theory *SALSafetyLogic-deep* = *SALSemantics-deep* + *VerificationConditionGenerator*:

1 SAL Safety Logic

In this theory we instantiate huge parts of our PCC Framework. These include the safety logic operators (connectives, judgements) , the safety policy (safeF) and the VCG parameter functions (succsF and wpF).

1.1 Evaluation of expressions

types $varint = var \Rightarrow tval$

consts
$eval{::}varint \Rightarrow SALstate \Rightarrow expr \Rightarrow tval$

primrec
$eval\ I\ s\ (V\ v) = (let\ (p,m,e){=}s\ in\ m\ v)$
$eval\ I\ s\ (Lv\ v) = I\ v$
$eval\ I\ s\ (C\ tv) = tv$
$eval\ I\ s\ PC = (let\ (p,m,e){=}s\ in\ (RA\ p))$
$eval\ I\ s\ LastRA = (let\ (p,m,e){=}s\ in\ (case\ (length\ (cs\ e))$
$\ of\ 0 \Rightarrow ILLEGAL\ |\ Suc\ n \Rightarrow (case\ n\ of\ 0 \Rightarrow (RA\ (0,0))$
$\qquad\qquad |\ Suc\ n' \Rightarrow RA\ (incA\ (callpc\ e)))))$
$eval\ I\ s\ Time = (let\ (p,m,e){=}s\ in\ NAT\ (length\ (h\ e)))$
$eval\ I\ s\ (Add\ e1\ e2) = (lift\ (op\ +)\ (eval\ I\ s\ e1)\ (eval\ I\ s\ e2))$
$eval\ I\ s\ (Minus\ e1\ e2) = (lift\ (op\ -)\ (eval\ I\ s\ e1)\ (eval\ I\ s\ e2))$
$eval\ I\ s\ (Mult\ e1\ e2) = (lift\ (op\ *)\ (eval\ I\ s\ e1)\ (eval\ I\ s\ e2))$
$eval\ I\ s\ (Deref\ e') = (case\ (eval\ I\ s\ e')$
$\qquad of\ ILLEGAL \Rightarrow ILLEGAL$
$\qquad |\ NAT\ v \Rightarrow (let\ (p,m,e){=}s\ in\ m\ v)$
$\qquad |\ RA\ r \Rightarrow ILLEGAL)$
$eval\ I\ s\ (Ifeq\ e0\ e1\ e2\ e3) = (case\ ((eval\ I\ s\ e0){=}(eval\ I\ s\ e1))$
$\ of\ True \Rightarrow (eval\ I\ s\ e2)\ |\ False \Rightarrow (eval\ I\ s\ e3))$
$eval\ I\ s\ (Old\ e') = (let\ (p,m,e){=}s\ in\ eval\ I\ (callstate\ e)\ e')$

1.2 Validity of formulae

consts
$validF :: varint \Rightarrow SALstate \Rightarrow SALform \Rightarrow bool \quad ((\text{-},\text{-} \models\ \text{-})\ [61,61,60]\ 60)$
$validFs :: varint \Rightarrow SALstate \Rightarrow SALform\ list \Rightarrow bool$

primrec
$validFs\ I\ s\ [] = True$
$validFs\ I\ s\ (f\#fs) = ((validF\ I\ s\ f) \wedge (validFs\ I\ s\ fs))$
$validF\ I\ s\ T = True$

validF I s F = False
validF I s (And fs) = (validFs I s fs)
validF I s (Imp f1 f2) = (validF I s f1 $\longrightarrow$ *(validF I s f2))*
validF I s (Neg f) = ($\neg$ *validF I s f)*
validF I s (Eq e1 e2) = ((eval I s e1) = (eval I s e2))
validF I s (Leq e1 e2) = (nv (eval I s e1) $\leq$ *nv (eval I s e2))*
validF I s (Less e1 e2) = (nv (eval I s e1) < nv (eval I s e2))
validF I s (Ty ex t) = (ty (eval I s ex) = t)
validF I s (Forall v f) = ($\forall$ *tv. validF (I[v::=tv]) s f)*

constdefs
valid :: SALprogram $\Rightarrow$ *SALstate* $\Rightarrow$ *SALform* $\Rightarrow$ *bool ((-,-* $\models$ *-) [61,61,60] 60)*
valid prg s f $\equiv$ *(*$\forall$ *I. validF I s f)*

1.3 Instantiating the Safety Logic Framework

constdefs
FalseF :: SALform ($\llcorner False \lrcorner$)
FalseF $\equiv$ *F*

constdefs
TrueF :: SALform ($\llcorner True \lrcorner$)
TrueF $\equiv$ *T*

constdefs
Conj :: SALform list $\Rightarrow$ *SALform* ($\llcorner\bigwedge\lrcorner$ - [70])
Conj fs $\equiv$ *And fs*

constdefs
Impl :: SALform $\Rightarrow$ *SALform* $\Rightarrow$ *SALform* (- $\llcorner\Longrightarrow\lrcorner$ - [61,60] 60)
Impl a b $\equiv$ *Imp a b*

