (*c*, *Bk* # *list*); *hd c* = *Bk*⟧
$\Longrightarrow$ *wadjust-on-left-moving-B m rs* (*tl c*, *Bk* # *Bk* # *list*)
apply(*auto simp*: *wadjust-on-left-moving-B.simps*)
apply(*case-tac ln*, *simp-all add*: *exp-ind-def*, *auto*)
done

lemma [*simp*]: ⟦*wadjust-on-left-moving-B m rs* (*c*, *Bk* # *list*); *hd c* = *Oc*⟧
$\Longrightarrow$ *wadjust-on-left-moving-O m rs* (*tl c*, *Oc* # *Bk* # *list*)
apply(*auto simp*: *wadjust-on-left-moving-O.simps*
wadjust-on-left-moving-B.simps)
apply(*case-tac ln*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: *wadjust-on-left-moving m rs* (*c*, *Bk* # *list*) $\Longrightarrow$
wadjust-on-left-moving m rs (*tl c*, *hd c* # *Bk* # *list*)
apply(*simp add*: *wadjust-on-left-moving.simps*)
apply(*case-tac hd c*, *simp-all*)
done

lemma [*simp*]: *wadjust-goon-left-moving m rs* (*c*, *b*) $\Longrightarrow$ *c* $\neq$ []

apply(*simp add*: *wadjust-goon-left-moving.simps*
wadjust-goon-left-moving-B.simps
wadjust-goon-left-moving-O.simps exp-ind-def, *auto*)
done

lemma [*simp*]: *wadjust-goon-left-moving-O m rs* (*c*, *Bk* # *list*) = *False*
apply(*simp add*: *wadjust-goon-left-moving-O.simps*, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: ⟦*wadjust-goon-left-moving-B m rs* (*c*, *Bk* # *list*); *hd c* = *Bk*⟧
⟹ *wadjust-backto-standard-pos-B m rs* (*tl c*, *Bk* # *Bk* # *list*)
apply(*auto simp*: *wadjust-goon-left-moving-B.simps*
wadjust-backto-standard-pos-B.simps exp-ind-def)
done

lemma [*simp*]: ⟦*wadjust-goon-left-moving-B m rs* (*c*, *Bk* # *list*); *hd c* = *Oc*⟧
⟹ *wadjust-backto-standard-pos-O m rs* (*tl c*, *Oc* # *Bk* # *list*)
apply(*auto simp*: *wadjust-goon-left-moving-B.simps*
wadjust-backto-standard-pos-O.simps exp-ind-def)
apply(*rule-tac x* = *m* **in** *exI*, *simp*, *auto*)
done

lemma [*simp*]: *wadjust-goon-left-moving m rs* (*c*, *Bk* # *list*) ⟹
wadjust-backto-standard-pos m rs (*tl c*, *hd c* # *Bk* # *list*)
apply(*case-tac hd c*, *simp-all add*: *wadjust-backto-standard-pos.simps*
wadjust-goon-left-moving.simps)
done

lemma [*simp*]: *wadjust-backto-standard-pos m rs* (*c*, *Bk* # *list*) ⟹
(*c* = [] ⟶ *wadjust-stop m rs* ([*Bk*], *list*)) ∧ (*c* ≠ [] ⟶ *wadjust-stop m rs* (*Bk* # *c*, *list*))
apply(*auto simp*: *wadjust-backto-standard-pos.simps*
wadjust-backto-standard-pos-B.simps
wadjust-backto-standard-pos-O.simps wadjust-stop.simps)
apply(*case-tac* [!] *mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: *wadjust-start m rs* (*c*, *Oc* # *list*)
⟹ (*c* = [] ⟶ *wadjust-loop-start m rs* ([*Oc*], *list*)) ∧
(*c* ≠ [] ⟶ *wadjust-loop-start m rs* (*Oc* # *c*, *list*))
apply(*auto simp*:*wadjust-loop-start.simps wadjust-start.simps*)
apply(*rule-tac x* = *ln* **in** *exI*, *rule-tac x* = *rn* **in** *exI*,
rule-tac x = *Suc 0* **in** *exI*, *simp*)
done

lemma [*simp*]: *wadjust-loop-start m rs* (*c*, *b*) ⟹ *c* ≠ []
apply(*simp add*: *wadjust-loop-start.simps*, *auto*)
done

lemma [*simp*]: *wadjust-loop-start m rs* (*c*, *Oc* # *list*)
  ⟹ *wadjust-loop-right-move m rs* (*Oc* # *c*, *list*)
apply(*simp add*: *wadjust-loop-start.simps wadjust-loop-right-move.simps*, *auto*)
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*, *auto*)
done

lemma [*simp*]: *wadjust-loop-right-move m rs* (*c*, *Oc* # *list*) ⟹
  wadjust-loop-check m rs (*Oc* # *c*, *list*)
apply(*simp add*: *wadjust-loop-right-move.simps*
  wadjust-loop-check.simps, *auto*)
apply(*rule-tac* [!] *x* = *ml* **in** *exI*, *simp-all*, *auto*)
apply(*case-tac nl*, *auto simp*: *exp-ind-def*)
apply(*rule-tac x* = *mr* − *1* **in** *exI*, *case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*case-tac* [!] *nr*, *simp-all add*: *exp-ind-def*, *auto*)
done

lemma [*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*)+
apply(*rule-tac x* = *ml* **in** *exI*, *rule-tac x* = *mr* **in** *exI*, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
apply(*case-tac rn*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: *wadjust-loop-erase m rs* (*c*, *Oc* # *list*) ⟹
  wadjust-loop-erase m rs (*c*, *Bk* # *list*)
apply(*auto simp*: *wadjust-loop-erase.simps*)
done

lemma [*simp*]: *wadjust-loop-on-left-moving-B m rs* (*c*, *Oc* # *list*) = *False*
apply(*auto simp*: *wadjust-loop-on-left-moving-B.simps*)
apply(*case-tac nr*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: *wadjust-loop-on-left-moving m rs* (*c*, *Oc* # *list*)
  ⟹ *wadjust-loop-right-move2 m rs* (*Oc* # *c*, *list*)
apply(*simp add*:*wadjust-loop-on-left-moving.simps*)
apply(*auto simp*: *wadjust-loop-on-left-moving-O.simps*
  wadjust-loop-right-move2.simps)
done

