PCC

Martin Wildmoser

7th February 2005

Contents

theory *VCGExec = ProofCalculus + VCOpt + TermCodegen*:

0.0.1 Control Flow Graph

lemma [*code*]:
addr-of-sys-xcpt s = (if s = NullPointer then 0 else if s = ClassCast then 1
else if s = OutOfMemory then 2 else (arbitrary::nat))

lemma [*code*]:
extractTy (Ty ex' tp, ex) = (if ex'=ex then [tp] else [])

0.0.2 Weakest Precondition Operator

0.0.3 Wellformedness

consts *prefix* :: *('a list × 'a list) ⇒ bool*

recdef *prefix measure* (λ (*xs*,*ys*). *length xs*)
prefix ([],*ys*) = *True*
prefix (*x*#*xs*,[]) = *False*
prefix (*x*#*xs*,*y*#*ys*) = ((*x* = *y*) ∧ *prefix* (*xs*,*ys*))

lemma *prefix-append*:
⋀ *ys*. *prefix* (*xs*,*ys*) = (∃ *zs*. *ys* = *xs* @ *zs*)
consts
checkPosS :: *jbc-prog ⇒ (pos list) ⇒ StrBool*

constdefs *toString*::*pos ⇒ string*
toString ≡ (λ (*C*,*M*,*pc*::*nat*). *''(''@C@'',''@M@'',''@BCVExec.toString pc@'')''*)

primrec
checkPosS Π [] = *TRUE*
checkPosS 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*)
⟶ *anF* Π *p'* ≠ *None*) ∧ *p' mem* (*domC* Π) ∧
(*p' mem scsn* ⟶ *handlesEx* (*fst* Π) *p'* = *None*))
(*scsn* @ *scse*) ∧ (*set scse* ⊂ *set* (*domC* Π))) ∧ *throwChk* (Π,*cmd* Π *p*,*anF* Π *p*,*p*) ∧
invokeChk (Π,*cmd* Π *p*,*anF* Π *p*,*p*)
then (*checkPosS* Π *ps*) *else FALSE* (*''Error at position::''*@(*toString p*)))

constdefs *wfS*::*jbc-prog ⇒ StrBool*
wfS Π == (*let cp* = (*checkPosS* Π (*domC* Π))
in (*if* (*cp* ≠ *TRUE*)

then cp
else (if (¬ (checkExTables Π))
then FALSE ''Exception tables malformed''
else (if (¬ (distinct (classnames (fst Π))))
then FALSE ''Classnames not distinct''
else (if (¬ (distinct (methodnames (fst Π))))
then FALSE ''Methodnames not distinct''
else (if (¬ (prefix (SystemClasses,fst Π)))
then FALSE ''Systemclasses are expected in front.''
else (if (¬ (ipc Π *mem (domC* Π)))
then FALSE ''initial position missing''
else (let wfTy-P = (wf-jvm-prog-phi (pTy (fst Π)) *(fst* Π))
in (if ¬ wfTy-P
then FALSE ''program not welltyped''
else (if (¬ (fst (snd (method (fst Π*) (fst (ipc* Π*)) (fst (snd (ipc* Π*))))) = []))*
then FALSE ''main method malformed''
else TRUE))))))))))

lemma *wf-eq* [*code*]:
wf Pi = (checkPos Pi (domC Pi) ∧ checkExTables Pi ∧
distinct (classnames (fst Pi)) ∧
distinct (methodnames (fst Pi)) ∧
prefix (SystemClasses,fst Pi) ∧ (ipc Pi mem (domC Pi))
∧ wf-jvm-prog-phi (pTy (fst Pi)) (fst Pi) ∧ fst (snd (method (fst Pi) (fst (ipc Pi)) (fst (snd (ipc Pi))))) = []
)

