

# Unspecified Veriopt Theory

July 6, 2021

## Contents

<b>1</b>	<b>Data-flow Semantics</b>	<b>1</b>
1.1	Data-flow Tree Representation . . . . .	3
1.2	Data-flow Tree Evaluation . . . . .	11
1.3	Data-flow Tree Refinement . . . . .	13
<b>2</b>	<b>Data-flow Expression-Tree Theorems</b>	<b>14</b>
2.1	Extraction and Evaluation of Expression Trees is Deterministic.	14
2.2	Example Data-flow Optimisations . . . . .	19
2.3	Monotonicity of Expression Optimization . . . . .	20
<b>3</b>	<b>Control-flow Semantics</b>	<b>21</b>
3.1	Heap . . . . .	21
3.2	Intraprocedural Semantics . . . . .	22
3.3	Interprocedural Semantics . . . . .	24
3.4	Big-step Execution . . . . .	25
3.4.1	Heap Testing . . . . .	26

## 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 calculated 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*  $\Rightarrow$  *Value*

**type-synonym** *Params* = *Value list*

**definition** *new-map-state* :: *MapState* **where**

*new-map-state* = ( $\lambda x$ . *UndefVal*)

**fun** *val-to-bool* :: *Value*  $\Rightarrow$  *bool* **where**

*val-to-bool* (*IntVal32 val*) = (if *val* = 0 then *False* else *True*) |

*val-to-bool* *v* = *False*

**fun** *bool-to-val* :: *bool*  $\Rightarrow$  *Value* **where**

*bool-to-val* *True* = (*IntVal32 1*) |

*bool-to-val* *False* = (*IntVal32 0*)

**fun** *find-index* :: '*a*  $\Rightarrow$  '*a list*  $\Rightarrow$  *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*  $\Rightarrow$  *ID*  $\Rightarrow$  *ID list* **where**

*phi-list* *g* *nid* =

(*filter* ( $\lambda x$ . (*is-PhiNode* (*kind g x*)))

(*sorted-list-of-set* (*usages g nid*)))

**fun** *input-index* :: *IRGraph*  $\Rightarrow$  *ID*  $\Rightarrow$  *ID*  $\Rightarrow$  *nat* **where**

*input-index g n n'* = *find-index n' (inputs-of (kind g n))*

**fun** *phi-inputs* :: *IRGraph*  $\Rightarrow$  *nat*  $\Rightarrow$  *ID list*  $\Rightarrow$  *ID list* **where**

*phi-inputs g i nodes* = (*map* ( $\lambda n$ . (*inputs-of* (*kind g n*))!(*i* + 1)) *nodes*)