1.4 Defining the VCG parameter functions succsF and wpF

constdefs
isCall :: (instr option) $\Rightarrow$ *pname* $\Rightarrow$ *bool*
isCall instr pn' $\equiv$ *(case instr of*
None $\Rightarrow$ *False*
| *Some c* $\Rightarrow$ *(case c of*
SET x n $\Rightarrow$ *False*
| *ADD x y* $\Rightarrow$ *False*
| *SUB x y* $\Rightarrow$ *False*
| *INC x* $\Rightarrow$ *False*
| *JMPEQ x y t* $\Rightarrow$ *False*
| *JMPL x y t* $\Rightarrow$ *False*
| *JLE x y t* $\Rightarrow$ *False*
| *JMPB t* $\Rightarrow$ *False*
| *CALL x pn* $\Rightarrow$ *(pn = pn')*

| *RET x* ⇒ *False*
| *MOV s t* ⇒ *False*
| *HALT* ⇒ *False*
))

constdefs
callpoints :: *SALprogram* ⇒ *pname* ⇒ *pos list*
callpoints prg pn ≡ [*cp* ∈ (*domC prg*). *isCall* (*cmd prg cp*) *pn*]

The contextUp and -Dn functions are needed by wpF to express the effect of procedure calls on the enironment (the callstack gets manipulated by these instructions)

constdefs
contextUpE:: *expr* ⇒ *expr*
contextUpE e ≡ *Old e*

consts
contextUp:: *SALform* ⇒ *SALform*
contextUpL:: *SALform list* ⇒ *SALform list*

primrec
contextUpL [] = []
contextUpL (*f*#*fs*) = (*contextUp f*)#(*contextUpL fs*)
contextUp T = *T*
contextUp F = *F*
contextUp (*And fs*) = (*And* (*contextUpL fs*))
contextUp (*Imp f1 f2*) = (*Imp* (*contextUp f1*) (*contextUp f2*))
contextUp (*Neg f*) = (*Neg* (*contextUp f*))
contextUp (*Eq e1 e2*) = (*Eq* (*contextUpE e1*) (*contextUpE e2*))
contextUp (*Leq e1 e2*) = (*Leq* (*contextUpE e1*) (*contextUpE e2*))
contextUp (*Less e1 e2*) = (*Less* (*contextUpE e1*) (*contextUpE e2*))
contextUp (*Ty e vt*) = (*Ty* (*contextUpE e*) *vt*)
contextUp (*Forall x f*) = (*Forall x* (*contextUp f*))

consts
contextDnE:: *expr* ⇒ *expr*

primrec
contextDnE (*V v*) = *V v*
contextDnE (*Lv v*) = (*Lv v*)
contextDnE (*C tv*) = (*C tv*)
contextDnE PC = *PC*
contextDnE LastRA = *LastRA*
contextDnE Time = *Time*
contextDnE (*Add e1 e2*) = (*Add* (*contextDnE e1*) (*contextDnE e2*))
contextDnE (*Minus e1 e2*) = (*Minus* (*contextDnE e1*) (*contextDnE e2*))
contextDnE (*Mult e1 e2*) = (*Mult* (*contextDnE e1*) (*contextDnE e2*))
contextDnE (*Deref e*) = (*Deref* (*contextDnE e*))
contextDnE (*Ifeq e0 e1 e2 e3*) = (*Ifeq* (*contextDnE e0*) (*contextDnE e1*) (*contextDnE*

e2) (contextDnE e3))
contextDnE (Old e) = e

consts
contextDn:: SALform ⇒ SALform
contextDnL:: SALform list ⇒ SALform list

primrec
contextDnL [] = []
contextDnL (f#fs) = (contextDn f)#(contextDnL fs)
contextDn T = T
contextDn F = F
contextDn (And fs) = (And (contextDnL fs))
contextDn (Imp f1 f2) = (Imp (contextDn f1) (contextDn f2))
contextDn (Neg f) = (Neg (contextDn f))
contextDn (Eq e1 e2) = (Eq (contextDnE e1) (contextDnE e2))
contextDn (Leq e1 e2) = (Leq (contextDnE e1) (contextDnE e2))
contextDn (Less e1 e2) = (Less (contextDnE e1) (contextDnE e2))
contextDn (Ty e vt) = (Ty (contextDnE e) vt)
contextDn (Forall n f) = (Forall n (contextDn f))

consts
contextOldUpE:: expr ⇒ expr

primrec
contextOldUpE (V v) = V v
contextOldUpE (Lv v) = Lv v
contextOldUpE (C tv) = C tv
contextOldUpE PC = PC
contextOldUpE LastRA = Old LastRA
contextOldUpE Time = Time
contextOldUpE (Add e1 e2) = (Add (contextOldUpE e1) (contextOldUpE e2))
contextOldUpE (Minus e1 e2) = (Minus (contextOldUpE e1) (contextOldUpE e2))
contextOldUpE (Mult e1 e2) = (Mult (contextOldUpE e1) (contextOldUpE e2))
contextOldUpE (Deref e) = (Deref (contextOldUpE e))
contextOldUpE (Ifeq e0 e1 e2 e3) = (Ifeq (contextOldUpE e0) (contextOldUpE e1)
(contextOldUpE e2) (contextOldUpE e3))
contextOldUpE (Old e) = (Old (Old e))

consts
contextOldUp:: SALform ⇒ SALform
contextOldUpL:: SALform list ⇒ SALform list

primrec
contextOldUpL [] = []
contextOldUpL (f#fs) = (contextOldUp f)#(contextOldUpL fs)
contextOldUp T = T
contextOldUp F = F
contextOldUp (And fs) = (And (contextOldUpL fs))

