theory *EX-RecMultMem* = *SALMemFWInst*:

1 SAL Example: Multiplication

This example is about a program that multplies variable C1 with C2 and writes the result to C3.

The variables C1, C2, C3 are just abbreviations for the addresses 1, 2 and 3.

constdefs
C1 :: *nat* — cell1
C1 ≡ *1*
C2 :: *nat* — cell2
C2 ≡ *2*
C3 :: *nat* — cell3
C3 ≡ *3*

At some point in the program we need to check whether a variable contains number one or zero. Since conditional jumps in SAL expect their arguments in form of variables, we introduce variables Zero and One. The progam uses them to the constants NAT 0 and NAT 1.

constdefs
Zero::*nat* — stores NAT 0
Zero ≡ *4*
One:: *nat* — stores NAT 1
One ≡ *5*

Our program will do multiplication by recursively calling a procedure that decrements C1 and adds C2 to C3. Whenever this procedure is called, the return address, which is the current address incremented by one, is dumped in a variable r. The address of r, which is 6, is kept in the variable Ra.

constdefs
R :: *nat* — return address buffer
R ≡ *6*
Ra :: *nat* — stores the address of R
Ra ≡ *7*

The program maintains a stack of return addresses. This stack starts at address Soff and grows towards higher addresses. The variable S contains the stack pointer. This is the address, where the next element of the stack is going to be stored.

constdefs
S:: *nat* — stack pointer
S ≡ *8*
Soff:: *nat* — stack base
Soff ≡ *9*

Since SAL programs are executed on a default initial state, we have to simulate the input phase of the program by SET operations. In this example we analyse the program for inputs arg1 and arg2.

constdefs
arg1::nat
arg1≡2
arg2::nat
arg2≡3

1.1 Program Code

Next, we present the program without annotations.

prog = [(*0*, [	Procedure 0 (Startup)
SET Zero 0,	Initialise Zero with NAT 0.
SET One 1,	Initialise One with NAT 1.
SET RA r,	Initialise RA with NAT r (r = 6).
SET S Soff,	Initialise S with NAT Soff (Soff = 9).
SET C1 arg1,	Initialise Argument C1 with NAT arg1.
SET C2 arg2,	Initialise Argument C2 with NAT arg2.
SET C3 0,	Initialise Result Variable C3 with NAT 0.
CALL r 1,	Start multplication procedure.
HALT]),	Stop execution
(*1*,[	Procedure 1 (Multiplication)
MOV Ra S,	Push the return address onto the Stack.
JMPEQ C1 Zero 7,	If C1 is NAT 0 we are done, otherwise . . .
SUB C1 One,	
	. . . we decrement C1
ADD C3 C2,	and add C2 to the result variable C3.
INC S,	We increment the stack pointer, before .
CALL r 1,	we do the recursive call.
SUB S One,	After return we restore the old stack pointer
MOV S Ra,	and copy the return address from the stack to r.
RET r])]	We finish this call by returning to the caller.

1.2 Program with Annotations

constdefs
prog :: *SALprogram*
prog ≡ [
(*0*, [(*SET Zero 0*,*None*),
(*SET One 1*,*None*),
(*SET Ra R*,*None*),
(*SET S Soff*,*None*),

```
      (SET C1 arg1,None),
      (SET C2 arg2,None),
      (SET C3 0,None),
      (CALL R 1,Some (λ(pc,m,e). m C1 = NAT arg1 ∧
                                m C2 = NAT arg2 ∧
                                m C3 = NAT 0 ∧
                                m Zero = NAT 0 ∧
                                m One = NAT 1 ∧
                                m Ra = NAT R ∧
                                m S = NAT Soff ∧
                               (∀ Z. Z∉{C1,C2,C3,Zero,One,Ra,S} ⟶  m Z = (←m e)
Z))),
      (JMPB 0,Some (λ(pc,m,e). m C3 = NAT (arg1 ∗ arg2)))]),
(1,[ (MOV Ra S,Some (λ(pc,m,e). (∃ n1. m C1 = NAT n1 ∧
                               (∃ n2. m C2 = NAT n2 ∧
                               (∃ n3. m C3 = NAT n3 ∧
                                n3+(n1∗n2)≤MAX))) ∧
                                m Zero = NAT 0 ∧
                                m One = NAT 1 ∧
                                m Ra = NAT R ∧
                               (∃ s. m S = NAT s ∧ Soff ≤ s ∧
                                  (∃ n1. m C1 = NAT n1 ∧ s+n1 ≤ MAX)) ∧
                                m R = RA (incA (←pc e)) ∧
                               (∀ Z. Z ≠ R ⟶ m Z = (←m e) Z))),
     (JMPEQ C1 Zero 7,None),
     (SUB C1 One,None),
     (ADD C3 C2,None),
     (INC S,None),
     (CALL R 1,Some (λ(pc,m,e). (∃ n1. m C1 = NAT n1 ∧
                               (∃ n2. m C2 = NAT n2 ∧
                               (∃ n3. m C3 = NAT n3 ∧
                                (∃ n1'. (←m e) C1 = NAT n1' ∧
                                (∃ n2'. (←m e) C2 = NAT n2' ∧
                                (∃ n3'. (←m e) C3 = NAT n3' ∧
                                 0 < n1' ∧ n1 = (n1'−(1::nat)) ∧
                                 n2=n2' ∧ n3=n3'+n2' ∧
                                 n3+(n1∗n2) ≤ MAX)))))) ∧
                                m Zero = NAT 0 ∧
                                m One = NAT 1 ∧
                                m Ra = NAT R ∧
                                (∃ s. m S = NAT s ∧ Soff ≤ s ∧
                                (∃ s'. (←m e) S = NAT s' ∧
                                  s = (s'+(1::nat)) ∧
                                  m s' = RA (incA (←pc e))) ∧
                                  (∃ n1. m C1 = NAT n1 ∧ s+n1 ≤ MAX)) ∧
                                (∀ Z. Z ≠ R ∧ Z≠S ∧ Z≠C1 ∧ Z≠C3 ∧
                                (∃ s. m S = NAT s ∧ (Suc Z) < s) ⟶ m Z = (←m e)
Z))),
     (SUB S One,Some (λ(pc,m,e). ∃ n3. m C3 = NAT n3 ∧
```

