theory *EX-SmartCardPurse-deep* = *VCOpt* + *AuxBox*:

1 Smart Card Purse

This program adds a credit C to a balance B if the new balance B + C does not execeed 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
$b \equiv 0$
 c :: *nat* — credit
$c \equiv 1$
 m :: *nat* — maximum
$m \equiv 2$
 p :: *nat* — return address storage
$p \equiv 3$

— initial values
constdefs
 $b0$ :: *nat*
$b0 \equiv 4$ — maximal intial balance
 $c0$ :: *nat*
$c0 \equiv 2$ — initial credit
x::*nat*
$x \equiv 5$

constdefs *prog*::*SALprogram*
prog ≡
$[(0,\ [(SET\ b\ b0,\ None),$
$\quad (SET\ c\ c0,\ None),$
$\quad (CALL\ p\ 1,\ Some\ (\Lambda\ [V\ b \doteq C\ (NAT\ b0),\ V\ c \doteq C\ (NAT\ c0)])),$
$\quad (ADD\ b\ c\ ,\ Some\ (\Lambda\ [V\ b \doteq C\ (NAT\ b0),\ Ty\ (V\ c)\ Nat,$
$\qquad (C\ (NAT\ 0) \prec V\ c) \supset$
$\qquad \Lambda\ [(V\ b \oplus V\ c) \preceq C\ (NAT\ MAX),\ V\ c \doteq C\ (NAT\ c0)]])),$
$\quad (HALT,\ None)]),$
$(1,\ [(SET\ m\ MAX,\ Some\ (\Lambda\ [V\ p \doteq Rp,\ Ty\ (V\ b)\ Nat,\ Ty\ (V\ c)\ Nat,$
$\qquad \prod x.\ Neg\ (Eq\ (Lv\ x)\ (C\ (NAT\ p))) \supset (Deref\ (Lv\ x) \doteq Old$
$(Deref\ (Lv\ x)))])),$
$\quad (SUB\ m\ c,\ None),$
$\quad (JMPL\ b\ m\ 2,\ None),$
$\quad (SET\ c\ 0,\ None),$
$\quad (RET\ p,\ Some\ (\Lambda\ [(\prod x.\ (\Lambda\ [Neg\ ((Lv\ x) \doteq (C\ (NAT\ c))),\ Neg\ ((Lv\ x) \doteq (C$
$(NAT\ m))),$
$\qquad Neg\ ((Lv\ x) \doteq C\ (NAT\ p))]) \supset$
$\qquad (Deref\ (Lv\ x) \doteq Old\ (Deref\ (Lv\ x)))),$
$\qquad (Neg\ (V\ c \doteq C\ (NAT\ 0)) \supset (\Lambda\ [V\ c \doteq Old\ (V\ c),$

$(V\ b \oplus V\ c) \preceq C\ (NAT\ MAX)]))]))])]$

1.1 The Verification Condition

generate-code [*term-of*]
vcg = *vcgSALDeep*
vcopt = *vcopt*
prg = *prog*

ML {∗ *set show-brackets*; ∗}

ML {∗ *val vc* = *vcg prog*; ∗}

ML {∗ *File.write* (*Path.unpack pvc.txt*) (*Pretty.str-of* (*Sign.pretty-term* (*sign-of* (*the-context* ())) (*term-of-form vc*))); ∗}

ML {∗ *val vco* = *vcopt* [] *vc*; ∗}

ML {∗ *File.write* (*Path.unpack opvc.txt*) (*Pretty.str-of* (*Sign.pretty-term* (*sign-of* (*the-context* ())) (*term-of-form vco*))); ∗}

ML {∗ *val pvc* = (*Pretty.str-of* (*Sign.pretty-term* (*sign-of* (*the-context* ())) (*term-of-form vco*))); ∗}