contextOldUp (*Imp f1 f2*) = (*Imp* (*contextOldUp f1*) (*contextOldUp f2*))
contextOldUp (*Neg f*) = (*Neg* (*contextOldUp f*))
contextOldUp (*Eq e1 e2*) = (*Eq* (*contextOldUpE e1*) (*contextOldUpE e2*))
contextOldUp (*Leq e1 e2*) = (*Leq* (*contextOldUpE e1*) (*contextOldUpE e2*))
contextOldUp (*Less e1 e2*) = (*Less* (*contextOldUpE e1*) (*contextOldUpE e2*))
contextOldUp (*Ty e vt*) = (*Ty* (*contextOldUpE e*) *vt*)
contextOldUp (*Forall n f*) = (*Forall n* (*contextOldUp f*))

```
consts
  ret-succs :: SALprogram ⇒ pos ⇒ loc ⇒ pos list ⇒ (pos × SALform) list
primrec
 ret-succs prg pc x [] = []
 ret-succs prg pc x (cp # cps) =
  ((let
    (pn,  i) = pc;
    (pn′, j) = cp;
     an = (case (anF prg cp) of
            None ⇒ TrueF
            | Some f ⇒ (contextUp f)
          )
   in ((pn′, Suc j),
       Conj [(Eq (V x) (C (RA (pn′, Suc j)))),(Eq PC (C (RA (pn,i)))), an])
   )# (ret-succs prg pc x cps)
  )
```

lemma *ret-succs-split*:
ret-succs prg pc x (*l1*@*l2*) = (*ret-succs prg pc x l1*) @ (*ret-succs prg pc x l2*)**done**

```
constdefs
  succsF::SALprogram ⇒ pos ⇒ (pos × SALform) list
 succsF prg pc ≡ (let (pn, i) = pc in (case (cmd prg pc) of
    None ⇒ []
  | Some ins ⇒ (case ins of
      SET x n ⇒ [((pn, i + 1), Eq PC (C (RA (pn,i))))]
    | ADD x y ⇒ [((pn, i + 1), Eq PC (C (RA (pn,i))))]
    | SUB x y ⇒ [((pn, i + 1), Eq PC (C (RA (pn,i))))]
    | INC x ⇒ [((pn, i + 1),  Eq PC (C (RA (pn,i))))]
    | JMPEQ x y t ⇒ [((pn, i + t), And [ (Ty (V x) N), (Ty (V y) N), (Eq (V
x) (V y)), (Eq PC (C (RA (pn,i))))]), ((pn, i + 1), And [ (Ty (V x) N), (Ty (V
y) N), (Neg (Eq (V x) (V y))), (Eq PC (C (RA (pn,i))))])
                    ]
    | JMPL x y t ⇒ [((pn, i + t), And [(Ty (V x) N), (Ty (V y) N), (Less (V x)
(V y)), (Eq  PC (C (RA (pn,i))))]), ((pn, i + 1), And [(Ty (V x) N), (Ty (V y)
N), (Neg (Less (V x) (V y))), (Eq PC (C (RA (pn,i))))])
                   ]
     | JLE x y t ⇒ [((pn, i + t), And [(Ty (V x) N), (Ty (V y) N), (Leq (V x)
(V y)), (Eq  PC (C (RA (pn,i))))]), ((pn, i + 1), And [(Ty (V x) N), (Ty (V y)
N), (Neg (Leq (V x) (V y))), (Eq PC (C (RA (pn,i))))])
                   ]
```

| *JMPB t* ⇒ [((*pn*, *i* − *t*), *Eq PC* (*C* (*RA* (*pn*,*i*))))]
| *CALL x pn′* ⇒ [((*pn′*, *0*), *Eq PC* (*C* (*RA* (*pn*,*i*))))]
| *RET x* ⇒ (*ret-succs prg pc x* (*callpoints prg pn*))
| *MOV s t* ⇒ [((*pn*,*i*+*1*), *Eq PC* (*C* (*RA* (*pn*,*i*))))]
| *HALT* ⇒ []

)))

constdefs
wpF :: *SALprogram* ⇒ *pos* ⇒ *pos* ⇒ *SALform* ⇒ *SALform*
wpF prg pc1 pc2 Q ≡ *let* (*pn*, *i*) = *pc1*; (*pn′*,*j*) = *pc2 in*
(*case* (*cmd prg* (*pn*,*i*)) *of*
None ⇒ *FalseF*
| *Some ins* ⇒ (*case ins*
of SET x n ⇒ *substF NoPt* [(*Time*,*Add Time* (*C* (*NAT 1*))),(*PC*,*C* (*RA pc2*)),(*V x*,*C* (*NAT n*))] *Q*
| *ADD x y* ⇒ *substF NoPt* [(*Time*,*Add Time* (*C* (*NAT 1*))),(*PC*,*C* (*RA pc2*)),(*V x*, *Add* (*V x*) (*V y*))] *Q*
| *SUB x y* ⇒ *substF NoPt* [(*Time*,*Add Time* (*C* (*NAT 1*))),(*PC*,*C* (*RA pc2*)),(*V x*, *Minus* (*V x*) (*V y*))] *Q*
| *INC x* ⇒ *substF NoPt* [(*Time*,*Add Time* (*C* (*NAT 1*))),(*PC*,*C* (*RA pc2*)),(*V x*, *Add* (*V x*) (*C* (*NAT 1*)))] *Q*
| *JMPEQ x y t* ⇒ *substF NoPt* [(*Time*,*Add Time* (*C* (*NAT 1*))),(*PC*,*C* (*RA pc2*))] *Q*
| *JMPL x y t* ⇒ *substF NoPt* [(*Time*,*Add Time* (*C* (*NAT 1*))),(*PC*,*C* (*RA pc2*))] *Q*
| *JLE x y t* ⇒ *substF NoPt* [(*Time*,*Add Time* (*C* (*NAT 1*))),(*PC*,*C* (*RA pc2*))] *Q*
| *JMPB t* ⇒ *substF NoPt* [(*Time*,*Add Time* (*C* (*NAT 1*))),(*PC*,*C* (*RA pc2*))] *Q*
| *CALL x pn″* ⇒ (*contextDn* (*substF NoPt* [(*Time*,*Add Time* (*C* (*NAT 1*))),(*LastRA*,*C* (*RA* (*pn*,*i*+*1*))),(*PC*,*C* (*RA pc2*)),(*V x*,*C* (*RA* (*pn*,*i*+*1*)))] *Q*))
| *RET x* ⇒ *contextOldUp* (*substF NoPt* [(*Time*,*Add Time* (*C* (*NAT 1*))),(*PC*,*C* (*RA pc2*))] *Q*)
| *MOV s t* ⇒ *substF* (*Mv s t*) [(*Time*,*Add Time* (*C* (*NAT 1*))),(*PC*,*C* (*RA pc2*))] *Q*
| *HALT* ⇒ *TrueF*
))

