theory *EX-SmartCardPurse = SALOverflowFWInst + AuxBox*:

1 Smart Card Purse

This program adds a credit C to a balance B if the new balance B + C does not exceeed an upper bound MAX (the highest number the safety policy accepts). To check this condition a procedure is called which sets a flag F to NAT 0 if this condition is violated

— program variables
constdefs
B :: *nat* — balance
B ≡ *0*
C :: *nat* — credit
C ≡ *1*
M :: *nat* — maximum
M ≡ *2*
P :: *nat* — return address storage
P ≡ *3*

— initial values
constdefs
b0 :: *nat*
b0 ≡ *4* — maximal intial balance
c0 :: *nat*
c0 ≡ *2* — initial credit

constdefs
prog :: *SALprogram*
prog ≡
[(*0*, [(*SET B b0*, *None*),
(*SET C c0*, *None*),
(*CALL P 1*, *Some* (λ (*pc*,*m*,*e*).*m B* = *NAT b0* ∧ *m C* = *NAT c0*)),
(*ADD B C*, *Some* (λ (*pc*,*m*,*e*). *m B* = *NAT b0* ∧
(∃ *c*. *m C* = *NAT c* ∧
c = (*if* (*b0* + *c0* < *MAX*)
then c0
else 0)
))),
(*HALT*, *None*)
]
),
(*1*,[(*SET M MAX*, *Some* (λ (*pc*,*m*,*e*). *m P* = *RA* (*incA* (*pc* e))
∧ (∃ *b*. *m B* = *NAT b*)
∧ (∃ *c*. *m C* = *NAT c*)
∧ (∀ *x*. *x* ≠ *P* ⟶
m x = (*m* e) *x*)
)),

$$
\begin{array}{l}
\quad (SUB\ M\ C,\ None),\\
\quad (JMPL\ B\ M\ 2,\ None),\\
\quad (SET\ C\ 0,\ None),\\
\quad (RET\ P,\ Some\ (\lambda\ (pc,m,e).\ (\forall\ x.\ x{\neq}C \wedge x \neq M \wedge x \neq P\\
\qquad\qquad\qquad\qquad \longrightarrow m\ x = (\overleftarrow{m}\ e)\ x)\\
\qquad\qquad\qquad \wedge\ (\exists\ b\ c\ c'.\ m\ B = NAT\ b\\
\qquad\qquad\qquad\quad \wedge\ m\ C = NAT\ c\\
\qquad\qquad\qquad\quad \wedge\ (\overleftarrow{m}\ e)\ C = NAT\ c'\\
\qquad\qquad\qquad\quad \wedge\ c = (if\ (b + c' < MAX)\\
\qquad\qquad\qquad\qquad\quad then\ c'\\
\qquad\qquad\qquad\qquad\quad else\ 0)\\
\qquad\qquad\qquad)))\\
\quad]\\
)\\
]
$$

1.1 The Verification Condition

Generate an ML file vcgSAL.ML that contains an executable VCG and the program source.

generate-code (*vcgSAL.ML*) [*term-of*]
  vcg = *vcgSAL*
  prg = *prog*

This is the output our ML program for the VCG yield for this example (copy and paste).