lemma [*simp*]: *wadjust-loop-right-move2 m rs* (*c*, *Oc* # *list*) = *False*
apply(*auto simp*: *wadjust-loop-right-move2.simps*)
apply(*case-tac ln*, *simp-all add*: *exp-ind-def*)
done

```
lemma [simp]: wadjust-erase2 m rs (c, Oc # list)
            ⟹ (c = [] ⟶ wadjust-erase2 m rs ([], Bk # list))
             ∧ (c ≠ [] ⟶ wadjust-erase2 m rs (c, Bk # list))
apply(auto simp: wadjust-erase2.simps )
done

lemma [simp]: wadjust-on-left-moving-B m rs (c, Oc # list) = False
apply(auto simp: wadjust-on-left-moving-B.simps)
done

lemma [simp]: ⟦wadjust-on-left-moving-O m rs (c, Oc # list); hd c = Bk⟧ ⟹
        wadjust-goon-left-moving-B m rs (tl c, Bk # Oc # list)
apply(auto simp: wadjust-on-left-moving-O.simps
     wadjust-goon-left-moving-B.simps exp-ind-def)
done

lemma [simp]: ⟦wadjust-on-left-moving-O m rs (c, Oc # list); hd c = Oc⟧
   ⟹ wadjust-goon-left-moving-O m rs (tl c, Oc # Oc # list)
apply(auto simp: wadjust-on-left-moving-O.simps
              wadjust-goon-left-moving-O.simps exp-ind-def)
apply(rule-tac x = rs in exI, simp)
apply(auto simp: exp-ind-def numeral-2-eq-2)
done


lemma [simp]: wadjust-on-left-moving m rs (c, Oc # list) ⟹
            wadjust-goon-left-moving m rs (tl c, hd c # Oc # list)
apply(simp add: wadjust-on-left-moving.simps
              wadjust-goon-left-moving.simps)
apply(case-tac hd c, simp-all)
done

lemma [simp]: wadjust-on-left-moving m rs (c, Oc # list) ⟹
  wadjust-goon-left-moving m rs (tl c, hd c # Oc # list)
apply(simp add: wadjust-on-left-moving.simps
  wadjust-goon-left-moving.simps)
apply(case-tac hd c, simp-all)
done

lemma [simp]: wadjust-goon-left-moving-B m rs (c, Oc # list) = False
apply(auto simp: wadjust-goon-left-moving-B.simps)
done

lemma [simp]: ⟦wadjust-goon-left-moving-O m rs (c, Oc # list); hd c = Bk⟧
             ⟹ wadjust-goon-left-moving-B m rs (tl c, Bk # Oc # list)
apply(auto simp: wadjust-goon-left-moving-O.simps wadjust-goon-left-moving-B.simps)
apply(case-tac [!] ml, auto simp: exp-ind-def)
done
```

lemma [*simp*]: ⟦*wadjust-goon-left-moving-O m rs* (*c*, *Oc* # *list*); *hd c* = *Oc*⟧ ⟹

wadjust-goon-left-moving-O m rs (*tl c*, *Oc* # *Oc* # *list*)
apply(*auto simp*: *wadjust-goon-left-moving-O.simps wadjust-goon-left-moving-B.simps*)
apply(*rule-tac x* = *ml* − *1* **in** *exI*, *simp*)
apply(*case-tac ml*, *simp-all add*: *exp-ind-def*)
apply(*rule-tac x* = *Suc mr* **in** *exI*, *auto simp*: *exp-ind-def*)
done

lemma [*simp*]: *wadjust-goon-left-moving m rs* (*c*, *Oc* # *list*) ⟹
wadjust-goon-left-moving m rs (*tl c*, *hd c* # *Oc* # *list*)
apply(*simp add*: *wadjust-goon-left-moving.simps*)
apply(*case-tac hd c*, *simp-all*)
done

lemma [*simp*]: *wadjust-backto-standard-pos-B m rs* (*c*, *Oc* # *list*) = *False*
apply(*simp add*: *wadjust-backto-standard-pos-B.simps*)
done

lemma [*simp*]: *wadjust-backto-standard-pos-O m rs* (*c*, *Bk* # *xs*) = *False*
apply(*simp add*: *wadjust-backto-standard-pos-O.simps*, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: *wadjust-backto-standard-pos-O m rs* ([], *Oc* # *list*) ⟹
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)
apply(*rule-tac x* = *rn* **in** *exI*, *simp*)
apply(*case-tac ml*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]:
⟦*wadjust-backto-standard-pos-O m rs* (*c*, *Oc* # *list*); *c* ≠ []; *hd c* = *Bk*⟧
⟹ *wadjust-backto-standard-pos-B m rs* (*tl c*, *Bk* # *Oc* # *list*)
apply(*simp add*:*wadjust-backto-standard-pos-O.simps*
wadjust-backto-standard-pos-B.simps, *auto*)
apply(*case-tac* [!] *ml*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: ⟦*wadjust-backto-standard-pos-O m rs* (*c*, *Oc* # *list*); *c* ≠ []; *hd c* =
Oc⟧
⟹ *wadjust-backto-standard-pos-O m rs* (*tl c*, *Oc* # *Oc* # *list*)
apply(*simp add*: *wadjust-backto-standard-pos-O.simps*, *auto*)
apply(*case-tac ml*, *simp-all add*: *exp-ind-def*, *auto*)

apply(*rule-tac x* = *nat* **in** *exI*, *auto simp*: *exp-ind-def*)
done

lemma [*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
thm *wadjust-loop-right-move.simps*

lemma [*simp*]: *wadjust-loop-right-move m rs* (*c*, []) = *False*
apply(*simp only*: *wadjust-loop-right-move.simps*)
apply(*rule-tac iffI*)
apply(*erule-tac exE*)+
apply(*case-tac nr*, *simp-all add*: *exp-ind-def*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: *wadjust-loop-erase m rs* (*c*, []) = *False*
apply(*simp only*: *wadjust-loop-erase.simps*, *auto*)
apply(*case-tac mr*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: ⟦*Suc* (*Suc rs*) = *a*; *wadjust-loop-erase m rs* (*c*, *Bk* # *list*)⟧
$\Longrightarrow$ *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*)))) $\vee$
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 [*simp*]:
⟦*Suc* (*Suc rs*) = *a*; *wadjust-loop-on-left-moving m rs* (*c*, *Bk* # *list*)⟧
$\Longrightarrow$ *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*)))) $\vee$
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(*subgoal-tac c* $\neq$ [])

apply(*case-tac c*, *simp-all*)
done

lemma *dropWhile-exp1*: *dropWhile* ($\lambda a.$ *a* = *Oc*) (Oc^n @ *xs*) = *dropWhile* ($\lambda a.$ *a* = *Oc*) *xs*
apply(*induct n*, *simp-all add*: *exp-ind-def*)
done
lemma *takeWhile-exp1*: *takeWhile* ($\lambda a.$ *a* = *Oc*) (Oc^n @ *xs*) = Oc^n @ *takeWhile* ($\lambda a.$ *a* = *Oc*) *xs*
apply(*induct n*, *simp-all add*: *exp-ind-def*)
done

lemma [*simp*]: ⟦*Suc* (*Suc rs*) = *a*; *wadjust-loop-right-move2 m rs* (*c*, *Bk* # *list*)⟧
⟹ *a* − *length* (*takeWhile* ($\lambda a.$ *a* = *Oc*) (*tl* (*dropWhile* ($\lambda a.$ *a* = *Oc*) (*rev c* @ *Oc* # *list*))))
< *a* − *length* (*takeWhile* ($\lambda a.$ *a* = *Oc*) (*tl* (*dropWhile* ($\lambda a.$ *a* = *Oc*) (*rev c* @ *Bk* # *list*))))
apply(*simp add*: *wadjust-loop-right-move2.simps*, *auto*)
apply(*simp add*: *dropWhile-exp1 takeWhile-exp1*)
apply(*case-tac ln*, *simp*, *simp add*: *exp-ind-def*)
done

lemma [*simp*]: *wadjust-loop-check m rs* ([], *b*) = *False*
apply(*simp add*: *wadjust-loop-check.simps*)
done

lemma [*simp*]: ⟦*Suc* (*Suc rs*) = *a*; *wadjust-loop-check m rs* (*c*, *Oc* # *list*)⟧
⟹ *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*)))) ∨
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 [*simp*]:
⟦*Suc* (*Suc rs*) = *a*; *wadjust-loop-erase m rs* (*c*, *Oc* # *list*)⟧
⟹ *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*)))) ∨
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

declare *numeral-2-eq-2*[*simp del*]

