1 Jinja Assertion Logic

theory *JBC-SafetyLogic* = *JBC-SafetyPolicy* + *VCG*:

1.1 Evaluation of Expressions

consts *callstate*::*jbc-state* ⇒ *jbc-state*
recdef *callstate* {}
callstate (*p*,(*x*,*h*,*fr*#(*st*,*rg*,*p*′)#*frs*),*e*) =
(*p*′,(*None*,*hd* (*cs e*),(*st*,*rg*,*p*′)#*frs*),*e*(|*cs*:=*tl* (*cs e*)|))

callstate s = *s*

consts
catchstate::(*jvm-prog* × *cname* × *jbc-state*) ⇒ *jbc-state*
recdef *catchstate measure* (λ(*P*,*cn*,(*p*,(*x*,*hp*,*frs*),*e*)). *length frs*)
catchstate (*P*,*X*,(*p*,(*x*,*h*,*fr*#(*st*,*rg*,*p*′)#*frs*),*e*)) =
(*let* (*C*,*M*,*pc*)=*p*′ *in* (*case* (*match-ex-table P X pc* (*ex-table-of P C M*))
of None ⇒ *catchstate*(*P*,*X*,(*p*′,(*None*,*hd* (*cs e*),(*st*,*rg*,*p*′)#*frs*),*e*(|*cs*:=*tl*(*cs e*)|)))
| *Some pc*′ ⇒ (*p*′,(*None*,*hd* (*cs e*),(*st*,*rg*,*p*′)#*frs*),*e*(|*cs*:=*tl* (*cs e*)|))))

catchstate (*P*,*X*,*s*) = *s*

consts
frameCnt:: *jbc-state* ⇒ *nat*
defs *frameCnt-def* [*simp*]:
frameCnt s ≡ (*let* (*p*,σ,*e*)=*s*; (*x*,*h*,*frs*)=σ *in length frs*)

consts *callers-sysinv*::(*jbc-prog* × *jbc-state*) ⇒ *bool*
recdef *callers-sysinv measure* (λ (Π,*s*). *frameCnt s*)
callers-sysinv (Π,(*ps*,(*x*,*h*,*frs*),*e*)) = (*case frs*
of [] ⇒ *True* | *fr* # *frs*′ ⇒ (*case frs*′ *of* [] ⇒ (*let* (*st*,*rg*,*p*)=*fr in p mem* (*domC* Π)
∧ *fst p* = *fst* (*ipc* Π)
∧ *fst* (*snd p*) = *fst* (*snd* (*ipc* Π)))
| *fr*′#*frs*′′ ⇒ (*let* (*st*,*rg*,*p*)=*fr*; (*st*′,*rg*′,*p*′)=*fr*′
in (*p mem* (*domC* Π) ∧ (*p*′ ∈ *set* (*callers* Π *p*) ∧ *callers-sysinv* (Π,(*p*′,(*None*,*hd* (*cs e*),*frs*′),*e*(|*cs*:=*tl* (*cs e*)|)))))))

constdefs *the-Bool*::*val option* ⇒ *bool*
the-Bool v ≡ (*v* = *Some* (*Bool True*))

constdefs
arb::*val*
arb ≡ *arbitrary*

consts
evalNewA::*heap* ⇒ *nat* ⇒ *val*
primrec
evalNewA h 0 = (*case new-Addr h of None* ⇒ *Null*
| *Some a* ⇒ *Addr a*)
evalNewA h (*Suc n*) = *evalNewA* (*h*(*the* (*new-Addr h*):= *Some arbitrary*)) *n*

consts
evalE::*jbc-prog* ⇒ *jbc-state* ⇒ *expr* ⇒ *val option*
evalEs::*jbc-prog* ⇒ *jbc-state* ⇒ *expr list* ⇒ (*val option*) *list*
primrec
evalEs-empty:
evalEs Π *s* [] = []

evalEs-cons:
evalEs Π *s* (*ex*#*exs*) = (*evalE* Π *s ex*)#(*evalEs* Π *s exs*)

evalE-Rg:
evalE Π *s* (*Rg k*) = (*let* (*p*,σ,*e*)=*s*;(*x*,*h*,*fs*)=σ; (*st*,*rg*,*p'*)=*hd fs*
in (*if k* < *length rg then Some* (*rg*!*k*) *else None*))