constdefs
  vc :: *SALform*
  vc ≡ (%*s*. (((%*s*. (((%(*pc*, *m*, *e*). ((*pc* = ((*0*::*nat*), (*0*::*nat*))) & (((*cs e*) = [((*0*::*nat*), *m*)]) & (((*h e*) = []) & (*ALL X*. ((*m X*) = *ILLEGAL*)))))) *s*) −−> ((%*s*. (((%(*pc*, *m*, *e*). ((*Suc* (*Suc* (*Suc* (*Suc* (*0*::*nat*))))) <= *MAX*)) *s*) & ((%*s*. (((%*s*. (((%(*pc*, *m*, *e*). (*pc* = ((*0*::*nat*), (*0*::*nat*)))) *s*) −−> ((%(*pc*, *m*, *e*). ((%*s*. (((%(*pc*, *m*, *e*). ((*Suc* (*Suc* (*0*::*nat*))) <= *MAX*)) *s*) & ((%*s*. (((%*s*. (((%(*pc*, *m*, *e*). (*pc* = ((*0*::*nat*), (*Suc* (*0*::*nat*))))) *s*) −−> ((%(*pc*, *m*, *e*). ((%*s*. (((%*s*. *True*) *s*) & ((%*s*. (((%(*pc*, *m*, *e*). (((*m B*) =
(*NAT b0*)) & ((*m C*) = (*NAT c0*)))) *s*) & ((%*s*. *True*) *s*))) *s*))) (((*0*::*nat*), (*Suc* (*Suc* (*0*::*nat*)))), (*update m* (*Suc* (*0*::*nat*)) (*NAT* (*Suc* (*Suc* (*0*::*nat*))))), (*h-update* ((*h e*) @ [((*0*::*nat*), (*Suc* (*0*::*nat*)))]) *e*)))) *s*))) *s*) & ((%*s*. *True*) *s*))) *s*))) (((*0*::*nat*), (*Suc* (*0*::*nat*))), (*update m* (*0*::*nat*) (*NAT* (*Suc* (*Suc* (*Suc* (*Suc* (*0*::*nat*)))))))), (*h-update* ((*h e*) @ [((*0*::*nat*), (*0*::*nat*))]) *e*)))) *s*))) *s*) & ((%*s*. *True*) *s*))) *s*))) *s*))) *s*) & ((%*s*. (((%*s*. (((%*s*. (((%*s*. (((%*s*. (((%*s*. *True*) *s*) &
((%*s*. (((%(*pc*, *m*, *e*). (((*m B*) = (*NAT b0*)) & ((*m C*) = (*NAT c0*)))) *s*) & ((%*s*. *True*) *s*))) *s*))) *s*) & ((%*s*. (((%(*pc*, *m*, *e*). (*pc* = ((*0*::*nat*), (*Suc* (*Suc* (*0*::*nat*)))))) *s*) & ((%*s*. *True*) *s*))) *s*))) *s*) −−> ((%(*pc*, *m*, *e*). ((%*s*. (((%(*pc*, *m*, *e*). ((*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*0*::*nat*))))))))))))))) <= *MAX*)) *s*) & ((%*s*. (((%(*pc*, *m*, *e*). (((*m P*) = (*RA* (*incA* (*callpc e*)))) & ((*EX b*. ((*m B*) = (*NAT b*))) & ((*EX c*. ((*m C*) = (*NAT*

