

# Unspecified Veriopt Theory

January 8, 2022

## Contents

<b>1</b>	<b>Data-flow Semantics</b>	<b>1</b>
1.1	Data-flow Tree Representation . . . . .	2
1.2	Data-flow Tree Evaluation . . . . .	3
1.3	Data-flow Tree Refinement . . . . .	6
<b>2</b>	<b>Tree to Graph</b>	<b>6</b>
<b>3</b>	<b>Data-flow Expression-Tree Theorems</b>	<b>16</b>
3.1	Extraction and Evaluation of Expression Trees is Deterministic.	16
3.2	Example Data-flow Optimisations . . . . .	23
3.3	Monotonicity of Expression Optimization . . . . .	24
<b>4</b>	<b>Control-flow Semantics</b>	<b>43</b>
4.1	Heap . . . . .	43
4.2	Intraprocedural Semantics . . . . .	43
4.3	Interprocedural Semantics . . . . .	46
4.4	Big-step Execution . . . . .	47
4.4.1	Heap Testing . . . . .	48
<b>5</b>	<b>Properties of Control-flow Semantics</b>	<b>49</b>

## 1 Data-flow Semantics

```
theory IRTreeEval
  imports
    Graph.Values
    Graph.Stamp
    HOL-Library.Word
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 ID = nat
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)
```

## 1.1 Data-flow Tree Representation

```
datatype IRUnaryOp =
  UnaryAbs
| UnaryNeg
| UnaryNot
| UnaryLogicNegation
| UnaryNarrow (ir-inputBits: nat) (ir-resultBits: nat)
| UnarySignExtend (ir-inputBits: nat) (ir-resultBits: nat)
| UnaryZeroExtend (ir-inputBits: nat) (ir-resultBits: nat)
```

```
datatype IRBinaryOp =
  BinAdd
| BinMul
| BinSub
| BinAnd
| BinOr
| BinXor
```

```

| BinLeftShift
| BinRightShift
| BinURightShift
| BinIntegerEquals
| BinIntegerLessThan
| BinIntegerBelow

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)

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

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

| ConstantExpr (ir-const: Value)
| ConstantVar (ir-name: string)
| VariableExpr (ir-name: string) (ir-stamp: Stamp)

fun is-ground :: IRExpr ⇒ bool where
  is-ground (UnaryExpr op e) = is-ground e |
  is-ground (BinaryExpr op e1 e2) = (is-ground e1 ∧ is-ground e2) |
  is-ground (ConditionalExpr b e1 e2) = (is-ground b ∧ is-ground e1 ∧ is-ground
e2) |
  is-ground (ParameterExpr i s) = True |
  is-ground (LeafExpr n s) = True |
  is-ground (ConstantExpr v) = True |
  is-ground (ConstantVar name) = False |
  is-ground (VariableExpr name s) = False

typedef GroundExpr = { e :: IRExpr . is-ground e }
using is-ground.simps(6) by blast

```

## 1.2 Data-flow Tree Evaluation

```

fun unary-eval :: IRUnaryOp ⇒ Value ⇒ Value where
  unary-eval UnaryAbs v = intval-abs v |
  unary-eval UnaryNeg v = intval-negate v |
  unary-eval UnaryNot v = intval-not v |
  unary-eval UnaryLogicNegation (IntVal32 v1) = (if v1 = 0 then (IntVal32 1) else
(IntVal32 0)) |
  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 |

```

$\text{bin-eval BinSub } v1 \ v2 = \text{intval-sub } v1 \ v2 \mid$   
 $\text{bin-eval BinAnd } v1 \ v2 = \text{intval-and } v1 \ v2 \mid$   
 $\text{bin-eval BinOr } v1 \ v2 = \text{intval-or } v1 \ v2 \mid$   
 $\text{bin-eval BinXor } v1 \ v2 = \text{intval-xor } v1 \ v2 \mid$   
 $\text{bin-eval BinLeftShift } v1 \ v2 = \text{intval-left-shift } v1 \ v2 \mid$   
 $\text{bin-eval BinRightShift } v1 \ v2 = \text{intval-right-shift } v1 \ v2 \mid$   
 $\text{bin-eval BinURightShift } v1 \ v2 = \text{intval-uright-shift } v1 \ v2 \mid$   
 $\text{bin-eval BinIntegerEquals } v1 \ v2 = \text{intval-equals } v1 \ v2 \mid$   
 $\text{bin-eval BinIntegerLessThan } v1 \ v2 = \text{intval-less-than } v1 \ v2 \mid$   
 $\text{bin-eval BinIntegerBelow } v1 \ v2 = \text{intval-below } v1 \ v2$

**inductive** *not-undef-or-fail* :: *Value*  $\Rightarrow$  *Value*  $\Rightarrow$  *bool* **where**  
 $\llbracket \text{value} \neq \text{UndefVal} \rrbracket \implies \text{not-undef-or-fail value value}$

**notation** (*latex output*)  
 $\text{not-undef-or-fail } (- = -)$

**inductive**  
 $\text{evaltree} :: \text{MapState} \Rightarrow \text{Params} \Rightarrow \text{IRExpr} \Rightarrow \text{Value} \Rightarrow \text{bool} \ (\llbracket -, - \rrbracket \vdash - \mapsto - \ 55)$   
**for** *m p* **where**

*ConstantExpr*:  
 $\llbracket \text{valid-value } (\text{constantAsStamp } c) \ c \rrbracket$   
 $\implies [m, p] \vdash (\text{ConstantExpr } c) \mapsto c \mid$

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

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

*UnaryExpr*:  
 $\llbracket [m, p] \vdash xe \mapsto v;$   
 $\text{result} = (\text{unary-eval op } v);$   
 $\text{result} \neq \text{UndefVal} \rrbracket$   
 $\implies [m, p] \vdash (\text{UnaryExpr op } xe) \mapsto \text{result} \mid$

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

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

*evalRules*

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

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

**inductive**

*evaltrees* :: *MapState*  $\Rightarrow$  *Params*  $\Rightarrow$  *IRExpr list*  $\Rightarrow$  *Value list*  $\Rightarrow$  *bool* ( $[-, \cdot] \vdash \cdot \mapsto_L$   
- 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* .

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

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

**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**  $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$  **by** simp  
**qed**  
**end**  
**end**

## 2 Tree to Graph

**theory** *TreeToGraph*  
**imports**  
*Semantics.IRTreeEval*  
*Graph.IRGraph*  
**begin**

**fun** *find-node-and-stamp* :: *IRGraph*  $\Rightarrow$  (*IRNode*  $\times$  *Stamp*)  $\Rightarrow$  *ID option* **where**  
*find-node-and-stamp* *g* (*n,s*) =  
*find* ( $\lambda i. kind\ g\ i = n \wedge stamp\ g\ i = s$ ) (*sorted-list-of-set*(*ids g*))

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

```
fun is-preevaluated :: IRNode ⇒ bool where
  is-preevaluated (InvokeNode n - - - -) = True |
  is-preevaluated (InvokeWithExceptionNode n - - - - -) = True |
  is-preevaluated (NewInstanceNode n - - -) = True |
  is-preevaluated (LoadFieldNode n - - -) = True |
  is-preevaluated (SignedDivNode n - - - - -) = True |
  is-preevaluated (SignedRemNode n - - - - -) = True |
  is-preevaluated (ValuePhiNode n - -) = 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]]
```

$$\implies g \vdash n \simeq (\text{UnaryExpr } \text{UnaryNeg } xe) \mid$$

*LogicNegationNode:*

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{LogicNegationNode } x; \\ & \quad g \vdash x \simeq xe \rrbracket \\ & \implies g \vdash n \simeq (\text{UnaryExpr } \text{UnaryLogicNegation } xe) \mid \end{aligned}$$

*AddNode:*

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{AddNode } x \ y; \\ & \quad g \vdash x \simeq xe; \\ & \quad g \vdash y \simeq ye \rrbracket \\ & \implies g \vdash n \simeq (\text{BinaryExpr } \text{BinAdd } xe \ ye) \mid \end{aligned}$$

*MulNode:*

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{MulNode } x \ y; \\ & \quad g \vdash x \simeq xe; \\ & \quad g \vdash y \simeq ye \rrbracket \\ & \implies g \vdash n \simeq (\text{BinaryExpr } \text{BinMul } xe \ ye) \mid \end{aligned}$$

*SubNode:*

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{SubNode } x \ y; \\ & \quad g \vdash x \simeq xe; \\ & \quad g \vdash y \simeq ye \rrbracket \\ & \implies g \vdash n \simeq (\text{BinaryExpr } \text{BinSub } xe \ ye) \mid \end{aligned}$$

*AndNode:*

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{AndNode } x \ y; \\ & \quad g \vdash x \simeq xe; \\ & \quad g \vdash y \simeq ye \rrbracket \\ & \implies g \vdash n \simeq (\text{BinaryExpr } \text{BinAnd } xe \ ye) \mid \end{aligned}$$

*OrNode:*

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{OrNode } x \ y; \\ & \quad g \vdash x \simeq xe; \\ & \quad g \vdash y \simeq ye \rrbracket \\ & \implies g \vdash n \simeq (\text{BinaryExpr } \text{BinOr } xe \ ye) \mid \end{aligned}$$

*XorNode:*

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{XorNode } x \ y; \\ & \quad g \vdash x \simeq xe; \\ & \quad g \vdash y \simeq ye \rrbracket \\ & \implies g \vdash n \simeq (\text{BinaryExpr } \text{BinXor } xe \ ye) \mid \end{aligned}$$

*IntegerBelowNode:*

$$\begin{aligned} & \llbracket \text{kind } g \ n = \text{IntegerBelowNode } x \ y; \\ & \quad g \vdash x \simeq xe; \\ & \quad g \vdash y \simeq ye \rrbracket \\ & \implies g \vdash n \simeq (\text{BinaryExpr } \text{BinIntegerBelow } xe \ ye) \mid \end{aligned}$$



*IntegerEqualsNode:*

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

*IntegerLessThanNode:*

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

*NarrowNode:*

$\llbracket \text{kind } g \ n = \text{NarrowNode } \text{inputBits } \text{resultBits } x; \\ g \vdash x \simeq xe \rrbracket \\ \implies g \vdash n \simeq (\text{UnaryExpr } (\text{UnaryNarrow } \text{inputBits } \text{resultBits}) \ xe) \mid$

*SignExtendNode:*

$\llbracket \text{kind } g \ n = \text{SignExtendNode } \text{inputBits } \text{resultBits } x; \\ g \vdash x \simeq xe \rrbracket \\ \implies g \vdash n \simeq (\text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \ xe) \mid$

*ZeroExtendNode:*

$\llbracket \text{kind } g \ n = \text{ZeroExtendNode } \text{inputBits } \text{resultBits } x; \\ g \vdash x \simeq xe \rrbracket \\ \implies g \vdash n \simeq (\text{UnaryExpr } (\text{UnaryZeroExtend } \text{inputBits } \text{resultBits}) \ xe) \mid$

*LeafNode:*

$\llbracket \text{is-preevaluated } (\text{kind } g \ n); \\ \text{stamp } g \ n = s \rrbracket \\ \implies g \vdash n \simeq (\text{LeafExpr } n \ s)$

**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* ( $- \vdash - \simeq_L -$  55)  
**for** *g* **where**

*RepNil:*

$g \vdash [] \simeq_L [] \mid$

*RepCons:*

$\llbracket g \vdash x \simeq xe; \\ g \vdash xs \simeq_L xse \rrbracket \\ \implies g \vdash x \# xs \simeq_L xe \# xse$