1.5 Instantiating the safety policy

constdefs
safeF :: *SALprogram* ⇒ *pos* ⇒ *SALform*
safeF prg pc ≡ (*let* (*pn*,*i*) = *pc in* (*case* (*cmd prg pc*) *of*
None ⇒ *FalseF*
| *Some ins* ⇒ (*case ins of*
SET x n ⇒ *Leq* (*C* (*NAT n*)) (*C* (*NAT MAX*))
| *ADD x y* ⇒ *Leq* (*Add* (*V x*) (*V y*)) (*C* (*NAT MAX*))
| *SUB x y* ⇒ *And* [(*Ty* (*V x*) *N*), (*Ty* (*V y*) *N*)]

$\quad | \ INC\ x\ \Rightarrow Less\ (V\ x)\ (C\ (NAT\ MAX))$
$\quad | \ JMPEQ\ x\ y\ t \Rightarrow And\ [(Ty\ (V\ x)\ N),\ (Ty\ (V\ y)\ N)]$
$\quad | \ JMPL\ x\ y\ t \Rightarrow And\ [(Ty\ (V\ x)\ N),\ (Ty\ (V\ y)\ N)]$
$\quad | \ JLE\ x\ y\ t \Rightarrow And\ [(Ty\ (V\ x)\ N),\ (Ty\ (V\ y)\ N)]$
$\quad | \ JMPB\ t \Rightarrow TrueF$
$\quad | \ CALL\ x\ pn' \Rightarrow TrueF$
$\quad | \ RET\ x \Rightarrow And\ [(Eq\ (V\ x)\ LastRA),\ (Ty\ (V\ x)\ R)]$
$\quad | \ MOV\ s\ t \Rightarrow And\ [(Ty\ (V\ s)\ N),\ (Ty\ (V\ t)\ N)]$
$\quad | \ HALT \Rightarrow TrueF$
)))

constdefs
$ipc :: SALprogram \Rightarrow pos$
$ipc\ prg \equiv (0,0)$

constdefs
$initF :: SALprogram \Rightarrow SALform$
$initF\ prg \equiv And\ [(Eq\ PC\ (C\ (RA\ (0,0)))),$
$(Forall\ 0\ (Ty\ (Deref\ (Lv\ 0))\ E)),$
$(Forall\ 0\ (Ty\ (Old\ (Deref\ (Lv\ 0)))\ E)),$
$(Eq\ LastRA\ (C\ (RA\ (0,0)))),$
$(Eq\ Time\ (C\ (NAT\ 0)))]$

constdefs
$isafeF::SALprogram \Rightarrow pos \Rightarrow SALform$
$isafeF\ prg\ pc \equiv isafe(domC\ prg,prg,anF\ prg,pc,FalseF,Conj,Impl,safeF,succsF,wpF)$

constdefs
$isafeP::SALprogram \Rightarrow SALstate\ set\ (isafe_{\square}\text{-}\ [70])$
$isafeP\ prg \equiv (isafeP'\ effS\ valid\ initF\ isafeF\ prg)$

lemma *isafeP-induct*:
$[\![\ s \in (isafeP\ prg);$
$\bigwedge s.\ [\![\ valid\ prg\ s\ (initF\ prg)\ ]\!] \Longrightarrow P\ s;$
$\bigwedge s\ s'.\ [\![\ s \in (isafeP\ prg);\ valid\ prg\ s\ (isafeF\ prg\ (fst\ s));\ valid\ prg\ s'\ (isafeF\ prg\ (fst\ s'));\ (s,s') \in (effS\ prg);\ P\ s\ ]\!] \Longrightarrow P\ s'\ ]\!] \Longrightarrow P\ s$**done**

lemma *doubleAllI*:
$(\forall\ x.\ P\ x = Q\ x) \Longrightarrow (\forall\ x.\ P\ x) = (\forall\ y.\ Q\ y)$**done**

1.6 Provability of formulae

In this section we define a proof calculus for our safety logic. We use natural deduction plus an extra rule HOLprf that allows us to defer the remainder of a proof to the Isabelle/HOL proof calculus.