evalE-St:
evalE Π *s* (*St k*) = (*let* (*p*,σ,*e*)=*s*;(*x*,*h*,*fs*)=σ; (*st*,*rg*,*p'*)=*hd fs*
in (*if k* < *length st then Some* (*st*!*k*) *else None*))

evalE-Lv:
evalE Π *s* (*Lv k*) = (*let* (*p*,σ,*e*)=*s in* ((*lv e*) *k*))

evalE-Cn:
evalE Π *s* (*Cn v*) = *Some v*

evalE-NewA:
evalE Π *s* (*NewA n*) = (*let* (*p*,σ,*e*)=*s*; (*x*,*h*,*frs*)=σ
in Some (*evalNewA h n*))

evalE-Gf:
evalE Π *s* (*Gf F C ex*) = (*case* (*evalE* Π *s ex*)
of Some v ⇒ (*case v of Addr a* ⇒ *Some* (*let* (*p*,σ,*e*)=*s*; (*x*,*h*,*frs*)=σ; (*D*,*fs*)=*the*
(*h a*)

$in\ the\ (fs\ (F,C)))$
$|\ \text{-} \Rightarrow None)$
$|\ \text{-} \Rightarrow None)$

evalE-Fr:
$evalE\ \Pi\ s\ FrNr = (let\ (p,\sigma,e){=}s;\ (x,h,frs){=}\sigma\ in\ Some\ (Intg\ (int\ (length\ frs))))$

evalE-Num:
$evalE\ \Pi\ s\ (Num\ e1\ no\ e2) = liftI\ (numop\ no,\ evalE\ \Pi\ s\ e1,\ evalE\ \Pi\ s\ e2)$

evalE-Rel:
$evalE\ \Pi\ s\ (Rel\ e1\ ro\ e2) = liftR\ (relop\ ro,\ evalE\ \Pi\ s\ e1,\ evalE\ \Pi\ s\ e2)$

evalE-Ite:
$evalE\ \Pi\ s\ (Ite\ b\ t\ e) = (if\ the\text{-}Bool\ (evalE\ \Pi\ s\ b)$
$then\ evalE\ \Pi\ s\ t\ else\ evalE\ \Pi\ s\ e)$

evalE-Eq:
$evalE\ \Pi\ s\ (Eq\ e1\ e2) = Some\ (Bool\ (evalE\ \Pi\ s\ e1 = (evalE\ \Pi\ s\ e2)))$

evalE-Neg:
$evalE\ \Pi\ s\ (Neg\ ex) = Some\ (Bool\ (\neg\ (the\text{-}Bool\ (evalE\ \Pi\ s\ ex))))$

evalE-Imp:
$evalE\ \Pi\ s\ (Imp\ e1\ e2) = Some\ (Bool\ (the\text{-}Bool\ (evalE\ \Pi\ s\ e1) \longrightarrow the\text{-}Bool\ (evalE\ \Pi\ s\ e2)))$

evalE-And:
$evalE\ \Pi\ s\ (And\ exs) = Some\ (Bool\ (list\text{-}all\ (\lambda v.\ the\text{-}Bool\ v)\ (evalEs\ \Pi\ s\ exs)))$

evalE-Forall:
$evalE\ \Pi\ s\ (Forall\ v\ ex) = (let\ (p,\sigma,e){=}s\ in\ Some\ (Bool\ (\forall\, v'.\ the\text{-}Bool\ (evalE\ \Pi\ (p,\sigma,e(\!|lv{:=}((lv\ e)(v{:=}v'))|\!))\ ex))))$

evalE-Ty:
$evalE\ \Pi\ s\ (Ty\ ex\ tp) = Some\ (Bool\ (case\ (evalE\ \Pi\ s\ ex)\ of\ None \Rightarrow False$
$|\ Some\ v \Rightarrow (case\ v$
$of\ Unit \Rightarrow tp{=}\ Void$
$|\ Null \Rightarrow tp{=}\ NT$
$|\ Bool\ b \Rightarrow tp{=}\ Boolean$
$|\ Intg\ i \Rightarrow tp{=}\ Integer$
$|\ Addr\ a \Rightarrow (let\ (p,\sigma,e){=}s;\ (x,hp,frs){=}\sigma\ in\ (case\ (hp\ a)\ of\ None \Rightarrow False\ |\ Some\ ob \Rightarrow tp{=}\ obj\text{-}ty\ ob)))))$

