Unspecified Veriopt Theory

July 3, 2021

Contents

1 Data-flow Semantics

theory *IRTreeEval*
  imports
    Graph.IRGraph
begin

We define a tree representation of data-flow nodes, as an abstraction of the graph view.

Data-flow trees are evaluated in the context of a method state (currently called MapState in the theories for historical reasons).

The method state consists of the values for each method parameter, references to method parameters use an index of the parameter within the parameter list, as such we store a list of parameter values which are looked up at parameter references.

The method state also stores a mapping of node ids to values. The contents of this mapping is calculates during the traversal of the control flow graph. As a concrete example, as the *SignedDivNode* can have side-effects (during division by zero), it is treated as part of the control-flow, since the data-flow phase is specified to be side-effect free. As a result, the control-flow semantics for *SignedDivNode* calculates the value of a node and maps the node identifier to the value within the method state. The data-flow semantics then just reads the value stored in the method state for the node.

type-synonym *MapState* = *ID* ⇒ *Value*
type-synonym *Params* = *Value list*

definition *new-map-state* :: *MapState* **where**
  new-map-state = (λ*x*. *UndefVal*)

fun *val-to-bool* :: *Value* ⇒ *bool* **where**
  val-to-bool (*IntVal32 val*) = (*if val* = *0 then False else True*) |
  val-to-bool v = *False*

fun *bool-to-val* :: *bool* ⇒ *Value* **where**
  bool-to-val True = (*IntVal32 1*) |
  bool-to-val False = (*IntVal32 0*)

fun *find-index* :: *'a* ⇒ *'a list* ⇒ *nat* **where**
  find-index - [] = *0* |
  find-index v (*x* # *xs*) = (*if* (*x*=*v*) *then 0 else find-index v xs* + *1*)

fun *phi-list* :: *IRGraph* ⇒ *ID* ⇒ *ID list* **where**
  phi-list g nid =
    (*filter* (λ*x*.(*is-PhiNode* (*kind g x*)))
      (*sorted-list-of-set* (*usages g nid*)))

fun *input-index* :: *IRGraph* ⇒ *ID* ⇒ *ID* ⇒ *nat* **where**
  input-index g n n' = *find-index n'* (*inputs-of* (*kind g n*))

fun *phi-inputs* :: *IRGraph* ⇒ *nat* ⇒ *ID list* ⇒ *ID list* **where**
  phi-inputs g i nodes = (*map* (λ*n*. (*inputs-of* (*kind g n*))!(*i* + *1*)) *nodes*)

fun *set-phis* :: *ID list* ⇒ *Value list* ⇒ *MapState* ⇒ *MapState* **where**
  set-phis [] [] *m* = *m* |
  set-phis (*nid* # *xs*) (*v* # *vs*) *m* = (*set-phis xs vs* (*m*(*nid* := *v*))) |
  set-phis [] (*v* # *vs*) *m* = *m* |
  set-phis (*x* # *xs*) [] *m* = *m*

fun *find-node-and-stamp* :: *IRGraph* ⇒ (*IRNode* × *Stamp*) ⇒ *ID option* **where**

find-node-and-stamp g (*n*,*s*) =
  find (λ*i*. *kind g i* = *n* ∧ *stamp g i* = *s*) (*sorted-list-of-set*(*ids g*))

export-code *find-node-and-stamp*

1.1 Data-flow Tree Representation

datatype *IRUnaryOp* =
  UnaryAbs
| *UnaryNeg*
| *UnaryNot*
| *UnaryLogicNegation*

datatype *IRBinaryOp* =
  BinAdd
| *BinMul*
| *BinSub*
| *BinAnd*
| *BinOr*
| *BinXor*
| *BinIntegerEquals*
| *BinIntegerLessThan*

datatype (*discs-sels*) *IRExpr* =
  UnaryExpr (*ir-uop*: *IRUnaryOp*) (*ir-value*: *IRExpr*)
| *BinaryExpr* (*ir-op*: *IRBinaryOp*) (*ir-x*: *IRExpr*) (*ir-y*: *IRExpr*)
| *ConditionalExpr* (*ir-condition*: *IRExpr*) (*ir-trueValue*: *IRExpr*) (*ir-falseValue*: *IRExpr*)
| *ConstantExpr* (*ir-const*: *Value*)

| *ParameterExpr* (*ir-index*: *nat*) (*ir-stamp*: *Stamp*)

| *LeafExpr* (*ir-nid*: *ID*) (*ir-stamp*: *Stamp*)

fun *is-preevaluated* :: *IRNode* ⇒ *bool* **where**
  is-preevaluated (*InvokeNode nid* - - - - -) = *True* |
  is-preevaluated (*InvokeWithExceptionNode nid* - - - - - -) = *True* |
  is-preevaluated (*NewInstanceNode nid* - - -) = *True* |
  is-preevaluated (*LoadFieldNode nid* - - -) = *True* |
  is-preevaluated (*SignedDivNode nid* - - - - -) = *True* |
  is-preevaluated (*SignedRemNode nid* - - - - -) = *True* |
  is-preevaluated (*ValuePhiNode nid* - -) = *True* |
  is-preevaluated - = *False*

```
inductive
  rep :: IRGraph ⇒ ID ⇒ IRExpr ⇒ bool (- ⊢ - ▷ - 55)
  for g where

  ConstantNode:
  ⟦kind g n = ConstantNode c⟧
    ⟹ g ⊢ n ▷ (ConstantExpr c) |

  ParameterNode:
  ⟦kind g n = ParameterNode i;
    stamp g n = s⟧
    ⟹ g ⊢ n ▷ (ParameterExpr i s) |

  ConditionalNode:
  ⟦kind g n = ConditionalNode c t f;
    g ⊢ c ▷ ce;
    g ⊢ t ▷ te;
    g ⊢ f ▷ fe⟧
    ⟹ g ⊢ n ▷ (ConditionalExpr ce te fe) |


  AbsNode:
  ⟦kind g n = AbsNode x;
    g ⊢ x ▷ xe⟧
    ⟹ g ⊢ n ▷ (UnaryExpr UnaryAbs xe) |

  NotNode:
  ⟦kind g n = NotNode x;
    g ⊢ x ▷ xe⟧
    ⟹ g ⊢ n ▷ (UnaryExpr UnaryNot xe) |

  NegateNode:
  ⟦kind g n = NegateNode x;
    g ⊢ x ▷ xe⟧
    ⟹ g ⊢ n ▷ (UnaryExpr UnaryNeg xe) |

  LogicNegationNode:
  ⟦kind g n = LogicNegationNode x;
    g ⊢ x ▷ xe⟧
    ⟹ g ⊢ n ▷ (UnaryExpr UnaryLogicNegation xe) |


  AddNode:
  ⟦kind g n = AddNode x y;
    g ⊢ x ▷ xe;
    g ⊢ y ▷ ye⟧
    ⟹ g ⊢ n ▷ (BinaryExpr BinAdd xe ye) |
```

MulNode:
$[\![kind\ g\ n = MulNode\ x\ y;$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye]\!]$
$\Longrightarrow g \vdash n \triangleright (BinaryExpr\ BinMul\ xe\ ye)\ |$

SubNode:
$[\![kind\ g\ n = SubNode\ x\ y;$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye]\!]$
$\Longrightarrow g \vdash n \triangleright (BinaryExpr\ BinSub\ xe\ ye)\ |$

AndNode:
$[\![kind\ g\ n = AndNode\ x\ y;$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye]\!]$
$\Longrightarrow g \vdash n \triangleright (BinaryExpr\ BinAnd\ xe\ ye)\ |$