1.6.1 Renaming and Instantiation for Integer Variables

The following is only required for the elimination rule of the Forall quantifier

consts
$minL{::}nat\ list \Rightarrow nat$
$minLe{::}nat \Rightarrow nat\ list \Rightarrow nat$

primrec
$minLe\ n\ [] = n$
$minLe\ n\ (x\#xs) = (case\ (x < n)$
$\quad of\ True \Rightarrow minLe\ x\ xs$
$\quad |\ False \Rightarrow minLe\ n\ xs)$

defs *minL-def*:
$minL\ ns == minLe\ (hd\ ns)\ ns$

lemma *minLe-le*:
$\bigwedge n\ n'.\ [\![\ n \le n'\ ]\!] \Longrightarrow (minLe\ n\ ns = n \lor ((minLe\ n\ ns = minLe\ n'\ ns) \land (minLe\ n\ ns < n))) \land (\forall\ n' \in set\ ns.\ minLe\ n\ ns \le n')$**done**

lemma *minL-in*:
$ns{\neq}[] \Longrightarrow minL\ ns \in set\ (ns)$**done**

lemma *minL-sem*:
$\forall\ n \in set\ ns.\ minL\ ns \le n$**done**

lemma *minL-switch*:
$minL\ (xs\ @\ y\#ys) = minL\ (y\#xs\ @\ ys)$**done**

lemma *minL-subset*:
$\bigwedge xs\ ys.\ ys \neq [] \Longrightarrow set\ ys \subseteq set\ xs \Longrightarrow minL\ xs \le minL\ ys$**done**

constdefs
$delL{::}'a \Rightarrow 'a\ list \Rightarrow 'a\ list$
$delL\ a\ xs == [x{\in}xs.\ x \neq a]$

lemma *delL-length*:
$\bigwedge a.\ a \in set\ xs \Longrightarrow length\ (delL\ a\ xs) < length\ xs$**done**

lemma *delL-setdiff*: $set\ (delL\ x\ xs) = (set\ xs) - \{x\}$**done**

lemma *setdiff-notin*:
$x \notin B \Longrightarrow (x \notin (A - B)) = (x \notin A)$
by *auto*

lemma *delL-subset*:
$\bigwedge x.\ set\ (delL\ x\ xs) \subseteq set\ xs$**done**

lemma *newVar-hint*: $\forall v\ vs.\ v\ mem\ vs \longrightarrow length\ (delL\ (minL\ vs)\ vs) < length\ vs$**done**

consts
$newVar :: var \times (var\ list) \Rightarrow var$

recdef *newVar measure* $(\lambda\ (v,vs).\ length\ vs)$
$newVar\ (v,vs) = (case\ v\ mem\ vs$
$\quad of\ True \Rightarrow newVar\ (Suc\ (minL\ vs),(delL\ (minL\ vs)\ vs))$
$\quad |\ False \Rightarrow v)$
(**hints** *simp*: *newVar-hint*)

lemma *newVar-res*:
$\bigwedge vs\ n\ v.\ [\![\ vs \neq [];\ \ v = (newVar\ (n,vs))\ ]\!] \Longrightarrow (v = n) \vee ((minL\ vs) < v)$**done**

lemma *newVar-notin*:
$\forall\ L\ n.\ newVar\ (n,L) \notin set\ L$**done**

lemma *newVarE*:
$newVar\ (n,L) = v \Longrightarrow v \notin set\ L$**done**

consts
$renLvE :: var \Rightarrow var \Rightarrow expr \Rightarrow expr$

primrec *renLvE*:
$renLvE\ v\ v'\ (V\ v'') = V\ v''$
$renLvE\ v\ v'\ (Lv\ v'') = Lv\ (if\ v{=}v''\ then\ v'\ else\ v'')$
$renLvE\ v\ v'\ (C\ tv) = C\ tv$
$renLvE\ v\ v'\ PC = PC$
$renLvE\ v\ v'\ Time = Time$
$renLvE\ v\ v'\ LastRA = LastRA$
$renLvE\ v\ v'\ (Add\ e1\ e2) = Add\ (renLvE\ v\ v'\ e1)\ (renLvE\ v\ v'\ e2)$
$renLvE\ v\ v'\ (Minus\ e1\ e2) = Minus\ (renLvE\ v\ v'\ e1)\ (renLvE\ v\ v'\ e2)$
$renLvE\ v\ v'\ (Mult\ e1\ e2) = Mult\ (renLvE\ v\ v'\ e1)\ (renLvE\ v\ v'\ e2)$
$renLvE\ v\ v'\ (Deref\ ex) = Deref\ (renLvE\ v\ v'\ ex)$
$renLvE\ v\ v'\ (Ifeq\ e1\ e2\ e3\ e4) = Ifeq\ (renLvE\ v\ v'\ e1)\ (renLvE\ v\ v'\ e2)\ (renLvE\ v\ v'\ e3)\ (renLvE\ v\ v'\ e4)$
$renLvE\ v\ v'\ (Old\ ex) = Old\ (renLvE\ v\ v'\ ex)$

consts
$renLvF :: var \Rightarrow var \Rightarrow SALform \Rightarrow SALform$
$renLvFs :: var \Rightarrow var \Rightarrow (SALform\ list) \Rightarrow (SALform\ list)$

primrec
$renLvFs\ v\ v'\ [] = []$
$renLvFs\ v\ v'\ (f\#fs) = (renLvF\ v\ v'\ f)\#(renLvFs\ v\ v'\ fs)$

