1 Jinja VCG

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

1.1 Control Flow Graph

consts *xcpt-cond*::*jbc-prog* ⇒ *cname* ⇒ *pos* ⇒ *expr*
defs *xcpt-cond-def*:
xcpt-cond Π *X p* ≡ (*case* (*cmd* Π *p*) *of None* ⇒ *TT* | *Some c* ⇒ (*case c*
of New C ⇒ *Eq* (*NewA 0*) (*Cn Null*)
| *Getfield F C* ⇒ *Eq* (*St 0*) (*Cn Null*)
| *Putfield F C* ⇒ *Eq* (*St 1*) (*Cn Null*)
| *Checkcast C* ⇒ *And* ((*Neg* (*Eq* (*St 0*) (*Cn Null*)))#(*map* (λ *Cl*. *Neg* (*Ty* (*St 0*) (*Class Cl*))) [*Cl* ∈ (*classnames* (*fst* Π)). (*fst* Π) ⊢ *Cl* $\preceq^*$ *C*]))
| *Invoke M n* ⇒ *Eq* (*St n*) (*Cn Null*)
| *Throw* ⇒ (*if X* = *NullPointer then Neg* (*And* [*Neg* (*Eq* (*St 0*) (*Cn Null*)), *Neg* (*Ty* (*St 0*) (*Class X*))]) *else Ty* (*St 0*) (*Class X*))
| - ⇒ *TT*))

consts
succsInvoke::(*jbc-prog* × *mname* × *nat* × *pos*) ⇒ (*pos* × *expr*) *list*
recdef *succsInvoke* {}
succsInvoke (Π,*M*,*n*,*p*) = (*case anF* Π *p of None* ⇒ []
| *Some A* ⇒ *concat* (*map* (λ *tp*. (*case tp of Void* ⇒ []
| *Boolean* ⇒ []
| *Integer* ⇒ []
| *NT* ⇒ []
| *Class X* ⇒ [((*fst* (*method* (*fst* Π) *X M*),*M*,*0*),*And* [*Neg* (*xcpt-cond* Π *NullPointer p*), *Ty* (*St n*) (*Class X*)])])) (*extractTy* (*A*,*St n*))))

constdefs
succsNormal::*jbc-prog* ⇒ *pos* ⇒ (*pos* × *expr*) *list*
succsNormal Π *p* ≡ (*case cmd* Π *p of None* ⇒ []
| *Some c* ⇒ (*case c*
of Load n ⇒ [(*incA p*,*TT*)]
| *Store n* ⇒ [(*incA p*,*TT*)]
| *Push v* ⇒ [(*incA p*,*TT*)]
| *New C* ⇒ [(*incA p*,*Neg* (*xcpt-cond* Π *OutOfMemory p*))]
| *Getfield F C* ⇒ [(*incA p*,*Neg* (*xcpt-cond* Π *NullPointer p*))]
| *Putfield F C* ⇒ [(*incA p*,*Neg* (*xcpt-cond* Π *NullPointer p*))]
| *Checkcast C* ⇒ [(*incA p*,*Neg* (*xcpt-cond* Π *ClassCast p*))]
| *Invoke M n* ⇒ *succsInvoke* (Π,*M*,*n*,*p*)
| *Return* ⇒ *map* (λ *p'*. (*incA p'*,*Call* (*And* [*aF* Π *p'*,*Pos p'*]))) (*callers* Π *p*)
| *Pop* ⇒ [(*incA p*,*TT*)]

| *IBin no* ⇒ [(*incA p*,*TT*)]
| *Goto t* ⇒ *let* (*C*,*M*,*n*)=*p in* [((*C*,*M*,*nat* ((*int n*)+*t*)),*TT*)]
| *CmpEq* ⇒ [(*incA p*,*TT*)]
| *IfIntCmp ro t* ⇒ *let* (*C*,*M*,*n*)=*p*
        in [((*C*,*M*,*nat* ((*int n*)+*t*)),*Rel* (*St 1*) *ro* (*St 0*)),
          (*incA p*, *Neg* (*Rel* (*St 1*) *ro* (*St 0*)))]
| *IfFalse t* ⇒ *let* (*C*,*M*,*n*)=*p*
        in [((*C*,*M*,*nat* ((*int n*)+*t*)),*Eq* (*St 0*) (*Cn* (*Bool False*))),
          (*incA p*, *Neg* (*Eq* (*St 0*) (*Cn* (*Bool False*))))]
| *Throw* ⇒ []
))

