1 Jinja Assertion Language

theory *Form* = *FiniteMap* + *Value*:

1.1 Syntax of formulae

types *var* = *nat*

types
— The position (C,M,n) identifies the n'th instruction in method M of class C.
pos = *cname* × *mname* × *nat*

datatype *expr* = *Rg var* | *St var* | *Lv var* | *Cn val* |
NewA nat | *Gf vname cname expr* | *FrNr* |
Num expr num-op expr | *Rel expr rel-op expr* |
Ite expr expr expr | *Eq expr expr* |
Neg expr | *Imp expr expr* | *And expr list* | *Forall var expr* |
Ty expr ty | *Pos pos* |
Call expr | *Catch cname expr*

syntax *Add-* :: *expr* ⇒ *expr* ⇒ *expr* (**infix** ⊕ *65*)
translations *Add- e1 e2* == *Num e1 Add e2*
syntax *Sub-* :: *expr* ⇒ *expr* ⇒ *expr* (**infix** ⊖ *65*)
translations *Sub- e1 e2* == *Num e1 Sub e2*
syntax *Mult-* :: *expr* ⇒ *expr* ⇒ *expr* (**infix** ⊗ *65*)
translations *Mult- e1 e2* == *Num e1 Mult e2*
syntax *Ite-* ::*expr* ⇒ *expr* ⇒ *expr* ⇒ *expr* ((*IF - THEN - ELSE -*) [*60*,*60*,*60*]*60*)
translations *Ite- e1 e2 e3* == *Ite e1 e2 e3*
syntax *Eq-* :: *expr* ⇒ *expr* ⇒ *expr* (**infix** ≐ *65*)
translations *Eq- e1 e2* == *Eq e1 e2*
syntax *Imp-* :: *expr* ⇒ *expr* ⇒ *expr* (**infix** ⊃ *65*)
translations *Imp- e1 e2* == *Imp e1 e2*
syntax *Leq-* :: *expr* ⇒ *expr* ⇒ *expr* (**infix** ⪯ *65*)
translations *Leq- e1 e2* == *Rel e1 Leq e2*
syntax *Less-* :: *expr* ⇒ *expr* ⇒ *expr* (**infix** ≺ *65*)
translations *Less- e1 e2* == *Rel e1 Less e2*
syntax *Req-* :: *expr* ⇒ *expr* ⇒ *expr* (**infix** ≜ *65*)
translations *Req- e1 e2* == *Rel e1 Eql e2*
syntax (*latex*) *Geq-* :: *expr* ⇒ *expr* ⇒ *expr* (**infix** ⪰ *65*)
translations *Geq- e1 e2* == *Rel e1 Geq e2*
syntax (*latex*) *Grtr-* :: *expr* ⇒ *expr* ⇒ *expr* (**infix** ≻ *65*)
translations *Grtr- e1 e2* == *Rel e1 Grtr e2*
syntax *TT* :: *expr*

translations *TT* == *Cn* (*Bool True*)
syntax *FF* :: *expr*
translations *FF* == *Cn* (*Bool False*)Abbreviations**constdefs** *Or*::*expr list* ⇒ *expr*
Or exs ≡ *Neg* (*And* (*map Neg exs*))

— A wrong typed expression evaluating to None
syntax *none* :: *expr*
translations *none* == *Num* (*Cn* (*Intg 0*)) *Add* (*Cn* (*Bool True*))

— Note: Before we turned none into a syntax translation, we modelled it as a constant and nonedef was used in proofs. We do not want to remove nonedef from proofs, because we may need to go back to the old definition of none in case syntax translations turn out to be too inefficient. For the meantime nonedef becomes a dummy lemma.

lemma *none-def*:
none ≠ *Num* (*Cn* (*Intg 1*)) *Add* (*Cn* (*Bool True*))
constdefs *not-none*::*expr* ⇒ *expr*
not-none ex ≡ *Neg* (*Eq ex none*)

