theory *EX-ListRev* = *SALMemFWInst*:

1 SAL Example: List Reversal

We analyse a program that reverts a list. We show this this program does not exceed its granted memory.

List are represented as chains of natural numbers finished by *NAT 0*, which acts as null pointer. Each number is the address of the next element. The address of the first element is expected the base location *B*. For example the list $[a,b,c]$ is represented as *m bse* = *NAT a*, *m a* = *NAT b*, *m b* = *NAT c*, *m c* = *NAT 0*.

constdefs
null::*nat* — stores NAT 0
null ≡ *0*
bs :: *nat* — base pointer
bs ≡ *1*
nxt:: *nat*
nxt ≡ *2* — stores pointer to the next list element
buf::*nat* — saves list elements temporarly
buf ≡ *3*
rt:: *nat* — stores return locations
rt ≡ *4*

Since MOV operates indireclty we also have to store the locations of our variables. We do this with so called location variables. The main procedure initialises them to point to the corresponding variable.

constdefs
lnull::*nat* — initialised with NAT null
lnull ≡ *5*
lbs::*nat* — initialised with NAT bs
lbs ≡ *6*
lnxt::*nat* — initialised with NAT nxt
lnxt ≡ *7*
lbuf::*nat* — initialised with NAT buf
lbuf ≡ *8*

Arguments

constdefs
e1::*nat*
e1 ≡ *10*
e2::*nat*
e2 ≡ *11*

1.1 Program with Annotations

consts *List*::(*loc* ⇒ *tval*) ⇒ *loc* ⇒ *loc list* ⇒ *bool*

primrec

```
List m l [] = (m l = NAT 0)
List m l (l'#ls) = (0<l ∧ (m l = NAT l') ∧ (List m l' ls))
```

constdefs

```
  prog :: SALprogram
prog ≡ [
(0, [ (SET lnull null,None),
      (SET lbs bs,None),
      (SET lnxt nxt,None),
      (SET lbuf buf,None),
      (SET null 0,None),
      (SET bs e1,None),
      (SET e1 e2,None),
      (SET e2 0,None),
      (CALL rt 1,Some (λ(p,m,e). m null = NAT 0 ∧ m bs = NAT e1 ∧
                     m lnull = NAT null ∧ m lbs = NAT bs ∧
                     m lnxt = NAT nxt ∧ m lbuf = NAT buf ∧
                     (List m bs [e1,e2]))),
      (HALT,Some TrueF)]),
(1,[
     (MOV bs lnxt, Some (λ(p,m,e).  m lnull = NAT null ∧ m lbs = NAT bs ∧
                       m lnxt = NAT nxt ∧ m lbuf = NAT buf ∧  m null = NAT 0
∧
                       m rt = RA (incA (←pc e)) ∧
                       (∃n. m bs = NAT n ∧ 10 ≤ n ∧ n < MAXMEM) ∧
                      (∃lst. List m bs lst ∧ List (←m e) bs lst ∧ ((∀l∈(set lst). 10 ≤
l ∧ l < MAXMEM))))),
     (MOV lnull bs,None),
     (JMPEQ nxt null 6,Some (λ(p,m,e).  m lnull = NAT null ∧ m lbs = NAT bs
∧
                     m lnxt = NAT nxt ∧ m lbuf = NAT buf ∧  m null = NAT 0 ∧
                       m rt = RA (incA (←pc e)) ∧
                       (∃n. m bs = NAT n ∧ 10 ≤ n ∧ n < MAXMEM) ∧
                       (∃l1. List m bs l1 ∧ (∃l2. List m nxt l2 ∧
                        (∃l3. List (←m e) bs l3 ∧ l3 = (rev l1) @ l2 ∧ (∀l∈(set l3).
10 ≤ l ∧ l < MAXMEM)))))),
     (MOV nxt lbuf,None),
     (MOV lbs nxt,None),
     (MOV lnxt lbs,None),
     (MOV lbuf lnxt ,None),
     (JMPB 5, None),
     (RET rt, Some (λ(p,m,e).  m lnull = NAT null ∧ m lbs = NAT bs ∧
                     m lnxt = NAT nxt ∧ m lbuf = NAT buf ∧  m null = NAT 0 ∧
                       m rt = RA (incA (←pc e)) ∧
                       (∃lst. List m bs lst ∧ List (←m e) bs (rev lst))))])]
```

constdefs

vc:: SALform

vc ≡ %s. (%s. (%(pc, m, e). pc = (0::nat, 0::nat) & env.cs e = [(0::nat, m)] & env.h e = [] & (ALL X. m X = ILLEGAL)) s --> (%s. (%s. (%(pc, m, e). (0::nat) <= MAX) s & (%s. (%(p, m, e). Suc (Suc (Suc (Suc (Suc (0::nat))))) < MAXMEM) s & (%s. True) s) s) s & (%s. (%s. (%(pc, m, e). pc = (0::nat, 0::nat)) s --> (%(pc, m, e). (%s. (%s. (%(pc, m, e). Suc (0::nat) <= MAX) s & (%s. (%(p, m, e). Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))) < MAXMEM) s & (%s. True) s) s) s & (%s. (%s. (%(pc, m, e). pc = (0::nat, Suc (0::nat))) s --> (%(pc, m, e). (%s. (%s. (%(pc, m, e). Suc (Suc (0::nat)) <= MAX) s & (%s. (%(p, m, e). Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))) < MAXMEM) s & (%s. True) s) s) s & (%s. (%s. (%(pc, m, e). pc = (0::nat, Suc (Suc (0::nat)))) s --> (%(pc, m, e). (%s. (%s. (%(pc, m, e). Suc (Suc (Suc (0::nat))) <= MAX) s & (%s. (%(p, m, e). Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) < MAXMEM) s & (%s. True) s) s) s & (%s. (%s. (%(pc, m, e). pc = (0::nat, Suc (Suc (Suc (0::nat))))) s --> (%(pc, m, e). (%s. (%s. (%(pc, m, e). (0::nat) <= MAX) s & (%s. (%(p, m, e). (0::nat) < MAXMEM) s & (%s. True) s) s) s & (%s. (%s. (%(pc, m, e). pc = (0::nat, Suc (Suc (Suc (Suc (0::nat)))))) s --> (%(pc, m, e). (%s. (%s. (%(pc, m, e). Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) <= MAX) s & (%s. (%(p, m, e). Suc (0::nat) < MAXMEM) s & (%s. True) s) s) s & (%s. (%s. (%(pc, m, e). pc = (0::nat, Suc (Suc (Suc (Suc (Suc (0::nat))))))) s -->