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*), *steps* (*Suc 0*, *Bk* # $Oc^{Suc\ m}$,
  Bk # *Oc* # Bk^{ln} @ *Bk* # $Oc^{Suc\ rs}$ @ Bk^{rn}) *t-wcode-adjust stp*)) *in*
  $\exists$ *n* .*P* (*f n*) $\wedge$ *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*), *steps* (*Suc 0*, *Bk* # $Oc^{Suc\ m}$,
  Bk # *Oc* # Bk^{ln} @ *Bk* # $Oc^{Suc\ rs}$ @ Bk^{rn}) *t-wcode-adjust stp*)
  have $\exists$ *n*. *?P* (*?f n*) $\wedge$ *?Q* (*?f n*)
  proof(*rule-tac halt-lemma2*)
    show *wf wadjust-le* **by** *auto*
  next
    show $\forall$ *n*. $\neg$ *?P* (*?f n*) $\wedge$ *?Q* (*?f n*) $\longrightarrow$
      ?Q (*?f* (*Suc n*)) $\wedge$ (*?f* (*Suc n*), *?f n*) $\in$ *wadjust-le*
    proof(*rule-tac allI*, *rule-tac impI*, *case-tac ?f n*,
      simp add: *tstep-red tstep.simps*, *rule-tac conjI*, *erule-tac conjE*,
      erule-tac conjE)
      fix *n a b c d*
      assume *0* < *b wadjust-inv b m rs* (*c*, *d*) *Suc* (*Suc rs*) = *a*
      thus *case case fetch t-wcode-adjust b* (*case d of* [] $\Rightarrow$ *Bk* | *x* # *xs* $\Rightarrow$ *x*)
        of (*ac*, *ns*) $\Rightarrow$ (*ns*, *new-tape ac* (*c*, *d*)) *of* (*st*, *x*) $\Rightarrow$ *wadjust-inv st m rs x*
        apply(*case-tac d*, *simp*, *case-tac* [*2*] *aa*)
        apply(*simp-all add*: *wadjust-inv.simps wadjust-le-def new-tape.simps*
          abacus.lex-triple-def abacus.lex-pair-def lex-square-def
          split: *if-splits*)
        done
    next
      fix *n a b c d*
      assume *0* < *b* $\wedge$ *wadjust-inv b m rs* (*c*, *d*)
        Suc (*Suc rs*) = *a* $\wedge$ *steps* (*Suc 0*, *Bk* # $Oc^{Suc\ m}$,
        Bk # *Oc* # Bk^{ln} @ *Bk* # $Oc^{Suc\ rs}$ @ Bk^{rn}) *t-wcode-adjust n* = (*b*, *c*, *d*)
      thus ((*a*, *case fetch t-wcode-adjust b* (*case d of* [] $\Rightarrow$ *Bk* | *x* # *xs* $\Rightarrow$ *x*)
        of (*ac*, *ns*) $\Rightarrow$ (*ns*, *new-tape ac* (*c*, *d*))), *a*, *b*, *c*, *d*) $\in$ *wadjust-le*
      proof(*erule-tac conjE*, *erule-tac conjE*, *erule-tac conjE*)
        assume *0* < *b wadjust-inv b m rs* (*c*, *d*) *Suc* (*Suc rs*) = *a*
        thus *?thesis*
          apply(*case-tac d*, *case-tac* [*2*] *aa*)
          apply(*simp-all add*: *wadjust-inv.simps wadjust-le-def new-tape.simps*
            abacus.lex-triple-def abacus.lex-pair-def lex-square-def

```
            split: if-splits)
          done
        qed
      qed
    next
      show ?Q (?f 0)
        apply(simp add: steps.simps wadjust-inv.simps wadjust-start.simps)
        apply(rule-tac x = ln in exI,auto)
        done
    next
      show ¬ ?P (?f 0)
        apply(simp add: steps.simps)
        done
    qed
    thus ?thesis
      apply(auto)
      done
qed

lemma [intro]: t-correct t-wcode-adjust
apply(auto simp: t-wcode-adjust-def t-correct.simps iseven-def)
apply(rule-tac x = 11 in exI, simp)
done

lemma wcode-lemma-pre':
  args ≠ [] ⟹
  ∃ stp rn. steps (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 ≠ []
  have ?P1 ⊢-> λ tp. (∃ stp tp'. steps (Suc 0, tp)
                 ((t-wcode-prepare |+| t-wcode-main) |+| t-wcode-adjust) stp =
(0, tp') ∧ ?Q2 tp')
  proof(rule-tac turing-merge.t-merge-halt[of t-wcode-prepare |+| t-wcode-main
            t-wcode-adjust ?P1 ?P2 ?P3 ?P3 ?Q1 ?Q2],
      auto simp: turing-merge-def)

    show ∃ stp. case steps (Suc 0, [], <m # args>) (t-wcode-prepare |+| t-wcode-main)
stp of
        (st, tp') ⇒ st = 0 ∧ (case tp' of (l, r) ⇒ l = Bk # Oc^(Suc m) ∧
              (∃ ln rn. r = Bk # Oc # Bk^ln @ Bk # Bk # Oc^(bl-bin (<args>)) @
Bk^rn))
```