consts
  match-ex-table-e :: *'m prog* ⇒ *cname* ⇒ *pc* ⇒ *ex-table* ⇒ *ex-entry option*
primrec
  match-ex-table-e P C pc []   = *None*
  match-ex-table-e P C pc (*e#es*) = (*if matches-ex-entry P C pc e*
                  then Some e
                  else match-ex-table-e P C pc es)

lemma *match-ex-table-e-sim*:
(*match-ex-table-e P C pc et* = *Some e*) ⟹ (*match-ex-table P C pc et* = *Some* (*snd* (*snd* (*snd e*))))
proof (*induct et*)
assume *A*: (*match-ex-table-e P C pc* [] = *Some e*)
from *A*
show (*match-ex-table P C pc* [] = *Some* (*snd* (*snd* (*snd e*))))
  by *simp*
next
fix *a et*
assume *hyp*: (*match-ex-table-e P C pc et* = *Some e*) ⟹ (*match-ex-table P C pc et* = *Some* (*snd* (*snd* (*snd e*))))
assume *A*: *match-ex-table-e P C pc* (*a # et*) = *Some e*
show *match-ex-table P C pc* (*a#et*) = *Some* (*snd* (*snd* (*snd e*)))
proof (*cases matches-ex-entry P C pc a*)
  case *True*
  from *True A* **show** *?thesis*
    by *simp*
next
  case *False*
  from *False A hyp* **show** *?thesis*
    by *simp*
qed
qed

lemma *match-ex-table-e-sim2*:

match-ex-table P C pc et = Some pc-h $\Longrightarrow$ ($\exists$ *e*.(*match-ex-table-e P C pc et = Some e*) $\wedge$ *snd* (*snd* (*snd e*)) = *pc-h*)
proof (*induct et*)
assume *A*: (*match-ex-table P C pc* [] = *Some pc-h*)
from *A*
show $\exists$ *e*. *match-ex-table-e P C pc* [] = *Some e* $\wedge$ *snd* (*snd* (*snd e*)) = *pc-h*
 by *simp*
next
fix *a et*
assume *hyp*: (*match-ex-table P C pc et = Some pc-h*) $\Longrightarrow$ ($\exists$ *e*. *match-ex-table-e P C pc et = Some e* $\wedge$ *snd* (*snd* (*snd e*)) = *pc-h*)
assume *A*: *match-ex-table P C pc* (*a* # *et*) = *Some pc-h*
show ($\exists$ *e*. *match-ex-table-e P C pc* (*a*#*et*) = *Some e* $\wedge$ *snd* (*snd* (*snd e*)) = *pc-h*)
proof (*cases matches-ex-entry P C pc a*)
 case *True*
 from *True A* **show** *?thesis*
  by *simp*
next
 case *False*
 from *False A hyp* **show** *?thesis*
  by *simp*
qed
qed

consts *match-ex-table*::'*m prog* $\Rightarrow$ *cname* $\Rightarrow$ *pc* $\Rightarrow$ *ex-table* $\Rightarrow$ *pc option*
defs *match-ex-table-def* [*simp*]:
match-ex-table P C pc et == (*case* (*JVMExceptions.match-ex-table P C pc et*) *of None* $\Rightarrow$ *None* | *Some h* $\Rightarrow$ *Some* (*fst h*))