(%(pc, m, e). (%s. (%s. (%(pc, m, e). Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))))) <= MAX) s & (%s. (%(p, m, e). Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) < MAXMEM) s & (%s. True) s) s) s & (%s. (%s. (%(pc, m, e). pc = (0::nat, Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) s --> (%(pc, m, e). (%s. (%s. (%(pc, m, e). (0::nat) <= MAX) s & (%s. (%(p, m, e). Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) < MAXMEM) s & (%s. True) s) s) s & (%s. (%s. (%(pc, m, e). pc = (0::nat, Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) s --> (%(pc, m, e). (%s. (%s. (%s. True) s & (%s. (%(p, m, e). Suc (Suc (Suc (0::nat)))) < MAXMEM) s & (%s. True) s) s) s & (%s. (%(p, m, e). m null = NAT (0::nat) & m bs = NAT e1 & m lnull = NAT null & m lbs = NAT bs & m lnxt = NAT nxt & m lbuf = NAT buf & List m bs [e1, e2]) s & (%s. True) s) s) ((0::nat, Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))), update m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))))) (NAT (0::nat)), env.h-update (env.h e @ [(0::nat, Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))]) e)) s) s & (%s. True) s) s) ((0::nat, Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))),

update m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) (NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))))))), env.h-update (env.h e @ [(0::nat, Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))]) e)) s) s & (%s. True) s) s) ((0::nat, Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))), update m (Suc (0::nat)) (NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))))), env.h-update (env.h e @ [(0::nat, Suc (Suc (Suc (Suc (Suc (0::nat))))))]) e)) s) s & (%s. True) s) s) ((0::nat, Suc (Suc (Suc (Suc (Suc (0::nat)))))), update m (0::nat) (NAT (0::nat)), env.h-update (env.h e @ [(0::nat, Suc (Suc (Suc (Suc (0::nat)))))]) e)) s) s & (%s. True) s) s) ((0::nat, Suc (Suc (Suc (Suc (0::nat))))), update m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) (NAT (Suc (Suc (Suc (0::nat))))), env.h-update (env.h e

@ [(0::nat, Suc (Suc (Suc (0::nat))))]) e)) s) s & (%s. True) s) s) ((0::nat, Suc (Suc (Suc (0::nat)))), update m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) (NAT (Suc (Suc (0::nat)))), env.h-update (env.h e @ [(0::nat, Suc (Suc (0::nat)))]) e)) s) s & (%s. True) s) s) ((0::nat, Suc (Suc (0::nat))), update m (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))) (NAT (Suc (0::nat))),
env.h-update (env.h e @ [(0::nat, Suc (0::nat))]) e)) s) s & (%s. True) s) s) ((0::nat, Suc (0::nat)), update m (Suc (Suc (Suc (Suc (Suc (0::nat)))))) (NAT (0::nat)), env.h-update (env.h e @ [(0::nat, 0::nat)]) e)) s) s & (%s. True) s) s) s) s & (%s. (%s. (%s. (%s. (%s. (%s. (%s. True) s & (%s. (%(p, m, e). Suc (Suc (Suc (Suc (0::nat)))) < MAXMEM) s & (%s. True) s) s) s & (%s. (%(p, m, e). m null = NAT (0::nat) & m bs = NAT e1 & m lnull = NAT null & m lbs = NAT bs & m lnxt = NAT nxt & m lbuf = NAT buf & List m bs [e1, e2]) s & (%s. True) s) s) s & (%s. (%(pc, m, e). pc = (0::nat, Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) s & (%s. True) s) s) s --> (%(pc, m, e). (%s. (%s. (%(pc, m, e). (EX sa. m (Suc (0::nat)) = NAT sa) & (EX ta. m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) = NAT ta)) s & (%s. (%(p, m, e). (ALL ns. m (Suc (0::nat)) = NAT ns --> ns < MAXMEM) & (ALL nt. m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) = NAT nt --> nt < MAXMEM)) s & (%s. True) s) s) s & (%s. (%(p, m, e). m lnull = NAT null & m lbs = NAT bs & m lnxt = NAT nxt & m lbuf = NAT buf & m null = NAT (0::nat) & m rt = RA (incA (callpc e)) & (EX n. m bs = NAT n & (10::nat) <= n & n < MAXMEM) & (EX lst. List m bs lst & List (callmem e) bs lst & (ALL l:set lst. (10::nat) <= l & l < MAXMEM))) s & (%s. True) s) s) ((Suc (0::nat), 0::nat), update m (Suc (Suc (Suc (Suc (0::nat))))) (RA (0::nat, Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))),
env.cs-update ((length (env.h e), m) # env.cs e) (env.h-update (env.h e @ [(0::nat, Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))]) e))) s) s & (%s. True) s) s & (%s. (%s. True) s & (%s. (%s. (%s. (%s. (%s. (%s. (%(pc, m, e). (EX sa. m (Suc (0::nat)) = NAT sa) & (EX ta. m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) = NAT ta)) s & (%s. (%(p, m, e). (ALL ns. m (Suc (0::nat)) = NAT ns --> ns < MAXMEM) & (ALL nt. m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) = NAT nt --> nt < MAXMEM)) s & (%s. True) s) s) s & (%s. (%(p, m, e). m lnull = NAT null & m lbs = NAT bs & m lnxt = NAT nxt & m lbuf = NAT buf & m null = NAT (0::nat) & m rt = RA (incA (callpc e)) & (EX n. m bs = NAT n & (10::nat) <= n & n < MAXMEM) & (EX lst. List m bs lst & List (callmem e) bs lst & (ALL l:set lst. (10::nat) <= l & l < MAXMEM))) s & (%s. True) s) s) s & (%s. (%(pc, m, e). pc = (Suc (0::nat), 0::nat)) s & (%s. True) s) s) s --> (%(pc, m, e). tval-case False (%sa. tval-case False (%ta. (%s. (%s. (%(pc, m, e). (EX sa. m (Suc (Suc (Suc (Suc (Suc (0::nat)))))) = NAT sa) & (EX ta. m (Suc (0::nat)) = NAT ta)) s & (%s. (%(p, m, e). (ALL ns. m (Suc (Suc (Suc (Suc (Suc (0::nat)))))) = NAT ns --> ns < MAXMEM) & (ALL nt. m (Suc (0::nat)) = NAT nt --> nt < MAXMEM)) s & (%s. True) s) s) s & (%s. (%s. (%(pc, m, e). pc = (Suc (0::nat),
Suc (0::nat))) s --> (%(pc, m, e). tval-case False (%sa. tval-case False (%ta. (%s. (%s. (%(pc, m, e). (EX n. m (Suc (Suc (0::nat))) = NAT n) & (EX n. m (0::nat) = NAT n)) s & (%s. (%(p, m, e). Suc (Suc (0::nat)) < MAXMEM & (0::nat) < MAXMEM) s & (%s. True) s) s) s & (%s. (%(p, m, e). m lnull = NAT null & m lbs = NAT bs & m lnxt = NAT nxt & m lbuf = NAT buf & m null = NAT (0::nat) & m rt = RA (incA (callpc e)) & (EX n. m bs = NAT n