1.2 Extracting information from expressions

consts
$foldE{::}(expr \times {'a} \Rightarrow {'a}) \Rightarrow ({'a} \Rightarrow {'a} \Rightarrow {'a}) \Rightarrow {'a} \Rightarrow expr \Rightarrow {'a}$
$foldEs{::}(expr \times {'a} \Rightarrow {'a}) \Rightarrow ({'a} \Rightarrow {'a} \Rightarrow {'a}) \Rightarrow {'a} \Rightarrow expr\ list \Rightarrow {'a}$

primrec
$foldEs\ f\ c\ a\ [] = a$
$foldEs\ f\ c\ a\ (ex\#exs) = c\ (foldE\ f\ c\ a\ ex)\ (foldEs\ f\ c\ a\ exs)$
$foldE\ f\ c\ a\ (And\ exs) = f\ (And\ exs,\ foldEs\ f\ c\ a\ exs)$
$foldE\ f\ c\ a\ (Rg\ k) = f\ (Rg\ k,\ a)$
$foldE\ f\ c\ a\ (St\ k) = f\ (St\ k,\ a)$
$foldE\ f\ c\ a\ (Lv\ k) = f\ (Lv\ k,\ a)$
$foldE\ f\ c\ a\ (Cn\ v) = f\ (Cn\ v,\ a)$
$foldE\ f\ c\ a\ (NewA\ n) = f\ (NewA\ n,\ a)$
$foldE\ f\ c\ a\ (Gf\ F\ C\ ex) = f\ (Gf\ F\ C\ ex,\ foldE\ f\ c\ a\ ex)$
$foldE\ f\ c\ a\ FrNr = f\ (FrNr,a)$
$foldE\ f\ c\ a\ (Num\ e\ no\ e') = f\ (Num\ e\ no\ e',\ c\ (foldE\ f\ c\ a\ e)\ (foldE\ f\ c\ a\ e'))$
$foldE\ f\ c\ a\ (Rel\ e1\ ro\ e2) = f\ (Rel\ e1\ ro\ e2,\ c\ (foldE\ f\ c\ a\ e1)\ (foldE\ f\ c\ a\ e2))$
$foldE\ f\ c\ a\ (Eq\ e1\ e2) = f\ (Eq\ e1\ e2,\ c\ (foldE\ f\ c\ a\ e1)\ (foldE\ f\ c\ a\ e2))$
$foldE\ f\ c\ a\ (Neg\ ex) = f\ (Neg\ ex,foldE\ f\ c\ a\ ex)$
$foldE\ f\ c\ a\ (Imp\ e1\ e2) = f\ (Imp\ e1\ e2,c\ (foldE\ f\ c\ a\ e1)\ (foldE\ f\ c\ a\ e2))$
$foldE\ f\ c\ a\ (Forall\ v\ ex) = f\ (Forall\ v\ ex,\ foldE\ f\ c\ a\ ex)$
$foldE\ f\ c\ a\ (Ite\ e1\ e2\ e3) = f\ (Ite\ e1\ e2\ e3,\ c\ (c\ (foldE\ f\ c\ a\ e1)\ (foldE\ f\ c\ a\ e2))\ (foldE\ f\ c\ a\ e3))$
$foldE\ f\ c\ a\ (Ty\ ex\ tp) = f\ (Ty\ ex\ tp,foldE\ f\ c\ a\ ex)$
$foldE\ f\ c\ a\ (Pos\ p) = f\ (Pos\ p,\ a)$
$foldE\ f\ c\ a\ (Call\ ex) = f\ (Call\ ex,\ foldE\ f\ c\ a\ ex)$
$foldE\ f\ c\ a\ (Catch\ cn\ ex) = f\ (Catch\ cn\ ex,\ foldE\ f\ c\ a\ ex)$

consts $noCC{::}expr \times {'a}\ list \Rightarrow {'a}\ list$

recdef $noCC$ $\{\}$
$noCC\ (Call\ ex,\ as) = []$
$noCC\ (Catch\ cn\ ex,\ as) = []$
$noCC\ (oth,\ as) = as$

consts $stkId{::}(expr \times var\ list) \Rightarrow var\ list$
recdef $stkId$ $\{\}$
$stkId\ (St\ k,\ as) = [k]$
$stkId\ (oth,\ as) = as$

constdefs
$stkIds{::}expr \Rightarrow var\ list$
$stkIds\ ex \equiv foldE\ (\lambda(ex,as).\ noCC\ (ex,stkId\ (ex,as)))\ (op\ @)\ []\ ex$