consts
succsXpt::(*jbc-prog* $\times$ *cname* $\times$ *pos list*) $\Rightarrow$ (*pos* $\times$ *expr*) *list*
recdef *succsXpt measure* (λ(Π,*X*,*ps*). *length* (*domC* Π) $-$ (*length ps*))
succsXpt ((*P*,*An*),*X*,*ps*) = (*case length* (*domC* (*P*,*An*)) $\leq$ *length ps*
of True $\Rightarrow$ *map* (λ*p*. (*p*,*TT*)) (*domC* (*P*,*An*))
| *False* $\Rightarrow$ (*case ps of* [] $\Rightarrow$ *map* (λ*p*. (*p*,*TT*)) (*domC* (*P*,*An*))
  | *p*#*pss* $\Rightarrow$ *let* (*C*,*M*,*pc*)=*p*; *et* = *ex-table-of P C M*; *A*=*aF* (*P*,*An*) *p*
   in (*case match-ex-table-e P X pc et*
    of None $\Rightarrow$ *concat* (*map* (λ*p*'. *succsXpt* ((*P*,*An*),*X*,*p*'#*ps*)) (*callers* (*P*,*An*) *p*))
    | *Some e* $\Rightarrow$ (*let* (*f*,*t*,*X*',*pc*',*d*) = *e*
     in [((*C*,*M*,*pc*'),*And* ((*if pss*=[] *then* [] *else* [*Catch X*' *A*,*Catch X* (*Pos p*)])@
      [*xcpt-cond* (*P*,*An*) *X* (*last ps*)]))]))))

constdefs

succsExcept::*jbc-prog* ⇒ *pos* ⇒ (*pos* × *expr*) *list*
succsExcept Π *p* ≡ (*case cmd* Π *p of None* ⇒ []
| *Some c* ⇒ (*case c*
of Load n ⇒ []
| *Store n* ⇒ []
| *Push v* ⇒ []
| *New C* ⇒ *succsXpt* (Π,*OutOfMemory*,[*p*])
| *Getfield F C* ⇒ *succsXpt* (Π,*NullPointer*,[*p*])
| *Putfield F C* ⇒ *succsXpt* (Π,*NullPointer*,[*p*])
| *Checkcast C* ⇒ *succsXpt* (Π,*ClassCast*,[*p*])
| *Invoke M n* ⇒ *succsXpt* (Π,*NullPointer*,[*p*])
| *Return* ⇒ []
| *Pop* ⇒ []
| *IBin no* ⇒ []
| *Goto t* ⇒ []
| *CmpEq* ⇒ []
| *IfIntCmp ro t* ⇒ []
| *IfFalse t* ⇒ []
| *Throw* ⇒ *succsXpt* (Π,*NullPointer*,[*p*]) @ (*case anF* Π *p of None* ⇒ []
| *Some a* ⇒ *concat* (*map* (λ *tp*. (*case tp of Void* ⇒ [] | *Boolean* ⇒ [] | *Integer* ⇒ [] | *NT* ⇒ []
| *Class X* ⇒ *succsXpt* (Π,*X*,[*p*]))) (*extractTy* (*a*,*St 0*))))
))

constdefs *addPos*::*pos* ⇒ ((*pos* × *expr*) *list*) ⇒ (*pos* × *expr*) *list*
addPos p es ≡ (*map* (λ (*p′*,*B*). (*p′*,*And* [*Pos p*,*B*])) *es*)

constdefs *succsF*::*jbc-prog* ⇒ *pos* ⇒ (*pos* × *expr*) *list*
succsF Π *p* ≡ *addPos p* (*succsNormal* Π *p* @ *succsExcept* Π *p*)

1.2 Static Semantics

constdefs
handlesEx′::*jvm-prog* ⇒ *pos* ⇒ *cname list*
handlesEx′ P p ≡ *remdups′* (*concat* (*map* (λ(*C*,(*S*,*Fs*,*Ms*)).
concat (*map* (λ(*M*,*Ts*,*T*,(*mxs*,*mxl*,*is*,*et*)).
concat (*map* (λ(*b*,*e*,*cn*,*h*,*d*).
if p=(*C*,*M*,*h*)
then [*cn*] *else* [])
et))
Ms))
P))

constdefs

$handlesEx::jvm\text{-}prog \Rightarrow pos \Rightarrow cname\ option$
$handlesEx\ P\ p \equiv (case\ handlesEx'\ P\ p\ of\ [] \Rightarrow None$
$\quad | \ cn\#cns \Rightarrow Some\ cn)$