(∃ *n3′*. ($\overleftarrow{m}$ *e*) *C3* = *NAT n3′* ∧
(∃ *n1′*. ($\overleftarrow{m}$ *e*) *C1* = *NAT n1′* ∧
(∃ *n2′*. ($\overleftarrow{m}$ *e*) *C2* = *NAT n2′* ∧
n3=*n3′*+(*n1′*∗*n2′*)))) ∧
m Zero = *NAT 0* ∧
m One = *NAT 1* ∧
m Ra = *NAT R* ∧
(∃ *s*. *m S* = *NAT s* ∧ *Soff* ≤ *s* ∧
(∃ *s′*. ($\overleftarrow{m}$ *e*) *S* = *NAT s′* ∧
s = (*s′*+(*1*::*nat*)) ∧
m s′ = *RA* (*incA* ($\overleftarrow{pc}$ *e*))) ∧
(∃ *n1*. *m C1* = *NAT n1* ∧ *s*+*n1* ≤ *MAX*)) ∧
(∀ *Z*. *Z*≠*R* ∧ *Z*≠*S* ∧ *Z*≠*C1* ∧ *Z*≠*C3* ∧ (∃ *s*. *m S* = *NAT*
s ∧ (*Suc Z*) < *s*)
⟶ *m Z* = ($\overleftarrow{m}$ *e*) *Z*))),
(*MOV S Ra*,*None*),
(*RET R*,*Some* (λ(*pc*,*m*,*e*). ∃ *n3*. *m C3* = *NAT n3* ∧
(∃ *n3′*. ($\overleftarrow{m}$ *e*) *C3* = *NAT n3′* ∧
(∃ *n1′*. ($\overleftarrow{m}$ *e*) *C1* = *NAT n1′* ∧
(∃ *n2′*. ($\overleftarrow{m}$ *e*) *C2* = *NAT n2′* ∧
n3=*n3′*+(*n1′*∗*n2′*)))) ∧
m Zero = *NAT 0* ∧
m One = *NAT 1* ∧
m Ra = *NAT R* ∧
m R = *RA* (*incA* ($\overleftarrow{pc}$ *e*)) ∧
(∃ *s*. *m S* = *NAT s* ∧ *Soff* ≤ *s* ∧
(∃ *n1*. *m C1* = *NAT n1* ∧ *s*+*n1* ≤ *MAX*)) ∧
(∀ *Z*. *Z*≠*R* ∧ *Z*≠*C1* ∧ *Z*≠*C3* ∧ (∃ *s*. *m S* = *NAT s* ∧ *Z*
< *s*)
⟶ *m Z* = ($\overleftarrow{m}$ *e*) *Z*)))

])
]

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* (*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* (*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* (*Suc* (*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 (0::nat)))))) s) --> ((%(pc,
m, e). ((%s. (((%s. (((%(pc, m, e). ((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 (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). ((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 (0::nat))))))))
s) --> ((%(pc, m, e). ((%s. (((%s. (((%(pc, m, e). ((Suc (Suc (Suc (0::nat))))
<= MAX)) s) & ((%s. (((%(p, m, e). ((Suc (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). ((0::nat) <= MAX)) s) & ((%s. (((%(p, m, e). ((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. (((%s. True) s) & ((%s. (((%(p, m, e). ((Suc (Suc
(Suc (Suc (Suc (Suc (0::nat))))))) < MAXMEM)) s) & ((%s. True) s))) s))) s)
& ((%s. (((%(pc, m, e). (((m C1) = (NAT arg1)) & (((m C2) = (NAT arg2)) &
(((m C3) = (NAT (0::nat))) & (((m Zero) = (NAT (0::nat))) & (((m One) =
(NAT (1::nat))) & (((m Ra) = (NAT R)) & (((m S) = (NAT Soff)) & (ALL Z. ((Z
~: {C1, C2, C3, Zero, One, Ra, S}) --> ((m Z) = (callmem e Z)))))))))))) s) &
((%s. True) s))) s))) (((0::nat), (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))),
(update m (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 (0::nat)))))))),
(update m (Suc (Suc (0::nat))) (NAT (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
(Suc (0::nat)) (NAT (Suc (Suc (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 (Suc
(Suc (Suc (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 (Suc (Suc (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 (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 (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 (Suc (Suc (0::nat))))))) < MAXMEM)) s) & ((%s. True) s)))
s))) s) & ((%s. (((%(pc, m, e). (((m C1) = (NAT arg1)) & (((m C2) = (NAT
arg2)) & (((m C3) = (NAT (0::nat))) & (((m Zero) = (NAT (0::nat))) & (((m
One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & (((m S) = (NAT Soff)) &
(ALL Z. ((Z ~: {C1, C2, C3, Zero, One, Ra, S}) --> ((m Z) = (callmem e

Z)))))))))))) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). (pc = ((0::nat), (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 (Suc (Suc (Suc (Suc (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 (Suc (Suc (Suc (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. (((%(pc, m, e). ((EX n1. (((m C1) = (NAT n1)) & (EX n2. (((m C2) = (NAT n2)) & (EX n3. (((m C3) = (NAT n3)) & ((n3 + (n1 * n2)) <= MAX))))))) & (((m Zero) = (NAT (0::nat))) & (((m One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & ((EX s. (((m S) = (NAT s)) & ((Soff <= s) & (EX n1. (((m C1) = (NAT n1)) &
((s + n1) <= MAX)))))) & (((m R) = (RA (incA (callpc e)))) & (ALL Z. ((Z ~= R) --> ((m Z) = (callmem e Z))))))))))) s) & ((%s. True) s))) s))) (((Suc (0::nat)), (0::nat)), (update m (Suc (Suc (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 (0::nat)))))))))]) e))))) s))) s) & ((%s. True) s))) s) & ((%s. (((%s. (((%s. (((%s. (((%s. (((%s. (((%s. True) s) & ((%s. (((%(p, m, e). True) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). ((m C3) = (NAT (arg1 * arg2)))) 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. (((%s. True) s) & ((%s. (((%(p, m, e). True) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). ((m C3) = (NAT (arg1 * arg2)))) s) & ((%s. True) s))) s))) (((0::nat), (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))), m, (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. (((%s. (((%s. (((%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 (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 (Suc (Suc (0::nat))))))))) = (NAT nt)) --> (nt < MAXMEM))))) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). ((EX n1. (((m C1) = (NAT n1)) & (EX n2. (((m C2) = (NAT n2)) & (EX n3. (((m C3) = (NAT n3)) & ((n3 + (n1 * n2)) <= MAX))))))) & (((m Zero) = (NAT (0::nat))) & (((m One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & ((EX s. (((m S) = (NAT s)) & ((Soff <= s) & (EX n1. (((m C1) = (NAT n1)) & ((s + n1) <= MAX)))))) & (((m R) = (RA (incA (callpc e)))) & (ALL Z. ((Z ~= R) --> ((m Z) = (callmem e Z))))))))))) 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 n. ((m (Suc (0::nat))) = (NAT n))) & (EX n. ((m (Suc (Suc (Suc (Suc (0::nat)))))) = (NAT n))))) s) & ((%s. (((%(p, m, e). (((Suc (0::nat)) < MAXMEM) & ((Suc (Suc (Suc (Suc (0::nat))))) < MAXMEM))) s) & ((%s. True) s))) s))) s) & ((%s. (((%s. (((%(pc, m, e). ((EX n n'. (((m (Suc (0::nat))) = (NAT n)) & (((m (Suc (Suc (Suc (Suc (0::nat)))))) = (NAT n')) & (n = n'))))