using *h prepare-mainpart-lemma*[*of args m*]
apply(*auto*)
apply(*rule-tac x* = *stp* **in** *exI*, *simp*)
apply(*rule-tac x* = *ln* **in** *exI*, *auto*)
done
next
fix *ln rn*
show ∃ *stp*. *case steps* (*Suc 0*, *Bk* # $Oc^{Suc\ m}$, *Bk* # *Oc* # Bk^{ln} @ *Bk* # *Bk* #
$Oc^{bl\text{-}bin\ (<args>)}$ @ Bk^{rn}) *t-wcode-adjust stp of*
(*st*, *tp*′) ⇒ *st* = *0* ∧ *wadjust-stop m* (*bl-bin* (<*args*>) − *Suc 0*) *tp*′
using *wadjust-correctness*[*of m bl-bin* (<*args*>) − *1 Suc ln rn*]
apply(*subgoal-tac bl-bin* (<*args*>) > *0*, *auto simp*: *wadjust-inv.simps*)
apply(*rule-tac x* = *n* **in** *exI*, *simp add*: *exp-ind*)
using *h*
apply(*case-tac args*, *simp-all*, *case-tac list*,
simp-all add: *tape-of-nl-abv tape-of-nat-list.simps exp-ind-def*
bl-bin.simps)
done
next
show *?Q1* ⊢−> *?P2*
by(*simp add*: *t-imply-def*)
qed
thus ∃ *stp rn*. *steps* (*Suc 0*, [], <*m* # *args*>) ((*t-wcode-prepare* |+| *t-wcode-main*) |+|
t-wcode-adjust) *stp* = (*0*, [*Bk*], $Oc^{Suc\ m}$ @ *Bk* # $Oc^{Suc\ (bl\text{-}bin\ (<args>))}$ @ Bk^{rn})
apply(*simp add*: *t-imply-def*)
apply(*erule-tac exE*)+
apply(*subgoal-tac bl-bin* (<*args*>) > *0*, *auto simp*: *wadjust-stop.simps*)
using *h*
apply(*case-tac args*, *simp-all*, *case-tac list*,
simp-all add: *tape-of-nl-abv tape-of-nat-list.simps exp-ind-def*
bl-bin.simps)
done
qed

The initialization TM *t-wcode*.

definition *t-wcode* :: *tprog*
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*. *steps* (*Suc 0*, [], <*m* # *args*>) (*t-wcode*) *stp* =
(*0*, [*Bk*], $Oc^{Suc\ m}$ @ *Bk* # $Oc^{Suc\ (bl\text{-}bin\ (<args>))}$ @ Bk^{rn})
apply(*simp add*: *wcode-lemma-pre*′ *t-wcode-def*)
done

lemma *wcode-lemma*:
  args ≠ [] ⟹
  ∃ *stp ln rn*. *steps* (*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-nl-abv tape-of-nat-list.simps*)
done

13 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* :: *tprog*
  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* @ *tMp* (*Suc* (*Suc 0*)) (*start-of* (*layout-of abc-F*)

                    (*length abc-F*) − *Suc 0*))))

definition *F-aprog* :: *abc-prog*
  where
  F-aprog ≡ (*let* (*aprog*, *rs-pos*, *a-md*) = *rec-ci rec-F in*
          aprog [+] *dummy-abc* (*Suc* (*Suc 0*)))

definition *F-tprog* :: *tprog*
  where
  F-tprog = *tm-of* (*F-aprog*)

definition *t-utm* :: *tprog*
  where
  t-utm ≡
   (*F-tprog*) @ *tMp* (*Suc* (*Suc 0*)) (*start-of* (*layout-of* (*F-aprog*))
          (*length* (*F-aprog*)) − *Suc 0*)

definition *UTM-pre* :: *tprog*
  where
  UTM-pre = *t-wcode* |+| *t-utm*

lemma *F-abc-halt-eq*:
  ⟦*turing-basic.t-correct tp*;
   length lm = *k*;
   steps (*Suc 0*, Bk^{l}, <*lm*>) *tp stp* = (*0*, Bk^{m}, Oc^{rs}@Bk^{n});
   rs > *0*⟧
   ⟹ ∃ *stp m*. *abc-steps-l* (*0*, [*code tp*, *bl2wc* (<*lm*>)]) (*F-aprog*) *stp* =
          (*length* (*F-aprog*), *code tp* # *bl2wc* (<*lm*>) # (*rs* − *1*) # 0^{m})

apply(*drule-tac F-t-halt-eq*, *simp*, *simp*, *simp*)
apply(*case-tac rec-ci rec-F*)
apply(*frule-tac abc-append-dummy-complie*, *simp*, *simp*, *erule-tac exE*, *erule-tac exE*)
apply(*rule-tac x = stp* **in** *exI*, *rule-tac x = m* **in** *exI*)
apply(*simp add*: *F-aprog-def dummy-abc-def*)
done

lemma *F-abc-utm-halt-eq*:
⟦*rs* > *0*;
abc-steps-l (*0*, [*code tp*, *bl2wc* (<*lm*>)]) *F-aprog stp* =
(*length F-aprog*, *code tp* # *bl2wc* (<*lm*>) # (*rs* − *1*) # 0^{m})⟧
⟹ ∃ *stp m n*.(*steps* (*Suc 0*, [*Bk*, *Bk*], <[*code tp*, *bl2wc* (<*lm*>)]>) *t-utm stp* =
(*0*, Bk^{m}, Oc^{rs} @ Bk^{n}))
thm *abacus-turing-eq-halt*
using *abacus-turing-eq-halt*
[*of layout-of F-aprog F-aprog F-tprog length* (*F-aprog*)
[*code tp*, *bl2wc* (<*lm*>)] *stp code tp* # *bl2wc* (<*lm*>) # (*rs* − *1*) # 0^{m} *Suc* (*Suc 0*)
start-of (*layout-of* (*F-aprog*)) (*length* (*F-aprog*)) [] *0*]
apply(*simp add*: *F-tprog-def t-utm-def abc-lm-v.simps nth-append*)
apply(*erule-tac exE*)+
apply(*rule-tac x = stpa* **in** *exI*, *rule-tac x = Suc* (*Suc ma*) **in** *exI*,
rule-tac x = l **in** *exI*, *simp add*: *exp-ind*)
done

declare *tape-of-nl-abv-cons*[*simp del*]

lemma *t-utm-halt-eq′*:
⟦*turing-basic.t-correct tp*;
0 < *rs*;
steps (*Suc 0*, Bk^{l}, <*lm*::*nat list*>) *tp stp* = (*0*, Bk^{m}, Oc^{rs}@Bk^{n})⟧
⟹ ∃ *stp m n*. *steps* (*Suc 0*, [*Bk*, *Bk*], <[*code tp*, *bl2wc* (<*lm*>)]>) *t-utm stp* =

(*0*, Bk^{m}, Oc^{rs} @ Bk^{n})
apply(*drule-tac l = l* **in** *F-abc-halt-eq*, *simp*, *simp*, *simp*)
apply(*erule-tac exE*, *erule-tac exE*)
apply(*rule-tac F-abc-utm-halt-eq*, *simp-all*)
done

lemma [*simp*]: *tinres xs* (*xs* @ Bk^{i})
apply(*auto simp*: *tinres-def*)
done

lemma [*elim*]: ⟦*rs* > *0*; Oc^{rs} @ Bk^{na} = *c* @ Bk^{n}⟧
⟹ ∃ *n*. *c* = Oc^{rs} @ Bk^{n}
apply(*case-tac na* > *n*)
apply(*subgoal-tac* ∃ *d*. *na* = *d* + *n*, *auto simp*: *exp-add*)
apply(*rule-tac x = na* − *n* **in** *exI*, *simp*)

apply(*subgoal-tac* ∃ *d*. *n* = *d* + *na*, *auto simp*: *exp-add*)
apply(*case-tac rs*, *simp-all add*: *exp-ind*, *case-tac d*,
simp-all add: *exp-ind*)
apply(*rule-tac x* = *n* − *na* **in** *exI*, *simp*)
done