**fun** *set-phis* :: *ID list*  $\Rightarrow$  *Value list*  $\Rightarrow$  *MapState*  $\Rightarrow$  *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*  $\Rightarrow$  (*IRNode*  $\times$  *Stamp*)  $\Rightarrow$  *ID option* **where**

```

find-node-and-stamp g (n,s) =
  find (λi. 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*  $\Rightarrow$  *ID*  $\Rightarrow$  *IRExpr*  $\Rightarrow$  *bool* (-  $\vdash$  -  $\triangleright$  - 55)  
**for** *g* **where**

*ConstantNode*:

$\llbracket \text{kind } g \ n = \text{ConstantNode } c \rrbracket$   
 $\implies g \vdash n \triangleright (\text{ConstantExpr } c) \mid$

*ParameterNode*:

$\llbracket \text{kind } g \ n = \text{ParameterNode } i; \text{stamp } g \ n = s \rrbracket$   
 $\implies g \vdash n \triangleright (\text{ParameterExpr } i \ s) \mid$

*ConditionalNode*:

$\llbracket \text{kind } g \ n = \text{ConditionalNode } c \ t \ f; \text{stamp } g \ n = s \rrbracket$   
 $\implies g \vdash c \triangleright ce;$   
 $\implies g \vdash t \triangleright te;$   
 $\implies g \vdash f \triangleright fe;$   
 $\implies g \vdash n \triangleright (\text{ConditionalExpr } ce \ te \ fe) \mid$

*AbsNode*:

$\llbracket \text{kind } g \ n = \text{AbsNode } x; \text{stamp } g \ n = s \rrbracket$   
 $\implies g \vdash x \triangleright xe;$   
 $\implies g \vdash n \triangleright (\text{UnaryExpr } \text{UnaryAbs } xe) \mid$

*NotNode*:

$\llbracket \text{kind } g \ n = \text{NotNode } x; \text{stamp } g \ n = s \rrbracket$   
 $\implies g \vdash x \triangleright xe;$   
 $\implies g \vdash n \triangleright (\text{UnaryExpr } \text{UnaryNot } xe) \mid$

*NegateNode*:

$\llbracket \text{kind } g \ n = \text{NegateNode } x; \text{stamp } g \ n = s \rrbracket$   
 $\implies g \vdash x \triangleright xe;$   
 $\implies g \vdash n \triangleright (\text{UnaryExpr } \text{UnaryNeg } xe) \mid$

*LogicNegationNode*:

$\llbracket \text{kind } g \ n = \text{LogicNegationNode } x; \text{stamp } g \ n = s \rrbracket$   
 $\implies g \vdash x \triangleright xe;$   
 $\implies g \vdash n \triangleright (\text{UnaryExpr } \text{UnaryLogicNegation } xe) \mid$

*AddNode*:

$\llbracket \text{kind } g \ n = \text{AddNode } x \ y; \text{stamp } g \ n = s \rrbracket$   
 $\implies g \vdash x \triangleright xe;$   
 $\implies g \vdash y \triangleright ye;$   
 $\implies g \vdash n \triangleright (\text{BinaryExpr } \text{BinAdd } xe \ ye) \mid$

*MulNode:*  
 $\llbracket \text{kind } g \ n = \text{MulNode } x \ y; \quad g \vdash x \triangleright xe; \quad g \vdash y \triangleright ye \rrbracket$   
 $\implies g \vdash n \triangleright (\text{BinaryExpr } \text{BinMul } xe \ ye) \mid$

*SubNode:*  
 $\llbracket \text{kind } g \ n = \text{SubNode } x \ y; \quad g \vdash x \triangleright xe; \quad g \vdash y \triangleright ye \rrbracket$   
 $\implies g \vdash n \triangleright (\text{BinaryExpr } \text{BinSub } xe \ ye) \mid$

*AndNode:*  
 $\llbracket \text{kind } g \ n = \text{AndNode } x \ y; \quad g \vdash x \triangleright xe; \quad g \vdash y \triangleright ye \rrbracket$   
 $\implies g \vdash n \triangleright (\text{BinaryExpr } \text{BinAnd } xe \ ye) \mid$

*OrNode:*  
 $\llbracket \text{kind } g \ n = \text{OrNode } x \ y; \quad g \vdash x \triangleright xe; \quad g \vdash y \triangleright ye \rrbracket$   
 $\implies g \vdash n \triangleright (\text{BinaryExpr } \text{BinOr } xe \ ye) \mid$

*XorNode:*  
 $\llbracket \text{kind } g \ n = \text{XorNode } x \ y; \quad g \vdash x \triangleright xe; \quad g \vdash y \triangleright ye \rrbracket$   
 $\implies g \vdash n \triangleright (\text{BinaryExpr } \text{BinXor } xe \ ye) \mid$

*IntegerEqualsNode:*  
 $\llbracket \text{kind } g \ n = \text{IntegerEqualsNode } x \ y; \quad g \vdash x \triangleright xe; \quad g \vdash y \triangleright ye \rrbracket$   
 $\implies g \vdash n \triangleright (\text{BinaryExpr } \text{BinIntegerEquals } xe \ ye) \mid$

*IntegerLessThanNode:*  
 $\llbracket \text{kind } g \ n = \text{IntegerLessThanNode } x \ y; \quad g \vdash x \triangleright xe; \quad g \vdash y \triangleright ye \rrbracket$   
 $\implies g \vdash n \triangleright (\text{BinaryExpr } \text{BinIntegerLessThan } xe \ ye) \mid$

*LeafNode:*  
 $\llbracket \text{is-preevaluated } (\text{kind } g \ n); \quad \text{stamp } g \ n = s \rrbracket$   
 $\implies g \vdash n \triangleright (\text{LeafExpr } n \ s)$

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

**inductive**

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

*RepNil*:

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

*RepCons*:

$\llbracket g \vdash x \triangleright xe;$   
 $g \vdash xs \triangleright_L xse \rrbracket$   
 $\implies g \vdash x \# xs \triangleright_L xe \# xse$

**code-pred** (*modes*:  $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$  as *exprListE*) *replist* .

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

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

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

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

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

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

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

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

**fun** *stamp-unary* :: *IRUnaryOp*  $\Rightarrow$  *Stamp*  $\Rightarrow$  *Stamp* **where**

*stamp-unary op* (*IntegerStamp* *b lo hi*) = *unrestricted-stamp* (*IntegerStamp* *b lo hi*) |

*stamp-unary op* - = *IllegalStamp*

**fun** *stamp-binary* :: *IRBinaryOp*  $\Rightarrow$  *Stamp*  $\Rightarrow$  *Stamp*  $\Rightarrow$  *Stamp* **where**

*stamp-binary op* (*IntegerStamp* *b1 lo1 hi1*) (*IntegerStamp* *b2 lo2 hi2*) =