OrNode:
$[\![kind\ g\ n = OrNode\ x\ y;$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye]\!]$
$\Longrightarrow g \vdash n \triangleright (BinaryExpr\ BinOr\ xe\ ye)\ |$

XorNode:
$[\![kind\ g\ n = XorNode\ x\ y;$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye]\!]$
$\Longrightarrow g \vdash n \triangleright (BinaryExpr\ BinXor\ xe\ ye)\ |$

IntegerEqualsNode:
$[\![kind\ g\ n = IntegerEqualsNode\ x\ y;$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye]\!]$
$\Longrightarrow g \vdash n \triangleright (BinaryExpr\ BinIntegerEquals\ xe\ ye)\ |$

IntegerLessThanNode:
$[\![kind\ g\ n = IntegerLessThanNode\ x\ y;$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye]\!]$
$\Longrightarrow g \vdash n \triangleright (BinaryExpr\ BinIntegerLessThan\ xe\ ye)\ |$

LeafNode:
$[\![is\text{-}preevaluated\ (kind\ g\ n);$
$stamp\ g\ n = s]\!]$
$\Longrightarrow g \vdash n \triangleright (LeafExpr\ n\ s)$

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$ *as exprE*) *rep* **.**

inductive
replist :: *IRGraph* ⇒ *ID list* ⇒ *IRExpr list* ⇒ *bool* $(- \vdash - \triangleright_L - 55)$
for *g* **where**

RepNil:
$g \vdash [] \triangleright_L [] \mid$

RepCons:
$[\![g \vdash x \triangleright xe;$
$g \vdash xs \triangleright_L xse]\!]$
$\Longrightarrow g \vdash x\#xs \triangleright_L xe\#xse$

code-pred (*modes*: *i* ⇒ *i* ⇒ *o* ⇒ *bool as exprListE*) *replist* **.**

$$\frac{kind\ g\ n = ConstantNode\ c}{g \vdash n \triangleright ConstantExpr\ c}$$

$$\frac{kind\ g\ n = ParameterNode\ i \qquad stamp\ g\ n = s}{g \vdash n \triangleright ParameterExpr\ i\ s}$$

$$\frac{kind\ g\ n = AbsNode\ x \qquad g \vdash x \triangleright xe}{g \vdash n \triangleright UnaryExpr\ UnaryAbs\ xe}$$

$$\frac{kind\ g\ n = AddNode\ x\ y \qquad g \vdash x \triangleright xe \qquad g \vdash y \triangleright ye}{g \vdash n \triangleright BinaryExpr\ BinAdd\ xe\ ye}$$

$$\frac{kind\ g\ n = MulNode\ x\ y \qquad g \vdash x \triangleright xe \qquad g \vdash y \triangleright ye}{g \vdash n \triangleright BinaryExpr\ BinMul\ xe\ ye}$$

$$\frac{kind\ g\ n = SubNode\ x\ y \qquad g \vdash x \triangleright xe \qquad g \vdash y \triangleright ye}{g \vdash n \triangleright BinaryExpr\ BinSub\ xe\ ye}$$

$$\frac{is\text{-}preevaluated\ (kind\ g\ n) \qquad stamp\ g\ n = s}{g \vdash n \triangleright LeafExpr\ n\ s}$$

values $\{t.\ eg2\text{-}sq \vdash 4 \triangleright t\}$

fun *stamp-unary* :: *IRUnaryOp* ⇒ *Stamp* ⇒ *Stamp* **where**
stamp-unary op (*IntegerStamp b lo hi*) = *unrestricted-stamp* (*IntegerStamp b lo hi*) |

stamp-unary op - = *IllegalStamp*

fun *stamp-binary* :: *IRBinaryOp* ⇒ *Stamp* ⇒ *Stamp* ⇒ *Stamp* **where**
stamp-binary op (*IntegerStamp b1 lo1 hi1*) (*IntegerStamp b2 lo2 hi2*) =

$(if\ (b1 = b2)\ then\ unrestricted\text{-}stamp\ (IntegerStamp\ b1\ lo1\ hi1)\ else\ IllegalStamp)$
|

stamp-binary op - - = IllegalStamp

fun *stamp-expr* :: *IRExpr* ⇒ *Stamp* **where**
stamp-expr (UnaryExpr op x) = stamp-unary op (stamp-expr x) |
stamp-expr (BinaryExpr bop x y) = stamp-binary bop (stamp-expr x) (stamp-expr y) |
stamp-expr (ConstantExpr val) = constantAsStamp val |
stamp-expr (LeafExpr i s) = s |
stamp-expr (ParameterExpr i s) = s |
stamp-expr (ConditionalExpr c t f) = meet (stamp-expr t) (stamp-expr f)

export-code *stamp-unary stamp-binary stamp-expr*

fun *unary-node* :: *IRUnaryOp* ⇒ *ID* ⇒ *IRNode* **where**
unary-node UnaryAbs v = AbsNode v |
unary-node UnaryNot v = NotNode v |
unary-node UnaryNeg v = NegateNode v |
unary-node UnaryLogicNegation v = LogicNegationNode v

fun *bin-node* :: *IRBinaryOp* ⇒ *ID* ⇒ *ID* ⇒ *IRNode* **where**
bin-node BinAdd x y = AddNode x y |
bin-node BinMul x y = MulNode x y |
bin-node BinSub x y = SubNode x y |
bin-node BinAnd x y = AndNode x y |
bin-node BinOr x y = OrNode x y |
bin-node BinXor x y = XorNode x y |
bin-node BinIntegerEquals x y = IntegerEqualsNode x y |
bin-node BinIntegerLessThan x y = IntegerLessThanNode x y

fun *unary-eval* :: *IRUnaryOp* ⇒ *Value* ⇒ *Value* **where**
unary-eval UnaryAbs (IntVal32 v1) = IntVal32 ((if sint(v1) < 0 then − v1 else v1)) |
unary-eval UnaryAbs (IntVal64 v1) = IntVal64 ((if sint(v1) < 0 then − v1 else v1)) |

unary-eval UnaryNot (IntVal32 v1) = IntVal32 (NOT v1) |
unary-eval UnaryNot (IntVal64 v1) = IntVal64 (NOT v1) |

unary-eval UnaryLogicNegation (IntVal32 v1) = (if v1 = 0 then (IntVal32 1) else (IntVal32 0)) |

unary-eval UnaryNeg v = intval-negate v |

unary-eval op v1 = UndefVal

fun *bin-eval* :: *IRBinaryOp* ⇒ *Value* ⇒ *Value* ⇒ *Value* **where**
bin-eval BinAdd v1 v2 = intval-add v1 v2 |
bin-eval BinMul v1 v2 = intval-mul v1 v2 |
bin-eval BinSub v1 v2 = intval-sub v1 v2 |
bin-eval BinAnd v1 v2 = intval-and v1 v2 |
bin-eval BinOr v1 v2 = intval-or v1 v2 |
bin-eval BinXor v1 v2 = intval-xor v1 v2 |
bin-eval BinIntegerEquals v1 v2 = intval-equals v1 v2 |
bin-eval BinIntegerLessThan v1 v2 = intval-less-than v1 v2

inductive *fresh-id* :: *IRGraph* ⇒ *ID* ⇒ *bool* **where**
nid ∉ *ids g* ⟹ *fresh-id g nid*

code-pred *fresh-id* **.**

fun *get-fresh-id* :: *IRGraph* ⇒ *ID* **where**

get-fresh-id g = last(sorted-list-of-set(ids g)) + 1

export-code *get-fresh-id*

value *get-fresh-id eg2-sq*
value *get-fresh-id (add-node 6 (ParameterNode 2, default-stamp) eg2-sq)*

inductive
unrep :: *IRGraph* ⇒ *IRExpr* ⇒ (*IRGraph* × *ID*) ⇒ *bool* (- ◃ - ⇝ - 55)
and
unrepList :: *IRGraph* ⇒ *IRExpr list* ⇒ (*IRGraph* × *ID list*) ⇒ *bool* (- $\triangleleft_L$ - ⇝ - 55)
where