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

*repRules*

$$\begin{array}{c}
\frac{\text{kind } g \ n = \text{ConstantNode } c}{g \vdash n \simeq \text{ConstantExpr } c} \\
\\
\frac{\text{kind } g \ n = \text{ParameterNode } i \quad \text{stamp } g \ n = s}{g \vdash n \simeq \text{ParameterExpr } i \ s} \\
\\
\frac{\text{kind } g \ n = \text{AbsNode } x \quad g \vdash x \simeq xe}{g \vdash n \simeq \text{UnaryExpr } \text{UnaryAbs } xe} \\
\\
\frac{\text{kind } g \ n = \text{AddNode } x \ y \quad g \vdash x \simeq xe \quad g \vdash y \simeq ye}{g \vdash n \simeq \text{BinaryExpr } \text{BinAdd } xe \ ye} \\
\\
\frac{\text{kind } g \ n = \text{MulNode } x \ y \quad g \vdash x \simeq xe \quad g \vdash y \simeq ye}{g \vdash n \simeq \text{BinaryExpr } \text{BinMul } xe \ ye} \\
\\
\frac{\text{kind } g \ n = \text{SubNode } x \ y \quad g \vdash x \simeq xe \quad g \vdash y \simeq ye}{g \vdash n \simeq \text{BinaryExpr } \text{BinSub } xe \ ye} \\
\\
\frac{\text{is-preevaluated } (\text{kind } g \ n) \quad \text{stamp } g \ n = s}{g \vdash n \simeq \text{LeafExpr } n \ s}
\end{array}$$

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

**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*

**definition** *fixed-32* :: *IRBinaryOp* *set* **where**  
*fixed-32* = {*BinIntegerEquals*, *BinIntegerLessThan*, *BinIntegerBelow*}

**fun** *stamp-binary* :: *IRBinaryOp*  $\Rightarrow$  *Stamp*  $\Rightarrow$  *Stamp*  $\Rightarrow$  *Stamp* **where**  
*stamp-binary* *op* (*IntegerStamp* *b1 lo1 hi1*) (*IntegerStamp* *b2 lo2 hi2*) =  
(case *op*  $\in$  *fixed-32* of *True*  $\Rightarrow$  *unrestricted-stamp* (*IntegerStamp* 32 *lo1 hi1*) |  
*False*  $\Rightarrow$   
(if (*b1* = *b2*) then *unrestricted-stamp* (*IntegerStamp* *b1 lo1 hi1*) else *IllegalStamp*)) |

*stamp-binary* *op* - - = *IllegalStamp*

**fun** *stamp-expr* :: *IRExpr*  $\Rightarrow$  *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 |
unary-node (UnaryNarrow ib rb) v = NarrowNode ib rb v |
unary-node (UnarySignExtend ib rb) v = SignExtendNode ib rb v |
unary-node (UnaryZeroExtend ib rb) v = ZeroExtendNode ib rb 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 BinLeftShift x y = LeftShiftNode x y |
bin-node BinRightShift x y = RightShiftNode x y |
bin-node BinURightShift x y = UnsignedRightShiftNode x y |
bin-node BinIntegerEquals x y = IntegerEqualsNode x y |
bin-node BinIntegerLessThan x y = IntegerLessThanNode x y |
bin-node BinIntegerBelow x y = IntegerBelowNode x y

```

```

fun choose-32-64 :: int ⇒ int64 ⇒ Value where
choose-32-64 bits val =
  (if bits = 32
   then (IntVal32 (ucast val))
   else (IntVal64 (val)))

```

```

inductive fresh-id :: IRGraph ⇒ ID ⇒ bool where
n ∉ ids g ⇒⇒ fresh-id g n

```

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

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

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

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

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

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

```

$$\implies g \triangleleft (\text{ConditionalExpr } ce \ te \ fe) \rightsquigarrow (g', n) \mid$$

*UnaryNodeSame:*

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

*UnaryNodeNew:*

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

*BinaryNodeSame:*

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

*BinaryNodeNew:*

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

*AllLeafNodes:*

$$\begin{aligned} & \text{stamp } g \ n = s \\ & \implies g \triangleleft (\text{LeafExpr } n \ s) \rightsquigarrow (g, n) \mid \end{aligned}$$

*UnrepNil:*

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

*UnrepCons:*

$$\begin{aligned} & \llbracket g \triangleleft xe \rightsquigarrow (g2, x); \\ & \quad g2 \triangleleft_L xes \rightsquigarrow (g3, xs) \rrbracket \\ & \implies g \triangleleft_L (xe \# xes) \rightsquigarrow (g3, x \# xs) \end{aligned}$$

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

*unrep* .

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{l} g \triangleleft xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op } (\text{stamp } g2 \text{ } x) \\ \text{find-node-and-stamp } g2 \text{ (unary-node op } x, s') = \text{Some } n \end{array}}{g \triangleleft \text{UnaryExpr op } xe \rightsquigarrow (g2, n)}$$

$$\frac{\begin{array}{l} g \triangleleft xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op } (\text{stamp } g2 \text{ } x) \\ \text{find-node-and-stamp } g2 \text{ (unary-node op } x, s') = \text{None} \\ n = \text{get-fresh-id } g2 \quad g' = \text{add-node } n \text{ (unary-node op } x, s') \end{array}}{g \triangleleft \text{UnaryExpr op } xe \rightsquigarrow (g', n)}$$

$$\frac{\text{stamp } g \text{ } n = s}{g \triangleleft \text{LeafExpr } n \text{ } s \rightsquigarrow (g, n)}$$

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

*sq-param0* = *BinaryExpr BinMul*

(*ParameterExpr* 0 (*IntegerStamp* 32 (- 2147483648) 2147483647))

(*ParameterExpr* 0 (*IntegerStamp* 32 (- 2147483648) 2147483647))

**values** {(*n*, *g*) . (*eg2-sq* < *sq-param0* ~> (*g*, *n*))}

**definition** *encodeeval* :: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *ID* ⇒ *Value* ⇒ *bool*

([*·*, *·*, *·*] ⊢ - ↦ - 50)

**where**

*encodeeval g m p n v* = (∃ *e*. (*g* ⊢ *n* ≃ *e*) ∧ ([*m*, *p*] ⊢ *e* ↦ *v*))

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

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

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

**definition** *graph-refinement* :: *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**

*graph-refinement g1 g2* =

(∀ *n* . *n* ∈ *ids g1* → (∀ *e1*. (*g1* ⊢ *n* ≃ *e1*) → (∃ *e2*. (*g2* ⊢ *n* ≃ *e2*) ∧ *e1* ≥ *e2*)))

**lemma** *graph-refinement*:

*graph-refinement g1 g2* ⇒ (∀ *n m p v*. *n* ∈ *ids g1* → ([*g1*, *m*, *p*] ⊢ *n* ↦ *v*) → ([*g2*, *m*, *p*] ⊢ *n* ↦ *v*))

**by** (*meson encodeeval-def graph-refinement-def le-expr-def*)

**definition** *graph-represents-expression* :: *IRGraph* ⇒ *ID* ⇒ *IRExpr* ⇒ *bool*

(- ⊢ - ≤ - 50)

**where**

*graph-represents-expression g n e* = (∀ *m p v*. ([*m*, *p*] ⊢ *e* ↦ *v*) → ([*g*, *m*, *p*] ⊢ *n* ↦ *v*))

**end**

### 3 Data-flow Expression-Tree Theorems

```

theory IRTreeEvalThms
  imports
    TreeToGraph
    HOL-Eisbach.Eisbach
begin

```

#### 3.1 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.

**named-theorems** *rep*

```

lemma rep-constant [rep]:
   $g \vdash n \simeq e \implies$ 
   $\text{kind } g \ n = \text{ConstantNode } c \implies$ 
   $e = \text{ConstantExpr } c$ 
by (induction rule: rep.induct; auto)

```

```

lemma rep-parameter [rep]:
   $g \vdash n \simeq e \implies$ 
   $\text{kind } g \ n = \text{ParameterNode } i \implies$ 
   $(\exists s. e = \text{ParameterExpr } i \ s)$ 
by (induction rule: rep.induct; auto)

```

```

lemma rep-conditional [rep]:
   $g \vdash n \simeq e \implies$ 
   $\text{kind } g \ n = \text{ConditionalNode } c \ t \ f \implies$ 
   $(\exists ce \ te \ fe. e = \text{ConditionalExpr } ce \ te \ fe)$ 
by (induction rule: rep.induct; auto)

```

```

lemma rep-abs [rep]:
   $g \vdash n \simeq e \implies$ 
   $\text{kind } g \ n = \text{AbsNode } x \implies$ 
   $(\exists xe. e = \text{UnaryExpr } \text{UnaryAbs } xe)$ 
by (induction rule: rep.induct; auto)

```

```

lemma rep-not [rep]:
   $g \vdash n \simeq e \implies$ 
   $\text{kind } g \ n = \text{NotNode } x \implies$ 
   $(\exists xe. e = \text{UnaryExpr } \text{UnaryNot } xe)$ 
by (induction rule: rep.induct; auto)

```

```

lemma rep-negate [rep]:
   $g \vdash n \simeq e \implies$ 
   $\text{kind } g \ n = \text{NegateNode } x \implies$ 

```



( $\exists xe. e = \text{UnaryExpr UnaryNeg } xe$ )  
**by** (induction rule: rep.induct; auto)

**lemma** rep-logicnegation [rep]:  
 $g \vdash n \simeq e \implies$   
 $\text{kind } g \ n = \text{LogicNegationNode } x \implies$   
( $\exists xe. e = \text{UnaryExpr UnaryLogicNegation } xe$ )  
**by** (induction rule: rep.induct; auto)

**lemma** rep-add [rep]:  
 $g \vdash n \simeq e \implies$   
 $\text{kind } g \ n = \text{AddNode } x \ y \implies$   
( $\exists xe \ ye. e = \text{BinaryExpr BinAdd } xe \ ye$ )  
**by** (induction rule: rep.induct; auto)

**lemma** rep-sub [rep]:  
 $g \vdash n \simeq e \implies$   
 $\text{kind } g \ n = \text{SubNode } x \ y \implies$   
( $\exists xe \ ye. e = \text{BinaryExpr BinSub } xe \ ye$ )  
**by** (induction rule: rep.induct; auto)

**lemma** rep-mul [rep]:  
 $g \vdash n \simeq e \implies$   
 $\text{kind } g \ n = \text{MulNode } x \ y \implies$   
( $\exists xe \ ye. e = \text{BinaryExpr BinMul } xe \ ye$ )  
**by** (induction rule: rep.induct; auto)

**lemma** rep-and [rep]:  
 $g \vdash n \simeq e \implies$   
 $\text{kind } g \ n = \text{AndNode } x \ y \implies$   
( $\exists xe \ ye. e = \text{BinaryExpr BinAnd } xe \ ye$ )  
**by** (induction rule: rep.induct; auto)

**lemma** rep-or [rep]:  
 $g \vdash n \simeq e \implies$   
 $\text{kind } g \ n = \text{OrNode } x \ y \implies$   
( $\exists xe \ ye. e = \text{BinaryExpr BinOr } xe \ ye$ )  
**by** (induction rule: rep.induct; auto)

**lemma** rep-xor [rep]:  
 $g \vdash n \simeq e \implies$   
 $\text{kind } g \ n = \text{XorNode } x \ y \implies$   
( $\exists xe \ ye. e = \text{BinaryExpr BinXor } xe \ ye$ )  
**by** (induction rule: rep.induct; auto)

**lemma** rep-integer-below [rep]:  
 $g \vdash n \simeq e \implies$   
 $\text{kind } g \ n = \text{IntegerBelowNode } x \ y \implies$   
( $\exists xe \ ye. e = \text{BinaryExpr BinIntegerBelow } xe \ ye$ )

**by** (*induction rule: rep.induct; auto*)

**lemma** *rep-integer-equals* [*rep*]:  
 $g \vdash n \simeq e \implies$   
 $\text{kind } g \ n = \text{IntegerEqualsNode } x \ y \implies$   
 $(\exists xe \ ye. \ e = \text{BinaryExpr } \text{BinIntegerEquals } xe \ ye)$   
**by** (*induction rule: rep.induct; auto*)

**lemma** *rep-integer-less-than* [*rep*]:  
 $g \vdash n \simeq e \implies$   
 $\text{kind } g \ n = \text{IntegerLessThanNode } x \ y \implies$   
 $(\exists xe \ ye. \ e = \text{BinaryExpr } \text{BinIntegerLessThan } xe \ ye)$   
**by** (*induction rule: rep.induct; auto*)