```

    (if (b1 = b2) then unrestricted-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*  $\Rightarrow$  *Value*  $\Rightarrow$  *Value*  $\Rightarrow$  *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*  $\Rightarrow$  *ID*  $\Rightarrow$  *bool* **where**  
*nid*  $\notin$  *ids g*  $\implies$  *fresh-id g nid*

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

**fun** *get-fresh-id* :: *IRGraph*  $\Rightarrow$  *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*  $\Rightarrow$  *IRExpr*  $\Rightarrow$  (*IRGraph*  $\times$  *ID*)  $\Rightarrow$  *bool* (-  $\triangleleft$  -  $\rightsquigarrow$  - 55)

**and**

*unrepList* :: *IRGraph*  $\Rightarrow$  *IRExpr list*  $\Rightarrow$  (*IRGraph*  $\times$  *ID list*)  $\Rightarrow$  *bool* (-  $\triangleleft_L$  -  $\rightsquigarrow$  - 55)

**where**

*ConstantNodeSame*:

$\llbracket \text{find-node-and-stamp } g \text{ (ConstantNode } c, \text{ constantAsStamp } c) = \text{Some } nid \rrbracket$   
 $\implies g \triangleleft (\text{ConstantExpr } c) \rightsquigarrow (g, nid) \mid$

*ConstantNodeNew*:

$\llbracket \text{find-node-and-stamp } g \text{ (ConstantNode } c, \text{ constantAsStamp } c) = \text{None};$   
 $nid = \text{get-fresh-id } g;$   
 $g' = \text{add-node } nid \text{ (ConstantNode } c, \text{ constantAsStamp } c) \text{ } g \rrbracket$   
 $\implies g \triangleleft (\text{ConstantExpr } c) \rightsquigarrow (g', nid) \mid$

*ParameterNodeSame*:

$\llbracket \text{find-node-and-stamp } g \text{ (ParameterNode } i, s) = \text{Some } nid \rrbracket$   
 $\implies g \triangleleft (\text{ParameterExpr } i \text{ } s) \rightsquigarrow (g, nid) \mid$



*ParameterNodeNew:*

$\llbracket \text{find-node-and-stamp } g \text{ (ParameterNode } i, s) = \text{None};$   
 $\text{nid} = \text{get-fresh-id } g;$   
 $g' = \text{add-node nid (ParameterNode } i, s) \text{ } g \rrbracket$   
 $\implies g \triangleleft (\text{ParameterExpr } i \text{ } s) \rightsquigarrow (g', \text{nid}) \mid$

*ConditionalNodeSame:*

$\llbracket g \triangleleft_L [ce, te, fe] \rightsquigarrow (g2, [c, t, f]);$   
 $s' = \text{meet (stamp } g2 \text{ } t) (\text{stamp } g2 \text{ } f);$   
 $\text{find-node-and-stamp } g2 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{Some nid} \rrbracket$   
 $\implies g \triangleleft (\text{ConditionalExpr } ce \text{ } te \text{ } fe) \rightsquigarrow (g2, \text{nid}) \mid$

*ConditionalNodeNew:*

$\llbracket g \triangleleft_L [ce, te, fe] \rightsquigarrow (g2, [c, t, f]);$   
 $s' = \text{meet (stamp } g2 \text{ } t) (\text{stamp } g2 \text{ } f);$   
 $\text{find-node-and-stamp } g2 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{None};$   
 $\text{nid} = \text{get-fresh-id } g2;$   
 $g' = \text{add-node nid (ConditionalNode } c \text{ } t \text{ } f, s') \text{ } g2 \rrbracket$   
 $\implies g \triangleleft (\text{ConditionalExpr } ce \text{ } te \text{ } fe) \rightsquigarrow (g', \text{nid}) \mid$

*UnaryNodeSame:*

$\llbracket g \triangleleft xe \rightsquigarrow (g2, x);$   
 $s' = \text{stamp-unary op (stamp } g2 \text{ } x);$   
 $\text{find-node-and-stamp } g2 \text{ (unary-node op } x, s') = \text{Some nid} \rrbracket$   
 $\implies g \triangleleft (\text{UnaryExpr op } xe) \rightsquigarrow (g2, \text{nid}) \mid$