$c))) \ \&\ (ALL\ x.\ ((x\ \sim=\ P)\ -->$
$((m\ x) = (callmem\ e\ x))))))))\ s)\ \&\ ((\%s.\ True)\ s)))\ s)))\ (((Suc\ (0::nat)),\ (0::nat)),$
$(update\ m\ (Suc\ (Suc\ (Suc\ (0::nat))))\ (RA\ ((0::nat),\ (Suc\ (Suc\ (Suc\ (0::nat)))))))),$
$(cs\text{-}update\ (((length\ (h\ e)),\ m)\ \#\ (cs\ e))\ (h\text{-}update\ ((h\ e)\ @\ [((0::nat),\ (Suc\ (Suc$
$(0::nat))))])\ e)))))\ s)))\ s)\ \&\ ((\%s.\ True)\ s)))\ s)\ \&\ ((\%s.\ (((\%s.\ (((\%s.\ (((\%s.$
$(((\%s.\ (((\%(pc,\ m,\ e).\ ((EX\ n.\ ((m\ (0::nat))\ =\ (NAT\ n)))\ \&\ ((EX\ n.\ ((m\ (Suc$
$(0::nat)))\ =\ (NAT\ n)))\ \&$
$(EX\ n.\ (((lift\ op\ +\ (m\ (0::nat))\ (m\ (Suc\ (0::nat))))\ =\ (NAT\ n))\ \&\ (n\ <=$
$MAX))))))\ s)\ \&\ ((\%s.\ (((\%(pc,\ m,\ e).\ (((m\ B)\ =\ (NAT\ b0))\ \&\ (EX\ c.\ (((m$
$C)\ =\ (NAT\ c))\ \&\ (c\ =\ (if\ ((b0\ +\ c0)\ <\ MAX)\ then\ c0\ else\ (0::nat)))))))\ s)\ \&$
$((\%s.\ True)\ s)))\ s)))\ s)\ \&\ ((\%s.\ (((\%(pc,\ m,\ e).\ (pc\ =\ ((0::nat),\ (Suc\ (Suc\ (Suc$
$(0::nat)))))))\ s)\ \&\ ((\%s.\ True)\ s)))\ s)))\ s)\ -->\ ((\%(pc,\ m,\ e).\ ((\%s.\ (((\%s.\ True)$
$s)\ \&\ ((\%s.\ True)\ s)))\ (((0::nat),\ (Suc\ (Suc\ (Suc\ (Suc\ (0::nat))))))),\ (update\ m$
$(0::nat)\ (lift\ op\ +\ (m\ (0::nat))\ (m\ (Suc\ (0::nat))))),\ (h\text{-}update\ ((h\ e)\ @\ [((0::nat),$
$(Suc\ (Suc\ (Suc\ (0::nat))))])\ e))))\ s)))\ s)$
$\&\ ((\%s.\ True)\ s)))\ s)\ \&\ ((\%s.\ (((\%s.\ (((\%s.\ (((\%s.\ (((\%(pc,\ m,\ e).$
$((Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc\ (Suc$
$(0::nat))))))))))))))) <= MAX))\ s)\ \&\ ((\%s.\ (((\%(pc,\ m,\ e).\ (((m\ P)\ =\ (RA\ (incA$
$(callpc\ e))))\ \&\ ((EX\ b.\ ((m\ B)\ =\ (NAT\ b)))\ \&\ ((EX\ c.\ ((m\ C)\ =\ (NAT\ c)))\ \&$
$(ALL\ x.\ ((x\ \sim=\ P)\ -->\ ((m\ x)\ =\ (callmem\ e\ x)))))))\ s)\ \&\ ((\%s.\ True)\ s)))$
$s)))\ s)\ \&$
$((\%s.\ (((\%(pc,\ m,\ e).\ (pc\ =\ ((Suc\ (0::nat)),\ (0::nat))))\ s)\ \&\ ((\%s.\ True)\ s)))\ s)))$
$s)\ -->\ ((\%(pc,\ m,\ e).\ ((\%s.\ (((\%(pc,\ m,\ e).\ ((EX\ n.\ ((m\ (Suc\ (Suc\ (0::nat))))$
$=\ (NAT\ n)))\ \&\ (EX\ n.\ ((m\ (Suc\ (0::nat)))\ =\ (NAT\ n)))))\ s)\ \&\ ((\%s.\ (((\%s.$
$(((\%(pc,\ m,\ e).\ (pc\ =\ ((Suc\ (0::nat)),\ (Suc\ (0::nat)))))\ s)\ -->\ ((\%(pc,\ m,\ e).$