consts $rgId$::($expr \times var\ list$) $\Rightarrow var\ list$
recdef $rgId$ {}
$rgId\ (Rg\ k,\ as) = [k]$
$rgId\ (oth,\ as) = as$

constdefs
$rgIds$::$expr \Rightarrow nat\ list$
$rgIds\ ex \equiv foldE\ (\lambda(ex,as).\ noCC\ (ex,rgId\ (ex,as)))\ (op\ @)\ []\ ex$

consts $gfEx$::($vname \times cname \times expr$) $\Rightarrow expr\ list$

recdef $gfEx$ {}
$gfEx\ (F,C,Gf\ F'\ C'\ ex) = (if\ F{=}F' \wedge C{=}C'\ then\ [ex]\ else\ [])$
$gfEx\ (F,C,oth) = []$

constdefs
$getGfEx$::$vname \Rightarrow cname \Rightarrow expr \Rightarrow expr\ list$
$getGfEx\ F\ C\ ex \equiv foldE\ (\lambda(ex,as).\ as\ @\ gfEx(F,C,ex))\ (op\ @)\ []\ ex$

consts $newEx$::$expr \Rightarrow nat\ list$

recdef $newEx$ {}
$newEx\ (NewA\ n) = [n]$
$newEx\ oth = []$

constdefs
$getNewEx$::$expr \Rightarrow nat\ list$
$getNewEx\ ex \equiv foldE\ (\lambda(ex,as).\ newEx\ ex\ @\ as)\ (op\ @)\ []\ ex$

consts $callEx$::$expr \Rightarrow expr\ list$
recdef $callEx$ {}
$callEx\ (Call\ ex) = [ex]$
$callEx\ oth = []$

constdefs
$getCallEx$::$expr \Rightarrow expr\ list$
$getCallEx\ ex \equiv foldE\ (\lambda(ex,as).\ callEx\ ex\ @\ noCC\ (ex,as))\ (op\ @)\ []\ ex$

consts $catchEx$::$expr \Rightarrow (cname \times expr)\ list$
recdef $catchEx$ {}
$catchEx\ (Catch\ cn\ ex) = [(cn,ex)]$
$catchEx\ oth = []$

constdefs
$getCatchEx$::$expr \Rightarrow (cname \times expr)\ list$

$getCatchEx\ ex \equiv foldE\ (\lambda(ex,as).\ catchEx\ ex\ @\ noCC\ (ex,as))\ (op\ @)\ []\ ex$

consts $posEx::expr \Rightarrow pos\ list$
recdef $posEx\ \{\}$
$posEx\ (Pos\ p) = [p]$
$posEx\ oth = []$

constdefs
$getPosEx::expr \Rightarrow pos\ list$
$getPosEx\ ex \equiv foldE\ (\lambda(ex,as).\ posEx\ ex\ @\ noCC\ (ex,as))\ (op\ @)\ []\ ex$

constdefs
$subExpr::expr \Rightarrow expr\ list$
$subExpr\ ex \equiv foldE\ (\lambda(ex,as).\ ex\#noCC(ex,as))\ (op\ @)\ []\ ex$

consts $logV::expr \Rightarrow var\ list$
recdef $logV\ \{\}$
$logV\ (Lv\ k) = [k]$
$logV\ (Forall\ k\ ex) = [k]$
$logV\ oth = []$

constdefs
$logVarsE :: expr \Rightarrow var\ list$
$logVarsE\ ex \equiv foldE\ (\lambda(ex,as).\ logV\ ex\ @\ as)\ (op\ @)\ []\ ex$

consts
$bindsVar ::expr \Rightarrow var\ option$

recdef $bindsVar\ \{\}$
$bindsVar\ (Forall\ v\ ex) = Some\ v$
$bindsVar\ oth = None$

constdefs
$freeLvs:: expr \Rightarrow var\ list$
$freeLvs\ ex \equiv foldE\ (\lambda(ex,as).\ (case\ bindsVar\ ex\ of\ None \Rightarrow as@(logV\ ex)$
$\qquad\qquad | \ Some\ v \Rightarrow [x{\in}as.\ x{\neq}v]))\ (op\ @)\ []\ ex$

consts
$extractEq::(expr \times expr) \Rightarrow expr\ option$

recdef $extractEq\ measure\ (\lambda\ (ex,ex').\ size\ ex)$
$extractEq\ (And\ [],ex) = None$
$extractEq\ (And\ (ex'\#exs),ex) = (case\ extractEq\ (ex',ex)$
$\ of\ None \Rightarrow extractEq\ (And\ exs,ex)\ |\ Some\ ex'' \Rightarrow Some\ ex'')$
$extractEq\ (Eq\ ex'\ ex'',ex) = (if\ ex'{=}ex\ then\ Some\ ex''\ else\ None)$