0.0.4 Setting up the verification environment

lemma *safeP-frs-length*:
⟦ *s ∈ safeP* Π; *fst (snd (fst s)) ≠ fst (snd (ipc* Π*))* ⟧ ⟹ *Suc 0 < length (snd (snd (fst (snd s))))*
lemma *evalE-Call-Cn*:
Suc 0 < length (snd (snd (fst (snd s)))) ⟹ evalE Π *s (Call (Cn v)) = v*
lemma *evalE-Call-Neg*:
Suc 0 < length (snd (snd (fst (snd s)))) ⟹ evalE Π *s (Call (Neg ex)) = Bool (¬ the-Bool (evalE* Π *s (Call ex)))*
lemma *evalE-Call-Imp*:
Suc 0 < length (snd (snd (fst (snd s)))) ⟹
evalE Π *s (Call (Imp ex ex')) = Bool (the-Bool (evalE* Π *s (Call ex)) ⟶ the-Bool (evalE* Π *s (Call ex')))*

lemma *evalE-Call-And*:
Suc 0 < length (snd (snd (fst (snd s)))) ⟹

$evalE\ \Pi\ s\ (Call\ (And\ es)) = Bool\ (list\text{-}all\ (\lambda\ ex.\ the\text{-}Bool\ (evalE\ \Pi\ s\ (Call\ ex)))\ es)$

lemma *evalE-Call-Num*:
$Suc\ 0 < length\ (snd\ (snd\ (fst\ (snd\ s)))) \Longrightarrow$
$evalE\ \Pi\ s\ (Call\ (Num\ ex\ no\ ex'))\ =\ liftI\ (numop\ no)\ (evalE\ \Pi\ s\ (Call\ ex))\ (evalE\ \Pi\ s\ (Call\ ex'))$

lemma *evalE-Call-Rel*:
$Suc\ 0 < length\ (snd\ (snd\ (fst\ (snd\ s)))) \Longrightarrow evalE\ \Pi\ s\ (Call\ (Rel\ ex\ ro\ ex')) = liftR\ (relop\ ro)\ (evalE\ \Pi\ s\ (Call\ ex))\ (evalE\ \Pi\ s\ (Call\ ex'))$
lemma *evalE-Call-Eq*:
$Suc\ 0 < length\ (snd\ (snd\ (fst\ (snd\ s)))) \Longrightarrow evalE\ \Pi\ s\ (Call\ (Eq\ ex\ ex')) = Bool\ (evalE\ \Pi\ s\ (Call\ ex) = (evalE\ \Pi\ s\ (Call\ ex')))$

lemma *triple-simp*:
$(fst\ x,\ fst\ (snd\ x),\ snd\ (snd\ x)) = x$
lemma *evalE-Call-Forall*:
$Suc\ 0 < length\ (snd\ (snd\ (fst\ (snd\ s)))) \Longrightarrow evalE\ \Pi\ s\ (Call\ (Forall\ v\ ex)) =$
$(let\ (p,\ \sigma,\ e) = s$
$in\ Bool\ (\forall\ v'.\ the\text{-}Bool\ (evalE\ \Pi\ (p,\ \sigma,\ e(\!|lv := (lv\ e)(v := v')|\!))\ (Call\ ex))))$
lemma *evalE-Ty-Integer*:
$evalE\ \Pi\ s\ (Ty\ ex\ Integer) = Bool\ (\exists\ x.\ evalE\ \Pi\ s\ ex = Intg\ x)$
lemma *evalE-FrNr'*:
$evalE\ \Pi\ s\ FrNr = Intg\ (int\ (length\ (snd\ (snd\ (fst\ (snd\ s))))))$

lemma *evalE-And'*:
$evalE\ \Pi\ s\ (And\ es) = Bool\ (list\text{-}all\ (\lambda\ ex.\ the\text{-}Bool\ (evalE\ \Pi\ s\ ex))\ es)$
lemma *evalE-Pos'*:
$the\text{-}Bool\ (evalE\ \Pi\ s\ (Pos\ p)) \Longrightarrow fst\ s = p$

lemmas *evalE-simps = evalEs-empty evalEs-cons evalE-Cn evalE-Num evalE-Imp evalE-Neg evalE-Rel evalE-Eq evalE-Forall*
evalE-Neg evalE-And' evalE-Ty-Integer evalE-Ite evalE-FrNr'
evalE-Call-Cn evalE-Call-And evalE-Call-Neg evalE-Call-Imp evalE-Call-Num evalE-Call-Rel evalE-Call-Eq
evalE-Call-Forall

lemmas *sem-simps = valid-def evalE-simps liftR-def relop-def liftI-def numop-def the-Intg.simps the-Bool.simps val.simps*

end