constdefs
$catchesEx::jvm\text{-}prog \Rightarrow cname \Rightarrow pos \Rightarrow bool$
$catchesEx\ P\ X\ p \equiv (let\ (C,M,pc) = p;\ m = match\text{-}ex\text{-}table\ P\ X\ pc\ (ex\text{-}table\text{-}of\ P\ C\ M)$
$\quad in\ (case\ m\ of\ None \Rightarrow False \mid Some\ pc' \Rightarrow True))$

constdefs $sys\text{-}xcpt\text{-}of :: instr \Rightarrow cname$
$sys\text{-}xcpt\text{-}of\ i \equiv (case\ i\ of\ New\ C \Rightarrow OutOfMemory$
$\quad | \ Getfield\ F\ C \Rightarrow NullPointer$
$\quad | \ Putfield\ F\ C \Rightarrow NullPointer$
$\quad | \ Checkcast\ C \Rightarrow ClassCast$
$\quad | \ Invoke\ M\ n \Rightarrow NullPointer$
$\quad | \ Throw \Rightarrow NullPointer$
$\quad | \ \text{-} \Rightarrow Exception)$

constdefs

```
wpF::jbc-prog ⇒ pos ⇒ pos ⇒ expr ⇒ expr
wpF Π p p′ Q ≡
(let pm=map (λq. (Pos q,if q=p′ then Pos p else FF)) (getPosEx Q) in
(case cmd Π p of None ⇒ FF | Some ins ⇒
(case handlesEx (fst Π) p′ of None ⇒ (case ins
 of Load n ⇒ substE (pm@(map (λk. (St k,if k=0 then Rg n
                                   else St (k − 1))) (stkIds Q))) Q

 | Store n ⇒ substE (pm@((Rg n,St 0)#
                   map (λk. (St k, St (k+1))) (stkIds Q))) Q

 | Push v ⇒ substE (pm@(map (λk. (St k,if k=0 then Cn v
                                   else St (k − 1))) (stkIds Q))) Q

 | New Cl ⇒ (let em=(pm@(map (λk. (St k,if k=0 then NewA 0
                   else St (k − 1))) (stkIds Q))@
                   (map (λn. (NewA n, NewA (n+1))) (getNewEx Q)));
               gfe′=foldl (λmp hex. (case hex
of GF F C ex ⇒ (let ex′=substE mp ex
               in (Gf F C ex,IF ex′ ≐ NewA 0
                        THEN Cn (the ((snd (blank (fst Π) Cl))(F,C)))
                        ELSE Gf F C ex′))
| TY ex ty   ⇒ (let ex′=substE mp ex
               in (Ty ex ty,IF ex′ ≐ NewA 0
                        THEN Cn (Bool ((Class Cl) = ty))
                        ELSE Ty ex′ ty)))#mp)
                  em (remdups′ (getHeapEx Q))
               in substE gfe′ Q)

 | Getfield F C ⇒ substE (pm@[(St 0,Gf F C (St 0))]) Q

 | Putfield F C ⇒ (let em=pm@(map (λk. (St k,St (k+2))) (stkIds Q));
                  gfe′=foldl (λmp ex. let ex′=substE mp ex
                               in (Gf F C ex,IF (ex′ ≐ St 1)
                                  THEN St 0 ELSE Gf F C ex′)#mp)
                         em (remdups′ (getGfEx F C Q))
               in substE gfe′ Q)

 | Checkcast C ⇒ substE pm Q

 | Invoke M n ⇒ substE (pm@(FrNr,FrNr ⊕ (Cn (Intg 1)))#
                  (map (λk. (Rg k,if k ≤ n then St (n−k)
                         else (if k ≤ n + fst (snd (snd (snd (snd (method (fst Π) (fst p′) M)))))
                               then Cn arb
```

```
                         else none))) (rgIds Q))@
          (map (λk. (St k,none)) (stkIds Q))@
          (map (λex. (Call ex,ex)) (getCallEx Q))@
          (concat (map (λ(cn',ex').
          (if catchesEx (fst Π) cn' p
           then [(Catch cn' ex',ex')]
           else [(Catch cn' ex',
                IF (FrNr ≐ Cn (Intg 1)) THEN ex'
                ELSE Catch cn' ex')])) (getCatchEx Q)))) Q

| Return ⇒ let (C,M,pc)=p; (P,An)=Π; n = length (fst (snd (method P C M)))
        in substE (pm@(FrNr,FrNr ⊖ (Cn (Intg 1)))#
            (map (λk. (St k,if 1 ≤ k then Call (St (n+k))
                        else St 0)) (stkIds Q))@
           (map (λk. (Rg k,Call (Rg k))) (rgIds Q))@
           (map (λex. (Call ex,Call (Call ex))) (getCallEx Q))@
           (map (λ(cn',ex'). (Catch cn' ex',Call (Catch cn' ex')))
              (getCatchEx Q))) Q

| Pop ⇒ substE (pm@(map (λk. (St k,St (k+1))) (stkIds Q))) Q

| IBin no ⇒ substE (pm@(map (λk. (St k,if k=0 then Num (St 1) no (St 0) else (St (k+1)))) (stkIds
Q))) Q

| Goto t ⇒ substE pm Q
| CmpEq ⇒ substE (pm@(map (λk. (St k,if k=0 then (St 0)≐(St 1)
                                  else (St (k+1)))) (stkIds Q))) Q
| IfIntCmp ro t ⇒ substE (pm@(map (λk. (St k,St (k+2))) (stkIds Q))) Q
| IfFalse t ⇒ substE (pm@(map (λk. (St k,St (k+1))) (stkIds Q))) Q
| Throw ⇒ FF )

| Some cn ⇒ (let mp=pm@(map (λk. (St k,if 1≤k then none
            else (if (ins = Throw)
                then (IF St 0 ≐ Cn (Null)
                     THEN (Cn (Addr (addr-of-sys-xcpt NullPointer)))
                     ELSE St 0)
                 else Cn (Addr (addr-of-sys-xcpt (sys-xcpt-of ins)))))) (stkIds Q))@
        (let (C,M,pc)=p; (C',M',pc')=p'; (P,An)=Π in
         (if match-ex-table P cn pc (ex-table-of P C M) = Some pc' then []
          else
          let rgm=map (λk. (Rg k,Catch cn (Rg k))) (rgIds Q);
              om =map (λex. (Call ex,Catch cn (Call ex))) (getCallEx Q);
              cm= map (λ(cn',ex'). (Catch cn' ex', Catch cn (Catch cn' ex')))
                    (getCatchEx Q)
          in (FrNr,Catch cn FrNr)#rgm@om@cm))
```