& (10::nat) <= n & n < MAXMEM) & (EX l1. List m bs l1 & (EX l2. List
m nxt l2 & (EX l3. List (callmem e) bs l3 & l3 = rev l1 @ l2 & (ALL l:set l3.
(10::nat) <= l & l < MAXMEM))))) s & (%s. True) s) s) ((Suc (0::nat), Suc
(Suc (0::nat))), update m ta (m sa), env.h-update (env.h e @ [(Suc (0::nat), Suc
(0::nat))]) e)) (%ra. False) (m (Suc (0::nat)))) (%ra. False) (m (Suc (Suc (Suc
(Suc (Suc (0::nat))))))) s) s & (%s. True) s) s) ((Suc (0::nat), Suc (0::nat)),
update m ta (m sa), env.h-update (env.h e @ [(Suc (0::nat), 0::nat)]) e)) (%ra.
False) (m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) (%ra. False) (m
(Suc (0::nat)))) s) s & (%s. True) s) s & (%s. (%s. (%s. (%s. (%s. (%s. (%(pc,
m, e). (EX n. m (Suc (Suc (0::nat))) = NAT n) & (EX n. m (0::nat) = NAT n))
s & (%s. (%(p, m, e). Suc (Suc (0::nat)) < MAXMEM & (0::nat) < MAXMEM)
s & (%s. True) s) s) s & (%s. (%(p, m, e). m lnull = NAT null & m lbs = NAT
bs & m lnxt = NAT nxt & m lbuf = NAT buf & m null = NAT (0::nat) &
m rt = RA (incA (callpc e)) & (EX n. m bs = NAT n & (10::nat) <= n & n
< MAXMEM) & (EX l1. List m bs l1 & (EX l2. List m nxt l2 & (EX l3. List
(callmem e) bs l3 & l3 = rev l1 @ l2 & (ALL l:set l3. (10::nat) <= l & l <
MAXMEM))))) s & (%s. True) s) s) s & (%s. (%(pc, m, e). (EX n n′. m (Suc
(Suc (0::nat))) = NAT n & m (0::nat) = NAT n′ & n = n′) & pc = (Suc (0::nat),
Suc (Suc (0::nat)))) s & (%s. True) s) s) s -->
(%(pc, m, e). (%s. (%s. (%(pc, m, e). EX pn′ i′. m (Suc (Suc (Suc (Suc (0::nat)))))
= RA (pn′, Suc i′) & (EX k m′ cl css. env.cs e = (k, m′) # cl # css & (pn′, i′) =
env.h e ! k)) s & (%s. (%(p, m, e). Suc (Suc (Suc (Suc (0::nat)))) < MAXMEM)
s & (%s. True) s) s) s & (%s. (%(p, m, e). m lnull = NAT null & m lbs = NAT
bs & m lnxt = NAT nxt & m lbuf = NAT buf & m null = NAT (0::nat) & m rt
= RA (incA (callpc e)) & (EX lst. List m bs lst & List (callmem e) bs (rev lst)))
s & (%s. True) s) s) ((Suc (0::nat), Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(0::nat)))))))))), m, env.h-update (env.h e @ [(Suc (0::nat), Suc (Suc (0::nat)))])
e)) s) s & (%s. (%s. (%s. (%s. (%s. (%(pc, m, e). (EX n. m (Suc (Suc (0::nat)))
= NAT n) & (EX n. m (0::nat) = NAT n)) s & (%s. (%(p, m, e). Suc (Suc
(0::nat)) < MAXMEM & (0::nat) < MAXMEM) s & (%s. True) s) s) s & (%s.
(%(p, m, e). m lnull = NAT null & m lbs = NAT bs & m lnxt = NAT nxt & m
lbuf = NAT buf & m null = NAT (0::nat) & m rt = RA (incA (callpc e)) & (EX
n. m bs = NAT n & (10::nat) <= n & n < MAXMEM) & (EX l1. List m bs l1
& (EX l2. List m nxt l2 & (EX l3. List (callmem e) bs l3 & l3 = rev l1 @ l2 &
(ALL l:set l3. (10::nat) <= l & l < MAXMEM))))) s & (%s. True) s) s) s &
(%s. (%(pc, m, e). (EX n n′. m (Suc (Suc (0::nat))) = NAT n & m (0::nat) =
NAT n′ & n ~= n′) & pc = (Suc (0::nat), Suc (Suc (0::nat)))) s & (%s. True) s)
s) s --> (%(pc, m, e). (%s. (%s. (%(pc, m, e). (EX sa. m (Suc (Suc (0::nat)))
= NAT sa) & (EX ta. m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))
= NAT ta)) s & (%s. (%(p, m, e). (ALL ns. m (Suc (Suc (0::nat))) = NAT ns
--> ns < MAXMEM) & (ALL nt. m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(0::nat))))))))) = NAT nt --> nt < MAXMEM)) s & (%s. True) s) s) s & (%s.
(%s. (%(pc, m, e). pc = (Suc (0::nat), Suc (Suc (Suc (0::nat))))) s -->
(%(pc, m, e). tval-case False (%sa. tval-case False (%ta. (%s. (%s. (%(pc, m, e).
(EX sa. m (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))) = NAT sa) & (EX ta. m
(Suc (Suc (0::nat))) = NAT ta)) s & (%s. (%(p, m, e). (ALL ns. m (Suc (Suc
(Suc (Suc (Suc (Suc (0::nat))))))) = NAT ns --> ns < MAXMEM) & (ALL nt.
m (Suc (Suc (0::nat))) = NAT nt --> nt < MAXMEM)) s & (%s. True) s) s)
s & (%s. (%s. (%(pc, m, e). pc = (Suc (0::nat), Suc (Suc (Suc (Suc (0::nat))))))