**lemma** *rep-narrow* [*rep*]:  
 $g \vdash n \simeq e \implies$   
 $\text{kind } g \ n = \text{NarrowNode } ib \ rb \ x \implies$   
 $(\exists x. \ e = \text{UnaryExpr } (\text{UnaryNarrow } ib \ rb) \ x)$   
**by** (*induction rule: rep.induct; auto*)

**lemma** *rep-sign-extend* [*rep*]:  
 $g \vdash n \simeq e \implies$   
 $\text{kind } g \ n = \text{SignExtendNode } ib \ rb \ x \implies$   
 $(\exists x. \ e = \text{UnaryExpr } (\text{UnarySignExtend } ib \ rb) \ x)$   
**by** (*induction rule: rep.induct; auto*)

**lemma** *rep-zero-extend* [*rep*]:  
 $g \vdash n \simeq e \implies$   
 $\text{kind } g \ n = \text{ZeroExtendNode } ib \ rb \ x \implies$   
 $(\exists x. \ e = \text{UnaryExpr } (\text{UnaryZeroExtend } ib \ rb) \ x)$   
**by** (*induction rule: rep.induct; auto*)

**lemma** *rep-load-field* [*rep*]:  
 $g \vdash n \simeq e \implies$   
 $\text{is-preevaluated } (\text{kind } g \ n) \implies$   
 $(\exists s. \ e = \text{LeafExpr } n \ s)$   
**by** (*induction rule: rep.induct; auto*)

**method** *solve-det* **uses** *node =*  
 $(\text{match } node \text{ in } kind \ - \ = \ node \text{ - for } node \Rightarrow$   
 $\langle \text{match } rep \text{ in } r: \ - \implies \ - = node \ - \implies \ - \Rightarrow$   
 $\langle \text{match } IRNode.inject \text{ in } i: (node \ - = node \ -) = \ - \Rightarrow$   
 $\langle \text{match } RepE \text{ in } e: \ - \implies (\bigwedge x. \ - = node \ x \implies \ -) \implies \ - \Rightarrow$   
 $\langle \text{metis } i \ e \ r \rangle \rangle \mid$   
 $\text{match } node \text{ in } kind \ - \ = \ node \text{ - for } node \Rightarrow$   
 $\langle \text{match } rep \text{ in } r: \ - \implies \ - = node \ - \implies \ - \Rightarrow$   
 $\langle \text{match } IRNode.inject \text{ in } i: (node \ - \ = \ node \ -) = \ - \Rightarrow$   
 $\langle \text{match } RepE \text{ in } e: \ - \implies (\bigwedge x \ y. \ - = node \ x \ y \implies \ -) \implies \ - \Rightarrow$

```

      ⟨metis i e r⟩⟩⟩ |
match node in kind - - = node - - - for node ⇒
  ⟨match rep in r: - ⇒ - = node - - - ⇒ - ⇒
    ⟨match IRNode.inject in i: (node - - - = node - - -) = - ⇒
      ⟨match RepE in e: - ⇒ (∧ x y z. - = node x y z ⇒ -) ⇒ - ⇒
        ⟨metis i e r⟩⟩⟩ |
match node in kind - - = node - - - for node ⇒
  ⟨match rep in r: - ⇒ - = node - - - ⇒ - ⇒
    ⟨match IRNode.inject in i: (node - - - = node - - -) = - ⇒
      ⟨match RepE in e: - ⇒ (∧ x. - = node - - x ⇒ -) ⇒ - ⇒
        ⟨metis i e r⟩⟩⟩)

```

Now we can prove that 'rep' and 'eval', and their list versions, are deterministic.

```

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 (solve-det node: ConditionalNode)
next
  case (AbsNode n x xe)
  then show ?case
    by (solve-det node: AbsNode)
next
  case (NotNode n x xe)
  then show ?case
    by (solve-det node: NotNode)
next
  case (NegateNode n x xe)
  then show ?case
    by (solve-det node: NegateNode)
next
  case (LogicNegationNode n x xe)
  then show ?case
    by (solve-det node: LogicNegationNode)
next
  case (AddNode n x y xe ye)
  then show ?case
    by (solve-det node: AddNode)
next
  case (MulNode n x y xe ye)
  then show ?case

```

```

      by (solve-det node: MulNode)
next
  case (SubNode n x y xe ye)
  then show ?case
      by (solve-det node: SubNode)
next
  case (AndNode n x y xe ye)
  then show ?case
      by (solve-det node: AndNode)
next
  case (OrNode n x y xe ye)
  then show ?case
      by (solve-det node: OrNode)
next
  case (XorNode n x y xe ye)
  then show ?case
      by (solve-det node: XorNode)
next
  case (IntegerBelowNode n x y xe ye)
  then show ?case
      by (solve-det node: IntegerBelowNode)
next
  case (IntegerEqualsNode n x y xe ye)
  then show ?case
      by (solve-det node: IntegerEqualsNode)
next
  case (IntegerLessThanNode n x y xe ye)
  then show ?case
      by (solve-det node: IntegerLessThanNode)
next
  case (NarrowNode n x xe)
  then show ?case
      by (metis IRNode.inject(28) NarrowNodeE rep-narrow)
next
  case (SignExtendNode n x xe)
  then show ?case
      using SignExtendNodeE rep-sign-extend IRNode.inject(39)
      by (metis IRNode.inject(39) SignExtendNodeE rep-sign-extend)
next
  case (ZeroExtendNode n x xe)
  then show ?case
      by (metis IRNode.inject(50) ZeroExtendNodeE rep-zero-extend)
next
  case (LeafNode n s)
  then show ?case using rep-load-field LeafNodeE by blast
qed

```

lemma repAllDet:  
 $g \vdash xs \simeq_L e1 \implies$

```

  g ⊢ xs ≃L e2 ⇒
  e1 = e2
proof (induction arbitrary: e2 rule: replist.induct)
  case RepNil
  then show ?case
    using replist.cases by auto
next
  case (RepCons x xe xs xse)
  then show ?case
    by (metis list.distinct(1) list.sel(1) list.sel(3) repDet replist.cases)
qed

```

```

lemma evalDet:
  [m,p] ⊢ e ↦ v1 ⇒
  [m,p] ⊢ e ↦ v2 ⇒
  v1 = v2
apply (induction arbitrary: v2 rule: evaltree.induct)
by (elim EvalTreeE; auto)+

```

```

lemma evalAllDet:
  [m,p] ⊢ e ↦L v1 ⇒
  [m,p] ⊢ e ↦L v2 ⇒
  v1 = v2
apply (induction arbitrary: v2 rule: evaltrees.induct)
apply (elim EvalTreeE; auto)
using evalDet by force

```

```

lemma encodeEvalDet:
  [g,m,p] ⊢ e ↦ v1 ⇒
  [g,m,p] ⊢ e ↦ v2 ⇒
  v1 = v2
by (metis encodeeval-def evalDet repDet)

```

```

lemma graphDet: ([g,m,p] ⊢ nid ↦ v1) ∧ ([g,m,p] ⊢ nid ↦ v2) ⇒ v1 = v2
using encodeEvalDet by blast

```

A valid value cannot be *UndefVal*.

```

lemma valid-not-undef:
  assumes a1: valid-value s val
  assumes a2: s ≠ VoidStamp
  shows val ≠ 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 ⇒
  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  $\implies$ 
    ( $\exists v. \text{val} = \text{ObjRef } v$ )
  using valid-value.simps by (metis IRTreeEval.val-to-bool.cases)

lemma valid-int32[elim]:
  shows valid-value (IntegerStamp 32 l h) val  $\implies$ 
    ( $\exists v. \text{val} = \text{IntVal32 } v$ )
  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  $\implies$ 
    ( $\exists v. \text{val} = \text{IntVal64 } v$ )
  apply (rule IRTreeEval.val-to-bool.cases[of val])
  using Value.distinct by simp+

TODO: could we prove that expression evaluation never returns UndefinedVal?
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  $\exists v. \text{val} = (\text{IntVal32 } v)$ 

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]  $\vdash$  LeafExpr i (IntegerStamp 64 lo hi)  $\mapsto$  val
  shows  $\exists v. \text{val} = (\text{IntVal64 } v)$ 

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  $\exists v. (\text{val} = (\text{IntVal32 } v) \wedge \text{lo} \leq \text{sint } v \wedge \text{sint } v \leq \text{hi})$ 

```

```

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

experiment begin
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
end

lemma valid32or64:
  assumes valid-value (IntegerStamp b lo hi) x
  shows  $(\exists v1. (x = IntVal32\ v1)) \vee (\exists v2. (x = IntVal64\ v2))$ 
  using valid32 valid64 assms valid-value.elims(2) by blast

lemma valid32or64-both:
  assumes valid-value (IntegerStamp b lox hix) x
  and valid-value (IntegerStamp b loy hiy) y
  shows  $(\exists v1\ v2. x = IntVal32\ v1 \wedge y = IntVal32\ v2) \vee (\exists v3\ v4. x = IntVal64\ v3 \wedge y = IntVal64\ v4)$ 
  using assms valid32or64 valid32 valid-value.elims(2) valid-value.simps(1) by
metis

```

### 3.2 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)
by (auto simp add: evaltree.LeanExpr)

```

```

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)

```

### 3.3 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  $y \geq y'$ 
shows  $(BinaryExpr\ op\ x\ y) \geq (BinaryExpr\ op\ x'\ y')$ 
using BinaryExpr assms by auto

lemma mono-conditional:
  assumes  $ce \geq ce'$ 
  assumes  $te \geq te'$ 
  assumes  $fe \geq fe'$ 
  shows  $(ConditionalExpr\ ce\ te\ fe) \geq (ConditionalExpr\ ce'\ te'\ fe')$ 
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  $ce'\ b'$ 
    using  $a$  by blast
qed

```

```

end
theory TreeToGraphThms
imports
  TreeToGraph
  IRTreeEvalThms
  HOL-Eisbach.Eisbach
begin

```

Lift refinement monotonicity to graph level. Hopefully these shouldn't really be required.

```

lemma mono-abs:
  assumes  $kind\ g1\ n = AbsNode\ x \wedge kind\ g2\ n = AbsNode\ x$ 
  assumes  $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$ 
  assumes  $xe1 \geq xe2$ 
  assumes  $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$ 
  shows  $e1 \geq e2$ 
by (metis AbsNode assms(1) assms(2) assms(3) assms(4) mono-unary repDet)

```

```

lemma mono-not:
  assumes  $kind\ g1\ n = NotNode\ x \wedge kind\ g2\ n = NotNode\ x$ 
  assumes  $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$ 
  assumes  $xe1 \geq xe2$ 
  assumes  $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$ 
  shows  $e1 \geq e2$ 

```

by (metis NotNode assms(1) assms(2) assms(3) assms(4) mono-unary repDet)

**lemma** *mono-negate*:

assumes  $\text{kind } g1 \ n = \text{NegateNode } x \wedge \text{kind } g2 \ n = \text{NegateNode } x$   
 assumes  $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$   
 assumes  $xe1 \geq xe2$   
 assumes  $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$   
 shows  $e1 \geq e2$   
 by (metis NegateNode assms(1) assms(2) assms(3) assms(4) mono-unary repDet)

**lemma** *mono-logic-negation*:

assumes  $\text{kind } g1 \ n = \text{LogicNegationNode } x \wedge \text{kind } g2 \ n = \text{LogicNegationNode } x$   
 assumes  $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$   
 assumes  $xe1 \geq xe2$   
 assumes  $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$   
 shows  $e1 \geq e2$   
 by (metis LogicNegationNode assms(1) assms(2) assms(3) assms(4) mono-unary repDet)

**lemma** *mono-narrow*:

assumes  $\text{kind } g1 \ n = \text{NarrowNode } ib \ rb \ x \wedge \text{kind } g2 \ n = \text{NarrowNode } ib \ rb \ x$   
 assumes  $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$   
 assumes  $xe1 \geq xe2$   
 assumes  $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$   
 shows  $e1 \geq e2$   
 using assms mono-unary repDet NarrowNode  
 by metis

**lemma** *mono-sign-extend*:

assumes  $\text{kind } g1 \ n = \text{SignExtendNode } ib \ rb \ x \wedge \text{kind } g2 \ n = \text{SignExtendNode } ib \ rb \ x$   
 assumes  $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$   
 assumes  $xe1 \geq xe2$   
 assumes  $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$   
 shows  $e1 \geq e2$   
 by (metis SignExtendNode assms(1) assms(2) assms(3) assms(4) mono-unary repDet)