evalE-Pos:
$evalE\ \Pi\ s\ (Pos\ p) = Some\ (Bool\ ((p = fst\ s) \wedge (let\ (p,\sigma,e){=}s;\ (x,h,frs){=}\sigma\ in\ (case\ frs\ of\ [] \Rightarrow False\ |\ fr\#frs' \Rightarrow (let\ (st,rg,ps){=}fr\ in\ p = ps)) \wedge x = None)$

$\wedge$ *callers-sysinv* $(\Pi,s)))$

evalE-Call:
evalE Π *s* (*Call ex*) = (*let* (p,σ,e)=*s*; $(x,h,frs)=\sigma$
in (*if length frs* $\leq$ *1 then None*
else evalE Π (*callstate s*) *ex*))

evalE-Catch:
evalE Π *s* (*Catch X ex*) = (*let* (p,σ,e)=*s*; $(x,h,frs)=\sigma$
in (*if length frs* $\leq$ *1 then None*
else evalE Π (*catchstate* (*fst* Π,*X*,*s*)) *ex*))

lemma *new-Addr-dom*:
$\bigwedge h\ h'.$ (*Map.dom h*) = (*Map.dom h'*) $\Longrightarrow$ *new-Addr h* = *new-Addr h'*

1.2 Lemmas on expression evaluation

lemma *evalNewA-dom*:
$\bigwedge h\ h'.$ (*Map.dom h*) = (*Map.dom h'*) $\Longrightarrow$ *evalNewA h n* = *evalNewA h' n*

lemma *evalEs-evalE*:
ex $\in$ *set exs* $\Longrightarrow$ *evalE* Π *s ex* $\in$ *set* (*evalEs* Π *s exs*)

lemma *evalE-And-rec*:
evalE Π *s* (*And* (*ex#es*)) = *Some* (*Bool* (*the-Bool* (*evalE* Π *s ex*) $\wedge$ *the-Bool* (*evalE* Π *s* (*And es*))))

lemma *evalE-And*:
evalE Π *s* (*And es*) = *Some* (*Bool* ($\forall$ *ex*$\in$*set es. the-Bool* (*evalE* Π *s ex*)))

lemma *subExpr-getPosEx*:
Pos x $\in$ *set* (*subExpr Q*) $\Longrightarrow$ *x* $\in$ *set* (*getPosEx Q*)

lemma *evalEs-map*:
evalEs Π *s exs* = *map* (*evalE* Π *s*) *exs*

1.3 Validity

constdefs
valid::*jbc-prog* $\Rightarrow$ *jbc-state* $\Rightarrow$ *expr* $\Rightarrow$ *bool* ((-,- $\models$ -) [*61*,*61*,*60*] *60*)
valid Π *s ex* $\equiv$ (*evalE* Π *s ex* = *Some* (*Bool True*))

1.4 Provability

Instead of explicit derivation rules, we reuse Isabelle/HOL's inference rules by defining provability as a HOL formula. We directly connect provability with validity.**consts**
safeP::*jbc-prog* $\Rightarrow$ *jbc-state set* ($safe_{\square}$- [*70*])

inductive *safeP* Π
intros
init: $[\![\ s \in (\mathit{initS}\ \Pi)\ \ ]\!] \Longrightarrow s \in (\mathit{safeP}\ \Pi)$
step: $[\![\ s \in (\mathit{safeP}\ \Pi);\ \ (s,s') \in (\mathit{effS}\ \Pi)\ ;$
$\quad \Pi,s \models (\mathit{safeF}\ \Pi\ (\mathit{fst}\ s)); \forall\, a.\ \mathit{anF}\ \Pi\ (\mathit{fst}\ s) = \mathit{Some}\ a \longrightarrow \Pi,s \models a;$
$\quad \Pi,s' \models (\mathit{safeF}\ \Pi\ (\mathit{fst}\ s')); \forall\, a'.\ \mathit{anF}\ \Pi\ (\mathit{fst}\ s') = \mathit{Some}\ a' \longrightarrow \Pi,s' \models a'\ ]\!]$
$\Longrightarrow s' \in (\mathit{safeP}\ \Pi)$