*UnaryNodeNew:*

$\llbracket g \triangleleft xe \rightsquigarrow (g2, x);$   
 $s' = \text{stamp-unary op (stamp } g2 \text{ } x);$   
 $\text{find-node-and-stamp } g2 \text{ (unary-node op } x, s') = \text{None};$   
 $\text{nid} = \text{get-fresh-id } g2;$   
 $g' = \text{add-node nid (unary-node op } x, s') \text{ } g2 \rrbracket$   
 $\implies g \triangleleft (\text{UnaryExpr op } xe) \rightsquigarrow (g', \text{nid}) \mid$

*BinaryNodeSame:*

$\llbracket g \triangleleft_L [xe, ye] \rightsquigarrow (g2, [x, y]);$   
 $s' = \text{stamp-binary op (stamp } g2 \text{ } x) (\text{stamp } g2 \text{ } y);$   
 $\text{find-node-and-stamp } g2 \text{ (bin-node op } x \text{ } y, s') = \text{Some nid} \rrbracket$   
 $\implies g \triangleleft (\text{BinaryExpr op } xe \text{ } ye) \rightsquigarrow (g2, \text{nid}) \mid$

*BinaryNodeNew:*

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

*AllLeafNodes:*

*stamp g nid = s*

$\implies g \triangleleft (\text{LeafExpr } \text{nid } s) \rightsquigarrow (g, \text{nid}) \mid$

*UnrepNil:*

$g \triangleleft_L [] \rightsquigarrow (g, []) \mid$

*UnrepCons:*

$\llbracket g \triangleleft xe \rightsquigarrow (g2, x);$

$g2 \triangleleft_L xes \rightsquigarrow (g3, xs) \rrbracket$

$\implies g \triangleleft_L (xe \# xes) \rightsquigarrow (g3, x \# xs)$

**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 .*

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

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

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

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

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

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

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

$$\begin{array}{c}
\frac{
\begin{array}{l}
g \triangleleft_L [xe, ye] \rightsquigarrow (g2, [x, y]) \quad s' = \text{stamp-binary op } (\text{stamp } g2 \ x) (\text{stamp } g2 \ y) \\
\quad \text{find-node-and-stamp } g2 \ (\text{bin-node op } x \ y, s') = \text{None} \\
nid = \text{get-fresh-id } g2 \quad g' = \text{add-node } nid \ (\text{bin-node op } x \ y, s') \ g2
\end{array}
}{g \triangleleft \text{BinaryExpr op } xe \ ye \rightsquigarrow (g', nid)} \\
\\
\frac{
\begin{array}{l}
g \triangleleft xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op } (\text{stamp } g2 \ x) \\
\text{find-node-and-stamp } g2 \ (\text{unary-node op } x, s') = \text{Some } nid
\end{array}
}{g \triangleleft \text{UnaryExpr op } xe \rightsquigarrow (g2, nid)} \\
\\
\frac{
\begin{array}{l}
g \triangleleft xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op } (\text{stamp } g2 \ x) \\
\text{find-node-and-stamp } g2 \ (\text{unary-node op } x, s') = \text{None} \\
nid = \text{get-fresh-id } g2 \quad g' = \text{add-node } nid \ (\text{unary-node op } x, s') \ g2
\end{array}
}{g \triangleleft \text{UnaryExpr op } xe \rightsquigarrow (g', nid)} \\
\\
\frac{
\text{stamp } g \ nid = s
}{g \triangleleft \text{LeafExpr } nid \ s \rightsquigarrow (g, nid)}
\end{array}$$

**definition** *sq-param0* :: *IRExpr* **where**

*sq-param0* = *BinaryExpr BinMul*  
*(ParameterExpr 0 (IntegerStamp 32 (- 2147483648) 2147483647))*  
*(ParameterExpr 0 (IntegerStamp 32 (- 2147483648) 2147483647))*

**values**  $\{(nid, g) . (eg2\text{-sq} \triangleleft sq\text{-param0} \rightsquigarrow (g, nid))\}$

## 1.2 Data-flow Tree Evaluation

**inductive**

*evaltree* :: *MapState*  $\Rightarrow$  *Params*  $\Rightarrow$  *IRExpr*  $\Rightarrow$  *Value*  $\Rightarrow$  *bool* (*[-,]*  $\vdash$  -  $\mapsto$  - 55)  
**for** *m p* **where**