$((\%s.\ (((\%(pc,\ m,\ e).\ ((EX\ n.\ ((m\ (0::nat))\ =\ (NAT\ n)))\ \&\ (EX\ n.\ ((m\ (Suc$
$(Suc\ (0::nat))))\ =\ (NAT\ n)))))\ s)\ \&\ ((\%s.\ (((\%s.\ (((\%(pc,\ m,\ e).\ ((EX\ n\ n'.\ (((m$
$(0::nat))\ =\ (NAT\ n))\ \&\ (((m\ (Suc\ (Suc\ (0::nat))))\ =\ (NAT\ n'))\ \&\ (n\ <\ n'))))$
$\&\ (pc\ =\ ((Suc\ (0::nat)),\ (Suc\ (Suc\ (0::nat)))))))\ s)\ -->\ ((\%(pc,\ m,\ e).\ ((\%s.$
$(((\%(pc,\ m,\ e).\ (EX\ pn'\ i'.\ (((m\ (Suc\ (Suc\ (Suc\ (0::nat)))))\ =\ (RA\ (pn',\ (Suc$
$i'))))\ \&\ (EX\ k\ m'\ cl\ css.\ (((cs\ e)\ =\ ((k,\ m')\ \#\ (cl\ \#\ css)))\ \&\ ((pn',\ i')\ =$
$((h\ e)\ !\ k)))))))\ s)\ \&\ ((\%s.\ (((\%(pc,\ m,\ e).\ ((ALL\ x.\ (((x\ \sim=\ C)\ \&\ ((x\ \sim=\ M)$
$\&\ (x\ \sim=\ P)))\ -->\ ((m\ x)\ =\ (callmem\ e\ x))))\ \&\ (EX\ b\ c\ c'.\ (((m\ B)\ =\ (NAT$
$b))\ \&\ (((m\ C)\ =\ (NAT\ c))\ \&\ (((callmem\ e\ C)\ =\ (NAT\ c'))\ \&\ (c\ =\ (if\ ((b\ +\ c')$
$<\ MAX)\ then\ c'\ else\ (0::nat)))))))))\ s)\ \&\ ((\%s.\ True)\ s)))\ s)))\ (((Suc\ (0::nat)),$
$(Suc\ (Suc\ (Suc\ (Suc\ (0::nat))))))),\ m,\ (h\text{-}update\ ((h\ e)\ @\ [((Suc\ (0::nat)),\ (Suc$
$(Suc\ (0::nat))))])\ e))))\ s)))\ s)\ \&\ ((\%s.\ (((\%s.\ (((\%(pc,\ m,\ e).\ ((EX\ n\ n'.\ (((m$
$(0::nat))\ =\ (NAT\ n))\ \&\ (((m\ (Suc\ (Suc\ (0::nat))))\ =\ (NAT\ n'))\ \&\ (\sim\ (n\ <$
$n')))))\ \&$
$(pc\ =\ ((Suc\ (0::nat)),\ (Suc\ (Suc\ (0::nat)))))))\ s)\ -->\ ((\%(pc,\ m,\ e).\ ((\%s.$
$(((\%(pc,\ m,\ e).\ ((0::nat)\ <=\ MAX))\ s)\ \&\ ((\%s.\ (((\%s.\ (((\%(pc,\ m,\ e).\ (pc\ =\ ((Suc$
$(0::nat)),\ (Suc\ (Suc\ (Suc\ (0::nat)))))))\ s)\ -->\ ((\%(pc,\ m,\ e).\ ((\%s.\ (((\%(pc,\ m,$
$e).\ (EX\ pn'\ i'.\ (((m\ (Suc\ (Suc\ (Suc\ (0::nat)))))\ =\ (RA\ (pn',\ (Suc\ i'))))\ \&\ (EX$
$k\ m'\ cl\ css.\ (((cs\ e)\ =\ ((k,\ m')\ \#\ (cl\ \#\ css)))\ \&\ ((pn',\ i')\ =\ ((h\ e)\ !\ k)))))))\ s)$
$\&\ ((\%s.\ (((\%(pc,\ m,\ e).\ ((ALL\ x.\ (((x\ \sim=\ C)\ \&\ ((x\ \sim=\ M)\ \&\ (x\ \sim=\ P)))\ -->$
$((m\ x)\ =\ (callmem\ e\ x))))\ \&\ (EX\ b\ c\ c'.\ (((m\ B)\ =\ (NAT\ b))\ \&\ (((m\ C)\ =\ (NAT$
$c))\ \&\ (((callmem\ e\ C)\ =\ (NAT\ c'))\ \&\ (c\ =\ (if\ ((b\ +\ c')\ <\ MAX)\ then\ c'\ else$
$(0::nat)))))))))\ s)\ \&\ ((\%s.\ True)\ s)))\ s)))\ (((Suc\ (0::nat)),\ (Suc\ (Suc\ (Suc\ (Suc$
$(0::nat))))))),$