consts
ReachablesAn::*jbc-prog* ⇒ *jbc-state set*

inductive *ReachablesAn* Π
intros
init: $[\![\ s \in (\mathit{initS}\ \Pi)\ \ ]\!] \Longrightarrow s \in (\mathit{ReachablesAn}\ \Pi)$
step: $[\![\ s \in (\mathit{ReachablesAn}\ \Pi);\ \ (s,s') \in (\mathit{effS}\ \Pi)\ ;$
$\quad \Pi,s \models \mathit{aF}\ \Pi\ (\mathit{fst}\ s); \Pi,s' \models \mathit{aF}\ \Pi\ (\mathit{fst}\ s')\ ]\!]$
$\Longrightarrow s' \in (\mathit{ReachablesAn}\ \Pi)$

constdefs
deriv :: *jbc-prog* ⇒ *expr list* ⇒ *expr* ⇒ *bool* ((-,- ⊢ -) [61,60,60] 60)
$\Pi,A \vdash f \equiv (\forall\, s.\ \Pi,s \models \mathit{Imp}\ (\mathit{And}\ A)\ f)$

constdefs
provable :: *jbc-prog* ⇒ *expr* ⇒ *bool* ((- ⊢ -) [61,60] 60)
$\Pi \vdash f \equiv \Pi,[] \vdash f$

lemma *provable-conv*:
$\Pi \vdash f = (\forall\ s.\ \Pi,s \models f)$

lemma *safeP-effS*:
$(p,\sigma,e) \in \mathit{safeP}\ \Pi \Longrightarrow \exists\ p0\ \sigma 0\ e0.\ (p0,\sigma 0,e0) \in (\mathit{initS}\ \Pi) \wedge ((p0,\sigma 0,e0),(p,\sigma,e)) \in (\mathit{effS}\ \Pi)^*$

lemma *safeP-Reachables*:
$s \in \mathit{safeP}\ \Pi \Longrightarrow s \in \mathit{Reachables}\ \Pi$

theorem *correctSafetyLogic*:
$\Pi \vdash f \Longrightarrow \Pi,s \models f$

theorem *completeSafetyLogic*:
$\forall\, s.\ \Pi,s \models f \Longrightarrow \Pi \vdash f$

constdefs *isSafe*::*jbc-prog* ⇒ *bool*
isSafe Π ≡ (∀ *s* ∈ *Reachables* Π. Π,*s* ⊨ *safeF* Π (*fst s*))

1.5 Theorems about safe states.

lemma *initF-startstate*:
ipc Π *mem* (*domC* Π) ⟹ (*let* σ=*start-state* (*fst* Π) (*fst* (*ipc* Π)) (*fst* (*snd* (*ipc* Π)))
in Π,(*ipc* Π,σ,*e*(|*cs*:=[]|)) ⊨ (*initF* Π))

lemma *initS-initF*:
assumes *ipc-domC*: *ipc* Π *mem* (*domC* Π)
assumes *s-initS*: *s* ∈ *initS* Π
shows Π,*s* ⊨ *initF* Π

lemma *initF-frs*:
Π, (*p*,(*x*,*hp*,*frs*),*e*) ⊨ *initF* Π ⟹ *length frs* = *1*

lemma *initS-frs*:
(*p*,(*x*,*hp*,*frs*),*e*) ∈ *initS* Π ⟹ *length frs* = *1*

lemma *frameCnt-callstate*:
⋀*s*. *frameCnt s* = *frameCnt* (*callstate s*) ⟹ *s* = *callstate s*

consts *callstates*::*jbc-state* ⇒ *jbc-state set*

inductive *callstates s*
intros
init: (*callstate s*) ∈ (*callstates s*)
step: *s'* ∈ (*callstates s*) ⟹ (*callstate s'*) ∈ (*callstates s*)