*ConstantExpr*:

$\llbracket c \neq \text{UndefVal} \rrbracket$   
 $\implies [m, p] \vdash (\text{ConstantExpr } c) \mapsto c \mid$

*ParameterExpr*:

$\llbracket \text{valid-value } s \ (p!i) \rrbracket$   
 $\implies [m, p] \vdash (\text{ParameterExpr } i \ s) \mapsto p!i \mid$

*ConditionalExpr*:

$\llbracket [m, p] \vdash ce \mapsto cond; \text{branch} = (\text{if val-to-bool } cond \text{ then } te \text{ else } fe); [m, p] \vdash \text{branch} \mapsto v \rrbracket$   
 $\implies [m, p] \vdash (\text{ConditionalExpr } ce \ te \ fe) \mapsto v \mid$

*UnaryExpr*:

$\llbracket [m, p] \vdash xe \mapsto v \rrbracket$   
 $\implies [m, p] \vdash (\text{UnaryExpr op } xe) \mapsto \text{unary-eval op } v \mid$

*BinaryExpr:*

$\llbracket [m,p] \vdash xe \mapsto x;$   
 $[m,p] \vdash ye \mapsto y \rrbracket$   
 $\implies [m,p] \vdash (BinaryExpr\ op\ xe\ ye) \mapsto bin\text{-}eval\ op\ x\ y \mid$

*LeafExpr:*

$\llbracket val = m\ nid;$   
 $valid\text{-}value\ s\ val \rrbracket$   
 $\implies [m,p] \vdash LeafExpr\ nid\ s \mapsto val$

$$\begin{array}{c}
\frac{c \neq UndefinedVal}{[m,p] \vdash ConstantExpr\ c \mapsto c} \\
\\
\frac{valid\text{-}value\ s\ p_{[i]}}{[m,p] \vdash ParameterExpr\ i\ s \mapsto p_{[i]}} \\
\\
\frac{[m,p] \vdash ce \mapsto cond \quad \text{branch} = (if\ IRTreeEval.val\text{-}to\text{-}bool\ cond\ then\ te\ else\ fe) \quad [m,p] \vdash branch \mapsto v}{[m,p] \vdash ConditionalExpr\ ce\ te\ fe \mapsto v} \\
\\
\frac{[m,p] \vdash xe \mapsto v}{[m,p] \vdash UnaryExpr\ op\ xe \mapsto unary\text{-}eval\ op\ v} \\
\\
\frac{[m,p] \vdash xe \mapsto x \quad [m,p] \vdash ye \mapsto y}{[m,p] \vdash BinaryExpr\ op\ xe\ ye \mapsto bin\text{-}eval\ op\ x\ y} \\
\\
\frac{val = m\ nid \quad valid\text{-}value\ s\ val}{[m,p] \vdash LeafExpr\ nid\ s \mapsto val}
\end{array}$$

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

**inductive**

$evaltrees :: MapState \Rightarrow Params \Rightarrow IRExpr\ list \Rightarrow Value\ list \Rightarrow bool\ ([-,] \vdash - \mapsto_L$   
- 55)

**for**  $m\ p$  **where**

*EvalNil:*

$[m,p] \vdash [] \mapsto_L [] \mid$

*EvalCons:*

$\llbracket [m,p] \vdash x \mapsto xval;$   
 $[m,p] \vdash yy \mapsto_L yyval \rrbracket$

```

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

code-pred (modes:  $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$  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*]:  $(e1 \leq e2) \longleftrightarrow (\forall m\ p\ v. ([m,p] \vdash e1 \mapsto v) \longrightarrow ([m,p] \vdash e2 \mapsto v))$

**definition**

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

**instance proof**

```

fix x y z :: IRExpr
show  $x < y \longleftrightarrow x \leq y \wedge \neg (y \leq x)$  by (simp add: equiv-exprs-def; auto)
show  $x \leq x$  by simp
show  $x \leq y \implies y \leq z \implies x \leq z$  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*:

```

 $g \vdash n \triangleright e \implies$ 
 $kind\ g\ n = ConstantNode\ c \implies$ 
 $e = ConstantExpr\ c$ 
by (induction rule: rep.induct; auto)

```

**lemma** *rep-parameter*:

```

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

```

**lemma** *rep-conditional*:

```

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

```

**lemma** *rep-abs*:

```

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

```

**lemma** *rep-not*:

```

 $g \vdash n \triangleright e \implies$ 
 $kind\ g\ n = NotNode\ x \implies$ 
 $(\exists xe. e = UnaryExpr\ UnaryNot\ xe)$ 
by (induction rule: rep.induct; auto)