(*update m (Suc (0::nat)) (NAT (0::nat))*), (*h-update ((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))))), m, (h-update ((h e) @ [((Suc (0::nat)), (Suc (Suc
(0::nat))))]) e)))) s))) s) & ((%s. True) s))) s))) s))) (((Suc (0::nat)), (Suc (Suc
(0::nat)))), (update m (Suc (Suc (0::nat))) (lift op − (m (Suc (Suc (0::nat))))
(m (Suc (0::nat))))), (h-update ((h e) @ [((Suc (0::nat)), (Suc (0::nat)))]) e))))
s))) s) & ((%s. True) s))) s))) (((Suc (0::nat)), (Suc (0::nat))), (update m (Suc
(Suc (0::nat))) (NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (0::nat))))))))))))))))), (h-update ((h e) @ [((Suc (0::nat)),
(0::nat))]) e)))) s))) s) & ((%s. True) s))) s) &
((%s. (((%s. (((%s. (((%s. (((%s. (((%(pc, m, e). (EX pn′ i′. (((m (Suc (Suc (Suc
(0::nat))))) = (RA (pn′, (Suc i′)))) & (EX k m′ cl css. (((cs e) = ((k, m′) # (cl
css))) & ((pn′, i′) = ((h e) ! k))))))) s) & ((%s. (((%(pc, m, e). ((ALL x. (((x
~= C) & ((x ~= M) & (x ~= P))) −−> ((m x) = (callmem e x)))) & (EX b c
c′. (((m B) = (NAT b)) & (((m C) = (NAT c)) & (((callmem e C) = (NAT c′))
& (c = (if ((b + c′) < MAX) then c′ else (0::nat)))))))) s) & ((%s. True) s)))
s))) s) & ((%s. (((%s. (((%(pc, m, e). (((m (Suc (Suc (Suc (0::nat))))) = (RA
((0::nat),
(Suc (Suc (Suc (0::nat))))))) & (pc = ((Suc (0::nat)), (Suc (Suc (Suc (Suc (0::nat)))))))))
s) & ((%s. (((%(pc, m, e). ((%(pc, m, e). (((m B) = (NAT b0)) & ((m C) = (NAT
c0)))) (let (k, m′) = (hd (cs e)); cs′ = (tl (cs e)); h′ = (take k (h e)) in (((h e)
! k), m′, (h-update h′ (cs-update cs′ e))))))) s) & ((%s. True) s))) s))) s) & ((%s.
True) s))) s))) s) −−> ((%(pc, m, e). ((%s. (((%(pc, m, e). ((EX n. ((m (0::nat))
= (NAT n))) & ((EX n. ((m (Suc (0::nat))) =
(NAT n))) & (EX n. (((lift op + (m (0::nat)) (m (Suc (0::nat)))) = (NAT n))
& (n <= MAX)))))) s) & ((%s. (((%(pc, m, e). (((m B) = (NAT b0)) & (EX c.
(((m C) = (NAT c)) & (c = (if ((b0 + c0) < MAX) then c0 else (0::nat))))))))
s) & ((%s. True) s))) s))) (((0::nat), (Suc (Suc (Suc (0::nat))))), m, (cs-update
(tl (cs e)) (h-update ((h e) @ [((Suc (0::nat)), (Suc (Suc (Suc (Suc (0::nat))))))])
e))))) s))) s) & ((%s. True) s))) s) & ((%s. True) s))) s))) s))) s))) s)))