ML {∗ *reset show-brackets* ∗}

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

1.2.1 Automated Proof

lemma *vc-prog-holds*:
provable prog vc
apply (*simp only*: *provable-def valid-def split-paired-all* | *rule HOLprf* | *rule ballI* | *rule allI* | *rule impI*)+
apply (*rename-tac pn i m e I*)
apply (*cut-tac wf-prog*, *drule vc-proof-startup*,*assumption*, (*erule conjE* | *erule exE*)+, (*simp only*: *fst-conv snd-conv*))

— Now, the prelude is finished. The main proof starts . . .

apply (*simp only*: *vc-def*)
apply (*auto simp add*: *vc-simps*)
apply (*simp add*: *vc-simps split add*: *tval.split tval.split-asm*)

apply (*simp add*: *vc-simps split add*: *nat.split*)

apply (*fastsimp simp add*: *vc-simps*)

apply (*simp add*: *vc-simps split add*: *tval.split tval.split-asm*)
apply *arith*

apply (*simp split add*: *nat.split*)
— nprfsize = 51.204
done

1.2.2 Manual Proof

— Lemmas about validity of formulae
lemma *validF-True*: $validF\ I\ s\ T = True$
by *simp*

lemma *validF-False*: $validF\ I\ s\ F = False$
by *simp*

lemma *validF-And-Nil*: $validF\ I\ s\ (\Lambda\ []) = True$
by *simp*

lemma *validF-And-Cons*: $validF\ I\ s\ (\Lambda\ (f\ \#\ fs)) = ((validF\ I\ s\ f) \wedge (validF\ I\ s\ (\Lambda\ fs)))$
by *simp*

lemma *validF-And-Cons-Nil*: $valid'\ (?s,\ (\Lambda\ [f])) = (valid'\ (?s,\ ?f))$
by *simp*

lemma *validF-Imp*: $validF\ I\ s\ (f1 \supset f2) = ((validF\ I\ s\ f1) \longrightarrow (validF\ I\ s\ f2))$
by *simp*

lemma *validF-Neg*: $validF\ I\ s\ (Neg\ f) = (\neg\ (validF\ I\ s\ f))$
by *simp*

lemma *validF-Eq*: $validF\ I\ s\ (e1 \doteq e2) = ((eval\ I\ s\ e1) = (eval\ I\ s\ e2))$
by *simp*

lemma *validF-Leq*: $validF\ I\ s\ (e1 \preceq e2) = ((nv\ (eval\ I\ s\ e1)) \leq (nv\ (eval\ I\ s\ e2)))$
by *simp*

lemma *validF-Less*: $validF\ I\ s\ (e1 \prec e2) = ((nv\ (eval\ I\ s\ e1)) < (nv\ (eval\ I\ s\ e2)))$

by *simp*

lemma *validF-Forall*: *validF I s* $(\prod v.\ f) = (\forall\, tv.\ (validF\ (I[v{::=}tv])\ s\ f))$
by (*simp add*: *Let-def*)

lemma *length-cons*:
length (*a#as*) = *Suc* (*length as*)
by *simp*

lemma *vc-prog-holds2*:
provable prog vc
apply (*simp only*: *provable-def valid-def split-paired-all* | *rule HOLprf* | *rule ballI* | *rule allI* | *rule impI*)+
apply (*rename-tac pn i m e I*)
apply (*cut-tac wf-prog*, *drule vc-proof-startup*,*assumption*, (*erule conjE* | *erule exE*)+, (*simp only*: *fst-conv snd-conv*))
apply (*simp add*: *vc-def validF-validFs.simps del*: *sysinv.simps sysinv2.simps*)

— Now, the prelude is finished. The main proof starts . . .
apply (*case-tac prog*,((*pn*,*i*),*m*,*e*) ⊨ (*initF prog*))
— (initF prg) implies (isafeF prg (ipc prg))
apply (*simp add*: *initF-def valid-def b-def MAX-def b0-def c-def c0-def update-def fun-upd-apply nv-def*)
— case not (initF prg)
apply (*erule isafeP-elims*)
apply (*simp only*: *simp-thms*)
apply (*drule-tac* $t{=}s''$ **in** *sym*)
apply (*simp only*: *fst-conv snd-conv*)
— now we know that ((pn,i),m,e) is inductively safe