$extractEq\ (ex',ex) = None$

consts
$isNegTy$::$(expr \times expr) \Rightarrow bool$

recdef $isNegTy\ measure\ (\lambda\ (ex',ex).\ size\ ex')$
$isNegTy\ (Neg\ (Ty\ ex\ tp),ex') = (ex = ex')$
$isNegTy\ (oth,ex') = False$

consts
$extractTy$::$(expr \times expr) \Rightarrow ty\ list$

— extractTy (ex,ex') lists the possible types of ex' in a state that satisfies ex

recdef $extractTy\ measure\ (\lambda\ (ex',ex).\ size\ ex')$
$extractTy\ (And\ [],ex) = []$
$extractTy\ (And\ (ex'\#exs),ex) = extractTy\ (ex',ex)\ @\ extractTy\ (And\ exs,ex)$
$extractTy\ (Ty\ ex'\ tp,ex) = (if\ ex'{=}ex\ then\ [tp]\ else\ [])$
$extractTy\ (Neg\ (And\ []),ex) = []$
$extractTy\ (Neg\ (And\ (Neg\ ex'\#exs)),ex) = (if\ (list\text{-}all\ (\lambda\ ex''.\ isNegTy\ (ex'',ex))\ (Neg\ ex'\#exs))$
$then\ (extractTy\ (ex',ex)\ @\ extractTy\ (Neg\ (And\ exs),ex))\ else\ [])$
$extractTy\ (oth,\ ex) = []$
(**hints** $cong\ del$: $option.weak\text{-}case\text{-}cong$)

constdefs $eqExMps$::$(expr\ \tilde{}\tilde{}{>}\ expr) \Rightarrow (expr\ \tilde{}\tilde{}{>}\ expr) \Rightarrow expr \Rightarrow bool$
$eqExMps\ em\ em'\ ex \equiv foldE\ (\lambda(ex,a).\ em\ ?\ ex = em'\ ?\ ex \wedge list\text{-}all\ (\lambda x.\ x)\ (noCC\ (ex,[a])))\ (op\ \wedge)$
$True\ ex$

datatype $heapexpr = GF\ vname\ cname\ expr \mid TY\ expr\ ty$

consts $heapEx$::$expr \Rightarrow heapexpr\ list$

recdef $heapEx\ \{\}$
$heapEx\ (Gf\ F\ C\ ex) = [GF\ F\ C\ ex]$
$heapEx\ (Ty\ ex\ ty) = [TY\ ex\ ty]$
$heapEx\ oth\ \ \ \ \ \ = []$

constdefs
$getHeapEx$::$expr \Rightarrow heapexpr\ list$
$getHeapEx\ ex \equiv foldE\ (\lambda(ex,as).\ as\ @\ heapEx\ ex)\ (op\ @)\ []\ ex$

1.3 Substitution Function

consts

$substE$::($expr$ ~~> $expr$) $\Rightarrow$ $expr$ $\Rightarrow$ $expr$
$substEs$::($expr$ ~~> $expr$) $\Rightarrow$ $expr\ list$ $\Rightarrow$ $expr\ list$

primrec

substEs-Nil: *substEs em* [] = []
substEs-Cons: *substEs em* (*ex*#*exs*) = (*substE em ex*)#(*substEs em exs*)