ConstantNodeSame:
⟦*find-node-and-stamp g (ConstantNode c, constantAsStamp c) = Some nid*⟧
⟹ *g* ◃ (*ConstantExpr c*) ⇝ (*g*, *nid*) |

ConstantNodeNew:
⟦*find-node-and-stamp g (ConstantNode c, constantAsStamp c) = None*;
nid = get-fresh-id g;
g′ = add-node nid (ConstantNode c, constantAsStamp c) g ⟧
⟹ *g* ◃ (*ConstantExpr c*) ⇝ (*g′*, *nid*) |

ParameterNodeSame:
⟦*find-node-and-stamp g (ParameterNode i, s) = Some nid*⟧
⟹ *g* ◃ (*ParameterExpr i s*) ⇝ (*g*, *nid*) |

ParameterNodeNew:
$[\![$*find-node-and-stamp g* (*ParameterNode i*, *s*) = *None*;
nid = *get-fresh-id g*;
g' = *add-node nid* (*ParameterNode i*, *s*) *g*$]\!]$
$\Longrightarrow g \lhd (\mathit{ParameterExpr}\ i\ s) \leadsto (g', \mathit{nid})$ |

ConditionalNodeSame:
$[\![g \lhd_L [\mathit{ce}, \mathit{te}, \mathit{fe}] \leadsto (g2, [c, t, f])$;
$s' = \mathit{meet}\ (\mathit{stamp}\ g2\ t)\ (\mathit{stamp}\ g2\ f)$;
find-node-and-stamp g2 (*ConditionalNode c t f*, s') = *Some nid*$]\!]$
$\Longrightarrow g \lhd (\mathit{ConditionalExpr}\ \mathit{ce}\ \mathit{te}\ \mathit{fe}) \leadsto (g2, \mathit{nid})$ |

ConditionalNodeNew:
$[\![g \lhd_L [\mathit{ce}, \mathit{te}, \mathit{fe}] \leadsto (g2, [c, t, f])$;
$s' = \mathit{meet}\ (\mathit{stamp}\ g2\ t)\ (\mathit{stamp}\ g2\ f)$;
find-node-and-stamp g2 (*ConditionalNode c t f*, s') = *None*;
nid = *get-fresh-id g2*;
g' = *add-node nid* (*ConditionalNode c t f*, s') *g2*$]\!]$
$\Longrightarrow g \lhd (\mathit{ConditionalExpr}\ \mathit{ce}\ \mathit{te}\ \mathit{fe}) \leadsto (g', \mathit{nid})$ |

UnaryNodeSame:
$[\![g \lhd \mathit{xe} \leadsto (g2, x)$;
s' = *stamp-unary op* (*stamp g2 x*);
find-node-and-stamp g2 (*unary-node op x*, s') = *Some nid*$]\!]$
$\Longrightarrow g \lhd (\mathit{UnaryExpr}\ \mathit{op}\ \mathit{xe}) \leadsto (g2, \mathit{nid})$ |

UnaryNodeNew:
$[\![g \lhd \mathit{xe} \leadsto (g2, x)$;
s' = *stamp-unary op* (*stamp g2 x*);
find-node-and-stamp g2 (*unary-node op x*, s') = *None*;
nid = *get-fresh-id g2*;
g' = *add-node nid* (*unary-node op x*, s') *g2*$]\!]$
$\Longrightarrow g \lhd (\mathit{UnaryExpr}\ \mathit{op}\ \mathit{xe}) \leadsto (g', \mathit{nid})$ |

BinaryNodeSame:
$[\![g \lhd_L [\mathit{xe}, \mathit{ye}] \leadsto (g2, [x, y])$;
s' = *stamp-binary op* (*stamp g2 x*) (*stamp g2 y*);
find-node-and-stamp g2 (*bin-node op x y*, s') = *Some nid*$]\!]$
$\Longrightarrow g \lhd (\mathit{BinaryExpr}\ \mathit{op}\ \mathit{xe}\ \mathit{ye}) \leadsto (g2, \mathit{nid})$ |

BinaryNodeNew:
$[\![g \lhd_L [\mathit{xe}, \mathit{ye}] \leadsto (g2, [x, y])$;
s' = *stamp-binary op* (*stamp g2 x*) (*stamp g2 y*);
find-node-and-stamp g2 (*bin-node op x y*, s') = *None*;
nid = *get-fresh-id g2*;
g' = *add-node nid* (*bin-node op x y*, s') *g2*$]\!]$
$\Longrightarrow g \lhd (\mathit{BinaryExpr}\ \mathit{op}\ \mathit{xe}\ \mathit{ye}) \leadsto (g', \mathit{nid})$ |

AllLeafNodes:
$stamp\ g\ nid = s$
$\Longrightarrow g \triangleleft (LeafExpr\ nid\ s) \rightsquigarrow (g,\ nid)$ |

UnrepNil:
$g \triangleleft_L [] \rightsquigarrow (g,\ [])$ |

UnrepCons:
$[\![g \triangleleft xe \rightsquigarrow (g2,\ x);$
$g2 \triangleleft_L xes \rightsquigarrow (g3,\ xs)]\!]$
$\Longrightarrow g \triangleleft_L (xe\#xes) \rightsquigarrow (g3,\ x\#xs)$

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$ *as unrepE*)
unrep **.**
code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$ *as unrepListE*) *unrepList* **.**

$$\frac{\textit{find-node-and-stamp}\ g\ (ConstantNode\ c,\ constantAsStamp\ c) = Some\ nid}{g \triangleleft ConstantExpr\ c \rightsquigarrow (g,\ nid)}$$

$$\frac{\begin{array}{c}\textit{find-node-and-stamp}\ g\ (ConstantNode\ c,\ constantAsStamp\ c) = None\\ nid = \textit{get-fresh-id}\ g\\ g' = \textit{add-node}\ nid\ (ConstantNode\ c,\ constantAsStamp\ c)\ g\end{array}}{g \triangleleft ConstantExpr\ c \rightsquigarrow (g',\ nid)}$$

$$\frac{\textit{find-node-and-stamp}\ g\ (ParameterNode\ i,\ s) = Some\ nid}{g \triangleleft ParameterExpr\ i\ s \rightsquigarrow (g,\ nid)}$$

$$\frac{\begin{array}{c}\textit{find-node-and-stamp}\ g\ (ParameterNode\ i,\ s) = None\\ nid = \textit{get-fresh-id}\ g \qquad g' = \textit{add-node}\ nid\ (ParameterNode\ i,\ s)\ g\end{array}}{g \triangleleft ParameterExpr\ i\ s \rightsquigarrow (g',\ nid)}$$

$$\frac{\begin{array}{c}g \triangleleft_L [ce,\ te,\ fe] \rightsquigarrow (g2,\ [c,\ t,\ f]) \qquad s' = meet\ (stamp\ g2\ t)\ (stamp\ g2\ f)\\ \textit{find-node-and-stamp}\ g2\ (ConditionalNode\ c\ t\ f,\ s') = Some\ nid\end{array}}{g \triangleleft ConditionalExpr\ ce\ te\ fe \rightsquigarrow (g2,\ nid)}$$