lemma *t-utm-halt-eq*′′:
⟦*turing-basic.t-correct tp*;
0 < *rs*;
steps (*Suc 0*, Bk^{l}, <*lm*::*nat list*>) *tp stp* = (*0*, Bk^{m}, Oc^{rs}@Bk^{n})⟧
⟹ ∃ *stp m n*. *steps* (*Suc 0*, [*Bk*, *Bk*], <[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{i}) *t-utm stp* =
(*0*, Bk^{m}, Oc^{rs} @ Bk^{n})
apply(*drule-tac t-utm-halt-eq*′, *simp-all*)
apply(*erule-tac exE*)+
proof −
fix *stpa ma na*
assume *steps* (*Suc 0*, [*Bk*, *Bk*], <[*code tp*, *bl2wc* (<*lm*>)]>) *t-utm stpa* = (*0*, Bk^{ma}, Oc^{rs} @ Bk^{na})
and *gr*: *rs* > *0*
thus ∃ *stp m n*. *steps* (*Suc 0*, [*Bk*, *Bk*], <[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{i}) *t-utm stp* = (*0*, Bk^{m}, Oc^{rs} @ Bk^{n})
apply(*rule-tac x* = *stpa* **in** *exI*, *rule-tac x* = *ma* **in** *exI*, *simp*)
proof(*case-tac steps* (*Suc 0*, [*Bk*, *Bk*], <[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{i}) *t-utm stpa*, *simp*)
fix *a b c*
assume *steps* (*Suc 0*, [*Bk*, *Bk*], <[*code tp*, *bl2wc* (<*lm*>)]>) *t-utm stpa* = (*0*, Bk^{ma}, Oc^{rs} @ Bk^{na})
steps (*Suc 0*, [*Bk*, *Bk*], <[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{i}) *t-utm stpa* = (*a*, *b*, *c*)
thus *a* = *0* ∧ *b* = Bk^{ma} ∧ (∃ *n*. *c* = Oc^{rs} @ Bk^{n})
using *tinres-steps2*[*of* <[*code tp*, *bl2wc* (<*lm*>)]> <[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{i}
Suc 0 [*Bk*, *Bk*] *t-utm stpa 0* Bk^{ma} Oc^{rs} @ Bk^{na} *a b c*]
apply(*simp*)
using *gr*
apply(*simp only*: *tinres-def*, *auto*)
apply(*rule-tac x* = *na* + *n* **in** *exI*, *simp add*: *exp-add*)
done
qed
qed

lemma [*simp*]: *tinres* [*Bk*, *Bk*] [*Bk*]
apply(*auto simp*: *tinres-def*)
done

lemma [*elim*]: Bk^{ma} = *b* @ Bk^{n} ⟹ ∃ *m*. *b* = Bk^{m}
apply(*subgoal-tac ma* = *length b* + *n*)

apply(*rule-tac x = ma − n* **in** *exI*, *simp add*: *exp-add*)
apply(*drule-tac length-equal*)
apply(*simp*)
done

lemma *t-utm-halt-eq*:
⟦*turing-basic.t-correct tp*;
0 < rs;
steps (*Suc 0*, Bk^{l}, <*lm*::*nat list*>) *tp stp* = (*0*, Bk^{m}, Oc^{rs}@Bk^{n})⟧
⟹ ∃ *stp m n*. *steps* (*Suc 0*, [*Bk*], <[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{i}) *t-utm stp* =
(*0*, Bk^{m}, Oc^{rs} @ Bk^{n})
apply(*drule-tac i* = *i* **in** *t-utm-halt-eq''*, *simp-all*)
apply(*erule-tac exE*)+
proof −
fix *stpa ma na*
assume *steps* (*Suc 0*, [*Bk*, *Bk*], <[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{i}) *t-utm stpa* = (*0*, Bk^{ma}, Oc^{rs} @ Bk^{na})
and *gr*: *rs* > *0*
thus ∃ *stp m n*. *steps* (*Suc 0*, [*Bk*], <[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{i}) *t-utm stp* = (*0*, Bk^{m}, Oc^{rs} @ Bk^{n})
apply(*rule-tac x = stpa* **in** *exI*)
proof(*case-tac steps* (*Suc 0*, [*Bk*], <[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{i}) *t-utm stpa*, *simp*)
fix *a b c*
assume *steps* (*Suc 0*, [*Bk*, *Bk*], <[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{i}) *t-utm stpa* = (*0*, Bk^{ma}, Oc^{rs} @ Bk^{na})
steps (*Suc 0*, [*Bk*], <[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{i}) *t-utm stpa* = (*a*, *b*, *c*)
thus *a* = *0* ∧ (∃ *m*. *b* = Bk^{m}) ∧ (∃ *n*. *c* = Oc^{rs} @ Bk^{n})
using *tinres-steps*[*of* [*Bk*, *Bk*] [*Bk*] *Suc 0* <[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{i} *t-utm stpa 0*
Bk^{ma} Oc^{rs} @ Bk^{na} *a b c*]
apply(*simp*)
apply(*auto simp*: *tinres-def*)
apply(*rule-tac x = ma + n* **in** *exI*, *simp add*: *exp-add*)
done
qed
qed

lemma [*intro*]: *t-correct t-wcode*
apply(*simp add*: *t-wcode-def*)
apply(*auto*)
done

lemma [*intro*]: *t-correct t-utm*
apply(*simp add*: *t-utm-def F-tprog-def*)
apply(*rule-tac t-compiled-correct*, *auto*)
done

```
lemma UTM-halt-lemma-pre:
  ⟦turing-basic.t-correct tp;
   0 < rs;
   args ≠ [];
   steps (Suc 0, Bk^i, <args::nat list>) tp stp = (0, Bk^m, Oc^rs@Bk^k)⟧
  ⟹ ∃stp m n. steps (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)
  term ?Q2
  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
  assume h: turing-basic.t-correct tp 0 < rs
        args ≠ [] steps (Suc 0, Bk^i, <args::nat list>) tp stp = (0, Bk^m, Oc^rs@Bk^k)
  have ?P1 ⊢−> λ tp. (∃ stp tp'. steps (Suc 0, tp)
                (t-wcode |+| t-utm) stp = (0, tp') ∧ ?Q2 tp')
  proof(rule-tac turing-merge.t-merge-halt [of t-wcode t-utm
        ?P1 ?P2 ?P3 ?P3 ?Q1 ?Q2], auto simp: turing-merge-def)
    show ∃stp. case steps (Suc 0, [], <code tp # args>) t-wcode stp of (st, tp') ⇒

      st = 0 ∧ (case tp' of (l, r) ⇒ l = [Bk] ∧
                        (∃rn. r = Oc^(Suc (code tp)) @ Bk # Oc^(Suc (bl-bin (<args>))) @
Bk^rn))
      using wcode-lemma-1[of args code tp] h
      apply(simp, auto)
      apply(rule-tac x = stpa in exI, auto)
      done
  next
    fix rn
    show ∃stp. case steps (Suc 0, [Bk], Oc^(Suc (code tp)) @
      Bk # Oc^(Suc (bl-bin (<args>))) @ Bk^rn) t-utm stp of
      (st, tp') ⇒ st = 0 ∧ (case tp' of (l, r) ⇒
      (∃ln. l = Bk^ln) ∧ (∃rn. r = Oc^rs @ Bk^rn))
      using t-utm-halt-eq[of tp rs i args stp m k rn] h
      apply(auto)
      apply(rule-tac x = stpa in exI, simp add: bin-wc-eq
        tape-of-nat-list.simps tape-of-nl-abv)
      apply(auto)
      done
  next
    show ?Q1 ⊢−> ?P2
      apply(simp add: t-imply-def)
      done
  qed
  thus ?thesis
```

apply(*simp add*: *t-imply-def*)
apply(*auto simp*: *UTM-pre-def*)
done
qed

The correctness of *UTM*, the halt case.