s −−> (%(pc, m, e). tval-case False (%sa. tval-case False (%ta. (%s. (%s. (%(pc, m, e). (EX sa. m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) = NAT sa) & (EX ta. m (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))) = NAT ta)) s & (%s. (%(p, m, e). (ALL ns. m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) = NAT ns −−> ns < MAXMEM) & (ALL nt. m (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))) = NAT nt −−> nt < MAXMEM)) s & (%s. True) s) s) s & (%s. (%s. (%(pc, m, e). pc = (Suc (0::nat), Suc (Suc (Suc (Suc (Suc (0::nat)))))) s −−> (%(pc, m, e). tval-case False (%sa. tval-case False (%ta. (%s. (%s. (%(pc, m, e). (EX sa. m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) = NAT sa) & (EX ta. m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) = NAT ta)) s & (%s. (%(p, m, e). (ALL ns. m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) = NAT ns −−> ns < MAXMEM) & (ALL nt. m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) = NAT nt −−> nt < MAXMEM)) s & (%s. True) s) s) s & (%s. (%s. (%(pc, m, e). pc = (Suc (0::nat),

Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))) s −−> (%(pc, m, e). tval-case False (%sa. tval-case False (%ta. (%s. (%s. (%s. True) s & (%s. (%(p, m, e). True) s & (%s. True) s) s) s & (%s. (%s. (%(pc, m, e). pc = (Suc (0::nat), Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) s −−> (%(pc, m, e). (%s. (%s. (%(pc, m, e). (EX n. m (Suc (Suc (0::nat))) = NAT n) & (EX n. m (0::nat) = NAT n)) s & (%s. (%(p, m, e). Suc (Suc (0::nat)) < MAXMEM & (0::nat) < MAXMEM) s & (%s. True) s) s) s & (%s. (%(p, m, e). m lnull = NAT null & m lbs = NAT bs & m lnxt = NAT nxt & m lbuf = NAT buf & m null = NAT (0::nat) & m rt = RA (incA (callpc e)) & (EX n. m bs = NAT n & (10::nat) <= n & n < MAXMEM) & (EX l1. List m bs l1 & (EX l2. List m nxt l2 & (EX l3. List (callmem e) bs l3 & l3 = rev l1 @ l2 & (ALL l:set l3. (10::nat) <= l & l < MAXMEM))))) s & (%s. True) s) s) ((Suc (0::nat), Suc (Suc (0::nat))), m, env.h-update (env.h e @ [(Suc (0::nat), Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))]) e)) s) s & (%s. True) s) s) ((Suc (0::nat), Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))), update m ta (m sa), env.h-update (env.h e @ [(Suc (0::nat), Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))]) e)) (%ra. False) (m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) (%ra. False) (m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) s) s