$$\frac{\begin{array}{c}g \triangleleft_L [ce,\ te,\ fe] \rightsquigarrow (g2,\ [c,\ t,\ f]) \qquad s' = meet\ (stamp\ g2\ t)\ (stamp\ g2\ f)\\ \textit{find-node-and-stamp}\ g2\ (ConditionalNode\ c\ t\ f,\ s') = None\\ nid = \textit{get-fresh-id}\ g2 \qquad g' = \textit{add-node}\ nid\ (ConditionalNode\ c\ t\ f,\ s')\ g2\end{array}}{g \triangleleft ConditionalExpr\ ce\ te\ fe \rightsquigarrow (g',\ nid)}$$

$$\frac{\begin{array}{c}g \triangleleft_L [xe,\ ye] \rightsquigarrow (g2,\ [x,\ y]) \qquad s' = \textit{stamp-binary}\ op\ (stamp\ g2\ x)\ (stamp\ g2\ y)\\ \textit{find-node-and-stamp}\ g2\ (\textit{bin-node}\ op\ x\ y,\ s') = Some\ nid\end{array}}{g \triangleleft BinaryExpr\ op\ xe\ ye \rightsquigarrow (g2,\ nid)}$$

$$\frac{g \triangleleft_L [xe, ye] \rightsquigarrow (g2, [x, y]) \qquad s' = \textit{stamp-binary op } (\textit{stamp g2 x}) \ (\textit{stamp g2 y}) \\ \textit{find-node-and-stamp g2 } (\textit{bin-node op x y}, s') = \textit{None} \\ \textit{nid} = \textit{get-fresh-id g2} \qquad g' = \textit{add-node nid } (\textit{bin-node op x y}, s') \ \textit{g2}}{g \triangleleft \textit{BinaryExpr op xe ye} \rightsquigarrow (g', \textit{nid})}$$

$$\frac{g \triangleleft xe \rightsquigarrow (g2, x) \qquad s' = \textit{stamp-unary op } (\textit{stamp g2 x}) \\ \textit{find-node-and-stamp g2 } (\textit{unary-node op x}, s') = \textit{Some nid}}{g \triangleleft \textit{UnaryExpr op xe} \rightsquigarrow (g2, \textit{nid})}$$

$$\frac{g \triangleleft xe \rightsquigarrow (g2, x) \qquad s' = \textit{stamp-unary op } (\textit{stamp g2 x}) \\ \textit{find-node-and-stamp g2 } (\textit{unary-node op x}, s') = \textit{None} \\ \textit{nid} = \textit{get-fresh-id g2} \qquad g' = \textit{add-node nid } (\textit{unary-node op x}, s') \ \textit{g2}}{g \triangleleft \textit{UnaryExpr op xe} \rightsquigarrow (g', \textit{nid})}$$

$$\frac{\textit{stamp g nid} = s}{g \triangleleft \textit{LeafExpr nid s} \rightsquigarrow (g, \textit{nid})}$$

definition *sq-param0* :: *IRExpr* **where**
sq-param0 = *BinaryExpr BinMul*
(*ParameterExpr 0* (*IntegerStamp 32* (− *2147483648*) *2147483647*))
(*ParameterExpr 0* (*IntegerStamp 32* (− *2147483648*) *2147483647*))

values {(*nid*, *g*) . (*eg2-sq* ◁ *sq-param0* ⇝ (*g*, *nid*))}

1.2 Data-flow Tree Evaluation

inductive
evaltree :: *MapState* ⇒ *Params* ⇒ *IRExpr* ⇒ *Value* ⇒ *bool* ([-,-] ⊢ - ↦ - *55*)
for *m p* **where**

ConstantExpr:
⟦*c* ≠ *UndefVal*⟧
⟹ [*m*,*p*] ⊢ (*ConstantExpr c*) ↦ *c* |

ParameterExpr:
⟦*valid-value s* (*p*!*i*)⟧
⟹ [*m*,*p*] ⊢ (*ParameterExpr i s*) ↦ *p*!*i* |

ConditionalExpr:
⟦[*m*,*p*] ⊢ *ce* ↦ *cond*;
branch = (*if val-to-bool cond then te else fe*);
[*m*,*p*] ⊢ *branch* ↦ *v*⟧
⟹ [*m*,*p*] ⊢ (*ConditionalExpr ce te fe*) ↦ *v* |

UnaryExpr:
⟦[*m*,*p*] ⊢ *xe* ↦ *v*⟧
⟹ [*m*,*p*] ⊢ (*UnaryExpr op xe*) ↦ *unary-eval op v* |

BinaryExpr:
$[\![[m,p] \vdash xe \mapsto x;$
$[m,p] \vdash ye \mapsto y]\!]$
$\Longrightarrow [m,p] \vdash (BinaryExpr\ op\ xe\ ye) \mapsto bin\text{-}eval\ op\ x\ y$ |

LeafExpr:
$[\![val = m\ nid;$
$valid\text{-}value\ s\ val]\!]$
$\Longrightarrow [m,p] \vdash LeafExpr\ nid\ s \mapsto val$

$$\frac{c \neq UndefVal}{[m,p] \vdash ConstantExpr\ c \mapsto c}$$

$$\frac{valid\text{-}value\ s\ p_{[i]}}{[m,p] \vdash ParameterExpr\ i\ s \mapsto p_{[i]}}$$

$$\frac{[m,p] \vdash ce \mapsto cond \qquad \begin{array}{c} branch = (\mathsf{if}\ IRTreeEval.val\text{-}to\text{-}bool\ cond\ \mathsf{then}\ te\ \mathsf{else}\ fe) \\ [m,p] \vdash branch \mapsto v \end{array}}{[m,p] \vdash ConditionalExpr\ ce\ te\ fe \mapsto v}$$

$$\frac{[m,p] \vdash xe \mapsto v}{[m,p] \vdash UnaryExpr\ op\ xe \mapsto unary\text{-}eval\ op\ v}$$

$$\frac{[m,p] \vdash xe \mapsto x \qquad [m,p] \vdash ye \mapsto y}{[m,p] \vdash BinaryExpr\ op\ xe\ ye \mapsto bin\text{-}eval\ op\ x\ y}$$

$$\frac{val = m\ nid \qquad valid\text{-}value\ s\ val}{[m,p] \vdash LeafExpr\ nid\ s \mapsto val}$$

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$ *as evalT*)
[*show-steps*,*show-mode-inference*,*show-intermediate-results*]
evaltree **.**

inductive
evaltrees :: *MapState* $\Rightarrow$ *Params* $\Rightarrow$ *IRExpr list* $\Rightarrow$ *Value list* $\Rightarrow$ *bool* ([-,-] $\vdash$ - $\mapsto_L$ - *55*)
for *m p* **where**

EvalNil:
$[m,p] \vdash [] \mapsto_L []$ |

EvalCons:
$[\![[m,p] \vdash x \mapsto xval;$
$[m,p] \vdash yy \mapsto_L yyval]\!]$

$\Longrightarrow [m,p] \vdash (x\#yy) \mapsto_L (xval\#yyval)$

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow$ *bool as evalTs*)
evaltrees **.**

values $\{v.$ *evaltree new-map-state* [*IntVal32 5*] *sq-param0 v*$\}$

declare *evaltree.intros* [*intro*]
declare *evaltrees.intros* [*intro*]

1.3 Data-flow Tree Refinement

We define the induced semantic equivalence relation between expressions. Note that syntactic equality implies semantic equivalence, but not vice versa.