lemma *UTM-halt-lemma*:
⟦*turing-basic.t-correct tp*;
$0 < rs$;
$args \neq []$;
steps (*Suc 0*, Bk^i, <*args*::*nat list*>) *tp stp* = (*0*, Bk^m, Oc^{rs}@Bk^k)⟧
⟹ ∃ *stp m n*. *steps* (*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*]
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*:: *t-conf* ⇒ *bool*
where
TSTD c = (*let* (*st*, *l*, *r*) = *c in*
st = *0* ∧ (∃ *m*. *l* = Bk^m) ∧ (∃ *rs n*. *r* = $Oc^{Suc\ rs}$ @ Bk^n))

thm *abacus-turing-eq-uhalt*

lemma *nstd-case1*: *0* < *a* ⟹ *NSTD* (*trpl-code* (*a*, *b*, *c*))
apply(*simp add*: *NSTD.simps trpl-code.simps*)
done

lemma [*simp*]: ∀ *m*. *b* ≠ Bk^m ⟹ *0* < *bl2wc b*
apply(*rule classical*, *simp*)
apply(*induct b*, *erule-tac x* = *0* **in** *allE*, *simp*)
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(*erule-tac x* = *Suc m* **in** *allE*, *simp add*: *exp-ind-def*, *simp*)
done
lemma *nstd-case2*: ∀ *m*. *b* ≠ Bk^m ⟹ *NSTD* (*trpl-code* (*a*, *b*, *c*))
apply(*simp add*: *NSTD.simps trpl-code.simps*)
done

thm *lg.simps*
thm *lgR.simps*

lemma [*elim*]: *Suc* (*2* ∗ *x*) = *2* ∗ *y* ⟹ *RR*
apply(*induct x arbitrary*: *y*, *simp*, *simp*)
apply(*case-tac y*, *simp*, *simp*)
done

```
lemma bl2nat-zero-eq[simp]: (bl2nat c 0 = 0) = (∃ n. c = Bk^n)
apply(auto)
apply(induct c, simp add: bl2nat.simps)
apply(rule-tac x = 0 in exI, simp)
apply(case-tac a, auto simp: bl2nat.simps bl2nat-double)
done

lemma bl2wc-exp-ex:
  ⟦Suc (bl2wc c) = 2 ^ m⟧ ⟹ ∃ rs n. c = Oc^rs @ Bk^n
apply(induct c arbitrary: m, simp add: bl2wc.simps bl2nat.simps)
apply(case-tac a, auto)
apply(case-tac m, simp-all add: bl2wc.simps, auto)
apply(rule-tac x = 0 in exI, rule-tac x = Suc n in exI,
  simp add: exp-ind-def)
apply(simp add: bl2wc.simps bl2nat.simps bl2nat-double)
apply(case-tac m, simp, simp)
proof −
  fix c m nat
  assume ind:
    ⋀m. Suc (bl2nat c 0) = 2 ^ m ⟹ ∃ rs n. c = Oc^rs @ Bk^n
  and h:
    Suc (Suc (2 * bl2nat c 0)) = 2 * 2 ^ nat
  have ∃ rs n. c = Oc^rs @ Bk^n
    apply(rule-tac m = nat in ind)
    using h
    apply(simp)
    done
  from this obtain rs n where  c = Oc^rs @ Bk^n by blast
  thus ∃ rs n. Oc # c = Oc^rs @ Bk^n
    apply(rule-tac x = Suc rs in exI, simp add: exp-ind-def)
    apply(rule-tac x = n in exI, simp)
    done
qed

lemma [elim]:
  ⟦∀ rs n. c ≠ Oc^(Suc rs) @ Bk^n;
  bl2wc c = 2 ^ lg (Suc (bl2wc c)) 2 − Suc 0⟧ ⟹ bl2wc c = 0
apply(subgoal-tac ∃ m. Suc (bl2wc c) = 2^m, erule-tac exE)
apply(drule-tac bl2wc-exp-ex, simp, erule-tac exE, erule-tac exE)
apply(case-tac rs, simp, simp, erule-tac x = nat in allE,
  erule-tac x = n in allE, simp)
using bl2wc-exp-ex[of c lg (Suc (bl2wc c)) 2]
apply(case-tac (2::nat) ^ lg (Suc (bl2wc c)) 2,
  simp, simp, erule-tac exE, erule-tac exE, simp)
apply(simp add: bl2wc.simps)
apply(rule-tac x = rs in exI)
apply(case-tac (2::nat)^rs, simp, simp)
done
```

lemma *nstd-case3*:
∀ *rs n*. $c \neq Oc^{Suc\ rs}$ @ Bk^{n} ⟹ *NSTD* (*trpl-code* (*a*, *b*, *c*))
apply(*simp add*: *NSTD.simps trpl-code.simps*)
apply(*rule-tac impI*)
apply(*rule-tac disjI2*, *rule-tac disjI2*, *auto*)
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*:
⟦*turing-basic.t-correct tp*;
steps (*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*:
⟦*turing-basic.t-correct tp*;
∀ *stp*. (¬ *TSTD* (*steps* (*Suc 0*, Bk^{l}, <*lm*>) *tp stp*))⟧
⟹ ∀ *y*. *rec-calc-rel rec-nonstop*
([*code tp*, *bl2wc* (<*lm*>), *y*]) (*Suc 0*)
apply(*rule-tac allI*, *erule-tac x* = *y* **in** *allE*)
apply(*case-tac steps* (*Suc 0*, Bk^{l}, <*lm*>) *tp y*, *simp*)
apply(*rule-tac nonstop-t-uhalt-eq*, *simp-all*)
done

declare *ci-cn-para-eq*[*simp*]

lemma *F-aprog-uhalt*:
⟦*turing-basic.t-correct tp*;
∀ *stp*. (¬ *TSTD* (*steps* (*Suc 0*, Bk^{l}, <*lm*>) *tp stp*));
rec-ci rec-F = (*F-ap*, *rs-pos*, *a-md*)⟧
⟹ ∀ *stp*. *case abc-steps-l* (*0*, [*code tp*, *bl2wc* (<*lm*>)] @ $0^{a\text{-}md\ -\ rs\text{-}pos}$