& (pc = ((Suc (0::nat)), (Suc (0::nat)))))) s) --> ((%(pc, m, e). ((%s. (((%s. (((%(pc, m, e). (EX pn' i'. (((m (Suc (Suc (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 (Suc (Suc (0::nat)))))) < MAXMEM)) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). (EX n3. (((m C3) = (NAT n3)) & ((EX n3'. (((callmem e C3) = (NAT n3')) & (EX n1'. (((callmem e C1) = (NAT n1')) & (EX n2'. (((callmem e C2) = (NAT n2')) & (n3 = (n3' + (n1' * n2')))))))))) & (((m Zero) = (NAT (0::nat))) & (((m One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & (((m R) = (RA (incA (callpc e)))) & ((EX s. (((m S) = (NAT s)) & ((Soff <= s) & (EX n1. (((m C1) = (NAT n1)) & ((s + n1) <= MAX)))))) & (ALL Z. (((Z ~= R) & ((Z ~= C1) & ((Z ~= C3) &
(EX s. (((m S) = (NAT s)) & (Z < s)))))) --> ((m Z) = (callmem e Z)))))))))))) 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 (0::nat)))]) e)))) s))) s) & ((%s. (((%s. (((%(pc, m, e). ((EX n n'. (((m (Suc (0::nat))) = (NAT n)) & (((m (Suc (Suc (Suc (Suc (0::nat)))))) = (NAT n')) & (n ~= n')))) & (pc = ((Suc (0::nat)), (Suc (0::nat)))))) s) --> ((%(pc, m, e). ((%s. (((%s. (((%(pc, m, e). ((EX n. ((m (Suc (0::nat))) = (NAT n))) & (EX n. ((m (Suc (Suc (Suc (Suc (Suc (0::nat))))))) = (NAT n))))) s) &
((%s. (((%(p, m, e). (((Suc (0::nat)) < MAXMEM) & ((Suc (Suc (Suc (Suc (Suc (0::nat)))))) < MAXMEM))) s) & ((%s. True) s))) s))) s) & ((%s. (((%s. (((%(pc, m, e). (pc = ((Suc (0::nat)), (Suc (Suc (0::nat)))))) s) --> ((%(pc, m, e). ((%s. (((%s. (((%(pc, m, e). ((EX n. ((m (Suc (Suc (Suc (0::nat))))) = (NAT n))) & ((EX n. ((m (Suc (Suc (0::nat)))) = (NAT n))) & (EX n. (((lift op + (m (Suc (Suc (Suc (0::nat))))) (m (Suc (Suc (0::nat))))) = (NAT n)) & (n <= MAX)))))) s) & ((%s. (((%(p, m, e). (((Suc (Suc (Suc (0::nat)))) < MAXMEM) & ((Suc (Suc (0::nat))) < 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). ((%s. (((%s. (((%(pc, m, e). (EX n. (((m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) = (NAT n)) & (n < 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 = ((Suc (0::nat)), (Suc (Suc (Suc (Suc (0::nat)))))))) s) --> ((%(pc, m, e). ((%s. (((%s. (((%s. True) s) & ((%s. (((%(p, m, e). ((Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))) < MAXMEM)) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). ((EX n1. (((m C1) = (NAT n1)) & (EX n2. (((m C2) = (NAT n2)) & (EX n3. (((m C3) = (NAT n3)) & (EX n1'. (((callmem e C1) = (NAT n1')) & (EX n2'. (((callmem e C2) = (NAT n2')) & (EX n3'. (((callmem e C3) = (NAT n3')) & (((0::nat) < n1') &
((n1 = (n1' - (1::nat))) & ((n2 = n2') & ((n3 = (n3' + n2')) & ((n3 + (n1 * n2)) <= MAX))))))))))))))))) & (((m Zero) = (NAT (0::nat))) & (((m One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & ((EX s. (((m S) = (NAT s)) & ((Soff <= s) & ((EX s'. (((callmem e S) = (NAT s')) & ((s = (s' + (1::nat))) & ((m s') = (RA (incA (callpc e))))))) & (EX n1. (((m C1) = (NAT n1)) & ((s + n1) <= MAX))))))) & (ALL Z. (((Z ~= R) & ((Z ~= S) & ((Z ~= C1) & ((Z ~= C3) & (EX s. (((m S) = (NAT s)) & ((Suc Z) < s))))))) --> ((m Z)