**lemma** *mono-zero-extend*:

assumes  $\text{kind } g1 \ n = \text{ZeroExtendNode } ib \ rb \ x \wedge \text{kind } g2 \ n = \text{ZeroExtendNode } ib \ rb \ x$   
 assumes  $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$   
 assumes  $xe1 \geq xe2$   
 assumes  $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$   
 shows  $e1 \geq e2$   
 using assms mono-unary repDet ZeroExtendNode  
 by metis

**lemma** *mono-conditional-graph*:

**assumes**  $\text{kind } g1 \ n = \text{ConditionalNode } c \ t \ f \wedge \text{kind } g2 \ n = \text{ConditionalNode } c \ t \ f$   
**assumes**  $(g1 \vdash c \simeq ce1) \wedge (g2 \vdash c \simeq ce2)$   
**assumes**  $(g1 \vdash t \simeq te1) \wedge (g2 \vdash t \simeq te2)$   
**assumes**  $(g1 \vdash f \simeq fe1) \wedge (g2 \vdash f \simeq fe2)$   
**assumes**  $ce1 \geq ce2 \wedge te1 \geq te2 \wedge fe1 \geq fe2$   
**assumes**  $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$   
**shows**  $e1 \geq e2$   
**by** (*metis ConditionalNodeE IRNode.inject(6) assms(1) assms(2) assms(3) assms(4) assms(5) assms(6) mono-conditional repDet rep-conditional*)

**lemma** *mono-add*:

**assumes**  $\text{kind } g1 \ n = \text{AddNode } x \ y \wedge \text{kind } g2 \ n = \text{AddNode } x \ y$   
**assumes**  $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$   
**assumes**  $(g1 \vdash y \simeq ye1) \wedge (g2 \vdash y \simeq ye2)$   
**assumes**  $xe1 \geq xe2 \wedge ye1 \geq ye2$   
**assumes**  $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$   
**shows**  $e1 \geq e2$   
**using** *mono-binary assms*  
**by** (*metis AddNodeE IRNode.inject(2) repDet rep-add*)

**lemma** *mono-mul*:

**assumes**  $\text{kind } g1 \ n = \text{MulNode } x \ y \wedge \text{kind } g2 \ n = \text{MulNode } x \ y$   
**assumes**  $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$   
**assumes**  $(g1 \vdash y \simeq ye1) \wedge (g2 \vdash y \simeq ye2)$   
**assumes**  $xe1 \geq xe2 \wedge ye1 \geq ye2$   
**assumes**  $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$   
**shows**  $e1 \geq e2$   
**using** *mono-binary assms*  
**by** (*metis IRNode.inject(27) MulNodeE repDet rep-mul*)

**lemma** *encodes-contains*:

$g \vdash n \simeq e \implies$   
 $\text{kind } g \ n \neq \text{NoNode}$   
**apply** (*induction rule: rep.induct*)  
**apply** (*match IRNode.distinct in e: ?n  $\neq$  NoNode  $\implies$*   
 $\langle \text{presburger add: } e \rangle +$   
**by** *fastforce*

**lemma** *no-encoding*:

**assumes**  $n \notin \text{ids } g$   
**shows**  $\neg(g \vdash n \simeq e)$   
**using** *assms* **apply** *simp* **apply** (*rule notI*) **by** (*induction e; simp add: encodes-contains*)

**lemma** *not-excluded-keep-type*:

**assumes**  $n \in \text{ids } g1$   
**assumes**  $n \notin \text{excluded}$   
**assumes**  $(\text{excluded} \leq \text{as-set } g1) \subseteq \text{as-set } g2$

```

shows  $kind\ g1\ n = kind\ g2\ n \wedge stamp\ g1\ n = stamp\ g2\ n$ 
using assms unfolding as-set-def domain-subtraction-def by blast

method metis-node-eq-unary for  $node :: 'a \Rightarrow IRNode =$ 
   $(match\ IRNode.inject\ in\ i: (node\ - = node\ -) = - \Rightarrow$ 
     $\langle metis\ i \rangle)$ 
method metis-node-eq-binary for  $node :: 'a \Rightarrow 'a \Rightarrow IRNode =$ 
   $(match\ IRNode.inject\ in\ i: (node\ -\ - = node\ -\ -) = - \Rightarrow$ 
     $\langle metis\ i \rangle)$ 
method metis-node-eq-ternary for  $node :: 'a \Rightarrow 'a \Rightarrow 'a \Rightarrow IRNode =$ 
   $(match\ IRNode.inject\ in\ i: (node\ -\ -\ - = node\ -\ -\ -) = - \Rightarrow$ 
     $\langle metis\ i \rangle)$ 

lemma graph-semantic-preservation:
  assumes  $a: e1' \geq e2'$ 
  assumes  $b: (\{n'\} \trianglelefteq as-set\ g1) \subseteq as-set\ g2$ 
  assumes  $c: g1 \vdash n' \simeq e1'$ 
  assumes  $d: g2 \vdash n' \simeq e2'$ 
  shows graph-refinement  $g1\ g2$ 
  unfolding graph-refinement-def
  apply (rule allI) apply (rule impI) apply (rule allI) apply (rule impI)
proof -
  fix  $n\ e1$ 
  assume  $e: n \in ids\ g1$ 
  assume  $f: (g1 \vdash n \simeq e1)$ 

  show  $\exists\ e2. (g2 \vdash n \simeq e2) \wedge e1 \geq e2$ 
  proof (cases  $n = n'$ )
    case True
    have  $g: e1 = e1'$  using  $c\ f\ True\ repDet$  by simp
    have  $h: (g2 \vdash n \simeq e2') \wedge e1' \geq e2'$ 
      using  $True\ a\ d$  by blast
    then show ?thesis
      using  $g$  by blast
  next
    case False
    have  $n \notin \{n'\}$ 
    using False by simp
    then have  $i: kind\ g1\ n = kind\ g2\ n \wedge stamp\ g1\ n = stamp\ g2\ n$ 
      using not-excluded-keep-type
      using  $b\ e$  by presburger
    show ?thesis using  $f\ i$ 
    proof (induction  $e1$ )
      case (ConstantNode  $n\ c$ )
      then show ?case
        by (metis eq-refl rep.ConstantNode)
    next
      case (ParameterNode  $n\ i\ s$ )
      then show ?case

```

```

    by (metis eq-refl rep.ParameterNode)
next
case (ConditionalNode n c t f ce1 te1 fe1)
have k: g1 ⊢ n ≃ ConditionalExpr ce1 te1 fe1 using f ConditionalNode
  by (simp add: ConditionalNode.hyps(2) rep.ConditionalNode)
obtain cn tn fn where l: kind g1 n = ConditionalNode cn tn fn
  using ConditionalNode.hyps(1) by blast
then have mc: g1 ⊢ cn ≃ ce1
  using ConditionalNode.hyps(1) ConditionalNode.hyps(2) by fastforce
from l have mt: g1 ⊢ tn ≃ te1
  using ConditionalNode.hyps(1) ConditionalNode.hyps(3) by fastforce
from l have mf: g1 ⊢ fn ≃ fe1
  using ConditionalNode.hyps(1) ConditionalNode.hyps(4) by fastforce
then show ?case
proof -
  have g1 ⊢ cn ≃ ce1 using mc by simp
  have g1 ⊢ tn ≃ te1 using mt by simp
  have g1 ⊢ fn ≃ fe1 using mf by simp
  have cer: ∃ ce2. (g2 ⊢ cn ≃ ce2) ∧ ce1 ≥ ce2
    using ConditionalNode
    using a b c d l no-encoding not-excluded-keep-type repDet singletonD
    by (metis-node-eq-ternary ConditionalNode)
  have ter: ∃ te2. (g2 ⊢ tn ≃ te2) ∧ te1 ≥ te2
    using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
    singletonD
    by (metis-node-eq-ternary ConditionalNode)
  have ∃ fe2. (g2 ⊢ fn ≃ fe2) ∧ fe1 ≥ fe2
    using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
    singletonD
    by (metis-node-eq-ternary ConditionalNode)
  then have ∃ ce2 te2 fe2. (g2 ⊢ n ≃ ConditionalExpr ce2 te2 fe2) ∧
    ConditionalExpr ce1 te1 fe1 ≥ ConditionalExpr ce2 te2 fe2
    using ConditionalNode.prem1 l mono-conditional rep.ConditionalNode cer
    ter
  by (smt (verit) IRTreeEvalThms.mono-conditional)
then show ?thesis
  by meson
qed
next
case (AbsNode n x xe1)
have k: g1 ⊢ n ≃ UnaryExpr UnaryAbs xe1 using f AbsNode
  by (simp add: AbsNode.hyps(2) rep.AbsNode)
obtain xn where l: kind g1 n = AbsNode xn
  using AbsNode.hyps(1) by blast
then have m: g1 ⊢ xn ≃ xe1
  using AbsNode.hyps(1) AbsNode.hyps(2) by fastforce
then show ?case
proof (cases xn = n')
case True

```

```

    then have n:  $xe1 = e1'$  using c m repDet by simp
    then have ev:  $g2 \vdash n \simeq \text{UnaryExpr UnaryAbs } e2'$  using AbsNode.hyps(1)
  l m n
    using AbsNode.premis True d rep.AbsNode by simp
    then have r:  $\text{UnaryExpr UnaryAbs } e1' \geq \text{UnaryExpr UnaryAbs } e2'$ 
      by (meson a mono-unary)
    then show ?thesis using ev r
      by (metis n)
  next
    case False
    have  $g1 \vdash xn \simeq xe1$  using m by simp
    have  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
      using AbsNode
    using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
      by (metis-node-eq-unary AbsNode)
    then have  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr UnaryAbs } xe2) \wedge \text{UnaryExpr}$ 
       $\text{UnaryAbs } xe1 \geq \text{UnaryExpr UnaryAbs } xe2$ 
      by (metis AbsNode.premis l mono-unary rep.AbsNode)
    then show ?thesis
      by meson
  qed
next
  case (NotNode n x xe1)
  have k:  $g1 \vdash n \simeq \text{UnaryExpr UnaryNot } xe1$  using f NotNode
    by (simp add: NotNode.hyps(2) rep.NotNode)
  obtain xn where l: kind  $g1$  n = NotNode xn
    using NotNode.hyps(1) by blast
  then have m:  $g1 \vdash xn \simeq xe1$ 
    using NotNode.hyps(1) NotNode.hyps(2) by fastforce
  then show ?case
  proof (cases xn = n')
    case True
    then have n:  $xe1 = e1'$  using c m repDet by simp
    then have ev:  $g2 \vdash n \simeq \text{UnaryExpr UnaryNot } e2'$  using NotNode.hyps(1)
  l m n
    using NotNode.premis True d rep.NotNode by simp
    then have r:  $\text{UnaryExpr UnaryNot } e1' \geq \text{UnaryExpr UnaryNot } e2'$ 
      by (meson a mono-unary)
    then show ?thesis using ev r
      by (metis n)
  next
    case False
    have  $g1 \vdash xn \simeq xe1$  using m by simp
    have  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
      using NotNode
    using False i b l not-excluded-keep-type singletonD no-encoding
      by (metis-node-eq-unary NotNode)
    then have  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr UnaryNot } xe2) \wedge \text{UnaryExpr}$ 
       $\text{UnaryNot } xe1 \geq \text{UnaryExpr UnaryNot } xe2$ 