@ *suflm*) (*F-ap*) *stp* *of* (*ss*, *e*) ⇒ *ss* < *length* (*F-ap*)
apply(*case-tac rec-ci* (*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*])]))
apply(*simp only*: *rec-F-def*, *rule-tac i* = *0* **and** *ga* = *a* **and** *gb* = *b* **and**
gc = *c* **in** *cn-gi-uhalt*, *simp*, *simp*, *simp*, *simp*, *simp*, *simp*, *simp*)
apply(*simp add*: *ci-cn-para-eq*)
apply(*case-tac rec-ci* (*Cn* (*Suc* (*Suc* *0*)) *rec-conf*
([*id* (*Suc* (*Suc* *0*)) *0*, *id* (*Suc* (*Suc* *0*)) (*Suc* *0*), *rec-halt*])))
apply(*rule-tac rf* = (*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*])])
and *n* = *Suc* (*Suc* *0*) **and** *f* = *rec-right* **and**
gs = [*Cn* (*Suc* (*Suc* *0*)) *rec-conf*
([*id* (*Suc* (*Suc* *0*)) *0*, *id* (*Suc* (*Suc* *0*)) (*Suc* *0*), *rec-halt*])]
and *i* = *0* **and** *ga* = *aa* **and** *gb* = *ba* **and** *gc* = *ca* **in**
cn-gi-uhalt)
apply(*simp*, *simp*, *simp*, *simp*, *simp*, *simp*, *simp*,
simp add: *ci-cn-para-eq*)
apply(*case-tac rec-ci rec-halt*)
apply(*rule-tac rf* = (*Cn* (*Suc* (*Suc* *0*)) *rec-conf*
([*id* (*Suc* (*Suc* *0*)) *0*, *id* (*Suc* (*Suc* *0*)) (*Suc* *0*), *rec-halt*]))
and *n* = *Suc* (*Suc* *0*) **and** *f* = *rec-conf* **and**
gs = ([*id* (*Suc* (*Suc* *0*)) *0*, *id* (*Suc* (*Suc* *0*)) (*Suc* *0*), *rec-halt*]) **and**
i = *Suc* (*Suc* *0*) **and** *gi* = *rec-halt* **and** *ga* = *ab* **and** *gb* = *bb* **and**
gc = *cb* **in** *cn-gi-uhalt*)
apply(*simp*, *simp*, *simp*, *simp*, *simp add*: *nth-append*, *simp*,
simp add: *nth-append*, *simp add*: *rec-halt-def*)
apply(*simp only*: *rec-halt-def*)
apply(*case-tac* [!] *rec-ci* ((*rec-nonstop*)))
apply(*rule-tac allI*, *rule-tac impI*, *simp*)
apply(*case-tac j*, *simp*)
apply(*rule-tac x* = *code tp* **in** *exI*, *rule-tac calc-id*, *simp*, *simp*, *simp*, *simp*)
apply(*rule-tac x* = *bl2wc* (<*lm*>) **in** *exI*, *rule-tac calc-id*, *simp*, *simp*, *simp*)
apply(*rule-tac rf* = *Mn* (*Suc* (*Suc* *0*)) (*rec-nonstop*)
and *f* = (*rec-nonstop*) **and** *n* = *Suc* (*Suc* *0*)
and *aprog′* = *ac* **and** *rs-pos′* = *bc* **and** *a-md′* = *cc* **in** *Mn-unhalt*)
apply(*simp*, *simp add*: *rec-halt-def* , *simp*, *simp*)
apply(*drule-tac nonstop-true*, *simp-all*)
apply(*rule-tac allI*)
apply(*erule-tac x* = *y* **in** *allE*)+
apply(*simp*)
done

thm *abc-list-crsp-steps*

lemma *uabc-uhalt′*:
⟦*turing-basic.t-correct tp*;
∀ *stp*. (¬ *TSTD* (*steps* (*Suc* *0*, Bk^l, <*lm*>) *tp stp*));
rec-ci rec-F = (*ap*, *pos*, *md*)⟧
⟹ ∀ *stp*. *case abc-steps-l* (*0*, [*code tp*, *bl2wc* (<*lm*>)]) *ap stp of* (*ss*, *e*)

```
      ⇒ ss < length ap
proof(frule-tac F-ap = ap and rs-pos = pos and a-md = md
    and suflm = [] in F-aprog-uhalt, auto)
  fix stp a b
  assume h:
    ∀ stp. case abc-steps-l (0, code tp # bl2wc (<lm>) # 0^(md - pos)) ap stp of
    (ss, e) ⇒ ss < length ap
    abc-steps-l (0, [code tp, bl2wc (<lm>)]) ap stp = (a, b)
    turing-basic.t-correct tp
    rec-ci rec-F = (ap, pos, md)
  moreover have ap ≠ []
    using h apply(rule-tac rec-ci-not-null, simp)
    done
  ultimately show a < length ap
  proof(erule-tac x = stp in allE,
  case-tac abc-steps-l (0, code tp # bl2wc (<lm>) # 0^(md - pos)) ap stp, simp)
    fix aa ba
    assume g: aa < length ap
      abc-steps-l (0, code tp # bl2wc (<lm>) # 0^(md - pos)) ap stp = (aa, ba)
      ap ≠ []
    thus ?thesis
      using abc-list-crsp-steps[of [code tp, bl2wc (<lm>)]
                              md - pos ap stp aa ba] h
      apply(simp)
      done
  qed
qed

lemma uabc-uhalt:
  ⟦turing-basic.t-correct tp;
  ∀ stp. (¬ TSTD (steps (Suc 0, Bk^l, <lm>) tp stp))⟧
  ⟹ ∀ stp. case abc-steps-l (0, [code tp, bl2wc (<lm>)]) F-aprog
       stp of (ss, e) ⇒ ss < length F-aprog
apply(case-tac rec-ci rec-F, simp add: F-aprog-def)
thm uabc-uhalt'
apply(drule-tac ap = a and pos = b and md = c in uabc-uhalt', simp-all)
proof -
  fix a b c
  assume
    ∀ stp. case abc-steps-l (0, [code tp, bl2wc (<lm>)]) a stp of (ss, e)
                                         ⇒ ss < length a
    rec-ci rec-F = (a, b, c)
  thus
    ∀ stp. case abc-steps-l (0, [code tp, bl2wc (<lm>)])
    (a [+] dummy-abc (Suc (Suc 0))) stp of (ss, e) ⇒
          ss < Suc (Suc (Suc (length a)))
    using abc-append-uhalt1[of a [code tp, bl2wc (<lm>)]
      a [+] dummy-abc (Suc (Suc 0)) [] dummy-abc (Suc (Suc 0))]
    apply(simp)
```