lemma *callstates-union*:
callstates s = ({*callstate s*} ∪ (*callstates* (*callstate s*)))

lemma *catchstate-callstates*:
⋀*s*. *catchstate*(*P*,*X*,*s*) ∈ (*callstates s*)

lemma *find-handler-frs*:
find-handler P x h frs = (*None*,*h*,*fr*#*frs'*)
⟹ ∃ *pfx*. *frs* = *pfx* @ *frs'* ∧ *pfx* ≠ []

lemma *callstate-idem-callstates*:
callstate s = *s* ⟹ *callstates s* = {*s*}

lemma *callstates-subset*:
s' ∈ *callstates s* ⟹ *callstates s'* ⊆ *callstates s*

lemma *callstates-frs*:
$\bigwedge$ *fr frs fr' frs' p e e' x h*. ⟦ *frs = pre @ (fr'#frs')*;
e' = e(|*cs:= drop (length pre) (cs e)*|) ⟧ $\Longrightarrow$
(snd (snd fr'),(None,hd (cs e'),fr'#frs'),e'(|*cs:= tl (cs e')*|)) $\in$
callstates (p,(x,h,fr#frs),e)

lemma *callstates-eq*:
$\bigwedge$ *s s' p x h fr frs e p' x' h' fr'*. ⟦ *s=(p,(x,h,fr#frs),e)*; *s'=(p',(x',h',fr'#frs),e)*; *frs* $\neq$ [] ⟧ $\Longrightarrow$
callstates s = callstates s'

lemma *callstates-safeP*:
assumes *s-safeP*: *s* $\in$ *(safeP* Π*)*
shows $\forall$ *s'* $\in$ *(callstates s)*. *s'* $\in$ *(safeP* Π*)* **using** *s-safeP*

lemma *callstate-safeP*:
$\bigwedge$ *s*. *s* $\in$ *(safeP* Π*)* $\Longrightarrow$ *callstate s* $\in$ *(safeP* Π*)*

lemma *catchstate-safeP*:
s $\in$ *safeP* Π $\Longrightarrow$ *catchstate(fst* Π*,X,s)* $\in$ *(safeP* Π*)*

lemma *callstates-ReachablesAn*:
assumes *s-ReachablesAn*: *s* $\in$ *(ReachablesAn* Π*)*
shows $\forall$ *s'* $\in$ *(callstates s)*. *s'* $\in$ *(ReachablesAn* Π*)* **using** *s-ReachablesAn*

lemma *callstate-ReachablesAn*:
$\bigwedge$ *s*. *s* $\in$ *(ReachablesAn* Π*)* $\Longrightarrow$ *callstate s* $\in$ *(ReachablesAn* Π*)*

lemma *catchstate-ReachablesAn*:
s $\in$ *ReachablesAn* Π $\Longrightarrow$ *catchstate(fst* Π*,X,s)* $\in$ *(ReachablesAn* Π*)*

lemma *ReachablesAn-Reachables*:
s $\in$ *ReachablesAn* Π $\Longrightarrow$ *s* $\in$ *Reachables* Π

lemma *ReachablesAn-initS-aF*:
$(p,\sigma,e) \in$ *ReachablesAn* $\Pi \Longrightarrow (p,\sigma,e) \in$ *initS* $\Pi \vee \Pi,(p,\sigma,e) \models$ *aF* Π *p*

lemma *ReachablesAn-initF-aF*:
⟦ *ipc* Π *mem (domC* Π*)*; $(p,\sigma,e) \in$ *ReachablesAn* Π ⟧ $\Longrightarrow (\Pi,(p,\sigma,e) \models$ *initF* $\Pi) \vee (\Pi,(p,\sigma,e) \models$ *aF*
Π *p*)

constdefs *assert* :: *jbc-prog* $\Rightarrow$ *pos* $\Rightarrow$ *expr*
assert Π *p* $\equiv$ *And* [*safeF* Π *p*,(*case anF* Π *p of None* $\Rightarrow$ *TT* | *Some a* $\Rightarrow$ *a*)]

lemma *safeP-initS-assert*:
$(p,\sigma,e) \in$ *safeP* $\Pi \Longrightarrow (p,\sigma,e) \in$ *initS* $\Pi \vee \Pi,(p,\sigma,e) \models$ *assert* Π *p*