definition *equiv-exprs* :: *IRExpr* $\Rightarrow$ *IRExpr* $\Rightarrow$ *bool* (- $\doteq$ - *55*) **where**
$(e1 \doteq e2) = (\forall\ m\ p\ v.\ (([m,p] \vdash e1 \mapsto v) \longleftrightarrow ([m,p] \vdash e2 \mapsto v)))$

We also prove that this is a total equivalence relation (*equivp equiv-exprs*) (HOL.Equiv_Relations), so that we can reuse standard results about equivalence relations.

lemma *equivp equiv-exprs*
apply (*auto simp add*: *equivp-def equiv-exprs-def*)
by (*metis equiv-exprs-def*)+

We define a refinement ordering over IRExpr and show that it is a preorder. Note that it is asymmetric because e2 may refer to fewer variables than e1.

instantiation *IRExpr* :: *preorder* **begin**

definition
le-expr-def [*simp*]: $(e1 \le e2) \longleftrightarrow (\forall\ m\ p\ v.\ (([m,p] \vdash e1 \mapsto v) \longrightarrow ([m,p] \vdash e2 \mapsto v)))$

definition
lt-expr-def [*simp*]: $(e1 < e2) \longleftrightarrow (e1 \le e2 \wedge \neg (e1 \doteq e2))$

instance proof
fix $x\ y\ z$ :: *IRExpr*
show $x < y \longleftrightarrow x \le y \wedge \neg (y \le x)$ **by** (*simp add*: *equiv-exprs-def*; *auto*)
show $x \le x$ **by** *simp*
show $x \le y \Longrightarrow y \le z \Longrightarrow x \le z$ **by** *simp*
qed
end

end

2 Data-flow Expression-Tree Theorems

theory *IRTreeEvalThms*
  imports
    Semantics.IRTreeEval
begin

2.1 Extraction and Evaluation of Expression Trees is Deterministic.

First, we prove some extra rules that relate each type of IRNode to the corresponding IRExpr type that 'rep' will produce. These are very helpful for proving that 'rep' is deterministic.

lemma *rep-constant*:
  $g \vdash n \triangleright e \Longrightarrow$
   $kind\ g\ n = ConstantNode\ c \Longrightarrow$
   $e = ConstantExpr\ c$
  by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-parameter*:
  $g \vdash n \triangleright e \Longrightarrow$
   $kind\ g\ n = ParameterNode\ i \Longrightarrow$
   $(\exists s.\ e = ParameterExpr\ i\ s)$
  by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-conditional*:
  $g \vdash n \triangleright e \Longrightarrow$
   $kind\ g\ n = ConditionalNode\ c\ t\ f \Longrightarrow$
   $(\exists\ ce\ te\ fe.\ e = ConditionalExpr\ ce\ te\ fe)$
  by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-abs*:
  $g \vdash n \triangleright e \Longrightarrow$
   $kind\ g\ n = AbsNode\ x \Longrightarrow$
   $(\exists xe.\ e = UnaryExpr\ UnaryAbs\ xe)$
  by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-not*:
  $g \vdash n \triangleright e \Longrightarrow$
   $kind\ g\ n = NotNode\ x \Longrightarrow$
   $(\exists xe.\ e = UnaryExpr\ UnaryNot\ xe)$
  by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-negate*:
  $g \vdash n \triangleright e \Longrightarrow$
   $kind\ g\ n = NegateNode\ x \Longrightarrow$
   $(\exists xe.\ e = UnaryExpr\ UnaryNeg\ xe)$
  by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-logicnegation*:
$g \vdash n \triangleright e \Longrightarrow$
$kind\ g\ n = LogicNegationNode\ x \Longrightarrow$
$(\exists xe.\ e = UnaryExpr\ UnaryLogicNegation\ xe)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-add*:
$g \vdash n \triangleright e \Longrightarrow$
$kind\ g\ n = AddNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinAdd\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-sub*:
$g \vdash n \triangleright e \Longrightarrow$
$kind\ g\ n = SubNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinSub\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-mul*:
$g \vdash n \triangleright e \Longrightarrow$
$kind\ g\ n = MulNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinMul\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-and*:
$g \vdash n \triangleright e \Longrightarrow$
$kind\ g\ n = AndNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinAnd\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-or*:
$g \vdash n \triangleright e \Longrightarrow$
$kind\ g\ n = OrNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinOr\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-xor*:
$g \vdash n \triangleright e \Longrightarrow$
$kind\ g\ n = XorNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinXor\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-integer-equals*:
$g \vdash n \triangleright e \Longrightarrow$
$kind\ g\ n = IntegerEqualsNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinIntegerEquals\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-integer-less-than*:

```
  g ⊢ n ▷ e ⟹
   kind g n = IntegerLessThanNode x y ⟹
   (∃ xe ye. e = BinaryExpr BinIntegerLessThan xe ye)
  by (induction rule: rep.induct; auto)

lemma rep-load-field:
  g ⊢ n ▷ e ⟹
   is-preevaluated (kind g n) ⟹
   (∃ s. e = LeafExpr n s)
  by (induction rule: rep.induct; auto)


lemma repDet:
  shows (g ⊢ n ▷ e1) ⟹ (g ⊢ n ▷ e2) ⟹ e1 = e2
proof (induction arbitrary: e2 rule: rep.induct)
  case (ConstantNode n c)
  then show ?case using rep-constant by auto
next
  case (ParameterNode n i s)
  then show ?case using rep-parameter by auto
next
case (ConditionalNode n c t f ce te fe)
  then show ?case
    by (metis rep-conditional ConditionalNodeE IRNode.inject(6))
next
  case (AbsNode n x xe)
  then show ?case
    by (metis rep-abs AbsNodeE IRNode.inject(1))
next
  case (NotNode n x xe)
  then show ?case
    by (metis rep-not NotNodeE IRNode.inject(29))
next
case (NegateNode n x xe)
  then show ?case
    by (metis  IRNode.inject(26) NegateNodeE rep-negate)
next
  case (LogicNegationNode n x xe)
  then show ?case
    by (metis IRNode.inject(19) LogicNegationNodeE rep-logicnegation)
next
  case (AddNode n x y xe ye)
  then show ?case
    by (metis AddNodeE IRNode.inject(2) rep-add)
next
case (MulNode n x y xe ye)
  then show ?case
```

by (*metis IRNode.inject*(*25*) *MulNodeE rep-mul*)
next
case (*SubNode n x y xe ye*)
then show *?case*
by (*metis IRNode.inject*(*39*) *SubNodeE rep-sub*)
next
case (*AndNode n x y xe ye*)
then show *?case*
by (*metis AndNodeE IRNode.inject*(*3*) *rep-and*)
next
case (*OrNode n x y xe ye*)
then show *?case*
by (*metis IRNode.inject*(*30*) *OrNodeE rep-or*)
next
case (*XorNode n x y xe ye*)
then show *?case*
by (*metis IRNode.inject*(*43*) *XorNodeE rep-xor*)
next
case (*IntegerEqualsNode n x y xe ye*)
then show *?case*
by (*metis IRNode.inject*(*12*) *IntegerEqualsNodeE rep-integer-equals*)
next
case (*IntegerLessThanNode n x y xe ye*)
then show *?case*
by (*metis IRNode.inject*(*13*) *IntegerLessThanNodeE rep-integer-less-than*)
next
case (*LeafNode n s*)
then show *?case* **using** *rep-load-field LeafNodeE* **by** *blast*
qed

lemma *evalDet*:
$[m,p] \vdash e \mapsto v1 \Longrightarrow$
$[m,p] \vdash e \mapsto v2 \Longrightarrow$
$v1 = v2$
apply (*induction arbitrary*: *v2 rule*: *evaltree.induct*)
by (*elim EvalTreeE*; *auto*)+

lemma *evalAllDet*:
$[m,p] \vdash e \mapsto_L v1 \Longrightarrow$
$[m,p] \vdash e \mapsto_L v2 \Longrightarrow$
$v1 = v2$
apply (*induction arbitrary*: *v2 rule*: *evaltrees.induct*)
apply (*elim EvalTreeE*; *auto*)
using *evalDet* **by** *force*