```

```

      by (metis NotNode.premis l mono-unary rep.NotNode)
    then show ?thesis
      by meson
  qed
next
case (NegateNode n x xe1)
have k:  $g1 \vdash n \simeq \text{UnaryExpr UnaryNeg xe1}$  using f NegateNode
  by (simp add: NegateNode.hyps(2) rep.NegateNode)
obtain xn where l: kind  $g1$   $n = \text{NegateNode xn}$ 
  using NegateNode.hyps(1) by blast
then have m:  $g1 \vdash xn \simeq xe1$ 
  using NegateNode.hyps(1) NegateNode.hyps(2) by fastforce
then show ?case
proof (cases  $xn = n'$ )
  case True
  then have n:  $xe1 = e1'$  using c m repDet by simp
  then have ev:  $g2 \vdash n \simeq \text{UnaryExpr UnaryNeg e2'}$  using NegateNode.hyps(1)
l m n
    using NegateNode.premis True d rep.NegateNode by simp
  then have r:  $\text{UnaryExpr UnaryNeg e1'} \geq \text{UnaryExpr UnaryNeg e2'}$ 
    by (meson a mono-unary)
  then show ?thesis using ev r
    by (metis n)
next
case False
have  $g1 \vdash xn \simeq xe1$  using m by simp
have  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
  using NegateNode
  using False i b l not-excluded-keep-type singletonD no-encoding
  by (metis node-eq-unary NegateNode)
then have  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr UnaryNeg xe2}) \wedge \text{UnaryExpr}$ 
UnaryNeg  $xe1 \geq \text{UnaryExpr UnaryNeg xe2}$ 
  by (metis NegateNode.premis l mono-unary rep.NegateNode)
then show ?thesis
  by meson
qed
next
case (LogicNegationNode n x xe1)
have k:  $g1 \vdash n \simeq \text{UnaryExpr UnaryLogicNegation xe1}$  using f LogicNegationNode
  by (simp add: LogicNegationNode.hyps(2) rep.LogicNegationNode)
obtain xn where l: kind  $g1$   $n = \text{LogicNegationNode xn}$ 
  using LogicNegationNode.hyps(1) by blast
then have m:  $g1 \vdash xn \simeq xe1$ 
  using LogicNegationNode.hyps(1) LogicNegationNode.hyps(2) by fastforce
then show ?case
proof (cases  $xn = n'$ )
  case True
  then have n:  $xe1 = e1'$  using c m repDet by simp

```

```

      then have ev:  $g2 \vdash n \simeq \text{UnaryExpr UnaryLogicNegation } e2'$  using
LogicNegationNode.hyps(1) l m n
      using LogicNegationNode.premis True d rep.LogicNegationNode by simp
      then have r:  $\text{UnaryExpr UnaryLogicNegation } e1' \geq \text{UnaryExpr UnaryLog-}$ 
icNegation  $e2'$ 
      by (meson a mono-unary)
      then show ?thesis using ev r
      by (metis n)
next
case False
have  $g1 \vdash xn \simeq xe1$  using m by simp
have  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
using LogicNegationNode
using False i b l not-excluded-keep-type singletonD no-encoding
by (metis-node-eq-unary LogicNegationNode)
then have  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr UnaryLogicNegation } xe2) \wedge$ 
UnaryExpr UnaryLogicNegation  $xe1 \geq \text{UnaryExpr UnaryLogicNegation } xe2$ 
by (metis LogicNegationNode.premis l mono-unary rep.LogicNegationNode)
then show ?thesis
by meson
qed
next
case (AddNode n x y xe1 ye1)
have k:  $g1 \vdash n \simeq \text{BinaryExpr BinAdd } xe1 ye1$  using f AddNode
by (simp add: AddNode.hyps(2) rep.AddNode)
obtain xn yn where l: kind  $g1 n = \text{AddNode } xn yn$ 
using AddNode.hyps(1) by blast
then have mx:  $g1 \vdash xn \simeq xe1$ 
using AddNode.hyps(1) AddNode.hyps(2) by fastforce
from l have my:  $g1 \vdash yn \simeq ye1$ 
using AddNode.hyps(1) AddNode.hyps(3) by fastforce
then show ?case
proof -
have  $g1 \vdash xn \simeq xe1$  using mx by simp
have  $g1 \vdash yn \simeq ye1$  using my by simp
have xer:  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
using AddNode
using a b c d l no-encoding not-excluded-keep-type repDet singletonD
by (metis-node-eq-binary AddNode)
have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
using AddNode
using a b c d l no-encoding not-excluded-keep-type repDet singletonD
by (metis-node-eq-binary AddNode)
then have  $\exists xe2 ye2. (g2 \vdash n \simeq \text{BinaryExpr BinAdd } xe2 ye2) \wedge \text{BinaryExpr}$ 
BinAdd  $xe1 ye1 \geq \text{BinaryExpr BinAdd } xe2 ye2$ 
by (metis AddNode.premis l mono-binary rep.AddNode xer)
then show ?thesis
by meson
qed

```



```

next
  case (MulNode n x y xe1 ye1)
  have k:  $g1 \vdash n \simeq \text{BinaryExpr BinMul } xe1 \text{ ye1}$  using f MulNode
    by (simp add: MulNode.hyps(2) rep.MulNode)
  obtain xn yn where l: kind g1 n = MulNode xn yn
    using MulNode.hyps(1) by blast
  then have mx:  $g1 \vdash xn \simeq xe1$ 
    using MulNode.hyps(1) MulNode.hyps(2) by fastforce
  from l have my:  $g1 \vdash yn \simeq ye1$ 
    using MulNode.hyps(1) MulNode.hyps(3) by fastforce
  then show ?case
  proof -
    have  $g1 \vdash xn \simeq xe1$  using mx by simp
    have  $g1 \vdash yn \simeq ye1$  using my by simp
    have xer:  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
      using MulNode
      using a b c d l no-encoding not-excluded-keep-type repDet singletonD
      by (metis-node-eq-binary MulNode)
    have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
      using MulNode
      using a b c d l no-encoding not-excluded-keep-type repDet singletonD
      by (metis-node-eq-binary MulNode)
    then have  $\exists xe2 ye2. (g2 \vdash n \simeq \text{BinaryExpr BinMul } xe2 \text{ ye2}) \wedge \text{BinaryExpr}$ 
      BinMul xe1 ye1  $\geq \text{BinaryExpr BinMul } xe2 \text{ ye2}$ 
      by (metis MulNode.prem1 l mono-binary rep.MulNode xer)
    then show ?thesis
      by meson
  qed
next
  case (SubNode n x y xe1 ye1)
  have k:  $g1 \vdash n \simeq \text{BinaryExpr BinSub } xe1 \text{ ye1}$  using f SubNode
    by (simp add: SubNode.hyps(2) rep.SubNode)
  obtain xn yn where l: kind g1 n = SubNode xn yn
    using SubNode.hyps(1) by blast
  then have mx:  $g1 \vdash xn \simeq xe1$ 
    using SubNode.hyps(1) SubNode.hyps(2) by fastforce
  from l have my:  $g1 \vdash yn \simeq ye1$ 
    using SubNode.hyps(1) SubNode.hyps(3) by fastforce
  then show ?case
  proof -
    have  $g1 \vdash xn \simeq xe1$  using mx by simp
    have  $g1 \vdash yn \simeq ye1$  using my by simp
    have xer:  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
      using SubNode
      using a b c d l no-encoding not-excluded-keep-type repDet singletonD
      by (metis-node-eq-binary SubNode)
    have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
      using SubNode a b c d l no-encoding not-excluded-keep-type repDet singletonD
      by (metis-node-eq-binary SubNode)
  
```

```

    then have  $\exists xe2 ye2. (g2 \vdash n \simeq BinaryExpr\ BinSub\ xe2\ ye2) \wedge BinaryExpr$ 
      BinSub xe1 ye1  $\geq BinaryExpr\ BinSub\ xe2\ ye2$ 
    by (metis SubNode.premis l mono-binary rep.SubNode xer)
    then show ?thesis
    by meson
  qed
next
case (AndNode n x y xe1 ye1)
have k:  $g1 \vdash n \simeq BinaryExpr\ BinAnd\ xe1\ ye1$  using f AndNode
by (simp add: AndNode.hyps(2) rep.AndNode)
obtain xn yn where l: kind g1 n = AndNode xn yn
using AndNode.hyps(1) by blast
then have mx:  $g1 \vdash xn \simeq xe1$ 
using AndNode.hyps(1) AndNode.hyps(2) by fastforce
from l have my:  $g1 \vdash yn \simeq ye1$ 
using AndNode.hyps(1) AndNode.hyps(3) by fastforce
then show ?case
proof -
  have  $g1 \vdash xn \simeq xe1$  using mx by simp
  have  $g1 \vdash yn \simeq ye1$  using my by simp
  have xer:  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
  using AndNode
  using a b c d l no-encoding not-excluded-keep-type repDet singletonD
  by (metis-node-eq-binary AndNode)
  have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
  using AndNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
  by (metis-node-eq-binary AndNode)
  then have  $\exists xe2 ye2. (g2 \vdash n \simeq BinaryExpr\ BinAnd\ xe2\ ye2) \wedge BinaryExpr$ 
    BinAnd xe1 ye1  $\geq BinaryExpr\ BinAnd\ xe2\ ye2$ 
  by (metis AndNode.premis l mono-binary rep.AndNode xer)
  then show ?thesis
  by meson
qed
next
case (OrNode n x y xe1 ye1)
have k:  $g1 \vdash n \simeq BinaryExpr\ BinOr\ xe1\ ye1$  using f OrNode
by (simp add: OrNode.hyps(2) rep.OrNode)
obtain xn yn where l: kind g1 n = OrNode xn yn
using OrNode.hyps(1) by blast
then have mx:  $g1 \vdash xn \simeq xe1$ 
using OrNode.hyps(1) OrNode.hyps(2) by fastforce
from l have my:  $g1 \vdash yn \simeq ye1$ 
using OrNode.hyps(1) OrNode.hyps(3) by fastforce
then show ?case
proof -
  have  $g1 \vdash xn \simeq xe1$  using mx by simp
  have  $g1 \vdash yn \simeq ye1$  using my by simp
  have xer:  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 

```

```

    using OrNode
    using a b c d l no-encoding not-excluded-keep-type repDet singletonD
    by (metis-node-eq-binary OrNode)
    have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
    using OrNode a b c d l no-encoding not-excluded-keep-type repDet singletonD
    by (metis-node-eq-binary OrNode)
    then have  $\exists xe2 ye2. (g2 \vdash n \simeq BinaryExpr BinOr xe2 ye2) \wedge BinaryExpr$ 
    BinOr xe1 ye1  $\geq BinaryExpr BinOr xe2 ye2$ 
    by (metis OrNode.premis l mono-binary rep.OrNode xer)
    then show ?thesis
    by meson
  qed
next
case (XorNode n x y xe1 ye1)
have k:  $g1 \vdash n \simeq BinaryExpr BinXor xe1 ye1$  using f XorNode
by (simp add: XorNode.hyps(2) rep.XorNode)
obtain xn yn where l: kind g1 n = XorNode xn yn
using XorNode.hyps(1) by blast
then have mx:  $g1 \vdash xn \simeq xe1$ 
using XorNode.hyps(1) XorNode.hyps(2) by fastforce
from l have my:  $g1 \vdash yn \simeq ye1$ 
using XorNode.hyps(1) XorNode.hyps(3) by fastforce
then show ?case
proof -
  have  $g1 \vdash xn \simeq xe1$  using mx by simp
  have  $g1 \vdash yn \simeq ye1$  using my by simp
  have xer:  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
  using XorNode
  using a b c d l no-encoding not-excluded-keep-type repDet singletonD
  by (metis-node-eq-binary XorNode)
  have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
  using XorNode a b c d l no-encoding not-excluded-keep-type repDet
  singletonD
  by (metis-node-eq-binary XorNode)
  then have  $\exists xe2 ye2. (g2 \vdash n \simeq BinaryExpr BinXor xe2 ye2) \wedge BinaryExpr$ 
  BinXor xe1 ye1  $\geq BinaryExpr BinXor xe2 ye2$ 
  by (metis XorNode.premis l mono-binary rep.XorNode xer)
  then show ?thesis
  by meson
qed
next
case (IntegerBelowNode n x y xe1 ye1)
have k:  $g1 \vdash n \simeq BinaryExpr BinIntegerBelow xe1 ye1$  using f IntegerBe-
lowNode
by (simp add: IntegerBelowNode.hyps(2) rep.IntegerBelowNode)
obtain xn yn where l: kind g1 n = IntegerBelowNode xn yn
using IntegerBelowNode.hyps(1) by blast
then have mx:  $g1 \vdash xn \simeq xe1$ 
using IntegerBelowNode.hyps(1) IntegerBelowNode.hyps(2) by fastforce