= (callmem e Z)))))))))) s) &
((%s. True) s))) s))) (((Suc (0::nat)), (Suc (Suc (Suc (Suc (Suc (0::nat))))))), (update m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) (lift op + (m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) (NAT (1::nat)))), (env.h-update ((env.h e) @ [((Suc (0::nat)), (Suc (Suc (Suc (Suc (0::nat))))))]) e)))) s))) s) & ((%s. True) s))) s))) (((Suc (0::nat)), (Suc (Suc (Suc (Suc (0::nat)))))), (update m (Suc (Suc (Suc (0::nat)))) (lift op + (m (Suc (Suc (Suc (0::nat))))) (m (Suc (Suc (0::nat)))))), (env.h-update ((env.h e) @ [((Suc (0::nat)), (Suc (Suc (Suc (0::nat)))))]) e)))) s))) s) &
((%s. True) s))) s))) (((Suc (0::nat)), (Suc (Suc (Suc (0::nat))))), (update m (Suc (0::nat)) (lift op − (m (Suc (0::nat))) (m (Suc (Suc (Suc (Suc (Suc (0::nat))))))))), (env.h-update ((env.h e) @ [((Suc (0::nat)), (Suc (Suc (0::nat))))]) e)))) s))) s) & ((%s. True) s))) s))) (((Suc (0::nat)), (Suc (Suc (0::nat)))), m, (env.h-update ((env.h e) @ [((Suc (0::nat)), (Suc (0::nat)))]) e)))) s))) s) & ((%s. True) s))) 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 (Suc (0::nat))))))))))) (%ra. False) (m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) s))) s) &
((%s. True) s))) s) & ((%s. (((%s. (((%s. (((%s. (((%s. (((%s. (((%s. True) s) & ((%s. (((%(p, m, e). ((Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))) < MAXMEM)) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). ((EX n1. (((m C1) = (NAT n1)) & (EX n2. (((m C2) = (NAT n2)) & (EX n3. (((m C3) = (NAT n3)) & (EX n1′. (((callmem e C1) = (NAT n1′)) & (EX n2′. (((callmem e C2) = (NAT n2′)) & (EX n3′. (((callmem e C3) = (NAT n3′)) & (((0::nat) < n1′) & ((n1 = (n1′ − (1::nat))) & ((n2 = n2′) & ((n3 = (n3′ + n2′)) & ((n3 + (n1 ∗ n2)) <= MAX))))))))))))))))) & (((m Zero) = (NAT (0::nat))) & (((m One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & ((EX s. (((m S) = (NAT s)) & ((Soff <= s) & ((EX s′. (((callmem e S) = (NAT s′)) & ((s = (s′ + (1::nat))) & ((m s′) = (RA (incA (callpc e))))))) &
(EX n1. (((m C1) = (NAT n1)) & ((s + n1) <= MAX))))))) & (ALL Z. (((Z ~= R) & ((Z ~= S) & ((Z ~= C1) & ((Z ~= C3) & (EX s. (((m S) = (NAT s)) & ((Suc Z) < s))))))) --> ((m Z) = (callmem e Z)))))))))) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). (pc = ((Suc (0::nat)), (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 (Suc (Suc (Suc (Suc (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 (Suc (Suc (Suc (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. (((%(pc, m, e). ((EX n1. (((m C1) = (NAT n1)) & (EX n2. (((m C2) = (NAT n2)) & (EX n3. (((m C3) = (NAT n3)) & ((n3 + (n1 ∗ n2)) <= MAX))))))) & (((m Zero) = (NAT (0::nat))) & (((m One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & ((EX s. (((m S) = (NAT s)) & ((Soff <= s) & (EX n1. (((m C1) = (NAT n1)) & ((s + n1) <= MAX)))))) & (((m R) = (RA (incA (callpc e)))) & (ALL Z. ((Z ~= R) -->
((m Z) = (callmem e Z))))))))))) s) & ((%s. True) s))) s))) (((Suc (0::nat)), (0::nat)), (update m (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))) (RA ((Suc

(0::nat)), (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) @ [((Suc (0::nat)), (Suc (Suc (Suc (Suc (Suc (0::nat)))))))]) e))))) s))) s) & ((%s. True) s))) s) & ((%s. (((%s. (((%s. (((%s. (((%s. (((%s. (((%(pc, m, e). ((EX n. ((m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) = (NAT n))) & (EX n. ((m (Suc (Suc (Suc (Suc (Suc (0::nat))))))) = (NAT n))))) s) &
((%s. (((%(p, m, e). (((Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) < MAXMEM) & ((Suc (Suc (Suc (Suc (Suc (0::nat)))))) < MAXMEM))) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). (EX n3. (((m C3) = (NAT n3)) & ((EX n3'. (((callmem e C3) = (NAT n3')) & (EX n1'. (((callmem e C1) = (NAT n1')) & (EX n2'. (((callmem e C2) = (NAT n2')) & (n3 = (n3' + (n1' * n2'))))))))) & (((m Zero) = (NAT (0::nat))) & (((m One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & ((EX s. (((m S) = (NAT s)) & ((Soff <= s) & ((EX s'. (((callmem e S) = (NAT s')) & ((s = (s' + (1::nat))) & ((m s') = (RA (incA (callpc e))))))) & (EX n1. (((m C1) = (NAT n1)) & ((s + n1) <= MAX))))))) & (ALL Z. (((Z ~= R) &
((Z ~= S) & ((Z ~= C1) & ((Z ~= C3) & (EX s. (((m S) = (NAT s)) & ((Suc Z) < s))))))) --> ((m Z) = (callmem e Z)))))))))))) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). (pc = ((Suc (0::nat)), (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 (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 (Suc (0::nat)))))))))) s) --> ((%(pc, m, e). (tval-case False (%sa. (tval-case False (%ta. ((%s. (((%s. (((%(pc, m, e). (EX pn' i'. (((m (Suc (Suc (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 (Suc (Suc (0::nat))))))) < MAXMEM)) s) &
((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). (EX n3. (((m C3) = (NAT n3)) & ((EX n3'. (((callmem e C3) = (NAT n3')) & (EX n1'. (((callmem e C1) = (NAT n1')) & (EX n2'. (((callmem e C2) = (NAT n2')) & (n3 = (n3' + (n1' * n2'))))))))) & (((m Zero) = (NAT (0::nat))) & (((m One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & (((m R) = (RA (incA (callpc e)))) & ((EX s. (((m S) = (NAT s)) & ((Soff <= s) & (EX n1. (((m C1) = (NAT n1)) & ((s + n1) <= MAX)))))) & (ALL Z. (((Z ~= R) & ((Z ~= C1) & ((Z ~= C3) & (EX s. (((m S) = (NAT s)) & (Z < s)))))) -->
((m Z) = (callmem e Z))))))))))))) s) & ((%s. True) s))) s))) (((Suc (0::nat)), (Suc (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 (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 (Suc (0::nat))))))))))) s))) s) &
((%s. True) s))) s))) (((Suc (0::nat)), (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))), (update m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) (lift op − (m

(Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) (m (Suc (Suc (Suc (Suc (Suc (0::nat))))))))), (env.h-update ((env.h e) @ [((Suc (0::nat)), (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))]) e)))) s))) s) & ((%s. True) s))) s) & ((%s. (((%s. (((%s. (((%s. (((%s. (((%s. (((%(pc, m, e). (EX pn′ i′. (((m (Suc (Suc (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 (Suc (Suc (0::nat))))))) < MAXMEM)) s) & ((%s. True) s))) s))) s) &
((%s. (((%(pc, m, e). (EX n3. (((m C3) = (NAT n3)) & ((EX n3′. (((callmem e C3) = (NAT n3′)) & (EX n1′. (((callmem e C1) = (NAT n1′)) & (EX n2′. (((callmem e C2) = (NAT n2′)) & (n3 = (n3′ + (n1′ ∗ n2′))))))))) & (((m Zero) = (NAT (0::nat))) & (((m One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & (((m R) = (RA (incA (callpc e)))) & ((EX s. (((m S) = (NAT s)) & ((Soff <= s) & (EX n1. (((m C1) = (NAT n1)) & ((s + n1) <= MAX)))))) & (ALL Z. (((Z ~= R) & ((Z ~= C1) & ((Z ~= C3) & (EX s. (((m S) = (NAT s)) & (Z < s)))))) -->
((m Z) = (callmem e Z))))))))))))) s) & ((%s. True) s))) s))) s) & ((%s. (((%s. (((%(pc, m, e). (((m (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) = (RA ((0::nat), (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))))))) & (pc = ((Suc (0::nat)), (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))))))) s) & ((%s. (((%(pc, m, e). ((%(pc, m, e). (((m C1) = (NAT arg1)) & (((m C2) = (NAT arg2)) & (((m C3) = (NAT (0::nat))) & (((m Zero) = (NAT (0::nat))) & (((m One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & (((m S) = (NAT Soff)) & (ALL Z. ((Z ~: {C1, C2, C3, Zero, One, Ra, S}) -->
((m Z) = (callmem e Z)))))))))))) (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. (((%(pc, m, e). ((m C3) = (NAT (arg1 ∗ arg2)))) s) & ((%s. True) s))) s))) (((0::nat), (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. (((%s. (((%s. (((%s. (((%s. (((%(pc, m, e). (EX pn′ i′. (((m (Suc (Suc (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 (Suc (Suc (0::nat))))))) < MAXMEM)) s) & ((%s. True) s))) s))) s) & ((%s. (((%(pc, m, e). (EX n3. (((m C3) = (NAT n3)) & ((EX n3′. (((callmem e C3) = (NAT n3′)) & (EX n1′. (((callmem e C1) = (NAT n1′)) & (EX n2′. (((callmem e C2) = (NAT n2′)) &
(n3 = (n3′ + (n1′ ∗ n2′))))))))) & (((m Zero) = (NAT (0::nat))) & (((m One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & (((m R) = (RA (incA (callpc e)))) & ((EX s. (((m S) = (NAT s)) & ((Soff <= s) & (EX n1. (((m C1) = (NAT n1)) & ((s + n1) <= MAX)))))) & (ALL Z. (((Z ~= R) & ((Z ~= C1) & ((Z ~= C3) & (EX s. (((m S) = (NAT s)) & (Z < s)))))) --> ((m Z) = (callmem e Z))))))))))))) s) & ((%s. True) s))) s))) s) & ((%s. (((%s. (((%(pc, m, e). (((m (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) = (RA ((Suc (0::nat)), (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))) & (pc = ((Suc (0::nat)), (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))))))))) s) &

((%s. (((%(pc, m, e). ((%(pc, m, e). ((EX n1. (((m C1) = (NAT n1)) & (EX n2. (((m C2) = (NAT n2)) & (EX n3. (((m C3) = (NAT n3)) & (EX n1'. (((callmem e C1) = (NAT n1')) & (EX n2'. (((callmem e C2) = (NAT n2')) & (EX n3'. (((callmem e C3) = (NAT n3')) & (((0::nat) < n1') & ((n1 = (n1' − (1::nat))) & ((n2 = n2') & ((n3 = (n3' + n2')) & ((n3 + (n1 ∗ n2)) <= MAX)))))))))))))))) & (((m Zero) = (NAT (0::nat))) & (((m One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & ((EX s. (((m S) = (NAT s)) & ((Soff <= s) & ((EX s'. (((callmem e S) = (NAT s')) & ((s = (s' + (1::nat))) & ((m s') = (RA (incA (callpc e))))))) & (EX n1. (((m C1) = (NAT n1)) & ((s + n1) <= MAX))))))) & (ALL Z. (((Z ~= R) & ((Z ~= S) &
((Z ~= C1) & ((Z ~= C3) & (EX s. (((m S) = (NAT s)) & ((Suc Z) < s))))))) --> ((m Z) = (callmem e Z)))))))))) (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. (((%(pc, m, e). ((EX n. ((m (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat))))))))) = (NAT n))) & (EX n. ((m (Suc (Suc (Suc (Suc (Suc (0::nat)))))) = (NAT n))))) s) & ((%s. (((%(p, m, e). (((Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (0::nat)))))))) < MAXMEM) & ((Suc (Suc (Suc (Suc (Suc (0::nat)))))) < MAXMEM))) s) & ((%s. True) s))) s))) s) &
((%s. (((%(pc, m, e). (EX n3. (((m C3) = (NAT n3)) & ((EX n3'. (((callmem e C3) = (NAT n3')) & (EX n1'. (((callmem e C1) = (NAT n1')) & (EX n2'. (((callmem e C2) = (NAT n2')) & (n3 = (n3' + (n1' ∗ n2')))))))))) & (((m Zero) = (NAT (0::nat))) & (((m One) = (NAT (1::nat))) & (((m Ra) = (NAT R)) & ((EX s. (((m S) = (NAT s)) & ((Soff <= s) & ((EX s'. (((callmem e S) = (NAT s')) & ((s = (s' + (1::nat))) & ((m s') = (RA (incA (callpc e))))))) & (EX n1. (((m C1) = (NAT n1)) & ((s + n1) <= MAX))))))) & (ALL Z. (((Z ~= R) & ((Z ~= S) & ((Z ~= C1) & ((Z ~= C3) & (EX s. (((m S) = (NAT s)) & ((Suc Z) < s))))))) -->
((m Z) = (callmem e Z)))))))))))) s) & ((%s. True) s))) s))) (((Suc (0::nat)), (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) & ((%s. True) s))) s))) s))) s))) s))) s))) s)))

1.3 Verifying the program

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 *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*)

apply (*simp add*: *numeral-0-eq-0* [*THEN sym*] *del*: *numeral-0-eq-0 numeral-1-eq-1*)
apply (*simp only*: *numeral-0-eq-0 numeral-1-eq-1*)
apply (*simp only*: *suc-eq-plus1 env-simp*)

— main proof
apply (*case-tac prog*,((*pn*,*i*),*m*,*e*) ⊨ (*initF prog*))
apply (*rule conjI'*)
— (initF prg) implies (isafeF prg (ipc prg))
apply (*rule impI*)
apply (*erule conjE*)+
apply (*simp* (*no-asm-simp*) *add*: *initF-def valid-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def MAX-def MAXMEM-def update-def fun-upd-apply Let-def arg1-def arg2-def*)
apply (*rule allI*)
apply (*rule impI*)
apply (*simp add*: *update-def nth-append callmem-def callstate-def*)

apply (*simp only*: *initF-def valid-def split-def fst-conv snd-conv*)
apply (*erule conjE*)+
apply (*simp add*: *initF-def valid-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def MAX-def MAXMEM-def update-def fun-upd-apply del*:*Let-def*)

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

apply (*rule conjI*)
— initF - - ¿ isafe (0,0)
apply (*rule impI*)
apply (*erule conjE*)+
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply*)
apply (*rule allI*)
apply (*rule impI*)
apply (*erule conjE*)+