apply (*rule conjI*)
— initF impliziert isafeF (0,0)
apply (*simp add*: *validF-validFs.simps Let-def split-def fst-conv snd-conv nv-def*)

apply (*simp only*: *Let-def split-def fst-conv snd-conv*)

apply (*rule conjI*)
— (1,0) nach (1,4), prozedurbody, zwei wege
apply (*case-tac css*)
apply (*simp only*:)
apply (*drule-tac* $P{=}\lambda\ s.\ s$ **in** *subst*[*OF sysinv.simps*])
apply (*simp add*: *validF-validFs.simps Let-def split-def fst-conv snd-conv nv-def lift-def*)

— case "css = a list"
apply (*simp add*: *validF-validFs.simps Let-def split-def fst-conv snd-conv nv-def*

lift-def tval.cases id-lookup-def)
apply (*rule impI*)
apply (*simp add*: *validF-validFs.simps Let-def split-def fst-conv snd-conv nv-def lift-def tval.cases id-lookup-def update-def split add*: *tval.split tval.split-asm*)
apply (*rule conjI* | *rule impI*)+

apply (*erule-tac x=NAT* (*Suc 0*) **in** *allE*)
apply (*erule conjE*)+
apply (*erule-tac x=Suc 0* **in** *allE*)
apply *simp*

apply *arith*

— Sprungbedingung gilt nicht

apply (*rule impI* | *rule allI* | *erule conjE*)+
apply (*rule conjI*)
apply (*simp add*: *lift-def nv-def update-def*)
apply (*erule-tac x=NAT 0* **in** *allE*)
apply *simp*
apply (*case-tac m* (*Suc 0*))
apply (*simp add*: *MAX-def*)
apply *simp*
apply (*case-tac 0* < *nat*)
apply *simp*
apply *simp*
apply *simp*

apply (*rule conjI*)
apply (*erule-tac x=NAT 0* **in** *allE*)
apply (*simp add*: *update-def*)

apply (*rule conjI*)
apply *simp*
apply (*case-tac m* (*Suc 0*))
apply *simp*
apply *simp*
apply *simp*

apply (*rule impI*)
apply (*simp add*: *nv-def lift-def*)
apply (*erule-tac x=NAT 0* **in** *allE*)
apply (*simp add*: *update-def*)

apply (*case-tac m* (*Suc 0*))
apply *simp*
apply *simp*
apply *simp*

done

1.2.3 Optimised VC

Vereinfachungen:

- verschachtelte Ands aufloesen (And [And [A1,..,An], ..] = And [A1,..,An, ..])
- konstante arithmetische Ausdruecke auswerten: Plus (C (NAT 3)) (C (NAT 3)) = C (NAT 6)
- Implikationen mit T,F vereinfachen: Imp F A = T, Imp T A = A, Imp A T = T
- Ands mit T,F vereinfachen: And [..,T,..] = And [..,..], And [..,F,..] = F, And [] = T
- Goals die direkt aus Assumptions folgen, herausnehmen, Imp And [..,A,..] And [..,A,..] = Imp And [..,..] And [..,..]
- von (V x) = (C (NAT 5)) auf Ty (V x) Nat schliessen (nicht implementiert)
- Ty (C (NAT 15) Nat) = T
- einelementige Ands aufloesen And [A] = A