in substE mp Q))))

1.3 Welformedness

constdefs *pTy::jvm-prog* ⇒ ty_P
pTy P ≡ (*map-of2* (*convert-pt* (*prog-kil P*)))

consts
throwChk::jbc-prog × *instr option* × *expr option* × *pos* ⇒ *bool*

— Throw instructions require a type annotation (disjunction of Ty (St 0) (Class X)). The successor function uses these to find proper handlers. The initial intruction must not be Throw. In typesafe programs (no exception on top of initial stack) this cannot happen anyway.

defs *throwChk-def*:

```
throwChk ≡ λ (Π,ins,an,p). (case ins of None ⇒ True
                                | Some c ⇒
(case c of Throw ⇒ (case an of None ⇒ False
                      | Some A  ⇒ (if p=ipc Π then False else
                        (case extractTy (A,St 0) of [] ⇒ False
                              | ty#tys ⇒ (list-all (λ tp.
                                          (case tp of Void ⇒ False | Boolean ⇒ False
                                                  | Integer ⇒ False | NT ⇒ False
                                                  | Class X ⇒ True)) (ty#tys)))))
        | - ⇒ True))
```

lemma *throwChk-Throw-A* [*simp*]:

```
throwChk (Π, Some Throw, Some A, p) =
(if p=ipc Π then False else
(case extractTy (A,St 0) of [] ⇒ False
 | ty#tys ⇒ (list-all (λ tp. (case tp of Void ⇒ False | Boolean ⇒ False
                        | Integer ⇒ False | NT ⇒ False
                        | Class X ⇒ True)) (ty#tys))))
```