A valid value cannot be $UndefVal$.

lemma *valid-not-undef*:
assumes *a1*: *valid-value s val*

assumes *a2*: $s \neq$ *VoidStamp*
shows *val* $\neq$ *UndefVal*
apply (*rule valid-value.elims*(*1*)[*of s val True*])
using *a1 a2* **by** *auto*

lemma *valid-VoidStamp*[*elim*]:
shows *valid-value VoidStamp val* $\Longrightarrow$
val = *UndefVal*
using *valid-value.simps* **by** (*metis IRTreeEval.val-to-bool.cases*)

lemma *valid-ObjStamp*[*elim*]:
shows *valid-value* (*ObjectStamp klass exact nonNull alwaysNull*) *val* $\Longrightarrow$
($\exists$ *v. val* = *ObjRef v*)
using *valid-value.simps* **by** (*metis IRTreeEval.val-to-bool.cases*)

lemma *valid-int32*[*elim*]:
shows *valid-value* (*IntegerStamp 32 l h*) *val* $\Longrightarrow$
($\exists$ *v. val* = *IntVal32 v*)
apply (*rule IRTreeEval.val-to-bool.cases*[*of val*])
using *Value.distinct* **by** *simp+*

lemma *valid-int64*[*elim*]:
shows *valid-value* (*IntegerStamp 64 l h*) *val* $\Longrightarrow$
($\exists$ *v. val* = *IntVal64 v*)
apply (*rule IRTreeEval.val-to-bool.cases*[*of val*])
using *Value.distinct* **by** *simp+*

TODO: could we prove that expression evaluation never returns $UndefVal$? But this might require restricting unary and binary operators to be total...

lemma *leafint32*:
assumes *ev*: [*m*,*p*] $\vdash$ *LeafExpr i* (*IntegerStamp 32 lo hi*) $\mapsto$ *val*
shows $\exists$ *v. val* = (*IntVal32 v*)

proof –
have *valid-value* (*IntegerStamp 32 lo hi*) *val*
using *ev* **by** (*rule LeafExprE*; *simp*)
then show *?thesis* **by** *auto*
qed

lemma *leafint64*:
assumes *ev*: [*m*,*p*] $\vdash$ *LeafExpr i* (*IntegerStamp 64 lo hi*) $\mapsto$ *val*
shows $\exists$ *v. val* = (*IntVal64 v*)

proof –
have *valid-value* (*IntegerStamp 64 lo hi*) *val*
using *ev* **by** (*rule LeafExprE*; *simp*)
then show *?thesis* **by** *auto*

qed

lemma *default-stamp* [*simp*]: *default-stamp* = *IntegerStamp 32* (−*2147483648*) *2147483647*
  using *default-stamp-def* **by** *auto*

lemma *valid32* [*simp*]:
  assumes *valid-value* (*IntegerStamp 32 lo hi*) *val*
  shows $\exists$ *v*. (*val* = (*IntVal32 v*) $\wedge$ *lo* $\leq$ *sint v* $\wedge$ *sint v* $\leq$ *hi*)
  using *assms valid-int32* **by** *force*

lemma *valid64* [*simp*]:
  assumes *valid-value* (*IntegerStamp 64 lo hi*) *val*
  shows $\exists$ *v*. (*val* = (*IntVal64 v*) $\wedge$ *lo* $\leq$ *sint v* $\wedge$ *sint v* $\leq$ *hi*)
  using *assms valid-int64* **by** *force*

lemma *int-stamp-implies-valid-value*:
  [*m*,*p*] $\vdash$ *expr* $\mapsto$ *val* $\Longrightarrow$
   valid-value (*stamp-expr expr*) *val*
proof (*induction rule*: *evaltree.induct*)
case (*ConstantExpr c*)
then show *?case* **sorry**
next
  case (*ParameterExpr s i*)
then show *?case* **sorry**
next
  case (*ConditionalExpr ce cond branch te fe v*)
  then show *?case* **sorry**
next
  case (*UnaryExpr xe v op*)
  then show *?case* **sorry**
next
  case (*BinaryExpr xe x ye y op*)
then show *?case* **sorry**
next
  case (*LeafExpr val nid s*)
  then show *?case* **sorry**
qed

2.2 Example Data-flow Optimisations

lemma *a0a-helper* [*simp*]:
  assumes *a*: *valid-value* (*IntegerStamp 32 lo hi*) *v*
  shows *intval-add v* (*IntVal32 0*) = *v*
proof −
  obtain *v32* :: *int32* **where** *v* = (*IntVal32 v32*) **using** *a valid32* **by** *blast*
  then show *?thesis* **by** *simp*
qed

lemma *a0a*: (*BinaryExpr BinAdd* (*LeafExpr 1 default-stamp*) (*ConstantExpr* (*IntVal32 0*)))
  $\leq$ (*LeafExpr 1 default-stamp*) (**is** *?L* $\leq$ *?R*)
by (*auto simp add*: *evaltree.LeafExpr*)

lemma *xyx-y-helper* [*simp*]:
assumes *valid-value* (*IntegerStamp 32 lox hix*) *x*
assumes *valid-value* (*IntegerStamp 32 loy hiy*) *y*
shows *intval-add x* (*intval-sub y x*) = *y*
proof –
obtain *x32* :: *int32* **where** *x*: *x* = (*IntVal32 x32*) **using** *assms valid32* **by** *blast*
obtain *y32* :: *int32* **where** *y*: *y* = (*IntVal32 y32*) **using** *assms valid32* **by** *blast*
show *?thesis* **using** *x y* **by** *simp*
qed

lemma *xyx-y*:
(*BinaryExpr BinAdd*
  (*LeafExpr x* (*IntegerStamp 32 lox hix*))
  (*BinaryExpr BinSub*
    (*LeafExpr y* (*IntegerStamp 32 loy hiy*))
    (*LeafExpr x* (*IntegerStamp 32 lox hix*))))
$\leq$ (*LeafExpr y* (*IntegerStamp 32 loy hiy*))
by (*auto simp add*: *LeafExpr*)

2.3 Monotonicity of Expression Optimization

We prove that each subexpression position is monotonic. That is, optimizing a subexpression anywhere deep inside a top-level expression also optimizes that top-level expression.