substE-Rg: *substE em* (*Rg k*) = *em* $?_=$ (*Rg k*)
substE-St: *substE em* (*St k*) = *em* $?_=$ (*St k*)
substE-Lv: *substE em* (*Lv k*) = *em* $?_=$ (*Lv k*)
substE-Cn: *substE em* (*Cn tv*) = *em* $?_=$ (*Cn tv*)
substE-NewA: *substE em* (*NewA n*) = *em* $?_=$ (*NewA n*)
substE-Gf: *substE em* (*Gf F C ex*) = (*case em ?* (*Gf F C ex*)
    of None $\Rightarrow$ *Gf F C* (*substE em ex*)
    | *Some ex'* $\Rightarrow$ *ex'*)
substE-FrNr: *substE em FrNr* = *em* $?_=$ *FrNr*
substE-Num: *substE em* (*Num e1 no e2*) = (*case em ?* (*Num e1 no e2*)
    of None $\Rightarrow$ *Num* (*substE em e1*) *no* (*substE em e2*)
    | *Some ex'* $\Rightarrow$ *ex'*)

substE-Rel: *substE em* (*Rel e1 ro e2*) = (*case em ?* (*Rel e1 ro e2*)
    of None $\Rightarrow$ *Rel* (*substE em e1*) *ro* (*substE em e2*)
    | *Some ex'* $\Rightarrow$ *ex'*)

substE-Ite: *substE em* (*Ite b t e*) = (*case em ?* (*Ite b t e*)
 of None $\Rightarrow$ *Ite* (*substE em b*) (*substE em t*) (*substE em e*)
| *Some ex'* $\Rightarrow$ *ex'*)

substE-Eq: *substE em* (*Eq e1 e2*) = (*case em ?* (*Eq e1 e2*)
    of None $\Rightarrow$ *Eq* (*substE em e1*) (*substE em e2*)
    | *Some ex'* $\Rightarrow$ *ex'*)

substE-Neg: *substE em* (*Neg ex*) = (*case em ?* (*Neg ex*)
    of None $\Rightarrow$ *Neg* (*substE em ex*)
    | *Some ex'* $\Rightarrow$ *ex'*)

substE-Imp: *substE em* (*Imp e1 e2*) = (*case em ?* (*Imp e1 e2*)
    of None $\Rightarrow$ *Imp* (*substE em e1*) (*substE em e2*)
    | *Some ex'* $\Rightarrow$ *ex'*)

substE-And: *substE em* (*And exs*) = *And* (*substEs em exs*)

substE-Forall: *substE em* (*Forall v ex*) = (*case em ?* (*Forall v ex*)

$\quad$ *of None* $\Rightarrow$ *Forall v (substE em ex)*
$\quad$ | *Some ex'* $\Rightarrow$ *ex'*)

substE-Ty: *substE em (Ty ex tp) = (case em ? (Ty ex tp)*
$\quad$ *of None* $\Rightarrow$ *Ty (substE em ex) tp*
$\quad$ | *Some ex'* $\Rightarrow$ *ex'*)

substE-Pos: *substE em (Pos p) =* $\;$ *em* $?_=$ *(Pos p)*

substE-Call: *substE em (Call ex) = em* $?_=$ *(Call ex)*

substE-Catch: *substE em (Catch cn ex) = em* $?_=$ *(Catch cn ex)*

1.4 Auxiliary Lemmas

lemma *expr-induct*:
$[\![\bigwedge nat.\ P1\ (Rg\ nat);$
$\bigwedge nat.\ P1\ (St\ nat);$
$\bigwedge nat.\ P1\ (Lv\ nat);$
$\bigwedge val.\ P1\ (Cn\ val);$
$\bigwedge\ n.\ P1\ (NewA\ n);$
$\bigwedge list1\ list2\ expr.\ P1\ expr \Longrightarrow P1\ (Gf\ list1\ list2\ expr);$
$P1\ FrNr;$
$\bigwedge expr1\ num\text{-}op\ expr2.\ [\![P1\ expr1;\ P1\ expr2]\!] \Longrightarrow P1\ (Num\ expr1\ num\text{-}op\ expr2);$
$\bigwedge expr1\ rel\text{-}op\ expr2.\ [\![P1\ expr1;\ P1\ expr2]\!] \Longrightarrow P1\ (Rel\ expr1\ rel\text{-}op\ expr2);$
$\bigwedge expr1\ expr2\ expr3.\ [\![P1\ expr1;\ P1\ expr2;\ P1\ expr3]\!] \Longrightarrow$
$P1\ (IF\ expr1\ THEN\ expr2\ ELSE\ expr3);$
$\bigwedge expr1\ expr2.\ [\![P1\ expr1;\ P1\ expr2]\!] \Longrightarrow P1\ (expr1 \doteq expr2);$
$\bigwedge expr.\ P1\ expr \Longrightarrow P1\ (Neg\ expr);$
$\bigwedge expr1\ expr2.\ [\![P1\ expr1;\ P1\ expr2]\!] \Longrightarrow P1\ (expr1 \supset expr2);$
$\bigwedge nat\ expr.\ P1\ expr \Longrightarrow P1\ (Forall\ nat\ expr);$
$\bigwedge expr\ ty.\ P1\ expr \Longrightarrow P1\ (Ty\ expr\ ty);$
$\bigwedge x.\ P1\ (Pos\ x);$
$\bigwedge expr.\ P1\ expr \Longrightarrow P1\ (Call\ expr);$
$\bigwedge list\ expr.\ P1\ expr \Longrightarrow P1\ (Catch\ list\ expr);$
$\bigwedge expr\ es.\ [\![\forall\, ex \in set\ es.\ P1\ ex]\!] \Longrightarrow P1\ (And\ es)]\!]$
$\Longrightarrow P1\ expr$