lemma *throwChk-Throw-None* [*simp*]:
throwChk (Π,*Some Throw*,*None*,*p*) = *False*

lemma *throwChk-oth-None* [*simp*]:
ins ≠ *Some Throw* ⟹ *throwChk* (Π,*ins*,*None*,*p*) = *True*

lemma *throwChk-oth-Some* [*simp*]:
ins ≠ *Some Throw* ⟹ *throwChk* (Π,*ins*,*Some A*,*p*) = *True*

consts
invokeChk::jbc-prog × (*instr option*) × (*expr option*) × *pos* ⇒ *bool*

— Invoke instructions require a type annotation (disjunction of Ty (St n) (Class X)). The successor function uses these to determine the potential method entry points. In addition the first instruction must not be Invoke. In typesafe programs this cannot happen anyway (no object reference on top of initial stack). We also forbid recursive calls of the main method (M = fst (snd (ipc Pi))). This ensures that the frame stack never becomes empty (A Return in method main stops execution).

defs *invokeChk-def*:

invokeChk ≡ λ (Π,*ins*,*an*,*p*). (*case ins of None* ⇒ *True* | *Some c* ⇒ *case c*
of Invoke M n ⇒ (*case an of None* ⇒ *False*
| *Some A* ⇒ (*if p*=*ipc* Π *then False else*
(*case extractTy* (*A*,*St n*) *of* [] ⇒ *False*
| *ty*#*tys* ⇒ (*list-all* (λ *tp*. (*case tp of Void* ⇒ *False* | *Boolean* ⇒ *False*
| *Integer* ⇒ *False* | *NT* ⇒ *True*
| *Class X* ⇒ *has-method* (*fst* Π) *X M*)) (*ty*#*tys*)) ∧ *M* ≠ *fst* (*snd* (*ipc* Π)))))
| - ⇒ *True*)

lemma *invokeChk-Invoke-A* [*simp*]:

invokeChk (Π,*Some* (*Invoke M n*),*Some A*,*p*) = (*if p*=*ipc* Π *then False else*
(*case extractTy* (*A*,*St n*) *of* [] ⇒ *False* | *ty*#*tys* ⇒ (*list-all* (λ *tp*. (*case tp of Void* ⇒ *False* | *Boolean* ⇒ *False*
| *Integer* ⇒ *False* | *NT* ⇒ *True*
| *Class X* ⇒ *has-method* (*fst* Π) *X M*)) (*ty*#*tys*)) ∧ *M* ≠ *fst* (*snd* (*ipc* Π))))

lemma *invokeChk-Invoke-None* [*simp*]:

invokeChk (Π,*Some* (*Invoke M n*),*None*,*p*) = *False*

lemma *invokeChk-oth-None* [*simp*]:

(∀ *M n*. *ins* ≠ *Some* (*Invoke M n*)) ⟹ *invokeChk* (Π,*ins*,*None*,*p*) = *True*

lemma *invokeChk-oth-Some* [*simp*]:

(∀ *M n*. *ins* ≠ *Some* (*Invoke M n*)) ⟹ *invokeChk* (Π,*ins*,*Some A*,*p*) = *True*

consts

checkPos :: *jbc-prog* ⇒ (*pos list*) ⇒ *bool*

— checkPos ensures that targets of backward jumps (idx (domC Pi) p' ¡= idx (domC Pi) p) are annotated. This ensures termination of the generic VCG. In addition all successors must be in the code range (p' mem (domC Pi)) and successors from the normal successor function must not be entry points of handlers. Throw and Invoke instructions are checked for the extra requirements described above.

primrec

checkPos Π [] = *True*
checkPos Π (*p*#*ps*) = (*if* (*let scsn* = *map fst* (*succsNormal* Π *p*);
scse = *map fst* (*succsExcept* Π *p*)
in list-all (λ*p*′. ((*idx* (*domC* Π) *p*′ ≤ *idx* (*domC* Π) *p*)