Note that we might also be able to do this via reusing Isabelle's 'mono' operator (HOL.Orderings theory), proving instantiations like 'mono (UnaryExpr op)', but it is not obvious how to do this for both arguments of the binary expressions.

lemma *mono-unary*:
assumes $e \leq e'$
shows (*UnaryExpr op e*) $\leq$ (*UnaryExpr op* e')
using *UnaryExpr assms* **by** *auto*

lemma *mono-binary*:
assumes $x \leq x'$
assumes $y \leq y'$
shows (*BinaryExpr op x y*) $\leq$ (*BinaryExpr op* x' y')
using *BinaryExpr assms* **by** *auto*

lemma *mono-conditional*:
assumes $ce \leq ce'$

assumes *te* $\leq$ *te'*
assumes *fe* $\leq$ *fe'*
shows (*ConditionalExpr ce te fe*) $\leq$ (*ConditionalExpr ce' te' fe'*)
proof (*simp only*: *le-expr-def*; (*rule allI*)+; *rule impI*)
fix *m p v*
assume *a*: [*m*,*p*] $\vdash$ *ConditionalExpr ce te fe* $\mapsto$ *v*
then obtain *cond* **where** *ce*: [*m*,*p*] $\vdash$ *ce* $\mapsto$ *cond* **by** *auto*
then have *ce'*: [*m*,*p*] $\vdash$ *ce'* $\mapsto$ *cond* **using** *assms* **by** *auto*
define *branch* **where** *b*: *branch* = (*if val-to-bool cond then te else fe*)
define *branch'* **where** *b'*: *branch'* = (*if val-to-bool cond then te' else fe'*)
then have [*m*,*p*] $\vdash$ *branch* $\mapsto$ *v* **using** *a b ce evalDet* **by** *blast*
then have [*m*,*p*] $\vdash$ *branch'* $\mapsto$ *v* **using** *assms b b'* **by** *auto*
then show [*m*,*p*] $\vdash$ *ConditionalExpr ce' te' fe'* $\mapsto$ *v*
using *ConditionalExpr ce' b'* **by** *auto*
qed

end

3 Control-flow Semantics

theory *IRStepObj*
imports
IRTreeEval
begin

3.1 Heap

The heap model we introduce maps field references to object instances to runtime values. We use the H[f][p] heap representation. See \cite{heap−reps−2011}.

We also introduce the DynamicHeap type which allocates new object references sequentially storing the next free object reference as 'Free'.

type-synonym ('*a*, '*b*) *Heap* = '*a* $\Rightarrow$ '*b* $\Rightarrow$ *Value*
type-synonym *Free* = *nat*
type-synonym ('*a*, '*b*) *DynamicHeap* = ('*a*, '*b*) *Heap* $\times$ *Free*

fun *h-load-field* :: '*a* $\Rightarrow$ '*b* $\Rightarrow$ ('*a*, '*b*) *DynamicHeap* $\Rightarrow$ *Value* **where**
h-load-field f r (*h*, *n*) = *h f r*

fun *h-store-field* :: '*a* $\Rightarrow$ '*b* $\Rightarrow$ *Value* $\Rightarrow$ ('*a*, '*b*) *DynamicHeap* $\Rightarrow$ ('*a*, '*b*) *DynamicHeap* **where**
h-store-field f r v (*h*, *n*) = (*h*(*f* := ((*h f*)(*r* := *v*))), *n*)

fun *h-new-inst* :: ('*a*, '*b*) *DynamicHeap* $\Rightarrow$ ('*a*, '*b*) *DynamicHeap* $\times$ *Value* **where**
h-new-inst (*h*, *n*) = ((*h*,*n*+*1*), (*ObjRef* (*Some n*)))

type-synonym *FieldRefHeap* = (*string*, *objref*) *DynamicHeap*

definition *new-heap* :: ('*a*, '*b*) *DynamicHeap* **where**
new-heap = ((λ*f*. λ*p*. *UndefVal*), *0*)

3.2 Intraprocedural Semantics

Intraprocedural semantics are given as a small-step semantics.

Within the context of a graph, the configuration triple, (ID, MethodState, Heap), is related to the subsequent configuration.

inductive *step* :: *IRGraph* ⇒ *Params* ⇒ (*ID* × *MapState* × *FieldRefHeap*) ⇒ (*ID* × *MapState* × *FieldRefHeap*) ⇒ *bool*
(-, - ⊢ - → - *55*) **for** *g p* **where**

SequentialNode:
⟦*is-sequential-node* (*kind g nid*);
nid' = (*successors-of* (*kind g nid*))!*0*⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid'*, *m*, *h*) |

IfNode:
⟦*kind g nid* = (*IfNode cond tb fb*);
g ⊢ *cond* ▷ *condE*;
[*m*, *p*] ⊢ *condE* ↦ *val*;
nid' = (*if val-to-bool val then tb else fb*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid'*, *m*, *h*) |

EndNodes:
⟦*is-AbstractEndNode* (*kind g nid*);
merge = *any-usage g nid*;
is-AbstractMergeNode (*kind g merge*);

i = *find-index nid* (*inputs-of* (*kind g merge*));
phis = (*phi-list g merge*);
inps = (*phi-inputs g i phis*);
g ⊢ *inps* $\triangleright_L$ *inpsE*;
[*m*, *p*] ⊢ *inpsE* $\mapsto_L$ *vs*;

m' = *set-phis phis vs m*⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*merge*, *m'*, *h*) |

NewInstanceNode:
⟦*kind g nid* = (*NewInstanceNode nid f obj nid'*);
(*h'*, *ref*) = *h-new-inst h*;
m' = *m*(*nid* := *ref*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid'*, *m'*, *h'*) |

LoadFieldNode:
$[\![kind\ g\ nid = (LoadFieldNode\ nid\ f\ (Some\ obj)\ nid');$
$g \vdash obj \triangleright objE;$
$[m, p] \vdash objE \mapsto ObjRef\ ref;$
$h\text{-}load\text{-}field\ f\ ref\ h = v;$
$m' = m(nid := v)]\!]$
$\Longrightarrow g, p \vdash (nid, m, h) \to (nid', m', h) \mid$

SignedDivNode:
$[\![kind\ g\ nid = (SignedDivNode\ nid\ x\ y\ zero\ sb\ nxt);$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye;$
$[m, p] \vdash xe \mapsto v1;$
$[m, p] \vdash ye \mapsto v2;$
$v = (intval\text{-}div\ v1\ v2);$
$m' = m(nid := v)]\!]$
$\Longrightarrow g, p \vdash (nid, m, h) \to (nxt, m', h) \mid$

SignedRemNode:
$[\![kind\ g\ nid = (SignedRemNode\ nid\ x\ y\ zero\ sb\ nxt);$
$g \vdash x \triangleright xe;$
$g \vdash y \triangleright ye;$
$[m, p] \vdash xe \mapsto v1;$
$[m, p] \vdash ye \mapsto v2;$
$v = (intval\text{-}mod\ v1\ v2);$
$m' = m(nid := v)]\!]$
$\Longrightarrow g, p \vdash (nid, m, h) \to (nxt, m', h) \mid$

StaticLoadFieldNode:
$[\![kind\ g\ nid = (LoadFieldNode\ nid\ f\ None\ nid');$
$h\text{-}load\text{-}field\ f\ None\ h = v;$
$m' = m(nid := v)]\!]$
$\Longrightarrow g, p \vdash (nid, m, h) \to (nid', m', h) \mid$

StoreFieldNode:
$[\![kind\ g\ nid = (StoreFieldNode\ nid\ f\ newval\ \text{-}\ (Some\ obj)\ nid');$
$g \vdash newval \triangleright newvalE;$
$g \vdash obj \triangleright objE;$
$[m, p] \vdash newvalE \mapsto val;$
$[m, p] \vdash objE \mapsto ObjRef\ ref;$
$h' = h\text{-}store\text{-}field\ f\ ref\ val\ h;$
$m' = m(nid := val)]\!]$
$\Longrightarrow g, p \vdash (nid, m, h) \to (nid', m', h') \mid$

StaticStoreFieldNode:
$[\![kind\ g\ nid = (StoreFieldNode\ nid\ f\ newval\ \text{-}\ None\ nid');$
$g \vdash newval \triangleright newvalE;$
$[m, p] \vdash newvalE \mapsto val;$
$h' = h\text{-}store\text{-}field\ f\ None\ val\ h;$