lemma *expr-induct'*:
$[\![\bigwedge nat.\ P1\ (Rg\ nat);$
$\bigwedge nat.\ P1\ (St\ nat);$
$\bigwedge nat.\ P1\ (Lv\ nat);$
$\bigwedge val.\ P1\ (Cn\ val);$
$\bigwedge\ n.\ P1\ (NewA\ n);$
$\bigwedge list1\ list2\ expr.\ P1\ expr \Longrightarrow P1\ (Gf\ list1\ list2\ expr);$
$P1\ FrNr;$

$\bigwedge expr1\ num\text{-}op\ expr2.\ [\![P1\ expr1;\ P1\ expr2]\!] \Longrightarrow P1\ (Num\ expr1\ num\text{-}op\ expr2);$
$\bigwedge expr1\ rel\text{-}op\ expr2.\ [\![P1\ expr1;\ P1\ expr2]\!] \Longrightarrow P1\ (Rel\ expr1\ rel\text{-}op\ expr2);$
$\bigwedge expr1\ expr2\ expr3.\ [\![P1\ expr1;\ P1\ expr2;\ P1\ expr3]\!] \Longrightarrow P1\ (IF\ expr1\ THEN\ expr2\ ELSE\ expr3);$
$\bigwedge expr1\ expr2.\ [\![P1\ expr1;\ P1\ expr2]\!] \Longrightarrow P1\ (expr1 \doteq expr2);$
$\bigwedge expr.\ P1\ expr \Longrightarrow P1\ (Neg\ expr);$
$\bigwedge expr1\ expr2.\ [\![P1\ expr1;\ P1\ expr2]\!] \Longrightarrow P1\ (expr1 \supset expr2);$
$\bigwedge nat\ expr.\ P1\ expr \Longrightarrow P1\ (Forall\ nat\ expr);$
$\bigwedge expr\ ty.\ P1\ expr \Longrightarrow P1\ (Ty\ expr\ ty);$
$\bigwedge x.\ P1\ (Pos\ x);$
$\bigwedge expr.\ P1\ expr \Longrightarrow P1\ (Call\ expr);$
$\bigwedge list\ expr.\ P1\ expr \Longrightarrow P1\ (Catch\ list\ expr);$
$\bigwedge expr\ ex\ es.\ [\![\ P1\ ex;\ P1\ (And\ es)]\!] \Longrightarrow P1\ (And\ (ex\#es));$
$P1\ (And\ [])]\!]$
$\Longrightarrow P1\ expr$

consts *parts*::*expr* ⇒ *expr list*
primrec
parts (*Rg k*) = []
parts (*St k*) = []
parts (*Lv k*) = []
parts (*Cn v*) = []
parts (*NewA n*) = []
parts (*Gf F C ex*) = [*ex*]
parts FrNr = []
parts (*Num e1 no e2*) = [*e1*,*e2*]
parts (*Rel e1 ro e2*) = [*e1*,*e2*]
parts (*Ite e1 e2 e3*) = [*e1*,*e2*,*e3*]
parts (*Eq e1 e2*) = [*e1*,*e2*]
parts (*Neg ex*) = [*ex*]
parts (*Imp e1 e2*) = [*e1*,*e2*]
parts (*And es*) = *es*
parts (*Forall k ex*) = [*ex*]
parts (*Ty ex tp*) = [*ex*]
parts (*Pos p*) = []
parts (*Call ex*) = []
parts (*Catch X ex*) = []