lemma *safeP-initF-assert*:
$[\![$ *ipc* Π *mem domC* Π; $(p,\sigma,e) \in$ *safeP* Π $]\!] \Longrightarrow \Pi,(p,\sigma,e) \models$ *initF* Π $\vee$ $\Pi,(p,\sigma,e) \models$ *assert* Π *p*

lemma *callstate-lv*:
$\bigwedge p\ \sigma\ e.$ *callstate* $(p,\sigma,e(\!|lv{:=}lv'|\!)) = ($*let* $(p',\sigma',e') =$ *callstate* (p,σ,e) *in* $(p',\sigma',e'(\!|lv{:=}lv'|\!)))$

lemma *catchstate-lv*:
$\bigwedge p\ \sigma\ e.$ *catchstate* $(P,X,(p,\sigma,e(\!|lv{:=}lv'|\!))) = ($*let* $(p',\sigma',e') =$ *catchstate* $(P,X,(p,\sigma,e))$ *in* $(p',\sigma',e'(\!|lv{:=}lv'|\!)))$

lemma *callstate-lv-inv*:
callstate $(p,\sigma,e) = (p',\sigma',e') \Longrightarrow lv\ e = lv\ e'$

lemma *catchstate-lv-inv*:
catchstate $(P,X,(p,\sigma,e)) = (p',\sigma',e') \Longrightarrow lv\ e = lv\ e'$

lemma *effS-lv*:
$((p,\sigma,e),(p',\sigma',e')) \in$ *effS* $\Pi \Longrightarrow (((p,\sigma,e(\!|lv{:=}lv'|\!))),(p',\sigma',e'(\!|lv{:=}lv'|\!))) \in$ *effS* Π

lemma *effS-lv-inv*:
$((p,\sigma,e),(p',\sigma',e')) \in ($*effS* $\Pi) \Longrightarrow lv\ e = lv\ e'$

lemma *find-handler-frs'*:
find-handler P x h frs = (*None*,*h*,*fr*#*frs'*)
$\Longrightarrow \exists$ *pfx*. *frs* = *pfx* @ *frs'* $\wedge$ *pfx* $\neq$ [] $\wedge$ *fst* (*snd* (*snd* (*snd fr*))) = *fst* (*snd* (*snd* (*snd* (*last pfx*))))
$\wedge$ *fst* (*snd* (*snd fr*)) = *fst* (*snd* (*snd* (*last pfx*)))

lemma *find-handler-simp*:
find-handler ?P ?a ?h (*?fr* # *?frs*) =
(*let* (*stk*, *loc*, *C*, *M*, *pc*) = *?fr*
in case JVMExceptions.match-ex-table ?P (*cname-of ?h ?a*) *pc* (*ex-table-of ?P C M*) *of None* ⇒ *find-handler ?P ?a ?h ?frs*
| ⌊*pc-d*⌋ ⇒ (*None*, *?h*, (*Addr ?a* # *drop* (*length stk* − *snd pc-d*) *stk*, *loc*, *C*, *M*, *fst pc-d*) # *?frs*))

lemma *safeP-state*:
assumes *s-safeP*: $s \in$ *safeP* Π
shows $\exists$ *C M pc h st rg frs e*. *s* = ((*C*,*M*,*pc*),(*None*,*h*,(*st*,*rg*,(*C*,*M*,*pc*))#*frs*),*e*)
$\wedge$ *fst* (*snd* (*last* (*map* (*snd* ∘ *snd*) ((*st*,*rg*,(*C*,*M*,*pc*))#*frs*))))= *fst* (*snd* (*ipc* Π)) **using** *s-safeP*

lemma *safeP-state'*:
$\bigwedge s.\ s \in$ *safeP* $\Pi \Longrightarrow \exists$ *p p' x h st rg frs e*. *s* = (*p*,(*x*,*h*,(*st*,*rg*,*p'*)#*frs*),*e*)

lemma *the-Bool-simp*:
(*the-Bool v*) = (*v* = *Some* (*Bool True*))

lemma *the-Bool-True*:

$\textit{the-Bool } x \Longrightarrow x = \textit{Some } (\textit{Bool True})$

lemma *the-Bool-False*:
$x \neq \textit{Some } (\textit{Bool True}) \Longrightarrow \textit{the-Bool } x = \textit{False}$

lemma *evalE-And'*:
$\textit{evalE } \Pi \ s \ (\textit{And es}) = \textit{Some } (\textit{Bool } (\textit{list-all } (\lambda \ ex. \ \textit{the-Bool } (\textit{evalE } \Pi \ s \ ex)) \ es))$

lemma *evalE-Pos'*:
$\textit{the-Bool } (\textit{evalE } \Pi \ s \ (\textit{Pos } p)) \Longrightarrow \textit{fst } s = p$

lemma *in-evalEs-conv*: $v \in \textit{set } (\textit{evalEs } \Pi \ s \ exs) \Longrightarrow \exists \ ex \in \textit{set } exs. \ v = \textit{evalE } \Pi \ s \ ex$

lemma *evalE-Or*:
$\textit{evalE } \Pi \ s \ (\textit{Or exs}) = \textit{Some } (\textit{Bool } (\exists \ ex \in \textit{set } exs. \ \textit{evalE } \Pi \ s \ ex = \textit{Some } (\textit{Bool True})))$

lemma *evalE-none* [*simp*]:
$\textit{evalE } \Pi \ s \ \textit{none} = \textit{None}$

constdefs *subclasses*::*jvm-prog* ⇒ *cname* ⇒ *cname list*
$\textit{subclasses } P \ \textit{Cl} \equiv \textit{filter } (\lambda \ \textit{Cl}'. \ P \vdash \textit{Cl}' \preceq^* \textit{Cl}) \ (\textit{map fst } P)$

constdefs *STy*::*jvm-prog* ⇒ *expr* ⇒ *ty* ⇒ *expr*
STy P ex tp ≡ (*case tp*
of Class Cl ⇒ *Or* (*Ty ex NT*#(*map* (λ *Cl*. *Ty ex* (*Class Cl*)) (*subclasses P Cl*)))
| - ⇒ *Ty ex tp*)

lemma *evalE-STy*:
$\textit{evalE } (P,\textit{An}) \ s \ (\textit{STy } P \ ex \ tp) = \textit{Some } (\textit{Bool } (\exists \ tp'. \ \textit{evalE } (P,\textit{An}) \ s \ (\textit{Ty } ex \ tp') = \textit{Some } (\textit{Bool True})$
$\land \ P \vdash tp' \leq tp \land tp' \in \textit{types } P))$

1.6 Auxiliary Lemmas

lemma *env-upd-cs*:
$e(\!|cs{:=}a, \ cs{:=}b|\!) = e(\!|cs{:=}b|\!)$

lemma *env-cs-upd-cs*:
$cs \ (e(\!|cs{:=} \ b|\!)) = b$

lemma *env-upd-id*:
$e(\!|cs{:=} \ cs \ e|\!) = e$

lemma *foldl-map-lookup*:
$\bigwedge \ es. \ \forall \ ex {\in} \ \textit{set } es. \ x {\neq} \textit{Gf } F \ C \ ex \Longrightarrow$
$\forall \ mp'. \ (\textit{foldl } (\lambda mp \ ex. \ (\textit{Gf } F \ C \ ex, \ \textit{IF substE } mp \ ex \doteq \textit{St } (\textit{Suc } 0) \ \textit{THEN St } 0$

ELSE Gf F C (substE mp ex)) # mp) mp′ es) ? x = mp′ ? x

lemma *is-Addr′-conv*:
is-Addr′ $(h,v) = (\exists\ a.\ v = Addr\ a \wedge h\ a \neq None)$

lemma *check-checkinstr*:
check P (None, h, (stk, loc, C, M, pc) # frs) $\Longrightarrow$ *check-instr ((instrs-of P C M)!pc) P h stk loc C M pc frs*

lemma *isIntg-conv*:
is-Intg $a = (\exists\ i.\ a = Intg\ i)$

lemma *evalEs-append*:
evalEs Π *s (exs@exs′) = evalEs* Π *s exs @ (evalEs* Π *s exs′)*

end