lemma *domC-prog*:*domC prog* = [(*0*,*0*),(*0*,*1*),(*0*,*2*),(*0*,*3*),(*0*,*4*),(*1*,*0*),(*1*,*1*),(*1*,*2*),(*1*,*3*),(*1*,*4*)]
apply (*simp add*: *domC.simps prog-def Let-def fst-conv snd-conv*)
done

lemma *anF-prog*:*domA prog* = [(*0*,*2*),(*0*,*3*),(*1*,*0*),(*1*,*4*)]
apply (*simp only*: *domA-def domC-prog*)
apply (*simp add*: *anF.simps prog-def noProc.simps Let-def fst-conv snd-conv split-def filter.simps*)
done

1.2 Program Verification

First we ensure that the program is wellformed.

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

Then, we prove the verification condition

lemmas *vc-simps* [*simp*] = *nat-number split-def Let-def fst-conv snd-conv B-def MAX-def b0-def C-def c0-def M-def P-def update-def fun-upd-apply callpc-def callmem-def callenv-def incA-def nth-append lift-def*

Automated Proof

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

— main proof
apply *auto*
apply *arith*
apply *arith*
done

Manually Guided Proof

declare *vc-simps* [*simp del*]

lemma *vc-prog-holds2*:
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 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*) ⊨ (*initF prog*))
apply (*simp add*: *initF-def valid-def B-def MAX-def b0-def C-def c0-def update-def fun-upd-apply*)
— case: not initF
apply (*erule isafeP-elims*)
apply (*simp only*: *simp-thms fst-conv snd-conv*)
— now we know that ((pn,i),m,e) is inductively safe
apply (*drule-tac t*=*s''* **in** *sym*)

apply (*rule conjI*)
— (initF prg) implies (isafeF prg (ipc prg))
apply (*simp add*: *MAX-def c0-def b0-def B-def C-def update-def fun-upd-apply*)

apply (*rule conjI*)
— (0,2) nach (1,0), prozedureinsprung
apply (*simp add*: *MAX-def c0-def b0-def B-def C-def P-def update-def fun-upd-apply callpc-def callmem-def nth-append incA-def fst-conv snd-conv*)

apply (*rule conjI*)
— (0,3) nach (0,4), programmende
apply (*simp add*: *fst-def*)

apply (*rule conjI*)
— (1,0) nach (1,4), prozedurbody, zwei wege
apply (*simp add*: *fst-conv snd-conv MAX-def c0-def b0-def B-def C-def P-def M-def update-def fun-upd-apply*)
apply (*rule impI*)+
apply (*erule conjE* | *erule exE*)+
apply (*case-tac css*)
apply (*simp add*: *Let-def split-def fst-conv snd-conv*)

— case: css = a list
apply (*simp add*: *Let-def split-def fst-conv snd-conv lift-def callpc-def del*: *sys-inv.simps sysinv2.simps*)
apply (*rule conjI*)
— Fall: Sprungbedingung gilt

apply (*rule impI*)
apply (*rule conjI*)
apply (*rule-tac x=fst* (*h e* ! *fst c*) **in** *exI*)
apply (*rule-tac x=snd* (*h e* ! *fst c*) **in** *exI*)
apply (*simp add*: *incA-def split-def nat-number Let-def fst-conv snd-conv*)
apply (*rule-tac x=fst c* **in** *exI*)
apply (*erule conjE*)+
apply (*simp add*: *nth-append*)
apply (*rule-tac x=snd c* **in** *exI*)
apply *simp*

apply (*rule conjI*)
apply (*rule allI*)
apply (*rule impI*)
apply (*erule-tac x=x* **in** *allE*)
apply (*simp add*: *callmem-def*)

apply (*rule-tac x=ca* **in** *exI*)
apply (*simp add*: *callmem-def*)
apply (*subgoal-tac* $(b + ca < MAX) = (b < MAX - ca)$)

prefer *2*
apply (*rule sym*)
apply (*rule less-diff-conv*)
apply (*simp add*: *nat-number MAX-def*)

— Sprungbedingung gilt nicht

apply (*rule impI*)
apply (*rule conjI*)
apply (*rule-tac x=fst* (*h e* ! *fst c*) **in** *exI*)
apply (*rule-tac x=snd* (*h e* ! *fst c*) **in** *exI*)
apply (*simp add*: *incA-def split-def nat-number Let-def fst-conv snd-conv*)
apply (*rule-tac x=fst c* **in** *exI*)
apply (*erule conjE*)+
apply (*simp add*: *nth-append*)
apply (*rule-tac x=snd c* **in** *exI*)
apply *simp*

apply (*rule conjI*)
apply (*rule allI*)
apply (*rule impI*)
apply (*erule-tac x=x* **in** *allE*)
apply (*simp add*: *callmem-def*)

apply (*rule-tac x=ca* **in** *exI*)
apply (*simp add*: *callmem-def*)
apply (*subgoal-tac* (*b* + *ca* < *MAX*) = (*b* < *MAX* − *ca*))
prefer *2*
apply (*rule sym*)
apply (*rule less-diff-conv*)
apply (*simp add*: *nat-number MAX-def*)

— (1,4) nach (0,3); prozeduraussprung
apply (*simp add*: *MAX-def c0-def b0-def B-def C-def P-def M-def update-def fun-upd-apply*)
apply (*rule impI*)+
apply (*erule conjE* | *erule exE*)+
apply (*simp add*: *Let-def split-def fst-conv snd-conv lift-def callmem-def*)
— nprfsize() = 245.170
done

end