& (%s. True) s) s) ((Suc (0::nat), Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))), update m ta (m sa), env.h-update (env.h e @ [(Suc (0::nat), Suc (Suc (Suc (Suc (Suc (0::nat))))))]) e)) (%ra. False) (m (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) (%ra. False) (m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) s) s & (%s. True) s) s) ((Suc (0::nat), Suc (Suc (Suc (Suc (Suc (0::nat)))))), update m ta (m sa), env.h-update (env.h e @ [(Suc (0::nat), Suc (Suc (Suc (Suc (0::nat)))))]) e)) (%ra. False) (m (Suc (Suc (0::nat))))) (%ra. False) (m (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) s) s & (%s. True) s) s) ((Suc (0::nat), Suc (Suc (Suc (Suc (0::nat))))), update m ta (m sa), env.h-update (env.h e @ [(Suc (0::nat), Suc (Suc (Suc (0::nat))))]) e)) (%ra. False) (m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) (%ra. False) (m (Suc (Suc (0::nat))))) s) s & (%s. True) s) s) ((Suc (0::nat), Suc (Suc (Suc (0::nat)))), m, env.h-update (env.h e @ [(Suc (0::nat), Suc (Suc (0::nat)))]) e)) s) s & (%s. True) s) s) s & (%s. (%s. (%s. (%s. (%s. (%s. (%(pc, m, e). EX pn' i'. m (Suc (Suc (Suc (Suc (0::nat))))) = RA (pn', Suc i') & (EX k m' cl css. env.cs e =

(k, m') # cl # css & (pn', i') = env.h e ! k)) s & (%s. (%(p, m, e). Suc (Suc (Suc (Suc (0::nat)))) < MAXMEM) s & (%s. True) s) s) s & (%s. (%(p, m, e).

m lnull = NAT null & m lbs = NAT bs & m lnxt = NAT nxt & m lbuf = NAT buf & m null = NAT (0::nat) & m rt = RA (incA (callpc e)) & (EX lst. List m bs lst & List (callmem e) bs (rev lst))) s & (%s. True) s) s) s & (%s. (%s. (%(pc, m, e). m (Suc (Suc (Suc (Suc (0::nat))))) = RA (0::nat, Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) & pc = (Suc (0::nat), Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) s & (%s. (%(pc, m, e). (%(p, m, e). m null = NAT (0::nat) & m bs = NAT e1 & m lnull = NAT null & m lbs = NAT bs & m lnxt = NAT nxt & m lbuf = NAT buf & List m bs [e1, e2]) (let (k, m′) = hd (env.cs e); cs′ = tl (env.cs e); h′ = take k (env.h e) in (env.h e ! k, m′, env.h-update h′ (env.cs-update cs′ e)))) s & (%s. True) s) s) s & (%s. True) s) s) s --> (%(pc, m, e). (%s. (%s. (%s. True) s & (%s. (%(p, m, e). True) s & (%s. True) s) s) s & (%s. (%s. True) s & (%s. True) s) s) ((0::nat, Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))), m, env.cs-update (tl (env.cs e)) (env.h-update (env.h e @ [(Suc (0::nat), Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))]) e))) s) s & (%s. True) s) s & (%s. True) s) s) s) s) s

1.2 Verifying the program

lemma *forall-switch4*:
$\forall\ w\ x\ y\ z.\ P\ w\ x\ y\ z \Longrightarrow \forall\ z.\ \forall\ x.\ \forall\ y.\ \forall\ w.\ P\ w\ x\ y\ z$
apply *simp*
done

lemma *List-unique*:
$\bigwedge ls\ ls'\ x.\ 0 \notin set\ (ls\ @\ ls') \Longrightarrow List\ m\ x\ ls \Longrightarrow List\ m\ x\ ls' \Longrightarrow ls = ls'$
apply (*subgoal-tac* $\exists k.\ k = size\ (ls\ @\ ls')$)
 prefer *2*
 apply *simp*
apply (*erule exE*)
apply (*erule rev-mp*)+
apply (*simp only*: *atomize-all*)
apply (*rule forall-switch4*)
apply (*rule allI*)
apply (*rule-tac n=k* **in** *nat-less-induct*)
apply (*rule allI*)+
apply (*rule impI*)+
apply (*case-tac ls*)
apply (*case-tac ls′*)
apply *simp*
apply *simp*
apply (*case-tac ls′*)
apply *simp*
apply (*erule-tac x=length* (*list @ lista*) **in** *allE*)
apply *simp*
apply (*erule-tac x=lista* **in** *allE*)
apply (*erule-tac x=a* **in** *allE*)
apply (*erule-tac x=list* **in** *allE*)
apply *simp*
done

lemma *List-split*:
$\bigwedge$ *x*. *List m x* (*ls*@*a*#*ls'*) $\Longrightarrow$ *List m a ls'*
apply (*induct ls*)
apply *simp*
apply *atomize*
apply (*erule-tac x*=*aa* **in** *allE*)
apply *simp*
done

First we ensure that the program is wellformed.

lemma *wf-prog*:
wf prog
apply (*simp add*: *wf-def domC-prog checkPos.simps prog-def Let-def split-def fst-conv snd-conv cmd.simps ret-succs.simps callpoints-def isCall-def anF.simps*)
done

Then, we prove the verification condition

lemma *List-m-upd*: $\bigwedge$ *z*. $x \neq z \Longrightarrow x \notin$ (*set lst*) $\Longrightarrow$ (*List* (*m*(*x* := *y*)) *z lst*) = (*List m z lst*)
apply (*induct lst*)
apply *simp*
apply *simp*
done

lemmas *prog-ids* = *bs-def lbs-def null-def lnull-def rt-def buf-def lbuf-def nxt-def lnxt-def e1-def e2-def MAX-def MAXMEM-def*

lemmas *vc-simps* = *prog-ids split-def fst-conv snd-conv update-def callstate-def incA-def callpc-def callmem-def nth-append nat-number*

lemma *vc-prog-holds*:
provable prog vc
— start up
apply (*simp add*: *provable-def valid-def* | *rule allI* | *rule impI*)+
apply (*rename-tac pn i m e*)
apply (*cut-tac wf-prog*)
apply (*drule isafeP-mono*)
apply (*drule vc-proof-startup*)
apply *assumption*
apply (*erule conjE* | *erule exE*)+
apply (*simp only*: *vc-def*)