constdefs *vco* :: *SALform*
vco ≡ (Λ [((Λ [(((V 0) ≐ (C (NAT (Suc (Suc (Suc (Suc 0))))))), (((V (Suc 0)) ≐ (C (NAT (Suc (Suc 0))))), (Pc ≐ (C (POS (0, (Suc (Suc 0))))))]) ⊃ (Λ [(Ty (V 0) vtype.Nat), (Ty (V (Suc 0)) vtype.Nat), (∏ (Suc (Suc (Suc (Suc (Suc 0))))). ((Neg ((Lv (Suc (Suc (Suc (Suc (Suc 0)))))) ≐ (C (NAT (Suc (Suc (Suc 0))))))) ⊃ ((IF (Lv (Suc (Suc (Suc (Suc (Suc 0)))))) ≐ (C (NAT (Suc (Suc (Suc 0))))) THEN (C (POS (0, (Suc (Suc (Suc 0)))))) ELSE (Deref (Lv (Suc (Suc (Suc (Suc (Suc 0)))))))) ≐ (Deref (Lv (Suc (Suc (Suc (Suc (Suc 0))))))))))]))), ((Λ [(((V (Suc (Suc (Suc 0)))) ≐ Rp), (Ty (V 0) vtype.Nat), (Ty (V (Suc 0)) vtype.Nat), (∏ (Suc (Suc (Suc (Suc (Suc 0))))). ((Neg ((Lv (Suc (Suc (Suc (Suc (Suc 0)))))) ≐ (C (NAT (Suc (Suc (Suc 0))))))) ⊃ ((Deref (Lv (Suc (Suc (Suc (Suc (Suc 0))))))) ≐ (Old (Deref (Lv (Suc (Suc (Suc (Suc (Suc 0))))))))))), (Pc ≐ (C (POS ((Suc 0), 0))))]) ⊃ (Λ [(Ty ((C (NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))))))))))) ⊖ (V (Suc 0))) vtype.Nat), ((Λ [(Ty ((C (NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0)))))))))))))))))) ⊖ (V (Suc 0))) vtype.Nat), (((V 0) ≺ ((C (NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))) ⊖ (V (Suc 0))))]) ⊃ (Λ [(Ty (V (Suc (Suc (Suc 0)))) Pos), (∏ (Suc (Suc (Suc (Suc (Suc 0))))). ((Λ [(Neg ((Lv (Suc (Suc (Suc (Suc (Suc 0)))))) ≐ (C (NAT (Suc 0))))), (Neg ((Lv (Suc (Suc (Suc (Suc (Suc 0)))))) ≐ (C (NAT (Suc (Suc 0)))))), (Neg ((Lv (Suc (Suc (Suc (Suc (Suc 0)))))) ≐ (C (NAT (Suc (Suc (Suc 0)))))))]) ⊃ ((IF (Lv (Suc (Suc (Suc (Suc (Suc 0)))))) ≐