$\longrightarrow$ *anF* Π $p' \neq$ *None*) $\wedge$ p' *mem* (*domC* Π) $\wedge$
(p' *mem scsn* $\longrightarrow$ *handlesEx* (*fst* Π) p' = *None*) $\wedge$ $p' \neq$ *ipc* Π)
(*scsn* @ *scse*) $\wedge$ (*set scse* $\subset$ *set* (*domC* Π))) $\wedge$ *throwChk* (Π,*cmd* Π *p*,*anF* Π *p*,*p*) $\wedge$
invokeChk (Π,*cmd* Π *p*,*anF* Π *p*,*p*)
then (*checkPos* Π *ps*) *else False*)

lemma *checkPos-split*:
checkPos Π (*l1*@*l2*) = ((*checkPos* Π *l1*) $\wedge$ (*checkPos* Π *l2*))

constdefs *checkExTables* ::*jbc-prog* $\Rightarrow$ *bool*

checkExTables $\Pi \equiv$ *list-all* ($\lambda x.$ *x*) (*concat* (*map* (λ(*C*,(*S*,*Fs*,*Ms*)).
concat (*map* (λ(*M*,*Ts*,*T*,(*mxs*,*mxl*,*is*,*et*)).
(*map* (λ(*b*,*e*,*cn*,*h*,*d*). *d* = *0* $\wedge$
(*C*,*M*,*h*) $\in$ *set* (*domC* Π) $\wedge$
remdups' (*concat* (*map* (λ(*b*, *e*, *cn*, *h'*,*d*). *if* $h' = h$ *then* [*cn*] *else* []) *et*)) = [*cn*]) *et*))
Ms))
(*fst* Π)))

— Programs are wellformed iff (1) all positions are wellformed (checkPos). That is - targets of backward jumps are annotated (enforces VCG termination). - Throw and Invoke instructions have type annotations (possibly inserted by the bytecode verifier). - the main method is not invoked (only automatically at start up). (2) all exception tables are wellformed. That is - remaining stack height d is 0 (no catch inside expressions). - the catching class is not Object. - handler entry points are within the code range. - handler entry points are distinct for each handler. (3) all classes and methods have distinct names. (4) the programs containts all system classes (object, exceptions). (5) the intial position is in the code range. (6) the main method has no arguments (type safety theorems from BV require this).

constdefs
wf :: *jbc-prog* $\Rightarrow$ *bool*
wf $\Pi \equiv$ *checkPos* Π (*domC* Π) $\wedge$
checkExTables Π $\wedge$
distinct (*classnames* (*fst* Π)) $\wedge$
distinct (*methodnames* (*fst* Π)) $\wedge$
($\exists$ *cdl*. *fst* Π = (*SystemClasses* @ *cdl*)) $\wedge$ (*ipc* $\Pi \in$ *set* (*domC* Π))
$\wedge$ *wf-jvm-prog-phi* (*pTy* (*fst* Π)) (*fst* Π)
$\wedge$ *fst* (*snd* (*method* (*fst* Π) (*fst* (*ipc* Π)) (*fst* (*snd* (*ipc* Π))))) = []

1.4 System Invariants

The following functions yield formulas that hold for all states reachable in wellformed programs

1.5 Position information

constdefs *inv-Pos::jbc-prog ⇒ pos ⇒ expr*
inv-Pos Π *p* ≡ *Pos p*

1.6 Frame Stack Size

constdefs *inv-FrNr:: jbc-prog ⇒ pos ⇒ expr*
inv-FrNr Π ≡ (λ(*C*,*M*,*pc*). *if* (*let* (*C0*,*M0*,*pc0*) = *ipc* Π *in C*=*C0* ∧ *M*=*M0*) *then Eq FrNr* (*Cn* (*Intg 1*)) *else* (*Rel* (*Cn* (*Intg 1*)) *Less FrNr*))

1.7 System Exception Types

constdefs *inv-ExTys::jbc-prog ⇒ pos ⇒ expr*
inv-ExTys Π *p* ≡ *And* [*Ty* (*Cn* (*Addr* (*addr-of-sys-xcpt NullPointer*))) (*Class NullPointer*),
Ty (*Cn* (*Addr* (*addr-of-sys-xcpt ClassCast*))) (*Class ClassCast*),
Ty (*Cn* (*Addr* (*addr-of-sys-xcpt OutOfMemory*))) (*Class OutOfMemory*)]