```

```

from l have my:  $g1 \vdash yn \simeq ye1$ 
  using IntegerBelowNode.hyps(1) IntegerBelowNode.hyps(3) by fastforce
then show ?case
proof -
  have  $g1 \vdash xn \simeq xe1$  using mx by simp
  have  $g1 \vdash yn \simeq ye1$  using my by simp
  have xer:  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
    using IntegerBelowNode
    using a b c d l no-encoding not-excluded-keep-type repDet singletonD
    by (metis-node-eq-binary IntegerBelowNode)
  have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
  using IntegerBelowNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
    by (metis-node-eq-binary IntegerBelowNode)
  then have  $\exists xe2 ye2. (g2 \vdash n \simeq BinaryExpr BinIntegerBelow xe2 ye2) \wedge$ 
BinaryExpr BinIntegerBelow  $xe1 ye1 \geq BinaryExpr BinIntegerBelow xe2 ye2$ 
    by (metis IntegerBelowNode.premis l mono-binary rep.IntegerBelowNode
xer)
  then show ?thesis
    by meson
qed
next
case (IntegerEqualsNode n x y xe1 ye1)
have k:  $g1 \vdash n \simeq BinaryExpr BinIntegerEquals xe1 ye1$  using f IntegerEqual-
sNode
  by (simp add: IntegerEqualsNode.hyps(2) rep.IntegerEqualsNode)
obtain xn yn where l: kind  $g1 n = IntegerEqualsNode xn yn$ 
  using IntegerEqualsNode.hyps(1) by blast
then have mx:  $g1 \vdash xn \simeq xe1$ 
  using IntegerEqualsNode.hyps(1) IntegerEqualsNode.hyps(2) by fastforce
from l have my:  $g1 \vdash yn \simeq ye1$ 
  using IntegerEqualsNode.hyps(1) IntegerEqualsNode.hyps(3) by fastforce
then show ?case
proof -
  have  $g1 \vdash xn \simeq xe1$  using mx by simp
  have  $g1 \vdash yn \simeq ye1$  using my by simp
  have xer:  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
    using IntegerEqualsNode
    using a b c d l no-encoding not-excluded-keep-type repDet singletonD
    by (metis-node-eq-binary IntegerEqualsNode)
  have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
    using IntegerEqualsNode a b c d l no-encoding not-excluded-keep-type
repDet singletonD
    by (metis-node-eq-binary IntegerEqualsNode)
  then have  $\exists xe2 ye2. (g2 \vdash n \simeq BinaryExpr BinIntegerEquals xe2 ye2) \wedge$ 
BinaryExpr BinIntegerEquals  $xe1 ye1 \geq BinaryExpr BinIntegerEquals xe2 ye2$ 
    by (metis IntegerEqualsNode.premis l mono-binary rep.IntegerEqualsNode
xer)
  then show ?thesis

```

```

      by meson
    qed
  next
    case (IntegerLessThanNode n x y xe1 ye1)
      have k:  $g1 \vdash n \simeq \text{BinaryExpr BinIntegerLessThan } xe1 \text{ } ye1$  using f IntegerLessThanNode
      by (simp add: IntegerLessThanNode.hyps(2) rep.IntegerLessThanNode)
      obtain xn yn where l: kind g1 n = IntegerLessThanNode xn yn
      using IntegerLessThanNode.hyps(1) by blast
      then have mx:  $g1 \vdash xn \simeq xe1$ 
      using IntegerLessThanNode.hyps(1) IntegerLessThanNode.hyps(2) by fastforce
      from l have my:  $g1 \vdash yn \simeq ye1$ 
      using IntegerLessThanNode.hyps(1) IntegerLessThanNode.hyps(3) by fastforce
      then show ?case
      proof -
        have  $g1 \vdash xn \simeq xe1$  using mx by simp
        have  $g1 \vdash yn \simeq ye1$  using my by simp
        have xer:  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
        using IntegerLessThanNode
        using a b c d l no-encoding not-excluded-keep-type repDet singletonD
        by (metis-node-eq-binary IntegerLessThanNode)
        have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
        using IntegerLessThanNode a b c d l no-encoding not-excluded-keep-type
        repDet singletonD
        by (metis-node-eq-binary IntegerLessThanNode)
        then have  $\exists xe2 ye2. (g2 \vdash n \simeq \text{BinaryExpr BinIntegerLessThan } xe2 \text{ } ye2) \wedge \text{BinaryExpr BinIntegerLessThan } xe1 \text{ } ye1 \geq \text{BinaryExpr BinIntegerLessThan } xe2 \text{ } ye2$ 
        by (metis IntegerLessThanNode.premis l mono-binary rep.IntegerLessThanNode xer)
        then show ?thesis
        by meson
      qed
    next
      case (NarrowNode n inputBits resultBits x xe1)
      have k:  $g1 \vdash n \simeq \text{UnaryExpr (UnaryNarrow inputBits resultBits) } xe1$  using f NarrowNode
      by (simp add: NarrowNode.hyps(2) rep.NarrowNode)
      obtain xn where l: kind g1 n = NarrowNode inputBits resultBits xn
      using NarrowNode.hyps(1) by blast
      then have m:  $g1 \vdash xn \simeq xe1$ 
      using NarrowNode.hyps(1) NarrowNode.hyps(2)
      by auto
      then show ?case
      proof (cases xn = n')
        case True
        then have n:  $xe1 = e1'$  using c m repDet by simp

```

```

    then have ev:  $g2 \vdash n \simeq \text{UnaryExpr } (\text{UnaryNarrow inputBits resultBits}) e2'$ 
using NarrowNode.hyps(1) l m n
    using NarrowNode.premis True d rep.NarrowNode by simp
    then have r:  $\text{UnaryExpr } (\text{UnaryNarrow inputBits resultBits}) e1' \geq \text{UnaryExpr } (\text{UnaryNarrow inputBits resultBits}) e2'$ 
    by (meson a mono-unary)
    then show ?thesis using ev r
    by (metis n)
next
case False
have  $g1 \vdash xn \simeq xe1$  using m by simp
have  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
using NarrowNode
using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
by (metis-node-eq-ternary NarrowNode)
    then have  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr } (\text{UnaryNarrow inputBits resultBits}) xe2) \wedge \text{UnaryExpr } (\text{UnaryNarrow inputBits resultBits}) xe1 \geq \text{UnaryExpr } (\text{UnaryNarrow inputBits resultBits}) xe2$ 
    by (metis NarrowNode.premis l mono-unary rep.NarrowNode)
    then show ?thesis
    by meson
qed
next
case (SignExtendNode n inputBits resultBits x xe1)
    have k:  $g1 \vdash n \simeq \text{UnaryExpr } (\text{UnarySignExtend inputBits resultBits}) xe1$ 
using f SignExtendNode
    by (simp add: SignExtendNode.hyps(2) rep.SignExtendNode)
    obtain xn where l: kind  $g1$   $n = \text{SignExtendNode inputBits resultBits } xn$ 
    using SignExtendNode.hyps(1) by blast
    then have m:  $g1 \vdash xn \simeq xe1$ 
    using SignExtendNode.hyps(1) SignExtendNode.hyps(2)
    by auto
    then show ?case
    proof (cases  $xn = n'$ )
    case True
    then have n:  $xe1 = e1'$  using c m repDet by simp
    then have ev:  $g2 \vdash n \simeq \text{UnaryExpr } (\text{UnarySignExtend inputBits resultBits}) e2'$ 
using SignExtendNode.hyps(1) l m n
    using SignExtendNode.premis True d rep.SignExtendNode by simp
    then have r:  $\text{UnaryExpr } (\text{UnarySignExtend inputBits resultBits}) e1' \geq \text{UnaryExpr } (\text{UnarySignExtend inputBits resultBits}) e2'$ 
    by (meson a mono-unary)
    then show ?thesis using ev r
    by (metis n)
next
case False
have  $g1 \vdash xn \simeq xe1$  using m by simp
have  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
using SignExtendNode

```

```

    using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
    by (metis-node-eq-ternary SignExtendNode)
  then have  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr } (\text{UnarySignExtend inputBits resultBits}) xe2) \wedge \text{UnaryExpr } (\text{UnarySignExtend inputBits resultBits}) xe1 \geq \text{UnaryExpr } (\text{UnarySignExtend inputBits resultBits}) xe2$ 
    by (metis SignExtendNode.premis l mono-unary rep.SignExtendNode)
  then show ?thesis
    by meson
qed
next
case (ZeroExtendNode n inputBits resultBits x xe1)
  have k:  $g1 \vdash n \simeq \text{UnaryExpr } (\text{UnaryZeroExtend inputBits resultBits}) xe1$ 
using f ZeroExtendNode
  by (simp add: ZeroExtendNode.hyps(2) rep.ZeroExtendNode)
obtain xn where l: kind g1 n = ZeroExtendNode inputBits resultBits xn
  using ZeroExtendNode.hyps(1) by blast
then have m:  $g1 \vdash xn \simeq xe1$ 
  using ZeroExtendNode.hyps(1) ZeroExtendNode.hyps(2)
  by auto
then show ?case
proof (cases xn = n')
  case True
    then have n:  $xe1 = e1'$  using c m repDet by simp
    then have ev:  $g2 \vdash n \simeq \text{UnaryExpr } (\text{UnaryZeroExtend inputBits resultBits}) e2'$ 
  using ZeroExtendNode.hyps(1) l m n
    using ZeroExtendNode.premis True d rep.ZeroExtendNode by simp
    then have r:  $\text{UnaryExpr } (\text{UnaryZeroExtend inputBits resultBits}) e1' \geq \text{UnaryExpr } (\text{UnaryZeroExtend inputBits resultBits}) e2'$ 
    by (meson a mono-unary)
    then show ?thesis using ev r
    by (metis n)
  next
  case False
    have  $g1 \vdash xn \simeq xe1$  using m by simp
    have  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
    using ZeroExtendNode
    using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
    by (metis-node-eq-ternary ZeroExtendNode)
    then have  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr } (\text{UnaryZeroExtend inputBits resultBits}) xe2) \wedge \text{UnaryExpr } (\text{UnaryZeroExtend inputBits resultBits}) xe1 \geq \text{UnaryExpr } (\text{UnaryZeroExtend inputBits resultBits}) xe2$ 
    by (metis ZeroExtendNode.premis l mono-unary rep.ZeroExtendNode)
    then show ?thesis
    by meson
  qed
next
case (LeafNode n s)
then show ?case
  by (metis eq-refl rep.LeafNode)

```

qed  
 qed  
 qed

**definition** *maximal-sharing*:  
 $maximal-sharing\ g = (\forall\ n1\ n2 . n1 \in ids\ g \wedge n2 \in ids\ g \longrightarrow$   
 $(\forall\ e . (g \vdash n1 \simeq e) \wedge (g \vdash n2 \simeq e) \longrightarrow n1 = n2))$

**lemma** *tree-to-graph-rewriting*:  
 $e1 \geq e2$   
 $\wedge (g1 \vdash n \simeq e1) \wedge maximal-sharing\ g1$   
 $\wedge (\{n\} \trianglelefteq as-set\ g1) \subseteq as-set\ g2$   
 $\wedge (g2 \vdash n \simeq e2) \wedge maximal-sharing\ g2$   
 $\implies graph-refinement\ g1\ g2$   
**using** *graph-semantics-preservation*  
**by** *auto*

**declare**  $[[simp-trace]]$   
**lemma** *equal-refines*:  
**fixes**  $e1\ e2 :: IRExpr$   
**assumes**  $e1 = e2$   
**shows**  $e1 \geq e2$   
**using** *assms*  
**by** *simp*  
**declare**  $[[simp-trace=false]]$