$renLvF\ v\ v'\ T = T$
$renLvF\ v\ v'\ F = F$
$renLvF\ v\ v'\ (And\ fs) = And\ (renLvFs\ v\ v'\ fs)$
$renLvF\ v\ v'\ (Imp\ f\ f') = Imp\ (renLvF\ v\ v'\ f)\ (renLvF\ v\ v'\ f')$
$renLvF\ v\ v'\ (Neg\ f) = Neg\ (renLvF\ v\ v'\ f)$
$renLvF\ v\ v'\ (Eq\ ex\ ex') = Eq\ (renLvE\ v\ v'\ ex)\ (renLvE\ v\ v'\ ex')$
$renLvF\ v\ v'\ (Leq\ ex\ ex') = Leq\ (renLvE\ v\ v'\ ex)\ (renLvE\ v\ v'\ ex')$
$renLvF\ v\ v'\ (Less\ ex\ ex') = Less\ (renLvE\ v\ v'\ ex)\ (renLvE\ v\ v'\ ex')$
$renLvF\ v\ v'\ (Ty\ ex\ tp) = Ty\ (renLvE\ v\ v'\ ex)\ tp$
$renLvF\ v\ v'\ (Forall\ x\ f) = (case\ x{=}v$
$\qquad of\ True \Rightarrow Forall\ x\ f$
$\qquad |\ False \Rightarrow Forall\ x\ (renLvF\ v\ v'\ f))$

consts
$intVarsF{::}SALform \Rightarrow var\ list$
$intVarsFs{::}SALform\ list \Rightarrow var\ list$

primrec
$intVarsFs\ [] = []$
$intVarsFs\ (f\#fs) = (intVarsF\ f)@(intVarsFs\ fs)$
$intVarsF\ T = []$
$intVarsF\ F = []$
$intVarsF\ (And\ fs) = intVarsFs\ fs$
$intVarsF\ (Imp\ f\ f') = (intVarsF\ f)\ @\ (intVarsF\ f')$
$intVarsF\ (Neg\ f) = (intVarsF\ f)$
$intVarsF\ (Eq\ ex\ ex') = (intVarsE\ ex)\ @\ (intVarsE\ ex')$
$intVarsF\ (Leq\ ex\ ex') = (intVarsE\ ex)\ @\ (intVarsE\ ex')$
$intVarsF\ (Less\ ex\ ex') = (intVarsE\ ex)\ @\ (intVarsE\ ex')$
$intVarsF\ (Ty\ ex\ tp) = (intVarsE\ ex)$
$intVarsF\ (Forall\ x\ f) = x\#(intVarsF\ f)$

consts $instLvE{::}\ expr \Rightarrow expr \Rightarrow var \Rightarrow expr$ (-[-'/-] $[300,\ 0,\ 0]\ 300$)

primrec $instLvE$:
$(V\ x)[ex/v] = (V\ x)$
$(Lv\ x)[ex/v] = (if\ (v{=}x)\ then\ ex\ else\ (Lv\ x))$
$(C\ tv)[ex/v] = C\ tv$
$PC[ex/v] = PC$
$Time[ex/v] = Time$
$LastRA[ex/v] = LastRA$
$(Add\ e1\ e2)[ex/v] = Add\ (e1[ex/v])\ (e2[ex/v])$
$(Minus\ e1\ e2)[ex/v] = Minus\ (e1[ex/v])\ (e2[ex/v])$
$(Mult\ e1\ e2)[ex/v] = Mult\ (e1[ex/v])\ (e2[ex/v])$
$(Deref\ ex')[ex/v] = Deref\ (ex'[ex/v])$
$(Ifeq\ e1\ e2\ e3\ e4)[ex/v] = Ifeq\ (e1[ex/v])\ (e2[ex/v])\ (e3[ex/v])\ (e4[ex/v])$
$(Old\ ex')[ex/v] = Old\ (ex'[ex/v])$

consts $instLvF{::}\ (SALform \times (var \times expr)) \Rightarrow SALform$

syntax *instLvF2* :: *SALform* ⇒ *expr* ⇒ *var* ⇒ *SALform* (-[-′/-] [*300*, *0*, *0*] *300*)

translations
instLvF2 f t v ⇌ *instLvF* (*f*,*v*,*t*)

consts
sizeF::*SALform* ⇒ *nat*
sizeFs::*SALform list* ⇒ *nat*

primrec
sizeFs [] = *0*
sizeFs (*f*#*fs*) = (*sizeF f*) + (*sizeFs fs*)
sizeF T = *0*
sizeF F = *0*
sizeF (*And fs*) = *Suc* (*sizeFs fs*) + *length fs*
sizeF (*Imp f1 f2*) = *Suc* ((*sizeF f1*) + (*sizeF f2*))
sizeF (*Neg f*) = *Suc* (*sizeF f*)
sizeF (*Eq e e*′) = *0*
sizeF (*Leq e e*′) = *0*
sizeF (*Less e e*′) = *0*
sizeF (*Ty e tp*) = *0*
sizeF (*Forall n f*) = *Suc* (*sizeF f*)

lemma *sizeF-sizeFs*:
∀ *fs f*. *f* ∈ *set fs* ⟶ *sizeF f* < *Suc* (*sizeFs fs* + *length fs*)
apply (*rule allI*)
apply (*induct-tac fs*)
apply *auto*
done

lemma *renLvF-sizeF*:
sizeF (*renLvF v v*′ *f*) = *sizeF f*
apply (*induct f*)
apply *simp*
apply *simp*
apply *simp*
prefer *9*
apply *simp*
prefer *8*
apply *simp+*
apply (*case-tac nat* = *v*)
apply *simp*
apply *simp*
done

consts *nV*::*var* ⇒ (*var list*) ⇒ *var*
defs *nV-def*:

$nV == (\lambda\ v\ L.\ newVar\ (v,L))$

recdef *instLvF measure* $(\lambda\ (f,(v,t)).\ sizeF\ f)$
$T[ex/v] = T$
$F[ex/v] = F$
$(And\ fs)[ex/v] = And\ (map\ (\lambda\ f.\ f[ex/v])\ fs)$
$(Imp\ f\ f')[ex/v] = Imp\ (f[ex/v])\ (f'[ex/v])$
$(Neg\ f)[ex/v] = Neg\ (f[ex/v])$
$(Eq\ e1\ e2)[ex/v] = Eq\ (e1[ex/v])\ (e2[ex/v])$
$(Leq\ e1\ e2)[ex/v] = Leq\ (e1[ex/v])\ (e2[ex/v])$
$(Less\ e1\ e2)[ex/v] = Less\ (e1[ex/v])\ (e2[ex/v])$
$(Ty\ ex'\ tp)[ex/v] = Ty\ (ex'[ex/v])\ tp$
$(Forall\ x\ f)[ex/v] = (case\ v{=}x$
$\quad of\ True \Rightarrow Forall\ x\ f$
$\quad |\ False \Rightarrow let\ nv = nV\ x\ (intVarsF\ f\ @(intVarsE\ ex)\ @[v])$
$\qquad in\ Forall\ nv\ (renLvF\ x\ nv\ f\ [ex/v]))$
(**hints** *simp*: *sizeF-sizeFs renLvF-sizeF*)

consts *stateInv::expr* $\Rightarrow$ *bool*

primrec
stateInv (*V v*) = *False*
stateInv (*Lv v*) = *True*
stateInv (*C tv*) = *True*
stateInv PC = *False*
stateInv Time = *False*
stateInv LastRA = *False*
stateInv (*Add e1 e2*) = (*stateInv e1* $\wedge$ *stateInv e2*)
stateInv (*Minus e1 e2*) = (*stateInv e1* $\wedge$ *stateInv e2*)
stateInv (*Mult e1 e2*) = (*stateInv e1* $\wedge$ *stateInv e2*)
stateInv (*Deref ex*) = *False*
stateInv (*Ifeq e1 e2 e3 e4*) =(*stateInv e1* $\wedge$ *stateInv e2* $\wedge$ *stateInv e3* $\wedge$ *stateInv e4*)
stateInv (*Old ex*) = (*stateInv ex*)

1.6.2 The Proof Calculus

consts *deriv* :: (*SALprogram* $\times$ (*SALform list*) $\times$ *SALform*) *set*

syntax *-deriv* :: *SALprogram* $\Rightarrow$ (*SALform list*) $\Rightarrow$ *SALform* $\Rightarrow$ *bool* ((-,- $\vdash$ -) [*61*,*61*,*60*] *60*)

translations
prg,*A* $\vdash$ *f* $\rightleftharpoons$ (*prg*,*A*,*f*) $\in$ *deriv*

inductive *deriv*
intros
HOLprf: ($\forall$ *s* $\in$ (*isafeP prg*). *prg*,*s* $\models$ *Imp* (*And A*) *f*) $\Longrightarrow$ *prg*,*A* $\vdash$ *f*

Asm: *f* $\in$ (*set A*) $\Longrightarrow$ *prg*,*A* $\vdash$ *f*

And0: *prg*,*A* $\vdash$ *And* []

AndI:*prg*,*A* $\vdash$ *And fs* $\Longrightarrow$ *prg*,*A* $\vdash$ *f* $\Longrightarrow$ *prg*,*A* $\vdash$ *And* (*f*#*fs*)

AndE:*f* $\in$ *set fs* $\Longrightarrow$ *prg*,*A* $\vdash$ *And fs* $\Longrightarrow$ *prg*,*A* $\vdash$ *f*

ImpI:*prg*,*f*#*A* $\vdash$ f' $\Longrightarrow$ *prg*,*A* $\vdash$ *Imp f* f'

ImpE:*prg*,*A* $\vdash$ *Imp f* f' $\Longrightarrow$ *prg*,*A* $\vdash$ *f* $\Longrightarrow$ *prg*,*A* $\vdash$ f'

AllI:($\forall$ *a*$\in$ (*set A*). *x* $\notin$ *set* (*freeIntVars a*)) $\Longrightarrow$ *prg*,*A* $\vdash$ *f* $\Longrightarrow$ *prg*,*A* $\vdash$ *Forall x f*

AllE:*stateInv ex* $\Longrightarrow$ *prg*,*A* $\vdash$ *Forall x f* $\Longrightarrow$ *prg*,*A* $\vdash$ *f*[*ex*/*x*]
— In AllE we can only substitute expressions that are state independent. The Integer variable Lv x, which we replace, might appear inside and outside of Old contexts. State dependent parts in ex, for example variables, would have a different

meaning inside and outside of Old.

consts
 provable :: *SALprogram* $\Rightarrow$ *SALform* $\Rightarrow$ *bool* ((- $\vdash$ -) [*61*,*60*] *60*)

defs *provable-def*:
 $prg \vdash f \equiv prg,[] \vdash f$

1.6.3 Verifiy the correctness of the proof system.

Here we prove that the provability judgement is correct. That is provable formulae are valid. The theorem correctSafetyLogic at the very bottom expresses this formally. It is one of the requirements of the PCC Framework.