(C (NAT (Suc (Suc 0)))) THEN ((C (NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))) ⊖ (V (Suc 0))) ELSE
(IF (Lv (Suc (Suc (Suc (Suc (Suc 0)))))) ≐ (C (NAT (Suc (Suc 0)))) THEN (C
(NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc 0)))))))))))))))) ELSE (Deref (Lv (Suc (Suc (Suc (Suc (Suc 0))))))))) ≐
(Old (Deref (Lv (Suc (Suc (Suc (Suc (Suc 0))))))))))), ((Neg ((V (Suc 0)) ≐ (C
(NAT 0)))) ⊃ (Λ [((V (Suc 0)) ≐ (Old (V (Suc 0)))), (((V 0) ⊕ (V (Suc 0)))
≼ (C (NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc 0)))))))))))))))))]))])), ((Λ [(Ty ((C (NAT (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))) ⊖ (V (Suc
0))) vtype.Nat), (Neg ((V 0) ≺ ((C (NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))) ⊖ (V (Suc 0)))))]) ⊃
(Λ [(Ty (V (Suc (Suc (Suc 0)))) Pos), (∏(Suc (Suc (Suc (Suc (Suc 0))))). ((Λ
[(Neg ((Lv (Suc (Suc (Suc (Suc (Suc 0)))))) ≐ (C (NAT (Suc 0))))), (Neg ((Lv
(Suc (Suc (Suc (Suc (Suc 0)))))) ≐ (C (NAT (Suc (Suc 0)))))), (Neg ((Lv (Suc
(Suc (Suc (Suc (Suc 0)))))) ≐ (C (NAT (Suc (Suc (Suc 0)))))))]) ⊃ ((IF (Lv (Suc
(Suc (Suc (Suc (Suc 0)))))) ≐ (C (NAT (Suc 0))) THEN (C (NAT 0)) ELSE
(IF (Lv (Suc (Suc (Suc (Suc (Suc 0)))))) ≐ (C (NAT (Suc (Suc 0)))) THEN ((C
(NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc 0)))))))))))))))) ⊖ (V (Suc 0))) ELSE (IF (Lv (Suc (Suc (Suc (Suc (Suc
0)))))) ≐ (C (NAT (Suc (Suc 0)))) THEN (C (NAT (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc 0))))))))))))))))) ELSE (Deref
(Lv (Suc (Suc (Suc (Suc (Suc 0)))))))))) ≐ (Old (Deref (Lv (Suc (Suc (Suc (Suc
(Suc 0)))))))))))]))])), ((Λ [((V (Suc (Suc (Suc 0)))) ≐ Rp), (Ty (V (Suc (Suc
(Suc 0)))) Pos), (∏(Suc (Suc (Suc (Suc (Suc 0))))). ((Λ [(Neg ((Lv (Suc (Suc
(Suc (Suc (Suc 0)))))) ≐ (C (NAT (Suc 0))))), (Neg ((Lv (Suc (Suc (Suc (Suc
(Suc 0)))))) ≐ (C (NAT (Suc (Suc 0)))))), (Neg ((Lv (Suc (Suc (Suc (Suc (Suc
0)))))) ≐ (C (NAT (Suc (Suc (Suc 0)))))))]) ⊃ ((Deref (Lv (Suc (Suc (Suc (Suc
(Suc 0))))))) ≐ (Old (Deref (Lv (Suc (Suc (Suc (Suc (Suc 0))))))))))), ((Neg ((V
(Suc 0)) ≐ (C (NAT 0)))) ⊃ (Λ [((V (Suc 0)) ≐ (Old (V (Suc 0)))), (((V 0) ⊕
(V (Suc 0))) ≼ (C (NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc 0)))))))))))))))))]))), ((V (Suc (Suc (Suc 0)))) ≐ (C
(POS (0, (Suc (Suc (Suc 0))))))), (Pc ≐ (C (POS ((Suc 0), (Suc (Suc (Suc
0)))))))), ((Old (V 0)) ≐ (Old (C (NAT (Suc (Suc (Suc (Suc 0)))))))), ((Old (V
(Suc 0))) ≐ (Old (C (NAT (Suc (Suc 0)))))))]) ⊃ (Λ [(((V 0) ⊕ (V (Suc 0))) ≼
(C (NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc 0)))))))))))))))))), ((V 0) ≐ (C (NAT (Suc (Suc (Suc (Suc 0))))))),
(Ty (V (Suc 0)) vtype.Nat), (((C (NAT 0)) ≺ (V (Suc 0))) ⊃ (Λ [(((V 0) ⊕
(V (Suc 0))) ≼ (C (NAT (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc (Suc
(Suc (Suc (Suc (Suc (Suc 0))))))))))))))))), ((V (Suc 0)) ≐ (C (NAT (Suc (Suc
0))))]))]))])

Now we prove the optimized verification condition

lemma *vco-prog-holds*:
provable prog vco
apply (*simp only*: *provable-def valid-def split-paired-all* | *rule HOLprf* | *rule ballI* | *rule allI* | *rule impI*)+
apply (*rename-tac pn i m e I*)
apply (*cut-tac wf-prog*, *drule vc-proof-startup*,*assumption*, (*erule conjE* | *erule*

exE)+, (*simp only*: *fst-conv snd-conv*))

— Now, the prelude is finished. The main proof starts . . .
apply (*simp only*: *vco-def*)
apply (*auto simp add*: *vc-simps*)
apply (*simp add*: *vc-simps split add*: *tval.split tval.split-asm*)

apply (*simp add*: *vc-simps split add*: *nat.split*)

apply (*fastsimp simp add*: *vc-simps*)

apply (*simp add*: *vc-simps split add*: *tval.split tval.split-asm*)
apply *arith*

apply (*simp split add*: *nat.split*)

done

end