**lemma** *subset-implies-evals*:  
**assumes**  $as-set\ g1 \subseteq as-set\ g2$   
**shows**  $(g1 \vdash n \simeq e) \implies (g2 \vdash n \simeq e)$   
**proof** (*induction e arbitrary: n*)  
**case** (*UnaryExpr op e*)  
**then have**  $n \in ids\ g1$   
**using** *no-encoding* **by** *force*  
**then have**  $kind\ g1\ n = kind\ g2\ n$   
**using** *assms* **unfolding** *as-set-def*  
**by** *blast*  
**then show** *?case* **using** *UnaryExpr UnaryRepE*  
**by** (*smt (verit, ccfv-threshold) AbsNode LogicNegationNode NarrowNode NegateNode NotNode SignExtendNode ZeroExtendNode*)  
**next**  
**case** (*BinaryExpr op e1 e2*)  
**then have**  $n \in ids\ g1$   
**using** *no-encoding* **by** *force*  
**then have**  $kind\ g1\ n = kind\ g2\ n$



```

    using assms unfolding as-set-def
  by blast
then show ?case using BinaryExpr BinaryRepE
  by (smt (verit, ccfv-threshold) AddNode MulNode SubNode AndNode OrNode
XorNode IntegerBelowNode IntegerEqualsNode IntegerLessThanNode)
next
  case (ConditionalExpr e1 e2 e3)
  then have  $n \in \text{ids } g1$ 
    using no-encoding by force
  then have  $\text{kind } g1 \ n = \text{kind } g2 \ n$ 
    using assms unfolding as-set-def
  by blast
  then show ?case using ConditionalExpr ConditionalExprE
    by (smt (verit, best) ConditionalNode ConditionalNodeE)
next
  case (ConstantExpr x)
  then have  $n \in \text{ids } g1$ 
    using no-encoding by force
  then have  $\text{kind } g1 \ n = \text{kind } g2 \ n$ 
    using assms unfolding as-set-def
  by blast
  then show ?case using ConstantExpr ConstantExprE
    by (metis ConstantNode ConstantNodeE)
next
  case (ParameterExpr x1 x2)
  then have  $\text{in-}g1: n \in \text{ids } g1$ 
    using no-encoding by force
  then have  $\text{kinds: kind } g1 \ n = \text{kind } g2 \ n$ 
    using assms unfolding as-set-def
  by blast
  from  $\text{in-}g1$  have  $\text{stamps: stamp } g1 \ n = \text{stamp } g2 \ n$ 
    using assms unfolding as-set-def
  by blast
  from  $\text{kinds stamps}$  show ?case using ParameterExpr ParameterExprE
    by (metis ParameterNode ParameterNodeE)
next
  case (LeafExpr nid s)
  then have  $\text{in-}g1: n \in \text{ids } g1$ 
    using no-encoding by force
  then have  $\text{kinds: kind } g1 \ n = \text{kind } g2 \ n$ 
    using assms unfolding as-set-def
  by blast
  from  $\text{in-}g1$  have  $\text{stamps: stamp } g1 \ n = \text{stamp } g2 \ n$ 
    using assms unfolding as-set-def
  by blast
  from  $\text{kinds stamps}$  show ?case using LeafExpr LeafExprE LeafNode
    by (smt (z3) IRExpr.distinct(29) IRExpr.simps(16) IRExpr.simps(28) rep.simps)
next
  case (ConstantVar x)

```

```

then have in-g1:  $n \in \text{ids } g1$ 
  using no-encoding by force
then have kinds:  $\text{kind } g1 \ n = \text{kind } g2 \ n$ 
  using assms unfolding as-set-def
  by blast
from in-g1 have stamps:  $\text{stamp } g1 \ n = \text{stamp } g2 \ n$ 
  using assms unfolding as-set-def
  by blast
from kinds stamps show ?case using ConstantVar
  using rep.simps by blast
next
case (VariableExpr x s)
then have in-g1:  $n \in \text{ids } g1$ 
  using no-encoding by force
then have kinds:  $\text{kind } g1 \ n = \text{kind } g2 \ n$ 
  using assms unfolding as-set-def
  by blast
from in-g1 have stamps:  $\text{stamp } g1 \ n = \text{stamp } g2 \ n$ 
  using assms unfolding as-set-def
  by blast
from kinds stamps show ?case using VariableExpr
  using rep.simps by blast
qed

```

```

lemma subset-refines:
  assumes  $\text{as-set } g1 \subseteq \text{as-set } g2$ 
  shows graph-refinement  $g1 \ g2$ 
proof -
  have  $\text{ids } g1 \subseteq \text{ids } g2$  using assms unfolding as-set-def
  by blast
show ?thesis unfolding graph-refinement-def
  apply (rule allI) apply (rule impI) apply (rule allI) apply (rule impI)
proof -
  fix  $n \ e1$ 
  assume 1:  $n \in \text{ids } g1$ 
  assume 2:  $g1 \vdash n \simeq e1$ 

  show  $\exists e2. (g2 \vdash n \simeq e2) \wedge e1 \geq e2$ 
  using assms 1 2 using subset-implies-evals
  by (meson equal-refines)
qed
qed

```

```

lemma graph-construction:
   $e1 \geq e2$ 
 $\wedge \text{as-set } g1 \subseteq \text{as-set } g2 \wedge \text{maximal-sharing } g1$ 
 $\wedge (g2 \vdash n \simeq e2) \wedge \text{maximal-sharing } g2$ 
 $\implies (g2 \vdash n \sqsubseteq e1) \wedge \text{graph-refinement } g1 \ g2$ 

```

```

using subset-refines
by (meson encodeeval-def graph-represents-expression-def le-expr-def)

end

```

## 4 Control-flow Semantics

```

theory IRStepObj
  imports
    TreeToGraph
begin

```

### 4.1 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'.

*heapdef*

```

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 = (( $\lambda f. \lambda p. \text{UndefVal}$ ), 0)

```

### 4.2 Intraprocedural Semantics

```

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 n =
  (filter ( $\lambda x. (is-PhiNode\ (kind\ g\ x))$ )
   (sorted-list-of-set (usages g n)))

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 (n # xs) (v # vs) m = (set-phis xs vs (m(n := v))) |
  set-phis [] (v # vs) m = m |
  set-phis (x # xs) [] m = m

```

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  $\Rightarrow$  Params  $\Rightarrow$  (ID  $\times$  MapState  $\times$  FieldRefHeap)  $\Rightarrow$  (ID
 $\times$  MapState  $\times$  FieldRefHeap)  $\Rightarrow$  bool
  ( $\neg, - \vdash - \rightarrow -$  55) for g p where

```

*SequentialNode*:

```

[[is-sequential-node (kind g nid);
  nid' = (successors-of (kind g nid))!0]]
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m, h)$  |

```

*IfNode*:

```

[[kind g nid = (IfNode cond tb fb);
   $g \vdash cond \simeq condE$ ;
  [m, p]  $\vdash condE \mapsto val$ ;
  nid' = (if val-to-bool val then tb else fb)]
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m, h)$  |

```

*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);
   $g \vdash inps \simeq_L inpsE$ ;
  [m, p]  $\vdash inpsE \mapsto_L vs$ ;

  m' = set-phis phis vs m]
 $\implies g, p \vdash (nid, m, h) \rightarrow (merge, m', h)$  |

```

*NewInstanceNode:*

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

*LoadFieldNode:*

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

*SignedDivNode:*

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

*SignedRemNode:*

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

*StaticLoadFieldNode:*

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

*StoreFieldNode:*

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

$$\Longrightarrow g, p \vdash (nid, m, h) \rightarrow (nid', m', h') \mid$$

*StaticStoreFieldNode:*

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

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

### 4.3 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*:

$$\begin{aligned} & \llbracket g, p \vdash (nid, m, h) \rightarrow (nid', m', h') \rrbracket \\ \Longrightarrow & P \vdash ((g, nid, m, p) \# stk, h) \longrightarrow ((g, nid', m', p) \# stk, h') \mid \end{aligned}$$

*InvokeNodeStep*:

$$\llbracket is\text{-}Invoke\ (kind\ g\ nid) ;$$

*callTarget* = *ir-callTarget* (*kind* *g* *nid*);

*kind* *g* *callTarget* = (*MethodCallTargetNode* *targetMethod* *arguments*);

*Some* *targetGraph* = *P* *targetMethod*;

*m'* = *new-map-state*;

*g*  $\vdash$  *arguments*  $\simeq_L$  *argsE*;

$[m, p] \vdash argsE \mapsto_L p \rrbracket$

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

|

*ReturnNode*:

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

*g*  $\vdash$  *expr*  $\simeq$  *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*:

$\llbracket \text{kind } g \text{ nid} = (\text{ReturnNode } \text{None } -);$   
 $\text{cm}' = \text{cm}(\text{cnid} := (\text{ObjRef } (\text{Some } (2048))));$

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

*UnwindNode*:

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

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

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

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

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

## 4.4 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*

$\Rightarrow (\text{IRGraph} \times \text{ID} \times \text{MapState} \times \text{Params}) \text{ list} \times \text{FieldRefHeap}$   
 $\Rightarrow \text{Trace}$   
 $\Rightarrow (\text{IRGraph} \times \text{ID} \times \text{MapState} \times \text{Params}) \text{ list} \times \text{FieldRefHeap}$   
 $\Rightarrow \text{Trace}$   
 $\Rightarrow \text{bool}$

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

**for** *P*

**where**

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

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

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

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

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

$\implies \text{exec } P \ ((g, \text{nid}, m, p) \# xs), h) \ l \ ((g', \text{nid}', m', p') \# ys), h') \ l'$   
**code-pred** (modes:  $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow \text{bool}$  as *Exec*) *exec* .

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

$\Rightarrow (IRGraph \times ID \times MapState \times Params) \text{ list} \times FieldRefHeap$   
 $\Rightarrow \text{nat}$   
 $\Rightarrow (IRGraph \times ID \times MapState \times Params) \text{ list} \times FieldRefHeap$   
 $\Rightarrow \text{bool}$

( $\vdash \rightarrow * - *$  -)

**where**

$\llbracket n > 0; \quad$   
 $p \vdash s \longrightarrow s';$   
 $\text{exec-debug } p \ s' \ (n - 1) \ s'' \rrbracket$   
 $\implies \text{exec-debug } p \ s \ n \ s'' \mid$

$\llbracket n = 0 \rrbracket$

$\implies \text{exec-debug } p \ s \ n \ s$

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

#### 4.4.1 Heap Testing

**definition** *p3* :: *Params* **where**

*p3* = [*IntVal32 3*]

**values**  $\{(prod.fst(prod.snd (prod.snd (hd (prod.fst res)))) \ 0$   
 $\mid res. (\lambda x. \text{Some } eg2\text{-sq}) \vdash ([ (eg2\text{-sq}, 0, \text{new-map-state}, p3), (eg2\text{-sq}, 0, \text{new-map-state}, p3)],$   
 $\text{new-heap}) \rightarrow * 2 * res\}$

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

*field-sq* = "sq"

**definition** *eg3-sq* :: *IRGraph* **where**

*eg3-sq* = *irgraph* [  
 $(0, \text{StartNode } \text{None } 4, \text{VoidStamp}),$   
 $(1, \text{ParameterNode } 0, \text{default-stamp}),$   
 $(3, \text{MulNode } 1 \ 1, \text{default-stamp}),$   
 $(4, \text{StoreFieldNode } 4 \ \text{field-sq } 3 \ \text{None } \text{None } 5, \text{VoidStamp}),$   
 $(5, \text{ReturnNode } (\text{Some } 3) \ \text{None}, \text{default-stamp})$   
 $]$

**values**  $\{h\text{-load-field } \text{field-sq } \text{None} \ (prod.snd \ res)$

$\mid res. (\lambda x. \text{Some } eg3\text{-sq}) \vdash ([ (eg3\text{-sq}, 0, \text{new-map-state}, p3), (eg3\text{-sq}, 0,$   
 $\text{new-map-state}, p3)], \text{new-heap}) \rightarrow * 3 * res\}$