— main proof
apply (*case-tac prog*,((*pn*,*i*),*m*,*e*) $\models$ (*initF prog*))

— (initF prg) implies (isafeF prg (ipc prg))
apply (*rule conjI*, *rule impI*)
apply (*simp add*: *initF-def valid-def vc-simps*)

apply (*simp add*: *initF-def valid-def vc-simps*)

— case not (initF prg)
apply (*erule isafeP-elims*)
apply *simp*

apply (*rule conjI*)
— initF - - ¿ isafe (0,0)
apply (*rule impI*)
apply (*drule-tac t=s''* **in** *sym*)
apply (*simp add*: *vc-simps*)

— (0,5) - - ¿ (1,0)
apply (*drule-tac t=s''* **in** *sym*)
apply (*rule conjI*)
apply (*simp add*: *vc-simps Let-def*)
apply (*rule impI*)
apply (*erule conjE* | *erule exE*)+
apply (*simp add*: *vc-simps*)
apply (*rule-tac x=[e1,e2]* **in** *exI*)
apply (*simp add*: *vc-simps*)

apply (*simp only*: *simp-thms split-def fst-conv snd-conv*)
— (1,0) nach (1,2)
apply (*rule conjI*)
apply (*rule impI*)
apply (*erule conjE* | *erule exE*)+
apply (*simp add*: *vc-simps*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*case-tac lst*)
apply (*simp add*: *List.simps*)
apply (*simp add*: *List.simps*)
apply (*case-tac list*)
apply (*simp add*: *List.simps*)
apply (*rule-tac x=[a]* **in** *exI*)
apply (*rule context-conjI*)
apply (*simp add*: *List-m-upd*)
apply *arith*
apply (*rule-tac x=[]* **in** *exI*)
apply *simp*

apply (*simp add*: *List.simps*)
apply (*rule-tac x=[a]* **in** *exI*)
apply (*simp add*: *List-m-upd*)

```
apply (rule conjI)
apply arith
apply (rule impI)
apply (rule-tac x=aa#lista in exI)
apply simp
apply (rule conjI)
apply (rule impI)
apply arith

apply (erule conjE | erule exE)+
apply (subgoal-tac 0 ∉ set (a#aa#lista))
 prefer 2
 apply simp
 apply (rule classical)
 apply (erule-tac x=0 in ballE)
 apply simp

 apply simp
apply (subgoal-tac distinct (a#aa#lista))
 prefer 2
 apply (rule-tac m=m and x=a in  List-distinct)
 apply simp
 apply (rule classical)
 apply (erule-tac x=0 in ballE)
 apply simp

 apply simp
apply (rule impI)
apply (subgoal-tac List (m(Suc (Suc 0) := NAT aa, a := NAT 0)) aa lista = List
(m(Suc (Suc 0) := NAT aa)) aa lista)
 prefer 2
 apply (rule List-m-upd)
 apply simp

 apply simp
apply (subgoal-tac List (m(Suc (Suc 0) := NAT aa)) aa lista = List m aa lista)
 prefer 2
 apply (rule List-m-upd)
 apply simp

 apply (rule classical)
 apply (erule-tac x=Suc (Suc 0) in ballE)
 apply simp

 apply simp
apply simp

apply (rule impI)
apply (case-tac css)
```

apply *simp*

apply (*simp add*: *Let-def*)

— (1,2) nach (1,8)
apply (*rule conjI*)
apply (*rule impI*)
apply (*erule conjE* | *erule exE*)+
apply (*simp add*: *vc-simps*)
apply (*case-tac css*)
apply (*simp add*: *Let-def*)

apply (*simp add*: *Let-def*)
apply (*rule conjI*)
apply (*rule-tac x=fst c* **in** *exI*)
apply *simp*
apply (*rule-tac x=snd c* **in** *exI*)
apply *simp*

apply (*rule-tac x=l1* **in** *exI*)
apply *simp*
apply (*subgoal-tac l2* = [])
 prefer *2*
 apply (*rule classical*)
 apply *simp*
 apply (*simp add*: *neq-Nil-conv*)
 apply (*erule exE*)+
 apply *simp*
apply *simp*

— (1,2) nach (1,2)
apply (*rule impI*)
apply (*erule conjE* | *erule exE*)+
apply (*case-tac css*)
apply *simp*

apply (*rule conjI*)
apply (*simp add*: *vc-simps*)
apply (*case-tac l2*)
apply *simp*

apply *simp*

apply (*simp add*: *Let-def vc-simps cases*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*rule conjI*)
apply (*case-tac l2*)

apply *simp*

apply *simp*
apply (*rule-tac x=l1* **in** *exI*)
apply (*rule conjI*)
apply (*subgoal-tac List* (*m*(*Suc na* := *m na*, *Suc 0* := *NAT nc*, *na* := *m na*)) (*Suc 0*) *l1* = *List* (*m*(*Suc na* := *m na*, *Suc 0* := *NAT nc*)) (*Suc 0*) *l1*)
prefer *2*
apply (*rule List-m-upd*)
apply *simp*

apply (*rule classical*)
apply (*erule-tac x=na* **in** *ballE*)
apply *simp*

apply *simp*
apply (*subgoal-tac m*(*Suc na* := *m na*, *Suc 0* := *NAT nc*) = *m*(*Suc na* := *m na*))
prefer *2*
apply (*rule ext*)
apply *simp*
apply *simp*
apply (*subgoal-tac List* (*m*(*Suc na* := *NAT na*)) (*Suc 0*) *l1* = *List m* (*Suc 0*) *l1*)
prefer *2*
apply (*rule List-m-upd*)
apply *simp*