```

**lemma** *rep-negate*:

```

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

```

**lemma** *rep-logicnegation*:

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

**lemma** *rep-add*:

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

**lemma** *rep-sub*:

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

**lemma** *rep-mul*:

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

**lemma** *rep-and*:

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

**lemma** *rep-or*:

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

**lemma** *rep-xor*:

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

**lemma** *rep-integer-equals*:

$g \vdash n \triangleright e \implies$   
 $kind\ g\ n = IntegerEqualsNode\ x\ y \implies$   
 $(\exists xe\ ye. e = BinaryExpr\ BinIntegerEquals\ xe\ ye)$   
**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*:

$$[m, p] \vdash e \mapsto v1 \implies [m, p] \vdash e \mapsto v2 \implies v1 = v2$$

**apply** (*induction arbitrary: v2 rule: evaltree.induct*)  
**by** (*elim EvalTreeE; auto*)+

**lemma** *evalAllDet*:

$$[m, p] \vdash e \mapsto_L v1 \implies [m, p] \vdash e \mapsto_L v2 \implies v1 = v2$$

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

```

```

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

```

```

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

```

```

lemma valid-int32[elim]:
shows valid-value (IntegerStamp 32 l h) val  $\implies$ 
   $(\exists v. val = \text{IntVal32 } v)$ 
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  $\implies$ 
   $(\exists v. val = \text{IntVal64 } v)$ 
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  $ev: [m, p] \vdash \text{LeafExpr } i \text{ (IntegerStamp 32 lo hi)} \mapsto val$ 
shows  $\exists v. val = (\text{IntVal32 } v)$ 

```

```

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

```

```

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

```

```

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*]  $\vdash expr \mapsto val \implies$   
*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)))
  ≤ (LeafExpr 1 default-stamp) (is ?L ≤ ?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))))
  ≤ (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 ≤ e'
  shows (UnaryExpr op e) ≤ (UnaryExpr op e')
  using UnaryExpr assms by auto

```

```

lemma mono-binary:
  assumes x ≤ x'
  assumes y ≤ y'
  shows (BinaryExpr op x y) ≤ (BinaryExpr op x' y')
  using BinaryExpr assms by auto

```

```

lemma mono-conditional:
  assumes ce ≤ 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  $b: branch = (if\ val\text{-}to\text{-}bool\ cond\ then\ te\ else\ fe)$ 
  define  $branch'$  where  $b': branch' = (if\ val\text{-}to\text{-}bool\ cond\ then\ te'\ else\ fe')$ 
  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**  $new\text{-}heap :: ('a, 'b) \ DynamicHeap$  **where**  
 $new\text{-}heap = ((\lambda f. \lambda p. \text{UndefVal}), 0)$

### 3.2 Intraprocedural Semantics

Intraprocedural semantics are given as a small-step semantics.

Within the context of a graph, the configuration triple,  $(ID, \text{MethodState}, \text{Heap})$ , is related to the subsequent configuration.

**inductive**  $step :: IRGraph \Rightarrow Params \Rightarrow (ID \times MapState \times FieldRefHeap) \Rightarrow (ID \times MapState \times FieldRefHeap) \Rightarrow bool$   
 $(-, - \vdash - \rightarrow - \ 55)$  **for**  $g \ p$  **where**

*SequentialNode:*

$\llbracket is\text{-}sequential\text{-}node \ (kind \ g \ nid);$   
 $nid' = (successors\text{-}of \ (kind \ g \ nid))!0 \rrbracket$   
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m, h) \mid$

*IfNode:*

$\llbracket kind \ g \ nid = (IfNode \ cond \ tb \ fb);$   
 $g \vdash cond \triangleright condE;$   
 $[m, p] \vdash condE \mapsto val;$   
 $nid' = (if \ val\text{-}to\text{-}bool \ val \ then \ tb \ else \ fb) \rrbracket$   
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m, h) \mid$

*EndNodes:*

$\llbracket is\text{-}AbstractEndNode \ (kind \ g \ nid);$   
 $merge = any\text{-}usage \ g \ nid;$   
 $is\text{-}AbstractMergeNode \ (kind \ g \ merge);$   
 $i = find\text{-}index \ nid \ (inputs\text{-}of \ (kind \ g \ merge));$   
 $phis = (phi\text{-}list \ g \ merge);$   
 $inps = (phi\text{-}inputs \ g \ i \ phis);$   
 $g \vdash inps \triangleright_L inpsE;$   
 $[m, p] \vdash inpsE \mapsto_L vs;$   
 $m' = set\text{-}phis \ phis \ vs \ m \rrbracket$   
 $\implies g, p \vdash (nid, m, h) \rightarrow (merge, m', h) \mid$