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

apply (*rule conjI*)
— (0,6) - - ¿ (1,0)
apply (*rule impI*)
apply (*erule conjE* | *erule exE*)+
apply (*simp only*: *MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def*)
apply (*rule conjI*)
apply (*rule-tac x=6* **in** *exI*)
apply (*simp add*: *update-def*)

apply (*rule conjI*)
apply (*rule-tac x=9* **in** *exI*)
apply (*simp add*: *update-def*)

apply (*rule conjI*)
apply (*rule allI*)
apply (*simp add*: *update-def*)

apply (*rule conjI*)
apply (*rule allI*)
apply (*simp add*: *update-def*)

apply (*rule conjI*)
apply (*rule-tac x=2* **in** *exI*)
apply (*simp add*: *update-def*)

apply (*simp add*: *update-def*)
apply (*rule conjI*)
apply (*simp add*: *callpc-def callstate-def nth-append incA-def*)

apply (*rule allI*)
apply (*rule impI*)
apply (*simp add*: *callmem-def*)

apply (*rule conjI*)
— (1,0) - - ¿ (1,5)

apply (*rule impI*)
apply (*erule conjE* | *erule exE*)+
apply (*simp add*: *numeral-0-eq-0* [*THEN sym*] *del*: *numeral-0-eq-0 numeral-1-eq-1*)
apply (*simp only*: *numeral-0-eq-0 numeral-1-eq-1*)
apply (*rule conjI*)
apply (*rule-tac x=n1* **in** *exI*)
apply (*simp add*: *MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply*)

apply (*rule conjI*)

apply (*rule-tac x=0* **in** *exI*)
apply (*simp add*: *MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply*)

apply (*rule conjI*)
apply (*simp add*:*MAXMEM-def*)

apply (*rule conjI*)
apply (*simp add*:*MAXMEM-def*)

apply (*rule conjI*)
apply (*rule impI*)
apply (*rule conjI'*)
apply (*case-tac css*)
apply (*simp add*: *Let-def split-def fst-conv snd-conv*)

— css = a list
apply (*simp add*:*Let-def split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def callmem-def callpc-def callenv-def incA-def*)
apply (*rule-tac x=fst* (*h e* ! *fst c*) **in** *exI*)
apply (*rule-tac x=snd* (*h e* ! *fst c*) **in** *exI*)
apply (*rule conjI*)
apply (*simp add*: *update-def fun-upd-apply*)

apply (*rule-tac x=fst c* **in** *exI*)
apply (*rule conjI'*)
apply (*rule-tac x=snd c* **in** *exI*)
apply *simp*

apply (*simp add*: *nth-append*)

apply (*rule conjI*)
apply (*simp add*: *MAXMEM-def*)

apply (*rule-tac x=n3* **in** *exI*)
apply (*rule conjI'*)
apply (*simp add*: *MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply*)

apply (*case-tac css*)
apply (*simp add*: *Let-def split-def fst-conv snd-conv*)

apply (*simp add*:*Let-def split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def callmem-def callpc-def callenv-def incA-def update-def*)
apply (*erule conjE* | *erule exE*)+
apply (*simp add*: *nth-append*)

apply (*case-tac css*)
apply (*simp add*: *Let-def split-def fst-conv snd-conv*)

apply (*rule impI*)
apply (*simp add*: *Let-def split-def fst-conv snd-conv*)
apply (*erule exE* | *erule conjE*)+

apply (*rule conjI'*)
apply (*rule-tac x=n* **in** *exI*)
apply *simp*

apply (*rule conjI*)
apply (*rule-tac x=Suc 0* **in** *exI*)
apply (*simp add*: *One-def S-def Soff-def update-def*)

apply (*rule conjI*)
apply (*simp add*: *MAXMEM-def*)

apply (*rule conjI*)
apply (*simp add*: *MAXMEM-def*)

apply (*rule conjI*)
apply (*rule-tac x=n3* **in** *exI*)
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)

apply (*rule conjI*)
apply (*rule-tac x=n2* **in** *exI*)
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)

apply (*rule conjI*)
apply (*rule-tac x=n3 + n2* **in** *exI*)
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)
apply (*subgoal-tac n2* $\leq$ *n* $*$ *n2*)
 prefer *2*
 apply (*rule nat-mult-mono*)
 apply *assumption*
apply *arith*

apply (*rule conjI*)

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

apply (*rule conjI*)
apply (*simp add*: *MAXMEM-def*)

apply (*rule conjI'*)
apply (*rule-tac x=s* **in** *exI*)
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)

apply (*rule conjI*)
apply (*simp add*: *MAXMEM-def*)

apply (*rule conjI*)
apply (*simp add*: *MAXMEM-def*)

— Problem: Stackueberlauf
apply (*rule conjI*)
apply (*rule-tac x=n1−(1::nat)* **in** *exI*)
apply (*rule conjI*)
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)

apply (*rule-tac x=n2* **in** *exI*)
apply (*rule conjI*)
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)

apply (*rule-tac x=n3+n2* **in** *exI*)
apply (*rule conjI*)
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)

apply (*rule-tac x=n1* **in** *exI*)
apply (*rule conjI*)
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)

apply (*rule-tac x=n2* **in** *exI*)

apply (*rule conjI*)
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)

apply (*rule-tac x=n3* **in** *exI*)
apply (*rule conjI*)
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)

apply (*rule conjI'*)
apply (*subgoal-tac* $n1 \neq 0$)
prefer *2*
apply (*subgoal-tac* $n=n1$)
prefer *2*
apply (*simp add*: *update-def MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def*)
apply (*subgoal-tac* $n'=0$)
prefer *2*
apply (*simp add*: *update-def MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def*)
apply *simp*
apply *simp*

apply *simp*
apply (*simp only*: *diff-mult-distrib*)
apply *simp*

apply (*rule conjI*)
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)

apply (*rule conjI*)
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)

apply (*rule conjI*)
apply (*simp add*: *split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)

apply (*rule conjI'*)
apply (*rule-tac x=s+(1::nat)* **in** *exI*)
apply (*simp add: split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)
apply *arith*

apply (*rule allI*)
apply (*rule impI*)
apply (*erule conjE* | *erule exE*)+
apply (*simp add: split-def fst-conv snd-conv MAX-def MAXMEM-def C1-def C2-def C3-def Zero-def One-def arg1-def arg2-def S-def Soff-def Ra-def R-def update-def fun-upd-apply incA-def callpc-def callmem-def callenv-def nth-append lift-def Let-def nth-append update-def*)

apply (*rule conjI*)
— (1,5) nach (1,0)
apply (*rule impI*)
apply (*erule conjE* | *erule exE*)+
apply (*rule conjI*)
apply (*rule-tac x=R* **in** *exI*)
apply (*simp add: R-def Ra-def update-def fun-upd-apply*)