apply (*rule classical*)
apply (*erule-tac x=Suc na* **in** *ballE*)
apply *simp*

apply *simp*
apply *simp*

apply (*rule-tac x=l2* **in** *exI*)
apply (*subgoal-tac List* (*m*(*Suc na* := *m na*, *Suc 0* := *NAT nc*, *na* := *m na*)) *na* *l2* = *List m* (*Suc* (*Suc 0*)) *l2*)
prefer *2*
apply (*subgoal-tac m*(*Suc na* := *m na*, *Suc 0* := *NAT nc*, *na* := *m na*) = *m*(*Suc na* := *m na*, *Suc 0* := *NAT nc*))
prefer *2*
apply (*rule ext*)
apply *simp*
apply *simp*
apply (*subgoal-tac List* (*m*(*Suc na* := *NAT na*, *Suc 0* := *NAT nc*)) *na* *l2* = *List* (*m*(*Suc na* := *NAT na*)) *na* *l2*)
prefer *2*
apply (*rule List-m-upd*)
apply *simp*

```
  apply (rule classical)
  apply (erule-tac x=Suc 0 in ballE)
  apply simp

  apply simp
 apply simp
 apply (subgoal-tac List (m(Suc na := NAT na)) na l2 = List m na l2)
  prefer 2
  apply (rule List-m-upd)
  apply simp

  apply (rule classical)
  apply (erule-tac x=Suc na in ballE)
  apply simp

  apply simp
 apply simp
apply simp

apply (rule impI)
apply (rule conjI)
apply (case-tac l2)
apply simp

apply simp
apply arith

apply (rule impI)
apply (rule conjI)
apply (case-tac l2)
apply simp

apply simp
apply arith

apply (rule impI)
apply (rule conjI)
apply (case-tac l2)
apply simp

apply simp
apply arith

apply (rule impI)
apply (rule conjI)
apply (case-tac l2)
apply simp

apply simp
```

```
apply arith

apply (rule impI)
apply (rule conjI)
apply (rule impI)
apply (case-tac l2)
apply simp

apply simp

apply (rule impI)
apply (rule conjI)
apply (rule impI)
apply (case-tac l2)
apply simp

apply simp

apply (rule impI)
apply (case-tac l2)
apply simp

apply simp
apply (rule conjI)
apply (case-tac lista)
apply simp

apply simp

apply (rule-tac x=aa#l1 in exI)
apply (rule conjI)
apply simp
apply (case-tac l1)
apply simp

apply simp
apply (rule conjI)
apply (rule impI)
apply (simp only:)
apply (erule conjE)+
apply (erule-tac P=?P ≤ Suc 0 in rev-mp)
apply (simp (no-asm))

apply (rule impI)
apply (subgoal-tac distinct (Suc 0 #(rev listb @ ab # aa # lista)))
 prefer 2
 apply (rule-tac m=snd c and x=Suc 0 in List-distinct)
 apply simp
 apply (rule conjI)
```

apply (*erule conjE*)+
apply (*erule-tac P*=*?P* $\leq$ *ab* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*rule conjI*)
apply (*rule classical*)
apply (*erule conjE*)+
apply (*erule-tac x*=*0* **in** *ballE*)
apply (*erule conjE*)
apply (*erule-tac P*=*?P* $\leq$ *0* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*erule-tac P*=*0* $\notin$ *?P* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*erule conjE*)+
apply (*rule classical*)
apply (*erule-tac x*=*0* **in** *ballE*)
apply (*erule conjE*)
apply (*erule-tac P*=*?P* $\leq$ *0* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*erule-tac P*=*0* $\notin$ *?P* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply *assumption*

apply (*erule conjE*)+
apply (*subgoal-tac List* (*m*(*Suc* (*Suc* (*Suc 0*)) := *m aa*, *aa* := *NAT ab*, *Suc 0* := *NAT aa*, *Suc* (*Suc 0*) := *m aa*)) *ab listb* = *List* (*m*(*Suc* (*Suc* (*Suc 0*)) := *m aa*, *aa* := *NAT ab*, *Suc 0* := *NAT aa*)) *ab listb*)
prefer *2*
apply (*rule List-m-upd*)
apply (*erule-tac P*=*?P* $\leq$ *ab* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*rule classical*)
apply (*erule-tac x*=*Suc* (*Suc 0*) **in** *ballE*)
apply (*simp only*:)
apply (*erule conjE*)
apply (*erule-tac P*=*?P* $\leq$ *Suc* (*Suc 0*) **in** *rev-mp*)
apply (*simp* (*no-asm*))
apply (*erule-tac P*=*Suc* (*Suc 0*) $\notin$ *?P* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*subgoal-tac List* (*m*(*Suc* (*Suc* (*Suc 0*)) := *m aa*, *aa* := *NAT ab*, *Suc 0* := *NAT aa*)) *ab listb* =
List (*m*(*Suc* (*Suc* (*Suc 0*)) := *m aa*, *aa* := *NAT ab*)) *ab listb*)
prefer *2*