$m' = \ \ m(nid := val)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')$

code-pred $(modes\colon i \Rightarrow i \Rightarrow i * i * i \Rightarrow o * o * o \Rightarrow bool)\ step$ **.**

3.3 Interprocedural Semantics

type-synonym *Signature* = *string*
type-synonym *Program* = *Signature* $\rightharpoonup$ *IRGraph*

inductive *step-top* :: *Program* $\Rightarrow$ (*IRGraph* $\times$ *ID* $\times$ *MapState* $\times$ *Params*) *list* $\times$ *FieldRefHeap* $\Rightarrow$ (*IRGraph* $\times$ *ID* $\times$ *MapState* $\times$ *Params*) *list* $\times$ *FieldRefHeap* $\Rightarrow$ *bool*
(- $\vdash$ - $\longrightarrow$ - *55*)
for *P* **where**

Lift:
$[\![g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#stk,\ h) \longrightarrow ((g,nid',m',p)\#stk,\ h') \mid$

InvokeNodeStep:
$[\![$*is-Invoke* (*kind g nid*);

callTarget = *ir-callTarget* (*kind g nid*);
kind g callTarget = (*MethodCallTargetNode targetMethod arguments*);
Some targetGraph = *P targetMethod*;
m' = *new-map-state*;
$g \vdash arguments \triangleright_L argsE$;
$[m,\ p] \vdash argsE\ \mapsto_L p']\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#stk,\ h) \longrightarrow ((targetGraph,0,m',p')\#(g,nid,m,p)\#stk,\ h)$
$\mid$

ReturnNode:
$[\![$*kind g nid* = (*ReturnNode* (*Some expr*) -);
$g \vdash expr \triangleright e$;
$[m,\ p] \vdash e \mapsto v$;

$cm' = cm(cnid := v)$;
$cnid'$ = (*successors-of* (*kind cg cnid*))!*0*$]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk,\ h) \longrightarrow ((cg,cnid',cm',cp)\#stk,\ h) \mid$

ReturnNodeVoid:
$[\![$*kind g nid* = (*ReturnNode None* -);
$cm' = cm(cnid := (ObjRef\ (Some\ (2048))))$;

$cnid'$ = (*successors-of* (*kind cg cnid*))!*0*$]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk,\ h) \longrightarrow ((cg,cnid',cm',cp)\#stk,\ h) \mid$

UnwindNode:
$[\![$*kind g nid* = (*UnwindNode exception*);

$g \vdash$ *exception* $\triangleright$ *exceptionE*;
$[m,\ p] \vdash$ *exceptionE* $\mapsto$ *e*;

kind cg cnid = (*InvokeWithExceptionNode* - - - - - - *exEdge*);

$cm' = cm(cnid := e)]\!]$
$\Longrightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk,\ h) \longrightarrow ((cg,exEdge,cm',cp)\#stk,\ h)$

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow$ *bool*) *step-top* **.**

3.4 Big-step Execution

type-synonym *Trace* = (*IRGraph* $\times$ *ID* $\times$ *MapState* $\times$ *Params*) *list*

fun *has-return* :: *MapState* $\Rightarrow$ *bool* **where**
has-return m = ($m\ 0 \neq$ *UndefVal*)

inductive *exec* :: *Program*
$\Rightarrow$ (*IRGraph* $\times$ *ID* $\times$ *MapState* $\times$ *Params*) *list* $\times$ *FieldRefHeap*
$\Rightarrow$ *Trace*
$\Rightarrow$ (*IRGraph* $\times$ *ID* $\times$ *MapState* $\times$ *Params*) *list* $\times$ *FieldRefHeap*
$\Rightarrow$ *Trace*
$\Rightarrow$ *bool*
(- $\vdash$ - | - $\longrightarrow*$ - | -)
for *P*
where
$[\![P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h')$;
$\neg$(*has-return* m');

$l' = (l\ @\ [(g,nid,m,p)])$;

exec P $(((g',nid',m',p')\#ys),h')\ l'$ *next-state* $l'']\!]$
$\Longrightarrow$ *exec P* $(((g,nid,m,p)\#xs),h)\ l$ *next-state* l''

|
$[\![P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h')$;
has-return m';

$l' = (l\ @\ [(g,nid,m,p)])]\!]$
$\Longrightarrow$ *exec P* $(((g,nid,m,p)\#xs),h)\ l\ (((g',nid',m',p')\#ys),h')\ l'$
code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow$ *bool as Exec*) *exec* **.**

inductive *exec-debug* :: *Program*
$\Rightarrow$ (*IRGraph* $\times$ *ID* $\times$ *MapState* $\times$ *Params*) *list* $\times$ *FieldRefHeap*
$\Rightarrow$ *nat*

$\Rightarrow$ (*IRGraph* $\times$ *ID* $\times$ *MapState* $\times$ *Params*) *list* $\times$ *FieldRefHeap*
$\Rightarrow$ *bool*
(-⊢-⟶∗-∗ -)
where
⟦$n > 0$;
$p \vdash s \longrightarrow s'$;
exec-debug $p\ s'\ (n - 1)\ s''$⟧
$\Longrightarrow$ *exec-debug* $p\ s\ n\ s''$ |

⟦$n = 0$⟧
$\Longrightarrow$ *exec-debug* $p\ s\ n\ s$
code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *exec-debug* **.**

3.4.1 Heap Testing

definition *p3*:: *Params* **where**
p3 = [*IntVal32 3*]

values {(*prod.fst*(*prod.snd* (*prod.snd* (*hd* (*prod.fst res*))))) *0*
| *res*. (λx . *Some eg2-sq*) ⊢ ([(*eg2-sq*,*0*,*new-map-state*,*p3*), (*eg2-sq*,*0*,*new-map-state*,*p3*)], *new-heap*) →∗*2*∗ *res*}

definition *field-sq* :: *string* **where**
field-sq = *''sq''*

definition *eg3-sq* :: *IRGraph* **where**
eg3-sq = *irgraph* [
(*0*, *StartNode None 4*, *VoidStamp*),
(*1*, *ParameterNode 0*, *default-stamp*),
(*3*, *MulNode 1 1*, *default-stamp*),
(*4*, *StoreFieldNode 4 field-sq 3 None None 5*, *VoidStamp*),
(*5*, *ReturnNode* (*Some 3*) *None*, *default-stamp*)
]

values {*h-load-field field-sq None* (*prod.snd res*)
| *res*. (λx. *Some eg3-sq*) ⊢ ([(*eg3-sq*, *0*, *new-map-state*, *p3*), (*eg3-sq*, *0*, *new-map-state*, *p3*)], *new-heap*) →∗*3*∗ *res*}

definition *eg4-sq* :: *IRGraph* **where**
eg4-sq = *irgraph* [
(*0*, *StartNode None 4*, *VoidStamp*),
(*1*, *ParameterNode 0*, *default-stamp*),
(*3*, *MulNode 1 1*, *default-stamp*),
(*4*, *NewInstanceNode 4 ''obj-class'' None 5*, *ObjectStamp ''obj-class'' True True True*),
(*5*, *StoreFieldNode 5 field-sq 3 None* (*Some 4*) *6*, *VoidStamp*),
(*6*, *ReturnNode* (*Some 3*) *None*, *default-stamp*)

]

values $\{$*h-load-field field-sq* (*Some 0*) (*prod.snd res*) $|$ *res*.
$(\lambda x.$ *Some eg4-sq*) $\vdash$ ([(*eg4-sq*, *0*, *new-map-state*, *p3*), (*eg4-sq*, *0*, *new-map-state*, *p3*)], *new-heap*) $\rightarrow *4*$ *res*$\}$

end