1.8 Bytecode Verifier Types

constdefs *conv-st::jvm-prog* ⇒ ty_i ⇒ *expr*
conv-st P ≡ (λ (*st*,*rt*). (*let ex-st* = *map* (λ *n*. *STy P* (*St n*) (*st*!*n*)) (*upt 0* (*length st*));
ex-rt = *map* (λ *n*. (*case rt*!*n of Err* ⇒ *not-none* (*Rg n*)
| *OK tp* ⇒ *STy P* (*Rg n*) *tp*)) (*upt 0* (*length rt*))
in (*And* (*ex-st* @ *ex-rt*))))

constdefs *annotate-types::jvm-prog* ⇒ ((*cname* × *mname*) × ($ty_i{}'$ *list*)) *list* ⇒ (*pos* ⇒ *expr*)
annotate-types P pt ≡ (λ (*C*,*M*,*pc*).
(*if* (*C*,*M*) *mem* (*methodnames P*) *then* (*case pt ?* (*C*,*M*) *of None* ⇒ *FF*
| *Some mt* ⇒ (*if pc* < *length mt then* (*case mt* ! *pc of None* ⇒ *FF* | *Some tyi* ⇒ *conv-st P tyi*)
else TT))
else TT))

constdefs *inv-Ty::jbc-prog ⇒ pos ⇒ expr*
inv-Ty Π *p* ≡ *annotate-types* (*fst* Π) (*convert-pt* (*prog-kil* (*fst* Π))) *p*

1.9 Instantiating the VCG

constdefs
vcg-jbc :: jbc-prog ⇒ expr
vcg-jbc prg ≡ *vcG And Imp FF ipc initF safeF succsF wpF domC domA anF prg*

1.10 Upgrade the VCG

Here we upgrade the VCG by instantiating it with successor functions that add invariants to

branch conditions.**constdefs** upg::($jbc\text{-}prog \Rightarrow pos \Rightarrow expr$) $\Rightarrow$ ($jbc\text{-}prog \Rightarrow pos \Rightarrow (pos \times expr)\ list$) $\Rightarrow$ ($jbc\text{-}prog \Rightarrow pos \Rightarrow (pos \times expr)\ list$)
$upg\ iF\ sucF \equiv (\lambda\ \Pi\ p.\ map\ (\lambda\ (p',B).\ (p',\ And\ [B,\ iF\ \Pi\ p]))\ (sucF\ \Pi\ p))$

1.11 Upgrade Frame Stack Size

constdefs $succsFrNrF$:: $jbc\text{-}prog \Rightarrow pos \Rightarrow (pos \times expr)\ list$
$succsFrNrF \equiv upg\ inv\text{-}FrNr\ succsF$

constdefs
$vcgFrNr$ :: $jbc\text{-}prog \Rightarrow expr$
$vcgFrNr\ prg \equiv vcG\ And\ Imp\ FF\ ipc\ initF\ safeF\ succsFrNrF\ wpF\ domC\ domA\ anF\ prg$

1.12 Upgrade System Exception Types.

constdefs $succsExTysF$:: $jbc\text{-}prog \Rightarrow pos \Rightarrow (pos \times expr)\ list$
$succsExTysF \equiv upg\ inv\text{-}ExTys\ succsFrNrF$

constdefs
$vcgExTys$ :: $jbc\text{-}prog \Rightarrow expr$
$vcgExTys\ prg \equiv vcG\ And\ Imp\ FF\ ipc\ initF\ safeF\ succsExTysF\ wpF\ domC\ domA\ anF\ prg$

1.13 Upgrade Types

constdefs $succsTyF$:: $jbc\text{-}prog \Rightarrow pos \Rightarrow (pos \times expr)\ list$
$succsTyF \equiv upg\ inv\text{-}Ty\ succsExTysF$

constdefs
$vcgTy$ :: $jbc\text{-}prog \Rightarrow expr$
$vcgTy\ \Pi \equiv (vcG\ And\ Imp\ FF\ ipc\ initF\ safeF\ succsTyF\ wpF\ domC\ domA\ anF\ \Pi)$

end