lemma *stateInv-callstate-eval*:
$\bigwedge s\ I.\ stateInv\ ex \Longrightarrow eval\ I\ (callstate\ (snd\ (snd\ s)))\ ex = eval\ I\ s\ ex$
lemma *intvar-upd-eval*:
$\bigwedge s\ v\ I\ tv.\ v \notin set\ (intVarsE\ ex) \Longrightarrow eval\ (I[v ::= tv])\ s\ ex = eval\ I\ s\ ex$
lemma *intvar-upd-validF*:
$\bigwedge s\ v\ I\ tv.\ v \notin set\ (freeIntVars\ f) \Longrightarrow ((I[v::=tv]),s \models f) = (I,s \models f)$
lemma *renLvE-id*:
$\bigwedge x.\ renLvE\ x\ x\ ex = ex$
lemma *renLvF-id*:
$\bigwedge x.\ renLvF\ x\ x\ f = f$
lemma *freeIntVars-intVarsF*:
$\bigwedge x.\ x \in set\ (freeIntVars\ f) \Longrightarrow x \in set\ (intVarsF\ f)$
lemma *notin-intVarsF-freeIntVars*:
$\bigwedge x.\ x \notin set\ (intVarsF\ f) \Longrightarrow x \notin set\ (freeIntVars\ f)$
lemma *renLvE-upd*:
$\bigwedge s.\ v' \notin (set\ (intVarsE\ ex)) \Longrightarrow (eval\ (I[v::=tv])\ s\ ex) = (eval\ (I[v'::=tv])\ s\ (renLvE\ v\ v'\ ex))$
lemma *renLvF-upd*:
$\bigwedge f\ v\ v'\ I\ tv\ s.\ v' \notin (set\ (intVarsF\ f)) \Longrightarrow (I[v::=tv],s \models f) = ((I[v'::=tv]),s \models (renLvF\ v\ v'\ f))$
lemma *instLvE-eval*:
$\bigwedge s\ I\ x\ ex'.\ stateInv\ ex' \Longrightarrow (eval\ (I[x ::=(eval\ I\ s\ ex')])\ s\ ex) = eval\ I\ s\ (ex[ex'/x])$
lemma *eval-intvar-update*:
$\bigwedge I\ s.\ x \notin set\ (intVarsE\ ex) \Longrightarrow eval\ (I[x ::= tv])\ s\ ex = eval\ I\ s\ ex$
lemma *validF-intvar-update*:
$\bigwedge I\ s.\ x \notin set\ (freeIntVars\ f) \Longrightarrow validF\ (I[x ::= tv])\ s\ f = validF\ I\ s\ f$
lemma *instLvF-validF*:
$\bigwedge f\ I\ x\ ex.\ stateInv\ ex \Longrightarrow ((I[x ::=(eval\ I\ s\ ex)]),s \models f) = (I,s \models f[ex/x])$
lemma *validFAndE*: $f \in set\ fs \Longrightarrow validF\ I\ s\ (And\ fs) \Longrightarrow validF\ I\ s\ f$**done**

lemma *validFs-validF-All*:
$validFs\ I\ s\ fs = (\forall f \in set\ fs.\ validF\ I\ s\ f)$
lemma *provable-validF*:
$\bigwedge A\ prg.\ [\![\ s \in (isafeP\ prg);\ \ prg,A \vdash f\ ]\!] \Longrightarrow (\forall\ I.\ (\forall\ a \in (set\ A).\ validF\ I\ s\ a) \longrightarrow validF\ I\ s\ f)$
theorem *correctSafetyLogic*:

$[\![$ $prg \vdash f$; $s \in (isafeP\ prg)$ $]\!] \Longrightarrow$ $prg,s \models f$

1.7 Some derived proof rules

lemma *AndSingle*:
$prg,A \vdash f \Longrightarrow prg,A \vdash And\ [f]$
apply (*rule AndI*)
apply (*rule And0*)
apply *assumption*
done

lemma *FlattenAsm*:
$prg,(fs'@fs) \vdash f \Longrightarrow$ $prg,((And\ fs)\#fs') \vdash f$
apply (*rule HOLprf*)
apply (*rule ballI*)
apply (*drule provable-validF*)
apply *simp*
apply (*simp add*: *valid-def validF-validFs.simps*)
apply (*rule allI*)
apply (*rule impI*)
apply (*erule conjE*)
apply (*erule-tac x=I* **in** *allE*)
apply (*subgoal-tac* ($\forall a \in set\ fs' \cup set\ fs.\ I,s \models a$))
 prefer *2*
 apply (*rule ballI*)
 apply (*simp add*: *validFs-validF-All*)
 apply (*erule disjE*)
 apply (*erule-tac x=a* **and** *A=set fs'* **in** *ballE*)
 apply *assumption*
 apply *simp*
 apply (*erule-tac x=a* **and** *A=set fs* **in** *ballE*)
 apply *assumption*
 apply *simp*
apply *simp*
done

lemma *FlattenAsmSingle*:
$prg,fs \vdash f \Longrightarrow$ $prg,[And\ fs] \vdash f$
apply (*simp add*: *FlattenAsm*)
done

lemma *AsmSwap*:
$prg,(fs@[f]) \vdash f' \Longrightarrow prg,(f\#fs) \vdash f'$
apply (*rule HOLprf*)
apply (*rule ballI*)
apply (*drule provable-validF*)
apply *simp*
apply (*simp add*: *valid-def validF-validFs.simps*)
apply (*rule allI*)

apply (*erule-tac* x=I **in** *allE*)
apply (*rule impI*)
apply (*simp add*: *validFs-validF-All*)
done

end