apply (*rule conjI*)
apply (*rule-tac x=s* **in** *exI*)
apply (*simp* (*no-asm*) *add: S-def update-def fun-upd-apply*)
apply (*erule-tac P=m S = NAT s* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add: S-def*)

apply (*rule conjI*)
apply (*rule allI*)
apply (*simp add: update-def MAXMEM-def R-def Ra-def*)

apply (*rule conjI*)
apply (*rule allI*)
apply (*erule-tac P=s + n1a ≤ MAX* **in** *rev-mp*)
apply (*erule-tac P=m S = NAT s* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add: MAXMEM-def MAX-def S-def update-def*)

apply (*rule conjI*)
apply (*rule-tac x=n1* **in** *exI*)
apply (*rule conjI*)
apply (*erule-tac P=m C1 = NAT n1* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add: C1-def update-def*)

apply (*rule-tac x=n2* **in** *exI*)
apply (*rule conjI*)
apply (*erule-tac P=m C2 = NAT n2* **in** *rev-mp*)

apply (*simp* (*no-asm*) *add*: *C2-def update-def*)

apply (*rule-tac x=n3* **in** *exI*)
apply (*erule-tac P=m C3 = NAT n3* **in** *rev-mp*)
apply (*simp add*: *C3-def update-def*)

apply (*rule conjI*)
apply (*erule-tac P=m Zero = NAT 0* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *Zero-def update-def*)

apply (*rule conjI*)
apply (*erule-tac P=m One = NAT 1* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *One-def update-def*)

apply (*rule conjI*)
apply (*erule-tac P=m Ra = NAT R* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *Ra-def update-def*)

apply (*rule conjI*)
apply (*rule-tac x=s* **in** *exI*)
apply (*rule conjI*)
apply (*erule-tac P=m S = NAT s* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *S-def update-def*)

apply (*rule conjI*)
apply *assumption*

apply (*rule-tac x=n1a* **in** *exI*)
apply (*rule conjI*)
apply (*erule-tac P=m C1 = NAT n1a* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *C1-def update-def*)
apply *assumption*

apply (*rule conjI*)
apply (*simp* (*no-asm*) *add*: *R-def update-def incA-def split-def fst-conv snd-conv nth-append callpc-def*)

apply (*rule allI*)
apply (*simp* (*no-asm*) *add*: *R-def update-def callmem-def*)

apply (*rule conjI*)
— (1,6) nach (1,8)
apply (*rule impI*)
apply (*erule conjE* | *erule exE*)+

apply (*rule conjI*)
apply (*rule-tac x=s'a* **in** *exI*)
apply (*erule-tac P=m S = NAT s* **in** *rev-mp*)
apply (*erule-tac P=m One = NAT 1* **in** *rev-mp*)

apply (*erule-tac P=s = s'a + 1* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *S-def One-def lift-def update-def*)

apply (*rule conjI*)
apply (*rule-tac x=R* **in** *exI*)
apply (*erule-tac P=m Ra = NAT R* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *Ra-def update-def*)

apply (*subgoal-tac* ($m[8 \mapsto$ *lift op* $-$ (*m 8*) (*m 5*)]) *8 = NAT s'a*)
prefer *2*
apply (*erule-tac P=m S = NAT s* **in** *rev-mp*)
apply (*erule-tac P=s = s'a + 1* **in** *rev-mp*)
apply (*erule-tac P=m One = NAT 1* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *One-def S-def lift-def update-def*)
apply (*erule-tac P=*($m[8 \mapsto$ *lift op* $-$ (*m 8*) (*m 5*)]) *8 = NAT s'a* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *cases*)
apply (*rule impI*)
apply (*subgoal-tac* ($m[8 \mapsto$ *lift op* $-$ (*m 8*) (*m 5*)])
(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))))) = *NAT 6*)
prefer *2*
apply (*erule-tac P=m Ra = NAT R* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *Ra-def R-def update-def nat-number*)
apply (*erule-tac P=*($m[8 \mapsto$ *lift op* $-$ (*m 8*) (*m 5*)])(*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))))) = *NAT 6* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *cases*)
apply (*rule impI*)
apply (*rule conjI*)
apply (*erule-tac P=s + n1* $\leq$ *MAX* **in** *rev-mp*)
apply (*erule-tac P=s = s'a + 1* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *MAX-def MAXMEM-def*)

apply (*rule conjI*)
apply (*rule allI*)
apply (*erule-tac P=m Ra = NAT R* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *Ra-def R-def update-def MAXMEM-def*)

apply (*rule conjI*)
apply (*rule-tac x=fst* ($\overleftarrow{pc}$ *e*) **in** *exI*)
apply (*rule-tac x=snd* ($\overleftarrow{pc}$ *e*) **in** *exI*)
apply (*rule conjI*)
apply (*erule-tac P=m s'a = RA* (*incA* ($\overleftarrow{pc}$ *e*)) **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *nat-number update-def incA-def*)
apply (*rule conjI*)
apply (*rule impI*)
apply (*erule-tac P=m S = NAT s* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *S-def nat-number*)
apply (*rule impI*)+
apply (*simp* (*no-asm*) *add*: *split-def*)

apply (*case-tac css*)
apply (*simp add*: *Let-def split-def fst-conv snd-conv*)

apply (*simp add*: *Let-def split-def fst-conv snd-conv*)
apply (*erule conjE*)+
apply (*rule-tac x=fst c* **in** *exI*)
apply (*rule conjI*)
apply (*rule-tac x=snd c* **in** *exI*)
apply (*simp* (*no-asm*))

apply (*erule-tac P=fst c < length* (*h e*) **in** *rev-mp*)
apply (*erule-tac P=cs e = c # a # list* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *nth-append callpc-def*)

apply (*rule conjI*)
apply (*simp* (*no-asm*) *add*: *MAXMEM-def*)

apply (*rule-tac x=n3* **in** *exI*)
apply (*rule conjI*)
apply (*simp add*: *C3-def update-def*)

apply (*rule conjI*)
apply (*rule-tac x=n3'* **in** *exI*)
apply (*simp add*: *C3-def update-def callmem-def*)

apply (*rule conjI*)
apply (*simp add*: *Zero-def update-def*)

apply (*rule conjI*)
apply (*simp add*: *One-def update-def*)

apply (*rule conjI*)
apply (*simp add*: *Ra-def R-def update-def*)

apply (*rule conjI*)
apply (*simp add*: *R-def Ra-def incA-def callpc-def update-def*)
apply (*rule conjI*)
apply (*rule classical*)
apply (*simp add*: *S-def*)

apply (*rule impI*)
apply (*case-tac css*)
apply (*simp add*: *Let-def split-def fst-conv snd-conv*)