    done
qed

lemma *tutm-uhalt'*:
  ⟦*turing-basic.t-correct tp*;
    ∀ *stp*. (¬ *TSTD* (*steps* (*Suc 0*, Bk^l, <*lm*>) *tp stp*))⟧
  ⟹ ∀ *stp*. ¬ *isS0* (*steps* (*Suc 0*, [*Bk*, *Bk*], <[*code tp*, *bl2wc* (<*lm*>)]>) *t-utm stp*)
  using *abacus-turing-eq-uhalt*[*of layout-of* (*F-aprog*)
      F-aprog F-tprog [*code tp*, *bl2wc* (<*lm*>)]
      start-of (*layout-of* (*F-aprog*)) (*length* (*F-aprog*))
      Suc (*Suc 0*)]
apply(*simp add*: *F-tprog-def*)
apply(*subgoal-tac* ∀ *stp*. *case abc-steps-l* (*0*, [*code tp*, *bl2wc* (<*lm*>)])
  (*F-aprog*) *stp of* (*as*, *am*) ⇒ *as* < *length* (*F-aprog*), *simp*)
thm *abacus-turing-eq-uhalt*
apply(*simp add*: *t-utm-def F-tprog-def*)
apply(*rule-tac uabc-uhalt*, *simp-all*)
done

lemma *tinres-commute*: *tinres r r'* ⟹ *tinres r' r*
apply(*auto simp*: *tinres-def*)
done

lemma *inres-tape*:
  ⟦*steps* (*st*, *l*, *r*) *tp stp* = (*a*, *b*, *c*); *steps* (*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 steps* (*st*, *l'*, *r*) *tp stp*)
  fix *aa ba ca*
  assume *h*: *steps* (*st*, *l*, *r*) *tp stp* = (*a*, *b*, *c*)
      steps (*st*, *l'*, *r'*) *tp stp* = (*a'*, *b'*, *c'*)
      tinres l l' tinres r r'
      steps (*st*, *l'*, *r*) *tp stp* = (*aa*, *ba*, *ca*)
  have *tinres b ba* ∧ *c* = *ca* ∧ *a* = *aa*
    using *h*
    apply(*rule-tac tinres-steps*, *auto*)
    done

  thm *tinres-steps2*
  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*: ∀ *stp*. ¬ *isS0* (*steps* (*Suc 0*, [*Bk*, *Bk*], <[*code tp*, *bl2wc* (<*lm*>)]>) *t-utm stp*)
⟹ ∀ *stp*. ¬ *isS0* (*steps* (*Suc 0*, Bk^{m}, <[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{n}) *t-utm stp*)
apply(*rule-tac allI*, *case-tac* (*steps* (*Suc 0*, Bk^{m},
<[*code tp*, *bl2wc* (<*lm*>)]> @ Bk^{n}) *t-utm stp*), *simp add*: *isS0-def*)
apply(*erule-tac x* = *stp* **in** *allE*)
apply(*case-tac steps* (*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 add*: *exp-ind-def*, *case-tac nat*, *simp-all add*: *exp-ind-def*)
apply(*rule-tac x* = *2* − *m* **in** *exI*, *simp add*: *exp-ind-def*[*THEN sym*] *exp-add*[*THEN sym*])
apply(*simp only*: *numeral-2-eq-2*, *simp add*: *exp-ind-def*)
done

lemma *tutm-uhalt*:
⟦*turing-basic.t-correct tp*;
∀ *stp*. (¬ *TSTD* (*steps* (*Suc 0*, Bk^{l}, <*args*>) *tp stp*))⟧
⟹ ∀ *stp*. ¬ *isS0* (*steps* (*Suc 0*, Bk^{m}, <[*code tp*, *bl2wc* (<*args*>)]> @ Bk^{n}) *t-utm stp*)
apply(*rule-tac tape-normalize*)
apply(*rule-tac tutm-uhalt′*, *simp-all*)
done

lemma *UTM-uhalt-lemma-pre*:
⟦*turing-basic.t-correct tp*;
∀ *stp*. (¬ *TSTD* (*steps* (*Suc 0*, Bk^{l}, <*args*>) *tp stp*));
args ≠ []⟧
⟹ ∀ *stp*. ¬ *isS0* (*steps* (*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* # $Oc^{Suc\ (bl\text{-}bin\ (<args>))}$ @ Bk^{rn}))
let *?P4* = *?Q1*
let *?P3* = λ (*l*, *r*). *False*
assume *h*: *turing-basic.t-correct tp* ∀ *stp*. ¬ *TSTD* (*steps* (*Suc 0*, Bk^{l}, <*args*>) *tp stp*)
args ≠ []
have *?P1* ⊢−> λ *tp*. ¬ (∃ *stp*. *isS0* (*steps* (*Suc 0*, *tp*) (*t-wcode* |+| *t-utm*) *stp*))
proof(*rule-tac turing-merge.t-merge-uhalt* [*of t-wcode t-utm*
?P1 ?P3 ?P3 ?P4 ?Q1 ?P3], *auto simp*: *turing-merge-def*)
show ∃ *stp*. *case steps* (*Suc 0*, [], <*code tp* # *args*>) *t-wcode stp of* (*st*, *tp′*) ⇒

st = *0* ∧ (*case tp′ of* (*l*, *r*) ⇒ *l* = [*Bk*] ∧
(∃ *rn*. *r* = $Oc^{Suc\ (code\ tp)}$ @ *Bk* # $Oc^{Suc\ (bl\text{-}bin\ (<args>))}$ @

```
Bk^rn))
      using wcode-lemma-1[of args code tp] h
      apply(simp, auto)
      apply(rule-tac x = stp in exI, auto)
      done
  next
    fix rn  stp
   show  isS0 (steps (Suc 0, [Bk], Oc^Suc (code tp) @ Bk # Oc^Suc (bl-bin (<args>))
@ Bk^rn) t-utm stp)
        ⟹ False
      using tutm-uhalt[of tp l args Suc 0 rn] h
      apply(simp)
      apply(erule-tac x = stp in allE)
      apply(simp add: tape-of-nl-abv tape-of-nat-list.simps bin-wc-eq)
      done
  next
    fix rn stp
   show isS0 (steps (Suc 0, [Bk], Oc^Suc (code tp) @ Bk # Oc^Suc (bl-bin (<args>))
@ Bk^rn) t-utm stp) ⟹
      isS0 (steps (Suc 0, [Bk], Oc^Suc (code tp) @ Bk # Oc^Suc (bl-bin (<args>)) @
Bk^rn) t-utm stp)
      by simp
  next
    show ?Q1 ⊢−> ?P4
      apply(simp add: t-imply-def)
      done
  qed
  thus ?thesis
    apply(simp add: t-imply-def UTM-pre-def)
    done
qed
```

The correctness of *UTM*, the unhalt case.

```
lemma UTM-uhalt-lemma:
  ⟦turing-basic.t-correct tp;
   ∀ stp. (¬ TSTD (steps (Suc 0, Bk^l, <args>) tp stp));
   args ≠ []⟧
  ⟹ ∀ stp. ¬ isS0 (steps (Suc 0, [], <code tp # args>)  UTM stp)
using UTM-uhalt-lemma-pre[of tp l args]
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

end
```