**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.
      (λx. Some eg4-sq) ⊢ ([ (eg4-sq, 0, new-map-state, p3), (eg4-sq, 0,
new-map-state, p3)], new-heap) →*4* res}

```

**end**

## 5 Properties of Control-flow Semantics

**theory** *IRStepThms*

**imports**

*IRStepObj*

*IRTreeEvalThms*

**begin**

We prove that within the same graph, a configuration triple will always transition to the same subsequent configuration. Therefore, our step semantics is deterministic.

**theorem** *stepDet*:

$$(g, p \vdash (nid, m, h) \rightarrow next) \implies \\ (\forall next'. ((g, p \vdash (nid, m, h) \rightarrow next') \longrightarrow next = next'))$$

**proof** (*induction rule: step.induct*)

**case** (*SequentialNode nid next m h*)

**have** *notif*:  $\neg(is\_IfNode\ (kind\ g\ nid))$

**using** *SequentialNode.hyps(1) is-sequential-node.simps*

**by** (*metis is-IfNode-def*)

**have** *notend*:  $\neg(is\_AbstractEndNode\ (kind\ g\ nid))$

**using** *SequentialNode.hyps(1) is-sequential-node.simps*

**by** (*metis is-AbstractEndNode.simps is-EndNode.elims(2) is-LoopEndNode-def*)

**have** *notnew*:  $\neg(is\_NewInstanceNode\ (kind\ g\ nid))$

**using** *SequentialNode.hyps(1) is-sequential-node.simps*

**by** (*metis is-NewInstanceNode-def*)

**have** *notload*:  $\neg(is\_LoadFieldNode\ (kind\ g\ nid))$

**using** *SequentialNode.hyps(1) is-sequential-node.simps*

**by** (*metis is-LoadFieldNode-def*)

**have** *notstore*:  $\neg(is\_StoreFieldNode\ (kind\ g\ nid))$

```

    using SequentialNode.hyps(1) is-sequential-node.simps
    by (metis is-StoreFieldNode-def)
    have notdivrem:  $\neg$ (is-IntegerDivRemNode (kind g nid))
    using SequentialNode.hyps(1) is-sequential-node.simps is-SignedDivNode-def
    is-SignedRemNode-def
    by (metis is-IntegerDivRemNode.simps)
    from notif notend notnew notload notstore notdivrem
    show ?case using SequentialNode.step.cases
    by (smt (z3) IRNode.disc(1028) IRNode.disc(2270) IRNode.discI(31) Pair-inject
    is-sequential-node.simps(18) is-sequential-node.simps(43) is-sequential-node.simps(44))
next
case (IfNode nid cond tb fb m val next h)
then have notseq:  $\neg$ (is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: IfNode.hyps(1))
have notend:  $\neg$ (is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: IfNode.hyps(1))
have notdivrem:  $\neg$ (is-IntegerDivRemNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: IfNode.hyps(1))
from notseq notend notdivrem show ?case using IfNode repDet evalDet IRN-
ode.distinct IRNode.inject(11) Pair-inject step.simps
    by (smt (z3) IRNode.distinct IRNode.inject(12) Pair-inject step.simps)
next
case (EndNodes nid merge iphis inputs m vs m' h)
have notseq:  $\neg$ (is-sequential-node (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps is-sequential-node.simps
    by (metis is-EndNode.elims(2) is-LoopEndNode-def)
have notif:  $\neg$ (is-IfNode (kind g nid))
    using EndNodes.hyps(1) is-IfNode-def is-AbstractEndNode.elims
    by (metis IRNode.distinct-disc(1058) is-EndNode.simps(12))
have notref:  $\neg$ (is-RefNode (kind g nid))
    using EndNodes.hyps(1) is-sequential-node.simps
    using IRNode.disc(1899) IRNode.distinct(1473) is-AbstractEndNode.simps
    is-EndNode.elims(2) is-LoopEndNode-def is-RefNode-def
    by metis
have notnew:  $\neg$ (is-NewInstanceNode (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps
    using IRNode.distinct-disc(1442) is-EndNode.simps(29) is-NewInstanceNode-def
    by (metis IRNode.distinct-disc(1901) is-EndNode.simps(32))
have notload:  $\neg$ (is-LoadFieldNode (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps
    using is-LoadFieldNode-def
    by (metis IRNode.distinct-disc(1706) is-EndNode.simps(21))
have notstore:  $\neg$ (is-StoreFieldNode (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps is-StoreFieldNode-def
    by (metis IRNode.distinct-disc(1926) is-EndNode.simps(44))
have notdivrem:  $\neg$ (is-IntegerDivRemNode (kind g nid))

```

```

    using EndNodes.hyps(1) is-AbstractEndNode.simps is-SignedDivNode-def is-SignedRemNode-def
    using IRNode.distinct-disc(1498) IRNode.distinct-disc(1500) is-IntegerDivRemNode.simps
is-EndNode.simps(36) is-EndNode.simps(37)
    by auto
    from notseq notif notref notnew notload notstore notdivrem
    show ?case using EndNodes repAllDet evalAllDet
    by (smt (z3) is-IfNode-def is-LoadFieldNode-def is-NewInstanceNode-def is-RefNode-def
is-StoreFieldNode-def is-SignedDivNode-def is-SignedRemNode-def Pair-inject is-IntegerDivRemNode.elims(3)
step.cases)
next
    case (NewInstanceNode nid f obj nxt h' ref h m' m)
    then have notseq: ¬(is-sequential-node (kind g nid))
        using is-sequential-node.simps is-AbstractMergeNode.simps
        by (simp add: NewInstanceNode.hyps(1))
    have notend: ¬(is-AbstractEndNode (kind g nid))
        using is-AbstractMergeNode.simps
        by (simp add: NewInstanceNode.hyps(1))
    have notif: ¬(is-IfNode (kind g nid))
        using is-AbstractMergeNode.simps
        by (simp add: NewInstanceNode.hyps(1))
    have notref: ¬(is-RefNode (kind g nid))
        using is-AbstractMergeNode.simps
        by (simp add: NewInstanceNode.hyps(1))
    have notload: ¬(is-LoadFieldNode (kind g nid))
        using is-AbstractMergeNode.simps
        by (simp add: NewInstanceNode.hyps(1))
    have notstore: ¬(is-StoreFieldNode (kind g nid))
        using is-AbstractMergeNode.simps
        by (simp add: NewInstanceNode.hyps(1))
    have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
        using is-AbstractMergeNode.simps
        by (simp add: NewInstanceNode.hyps(1))
    from notseq notend notif notref notload notstore notdivrem
    show ?case using NewInstanceNode step.cases
        by (smt (z3) IRNode.disc(1028) IRNode.disc(2270) IRNode.discI(11) IRN-
ode.distinct(2311) IRNode.distinct(2313) IRNode.inject(31) Pair-inject)
next
    case (LoadFieldNode nid f obj nxt m ref h v m')
    then have notseq: ¬(is-sequential-node (kind g nid))
        using is-sequential-node.simps is-AbstractMergeNode.simps
        by (simp add: LoadFieldNode.hyps(1))
    have notend: ¬(is-AbstractEndNode (kind g nid))
        using is-AbstractEndNode.simps
        by (simp add: LoadFieldNode.hyps(1))
    have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
        using is-AbstractEndNode.simps
        by (simp add: LoadFieldNode.hyps(1))
    from notseq notend notdivrem
    show ?case using LoadFieldNode step.cases repDet evalDet

```

```

    by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject Value.inject(3)
option.distinct(1) option.inject)
next
case (StaticLoadFieldNode nid f nrt h v m' m)
then have notseq: ¬(is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: StaticLoadFieldNode.hyps(1))
have notend: ¬(is-AbstractEndNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: StaticLoadFieldNode.hyps(1))
have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
  by (simp add: StaticLoadFieldNode.hyps(1))
from notseq notend notdivrem
show ?case using StaticLoadFieldNode step.cases
  by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject option.distinct(1))
next
case (StoreFieldNode nid f newval uu obj nrt m val ref h' h m')
then have notseq: ¬(is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: StoreFieldNode.hyps(1))
have notend: ¬(is-AbstractEndNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: StoreFieldNode.hyps(1))
have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
  by (simp add: StoreFieldNode.hyps(1))
from notseq notend notdivrem
show ?case using StoreFieldNode step.cases repDet evalDet
  by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject Value.inject(3)
option.distinct(1) option.inject)
next
case (StaticStoreFieldNode nid f newval uv nrt m val h' h m')
then have notseq: ¬(is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: StaticStoreFieldNode.hyps(1))
have notend: ¬(is-AbstractEndNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: StaticStoreFieldNode.hyps(1))
have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
  by (simp add: StaticStoreFieldNode.hyps(1))
from notseq notend notdivrem
show ?case using StoreFieldNode step.cases repDet evalDet
  by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject Static-
StoreFieldNode.hyps(1) StaticStoreFieldNode.hyps(2) StaticStoreFieldNode.hyps(3)
StaticStoreFieldNode.hyps(4) StaticStoreFieldNode.hyps(5) option.distinct(1))
next

```

```

case (SignedDivNode nid x y zero sb nxt m v1 v2 v m' h)
then have notseq:  $\neg(\text{is-sequential-node } (\text{kind } g \text{ } nid))$ 
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: SignedDivNode.hyps(1))
have notend:  $\neg(\text{is-AbstractEndNode } (\text{kind } g \text{ } nid))$ 
  using is-AbstractEndNode.simps
  by (simp add: SignedDivNode.hyps(1))
from notseq notend
show ?case using SignedDivNode step.cases repDet evalDet
  by (smt (z3) IRNode.distinct(1091) IRNode.distinct(1739) IRNode.distinct(2311)
IRNode.distinct(2601) IRNode.distinct(2605) IRNode.inject(40) Pair-inject)
next
  case (SignedRemNode nid x y zero sb nxt m v1 v2 v m' h)
  then have notseq:  $\neg(\text{is-sequential-node } (\text{kind } g \text{ } nid))$ 
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: SignedRemNode.hyps(1))
  have notend:  $\neg(\text{is-AbstractEndNode } (\text{kind } g \text{ } nid))$ 
    using is-AbstractEndNode.simps
    by (simp add: SignedRemNode.hyps(1))
  from notseq notend
  show ?case using SignedRemNode step.cases repDet evalDet
    by (smt (z3) IRNode.distinct(1093) IRNode.distinct(1741) IRNode.distinct(2313)
IRNode.distinct(2601) IRNode.distinct(2627) IRNode.inject(41) Pair-inject)
qed

```

**lemma** *stepRefNode*:

```

 $\llbracket \text{kind } g \text{ } nid = \text{RefNode } nid' \rrbracket \implies g, p \vdash (nid, m, h) \rightarrow (nid', m, h)$ 
by (simp add: SequentialNode)

```

**lemma** *IfNodeStepCases*:

```

assumes kind g nid = IfNode cond tb fb
assumes g  $\vdash$  cond  $\simeq$  condE
assumes [m, p]  $\vdash$  condE  $\mapsto$  v
assumes g, p  $\vdash$  (nid, m, h)  $\rightarrow$  (nid', m, h)
shows nid'  $\in$  {tb, fb}
using step.IfNode repDet stepDet assms
by (metis insert-iff old.prod.inject)

```

**lemma** *IfNodeSeq*:

```

shows kind g nid = IfNode cond tb fb  $\longrightarrow$   $\neg(\text{is-sequential-node } (\text{kind } g \text{ } nid))$ 
unfolding is-sequential-node.simps by simp

```

**lemma** *IfNodeCond*:

```

assumes kind g nid = IfNode cond tb fb
assumes g, p  $\vdash$  (nid, m, h)  $\rightarrow$  (nid', m, h)
shows  $\exists$  condE v. ((g  $\vdash$  cond  $\simeq$  condE)  $\wedge$  ([m, p]  $\vdash$  condE  $\mapsto$  v))
using assms(2,1) by (induct (nid,m,h) (nid',m,h) rule: step.induct; auto)

```

```

lemma step-in-ids:
  assumes  $g, p \vdash (nid, m, h) \rightarrow (nid', m', h')$ 
  shows  $nid \in ids\ g$ 
  using assms apply (induct ( $nid, m, h$ ) ( $nid', m', h'$ ) rule: step.induct)
  using is-sequential-node.simps(45) not-in-g
  apply simp
  apply (metis is-sequential-node.simps(53))
  using ids-some
  using IRNode.distinct(1113) apply presburger
  using EndNodes(1) is-AbstractEndNode.simps is-EndNode.simps(45) ids-some
  apply (metis IRNode.disc(1218) is-EndNode.simps(52))
  by simp+

end

```