apply (*rule List-m-upd*)
apply (*erule-tac P=?P ≤ ab* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*rule classical*)
apply (*erule-tac x=Suc 0* **in** *ballE*)
apply (*simp only*:)
apply (*erule conjE*)
apply (*erule-tac P=?P ≤ Suc 0* **in** *rev-mp*)
apply (*simp* (*no-asm*))
apply (*erule-tac P=Suc 0 ∉ ?P* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*subgoal-tac List* (*m*(*Suc* (*Suc* (*Suc 0*)) := *m aa*, *aa* := *NAT ab*)) *ab listb*
=
List (*m*(*Suc* (*Suc* (*Suc 0*)) := *m aa*)) *ab listb*)
prefer *2*
apply (*rule List-m-upd*)
apply (*erule-tac P=distinct ?P* **in** *rev-mp*)
apply (*simp* (*no-asm*))
apply (*rule impI*)
apply (*erule conjE*)+
apply (*rule not-sym*)
apply *assumption*

apply (*erule-tac P=distinct ?P* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*subgoal-tac List* (*m*(*Suc* (*Suc* (*Suc 0*)) := *m aa*)) *ab listb* = *List m ab listb*)
prefer *2*
apply (*rule List-m-upd*)
apply (*erule-tac P=?P ≤ ab* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*rule classical*)
apply (*erule-tac x=Suc* (*Suc* (*Suc 0*)) **in** *ballE*)
apply (*simp only*:)
apply (*erule conjE*)
apply (*erule-tac P=?P ≤ Suc* (*Suc* (*Suc 0*)) **in** *rev-mp*)
apply (*simp* (*no-asm*))
apply (*erule-tac P=Suc* (*Suc* (*Suc 0*)) *∉ ?P* **in** *rev-mp*)
apply (*simp* (*no-asm*))
apply (*simp only*:)

apply (*rule-tac x=lista* **in** *exI*)
apply (*rule conjI*)
apply (*case-tac lista*)
apply *simp*

apply *simp*
apply (*subgoal-tac distinct* (*Suc 0* #(*rev l1* @ *aa* # *ab* # *listb*)))
prefer *2*
apply (*rule-tac m=snd c* **and** *x=Suc 0* **in** *List-distinct*)
apply *simp*
apply (*rule conjI*)
apply (*erule conjE*)+
apply (*erule-tac P=?P* ≤ *ab* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*rule conjI*)
apply (*rule classical*)
apply (*erule conjE*)+
apply (*erule-tac x=0* **in** *ballE*)
apply (*erule conjE*)
apply (*erule-tac P=?P* ≤ *0* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*erule-tac P=0* ∉ *?P* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*erule conjE*)+
apply (*rule classical*)
apply (*erule-tac x=0* **in** *ballE*)
apply (*erule conjE*)
apply (*erule-tac P=?P* ≤ *0* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*erule-tac P=0* ∉ *?P* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply *assumption*

apply (*erule conjE*)+
apply (*subgoal-tac List* (*m*(*Suc* (*Suc* (*Suc 0*)) := *NAT ab*, *aa* := *NAT nc*, *Suc 0* := *NAT aa*, *Suc* (*Suc 0*) := *NAT ab*)) *ab listb* = *List* (*m*(*Suc* (*Suc* (*Suc 0*)) := *NAT ab*, *aa* := *NAT nc*, *Suc 0* := *NAT aa*)) *ab*
listb)
prefer *2*
apply (*rule List-m-upd*)
apply (*erule-tac P=?P* ≤ *ab* **in** *rev-mp*)
apply (*simp* (*no-asm*))

apply (*rule classical*)
apply (*erule-tac x=Suc* (*Suc 0*) **in** *ballE*)
apply (*simp only*:)
apply (*erule conjE*)
apply (*erule-tac P=?P* ≤ *Suc* (*Suc 0*) **in** *rev-mp*)
apply (*simp* (*no-asm*))

```
 apply (erule-tac P=Suc (Suc 0) ∉ ?P in rev-mp)
 apply (simp (no-asm))

apply (subgoal-tac List (m(Suc (Suc (Suc 0)) := NAT ab, aa := NAT nc, Suc 0
:= NAT aa)) ab listb =
List (m(Suc (Suc (Suc 0)) := NAT ab, aa := NAT nc)) ab listb)
 prefer 2
 apply (rule List-m-upd)
 apply (erule-tac P=?P ≤ ab in rev-mp)
 apply (simp (no-asm))

 apply (rule classical)
 apply (erule-tac x=Suc 0 in ballE)
 apply (simp only:)
 apply (erule conjE)
 apply (erule-tac P=?P ≤ Suc 0 in rev-mp)
 apply (simp (no-asm))
 apply (erule-tac P=Suc 0 ∉ ?P in rev-mp)
 apply (simp (no-asm))

apply (subgoal-tac List (m(Suc (Suc (Suc 0)) := NAT ab, aa := NAT nc)) ab
listb =
 List (m(Suc (Suc (Suc 0)) := NAT ab)) ab listb)
 prefer 2
 apply (rule List-m-upd)
 apply (erule-tac P=distinct ?P in rev-mp)
 apply (simp (no-asm))

 apply (erule-tac P=distinct ?P in rev-mp)
 apply (simp (no-asm))

apply (subgoal-tac List (m(Suc (Suc (Suc 0)) := NAT ab)) ab listb = List m ab
listb)
 prefer 2
 apply (rule List-m-upd)
 apply (erule-tac P=?P ≤ ab in rev-mp)
 apply (simp (no-asm))

 apply (rule classical)
 apply (erule-tac x=Suc (Suc (Suc 0)) in ballE)
 apply (simp only:)
 apply (erule conjE)
 apply (erule-tac P=?P ≤ Suc (Suc (Suc 0)) in rev-mp)
 apply (simp (no-asm))
 apply (erule-tac P=Suc (Suc (Suc 0)) ∉ ?P in rev-mp)
 apply (simp (no-asm))
apply (simp only:)

apply (rule conjI)
```

apply *simp*

apply *simp*
apply (*erule conjE*)+
apply *assumption*
done

end