apply (*simp add*: *Let-def split-def nth-append*)

apply (*rule conjI*)
apply (*rule-tac x=s'a* **in** *exI*)
apply (*rule conjI*)

apply (*erule-tac P=m S = NAT s* **in** *rev-mp*)
apply (*erule-tac P=s = s'a + 1* **in** *rev-mp*)
apply (*erule-tac P=m One = NAT 1* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *S-def update-def One-def lift-def*)
apply (*rule conjI*)
— Idee: Wenn S=Soff=9, dann s'a=8, dann m 8 = NAT und m 8 = RA .. Typkonflikt
apply (*case-tac s = 9*)
apply (*subgoal-tac s'a = 8*)
prefer *2*
apply *arith*
apply (*erule-tac P=m S = NAT s* **in** *rev-mp*)
apply (*erule-tac P=m s'a = RA* (*incA* $\overleftarrow{pc}$ *e*) **in** *rev-mp*)
apply (*erule-tac P=s'a = 8* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *S-def*)

apply (*erule-tac P=Soff* $\leq$ *s* **in** *rev-mp*)
apply (*erule-tac P=s* $\neq$ *9* **in** *rev-mp*)
apply (*erule-tac P=s=s'a +1* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *S-def Soff-def*)

apply (*rule-tac x=n1* **in** *exI*)
apply (*simp add*: *C1-def update-def*)

apply (*rule allI*)
apply (*rule impI*)+

apply (*erule conjE* | *erule exE*)+
apply (*subgoal-tac sa=s'a*)
prefer *2*
apply (*erule-tac P=*(*m*[*8* $\mapsto$ *lift op* $-$ (*m 8*) (*m 5*)]) *8* = *NAT s'a* **in** *rev-mp*)
apply (*erule-tac P=*(*m*[*8* $\mapsto$ *lift op* $-$ (*m 8*) (*m 5*)][*6* $\mapsto$
(*m*[*8* $\mapsto$ *lift op* $-$ (*m 8*) (*m 5*)]) *s'a*])
S =
NAT sa **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *update-def S-def*)

apply (*erule-tac P=Z*$\neq$*R* **in** *rev-mp*)
apply (*case-tac Z = S*)
apply (*erule-tac P=Z = S* **in** *rev-mp*)
apply (*erule-tac P=*(*m*[*8* $\mapsto$ *lift op* $-$ (*m 8*) (*m 5*)]) *8* = *NAT s'a* **in** *rev-mp*)
apply (*erule-tac P=*$\overleftarrow{m}$ *e S = NAT s'a* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *update-def S-def callmem-def*)

apply (*erule-tac P=Z*$\neq$*S* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *S-def R-def update-def*)
apply (*rule impI*)+
apply (*erule-tac x=Z* **in** *allE*)
apply (*subgoal-tac* ($\exists$ *s. m S = NAT s* $\wedge$ *Z + 1* $<$ *s*))

prefer *2*
apply (*rule-tac x=s* **in** *exI*)
apply *simp*
apply (*simp add*: *S-def R-def callmem-def*)

apply (*rule conjI*)
— (1,8) nach (0,8)
apply (*rule impI*)
apply (*erule conjE* | *erule exE*)+
apply (*simp add*: *Let-def split-def fst-conv snd-conv*)
apply (*erule conjE*)+
apply (*simp add*: *callmem-def*)

apply (*rule impI*)
apply (*erule conjE* | *erule exE*)+
apply (*simp add*: *Let-def split-def fst-conv snd-conv*)
apply (*erule conjE* | *erule exE*)+
apply (*rule conjI*)
apply (*rule-tac x=s* **in** *exI*)
apply (*erule-tac P=m S = NAT s* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *S-def*)

apply (*rule conjI*)
apply (*rule-tac x=Suc 0* **in** *exI*)
apply (*erule-tac P=m One = NAT* (*Suc 0*) **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *One-def*)

apply (*rule conjI*)
apply (*simp* (*no-asm*) *add*: *MAXMEM-def*)

apply (*rule conjI*)
apply (*rule-tac x=n3'a* **in** *exI*)
apply (*rule conjI*)

apply (*erule-tac P=$\overleftarrow{m}$ e(|cs := (a, b) # cssa, h := take k (h e)|) C3 = NAT n3'a* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *callmem-def*)

apply (*rule-tac x=n1'a* **in** *exI*)
apply (*rule conjI*)
apply (*erule-tac P=$\overleftarrow{m}$ e(|cs := (a, b) # cssa, h := take k (h e)|) C1 = NAT n1'a* **in** *rev-mp*)
apply (*simp* (*no-asm*) *add*: *callmem-def*)

apply (*rule-tac x=n2'a* **in** *exI*)
apply (*simp add*: *callmem-def*)
apply (*drule-tac t=n3'* **in** *sym*)
apply (*drule-tac t=n1'* **in** *sym*)
apply (*simp add*: *diff-mult-distrib*)

```
apply(rule conjI)
apply (rule-tac x=s'a in exI)
apply (simp add: callmem-def callpc-def Let-def split-def fst-conv snd-conv)
apply (case-tac cssa)
apply (drule-tac s=(Suc 0,5) in sym)
apply (simp add: Let-def split-def fst-conv snd-conv)

apply (simp add: Let-def split-def fst-conv snd-conv nth-append)
apply (rule conjI')
apply (subgoal-tac m S = m' S)
 prefer 2
 apply (erule-tac x=S in allE)
 apply (simp add: S-def R-def C1-def C3-def Soff-def)
apply simp
apply (subgoal-tac m s'a = m' s'a)
 prefer 2
 apply (erule-tac x=s'a in allE)
 apply (simp add: Soff-def R-def C1-def C3-def)
apply simp

apply (rule allI)
apply (rule impI)
apply (erule conjE)+
apply (subgoal-tac s = Suc s'a)
 prefer 2
 apply (erule-tac x=S in allE)
 apply (subgoal-tac m S = m' S)
  prefer 2
  apply (subgoal-tac S ≠ R ∧ S ≠ C1 ∧ S ≠ C3 ∧ S < s)
   prefer 2
   apply (erule-tac P=Soff ≤ s in rev-mp)
   apply  (simp (no-asm) add: Soff-def S-def R-def C1-def C3-def Soff-def)
  apply (drule mp, assumption)
  apply (simp add: callmem-def)
 apply simp
apply (erule-tac x=Z in allE)
apply (erule-tac x=Z in allE)
apply (subgoal-tac m Z = m⃖ e Z)
 prefer 2
 apply simp
apply (subgoal-tac m⃖ e Z = b Z)
 prefer 2
 apply (simp add: callmem-def)
apply (simp add: callmem-def)
done
```

end