*NewInstanceNode:*

$\llbracket kind \ g \ nid = (NewInstanceNode \ nid \ f \ obj \ nid');$   
 $(h', ref) = h\text{-}new\text{-}inst \ h;$   
 $m' = m(nid := ref) \rrbracket$   
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m', h') \mid$

*LoadFieldNode:*

$$\begin{aligned} & \llbracket \text{kind } g \text{ nid} = (\text{LoadFieldNode } \text{nid } f \text{ (Some obj) nid'}) \rrbracket; \\ & g \vdash \text{obj} \triangleright \text{objE}; \\ & [m, p] \vdash \text{objE} \mapsto \text{ObjRef ref}; \\ & h\text{-load-field } f \text{ ref } h = v; \\ & m' = m(\text{nid} := v) \\ \implies & g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m', h) \mid \end{aligned}$$

*SignedDivNode:*

$$\begin{aligned} & \llbracket \text{kind } g \text{ nid} = (\text{SignedDivNode } \text{nid } x \text{ y zero sb nxt}) \rrbracket; \\ & g \vdash x \triangleright xe; \\ & g \vdash y \triangleright ye; \\ & [m, p] \vdash xe \mapsto v1; \\ & [m, p] \vdash ye \mapsto v2; \\ & v = (\text{intval-div } v1 \text{ } v2); \\ & m' = m(\text{nid} := v) \\ \implies & g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nxt}, m', h) \mid \end{aligned}$$

*SignedRemNode:*

$$\begin{aligned} & \llbracket \text{kind } g \text{ nid} = (\text{SignedRemNode } \text{nid } x \text{ y zero sb nxt}) \rrbracket; \\ & g \vdash x \triangleright xe; \\ & g \vdash y \triangleright ye; \\ & [m, p] \vdash xe \mapsto v1; \\ & [m, p] \vdash ye \mapsto v2; \\ & v = (\text{intval-mod } v1 \text{ } v2); \\ & m' = m(\text{nid} := v) \\ \implies & g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nxt}, m', h) \mid \end{aligned}$$

*StaticLoadFieldNode:*

$$\begin{aligned} & \llbracket \text{kind } g \text{ nid} = (\text{LoadFieldNode } \text{nid } f \text{ None nid'}) \rrbracket; \\ & h\text{-load-field } f \text{ None } h = v; \\ & m' = m(\text{nid} := v) \\ \implies & g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m', h) \mid \end{aligned}$$

*StoreFieldNode:*

$$\begin{aligned} & \llbracket \text{kind } g \text{ nid} = (\text{StoreFieldNode } \text{nid } f \text{ newval - (Some obj) nid'}) \rrbracket; \\ & g \vdash \text{newval} \triangleright \text{newvalE}; \\ & g \vdash \text{obj} \triangleright \text{objE}; \\ & [m, p] \vdash \text{newvalE} \mapsto \text{val}; \\ & [m, p] \vdash \text{objE} \mapsto \text{ObjRef ref}; \\ & h' = h\text{-store-field } f \text{ ref val } h; \\ & m' = m(\text{nid} := \text{val}) \\ \implies & g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m', h') \mid \end{aligned}$$

*StaticStoreFieldNode:*

$$\begin{aligned} & \llbracket \text{kind } g \text{ nid} = (\text{StoreFieldNode } \text{nid } f \text{ newval - None nid'}) \rrbracket; \\ & g \vdash \text{newval} \triangleright \text{newvalE}; \\ & [m, p] \vdash \text{newvalE} \mapsto \text{val}; \\ & h' = h\text{-store-field } f \text{ None val } h; \end{aligned}$$

$$m' = m(nid := val) \\ \implies g, p \vdash (nid, m, h) \rightarrow (nid', m', h')$$

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

### 3.3 Interprocedural Semantics

**type-synonym** *Signature* = *string*

**type-synonym** *Program* = *Signature*  $\rightarrow$  *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*:

$$\llbracket g, p \vdash (nid, m, h) \rightarrow (nid', m', h') \rrbracket \\ \implies P \vdash ((g, nid, m, p) \# stk, h) \longrightarrow ((g, nid', m', p) \# stk, h') \mid$$

*InvokeNodeStep*:

$\llbracket is-Invoke (kind\ g\ nid) \rrbracket$ ;

*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*;

$\llbracket m, p \rrbracket \vdash argsE \mapsto_L p \rrbracket$

$$\implies P \vdash ((g, nid, m, p) \# stk, h) \longrightarrow ((targetGraph, 0, m', p') \# (g, nid, m, p) \# stk, h)$$

|

*ReturnNode*:

$\llbracket kind\ g\ nid = (ReturnNode\ (Some\ expr)\ -) \rrbracket$ ;

*g*  $\vdash$  *expr*  $\triangleright$  *e*;

$\llbracket m, p \rrbracket \vdash e \mapsto v$ ;

*cm'* = *cm*(*cnid* := *v*);

*cnid'* = (*successors-of* (*kind cg cnid*))!0

$$\implies P \vdash ((g, nid, m, p) \# (cg, cnid, cm, cp) \# stk, h) \longrightarrow ((cg, cnid', cm', cp) \# stk, h) \mid$$

*ReturnNodeVoid*:

$\llbracket kind\ g\ nid = (ReturnNode\ None\ -) \rrbracket$ ;

*cm'* = *cm*(*cnid* := (*ObjRef* (*Some* (*2048*))));

*cnid'* = (*successors-of* (*kind cg cnid*))!0

$$\implies P \vdash ((g, nid, m, p) \# (cg, cnid, cm, cp) \# stk, h) \longrightarrow ((cg, cnid', cm', cp) \# stk, h) \mid$$



*UnwindNode:*

$\llbracket \text{kind } g \text{ nid} = (\text{UnwindNode } \text{exception}) \rrbracket$ ;

$g \vdash \text{exception} \triangleright \text{exceptionE};$

$[m, p] \vdash \text{exceptionE} \mapsto e;$

$\text{kind } cg \text{ cnid} = (\text{InvokeWithExceptionNode} \text{ - - - - - } \text{exEdge});$

$cm' = cm(\text{cnid} := e)$

$\implies P \vdash ((g, \text{nid}, m, p) \# (cg, \text{cnid}, cm, cp) \# \text{stk}, h) \longrightarrow ((cg, \text{exEdge}, cm', cp) \# \text{stk}, h)$

**code-pred** (*modes*:  $i \Rightarrow i \Rightarrow o \Rightarrow \text{bool}$ ) *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*

( $- \vdash - \mid - \longrightarrow^* - \mid -$ )

**for** *P*

**where**

$\llbracket P \vdash (((g, \text{nid}, m, p) \# xs), h) \longrightarrow (((g', \text{nid}', m', p') \# ys), h') \rrbracket$ ;

$\neg(\text{has-return } m')$ ;

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

$\text{exec } P (((g', \text{nid}', m', p') \# ys), h') \text{ } l' \text{ next-state } l''$

$\implies \text{exec } P (((g, \text{nid}, m, p) \# xs), h) \text{ } l \text{ next-state } l''$

$\mid$   
 $\llbracket P \vdash (((g, \text{nid}, m, p) \# xs), h) \longrightarrow (((g', \text{nid}', m', p') \# ys), h') \rrbracket$ ;  
*has-return* *m'*;

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

$\implies \text{exec } P (((g, \text{nid}, m, p) \# xs), h) \text{ } l \text{ } (((g', \text{nid}', m', p') \# ys), h') \text{ } l'$

**code-pred** (*modes*:  $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow \text{bool}$  as *Exec*) *exec* .

**inductive** *exec-debug* :: *Program*

$\Rightarrow$  (*IRGraph*  $\times$  *ID*  $\times$  *MapState*  $\times$  *Params*) *list*  $\times$  *FieldRefHeap*

$\Rightarrow$  *nat*

```

    ⇒ (IRGraph × ID × MapState × Params) list × FieldRefHeap
    ⇒ bool
  (⊢ → * -)
  where
    ⌊n > 0;
    p ⊢ s → s';
    exec-debug p s' (n - 1) s'⌋
    ⇒ exec-debug p s n s'' |

    ⌊n = 0⌋
    ⇒ exec-debug p s n s
  code-pred (modes: i ⇒ i ⇒ i ⇒ o ⇒ 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.*  
          ( $\lambda x. \text{Some } eg4\text{-sq}$ )  $\vdash$   $[(eg4\text{-sq}, 0, \text{new-map-state}, p3), (eg4\text{-sq}, 0,$   
*new-map-state, p3)], new-heap*)  $\rightarrow^*4^*$  *res*}

**end**