lemma *getExpr-refl*: *ex* ∈ *set* (*subExpr ex*)

lemma *subExpr-And-cons*:
subExpr (*And* (*ex*#*es*)) =
[*And* (*ex*#*es*)] @ (*subExpr ex* @ (*tl* (*subExpr* (*And es*))))

lemma *subExpr-And-in-list*:
ex ∈ *set es* ⟹ *ex* ∈ *set* (*tl* (*subExpr* (*And es*)))

lemma *subExpr-And-map*:
$subExpr\ (And\ es) = And\ es\ \#\ (concat\ (map\ subExpr\ es))$
lemma *subExpr-Gf*:
$\bigwedge ex'\ ex''.\ [\![\ ex' \in set\ (subExpr\ ex);\ ex'' \in set\ (parts\ ex')\ ]\!]$
$\Longrightarrow ex'' \in set\ (subExpr\ ex)$

lemma *subExpr-rgIds*:
$Rg\ k \in set\ (subExpr\ ex) \Longrightarrow k \in set\ (rgIds\ ex)$

lemma *subExpr-stkIds*:
$St\ k \in set\ (subExpr\ ex) \Longrightarrow k \in set\ (stkIds\ ex)$

lemma *subExpr-NewEx*:
$NewA\ n \in set\ (subExpr\ ex) \Longrightarrow n \in set\ (getNewEx\ ex)$

lemma *subExpr-getCatchEx*:
$Catch\ X\ ex \in set\ (subExpr\ ex') \Longrightarrow (X,ex) \in set\ (getCatchEx\ ex')$

lemma *subExpr-getCallEx*:
$Call\ ex \in set\ (subExpr\ ex') \Longrightarrow ex \in set\ (getCallEx\ ex')$

lemma *subExpr-getGfEx*:
$Gf\ F\ C\ ex\ \in set\ (subExpr\ ex') \Longrightarrow ex \in set\ (getGfEx\ F\ C\ ex')$

lemma *subExpr-getHeapEx*:
$Gf\ F\ C\ ex\ \in set\ (subExpr\ ex') \Longrightarrow GF\ F\ C\ ex \in set\ (getHeapEx\ ex')$

lemma *subExpr-getHeapEx-TY*:
$Ty\ ex\ ty \in set\ (subExpr\ ex') \Longrightarrow TY\ ex\ ty \in set\ (getHeapEx\ ex')$

lemma *eqExMps-Call*:
$eqExMps\ em\ em'\ (Call\ ex) = (em\ ?\ (Call\ ex) = em'\ ?\ (Call\ ex))$

lemma *eqExMps-Catch*:
$eqExMps\ em\ em'\ (Catch\ X\ ex) = (em\ ?\ (Catch\ X\ ex) = em'\ ?\ (Catch\ X\ ex))$

lemma *eqExMps-And'*:
$eqExMps\ em\ em'\ (And\ es) = ((em\ ?\ (And\ es) = em'\ ?\ (And\ es)) \land$
$(\forall ex \in set\ es.\ eqExMps\ em\ em'\ ex))$
lemma *substE-empty*:
$substE\ []\ ex = ex$

lemma *substEs-map*:
$substEs\ em\ es = map\ (substE\ em)\ es$

lemma *foldEs-append*:
$[\![\ \forall\, ex.\ c\ a\ ex = ex;\ \forall\ \ ex\ ex'\ ex''.\ c\ ex\ (c\ ex'\ ex'') = c\ (c\ ex\ ex')\ ex''\ ]\!]$
$\Longrightarrow (foldEs\ f\ c\ a\ (es@es')) = c\ (foldEs\ f\ c\ a\ es)\ (foldEs\ f\ c\ a\ es')$

lemma *substE-eq*:
$eqExMps\ em\ em'\ ex \Longrightarrow substE\ em\ ex = substE\ em'\ ex$

lemma *eqExMps-ren*:
$[\![\ ex = ex';\ eqExMps\ em\ em'\ ex']\!] \Longrightarrow eqExMps\ em\ em'\ ex$
end