Unspecified Veriopt Theory

December 14, 2021

Contents

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 calculates during the traversal of the control flow graph.

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

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

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

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

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

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* |

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

inductive *not-undef-or-fail* :: *Value* ⇒ *Value* ⇒ *bool* **where**
⟦*value* ≠ *UndefVal*⟧ ⟹ *not-undef-or-fail value value*

notation (*latex* **output**)
not-undef-or-fail (- = -)

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

ConstantExpr:
⟦*valid-value* (*constantAsStamp c*) *c*⟧
⟹ [*m*,*p*] ⊢ (*ConstantExpr c*) ↦ *c* |

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

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

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

BinaryExpr:
⟦[*m*,*p*] ⊢ *xe* ↦ *x*;
[*m*,*p*] ⊢ *ye* ↦ *y*;
result = (*bin-eval op x y*);
result ≠ *UndefVal*⟧
⟹ [*m*,*p*] ⊢ (*BinaryExpr op xe ye*) ↦ *result* |

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

evalRules

$$\frac{valid\text{-}value\ (constantAsStamp\ c)\ c}{[m,p] \vdash ConstantExpr\ c \mapsto c}$$

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

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

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

$$\frac{\begin{array}{c}[m,p] \vdash xe \mapsto x \\ [m,p] \vdash ye \mapsto y \qquad result = bin\text{-}eval\ op\ x\ y \qquad result \neq UndefVal\end{array}}{[m,p] \vdash BinaryExpr\ op\ xe\ ye \mapsto result}$$

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

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

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

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

EvalCons:
$\llbracket [m,p] \vdash x \mapsto xval;$
$\quad [m,p] \vdash yy \mapsto_L yyval\rrbracket$
$\quad \Longrightarrow [m,p] \vdash (x\#yy) \mapsto_L (xval\#yyval)$

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

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

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

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

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

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

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

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

instance proof
  fix $x\ y\ z$ :: *IRExpr*
  show $x < y \longleftrightarrow x \leq y \land \neg\ (y \leq x)$ **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* ⇒ (*IRNode* × *Stamp*) ⇒ *ID option* **where**
  find-node-and-stamp g (*n*,*s*) =
    find (λ*i*. *kind g i* = *n* ∧ *stamp g i* = *s*) (*sorted-list-of-set*(*ids g*))

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

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*⟧

$\Longrightarrow g \vdash n \simeq (\textit{UnaryExpr UnaryNeg xe})$ |

LogicNegationNode:
$[\![\textit{kind } g\ n = \textit{LogicNegationNode } x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (\textit{UnaryExpr UnaryLogicNegation xe})$ |

AddNode:
$[\![\textit{kind } g\ n = \textit{AddNode } x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (\textit{BinaryExpr BinAdd xe ye})$ |

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

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

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

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

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

IntegerBelowNode:
$[\![\textit{kind } g\ n = \textit{IntegerBelowNode } x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (\textit{BinaryExpr BinIntegerBelow xe ye})$ |

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

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

NarrowNode:
$[\![kind\ g\ n = NarrowNode\ inputBits\ resultBits\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ (UnaryNarrow\ inputBits\ resultBits)\ xe)$ |

SignExtendNode:
$[\![kind\ g\ n = SignExtendNode\ inputBits\ resultBits\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ (UnarySignExtend\ inputBits\ resultBits)\ xe)$ |

ZeroExtendNode:
$[\![kind\ g\ n = ZeroExtendNode\ inputBits\ resultBits\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ (UnaryZeroExtend\ inputBits\ resultBits)\ xe)$ |

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

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

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

RepNil:
$g \vdash [] \simeq_L []$ |

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

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

repRules

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

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

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

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

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

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

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

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

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

stamp-unary op - = *IllegalStamp*

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

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

stamp-binary op - - = *IllegalStamp*

fun *stamp-expr* :: *IRExpr* ⇒ *Stamp* **where**
stamp-expr (*UnaryExpr op x*) = *stamp-unary op* (*stamp-expr x*) |

stamp-expr (*BinaryExpr bop x y*) = *stamp-binary bop* (*stamp-expr x*) (*stamp-expr y*) |
stamp-expr (*ConstantExpr val*) = *constantAsStamp val* |
stamp-expr (*LeafExpr i s*) = *s* |
stamp-expr (*ParameterExpr i s*) = *s* |
stamp-expr (*ConditionalExpr c t f*) = *meet* (*stamp-expr t*) (*stamp-expr f*)

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

fun *unary-node* :: *IRUnaryOp* ⇒ *ID* ⇒ *IRNode* **where**
unary-node UnaryAbs v = *AbsNode v* |
unary-node UnaryNot v = *NotNode v* |
unary-node UnaryNeg v = *NegateNode v* |
unary-node UnaryLogicNegation v = *LogicNegationNode v* |
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**
$n \notin ids\ g \Longrightarrow fresh\text{-}id\ g\ n$

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

fun $get\text{-}fresh\text{-}id :: IRGraph \Rightarrow ID$ **where**

$get\text{-}fresh\text{-}id\ g = last(sorted\text{-}list\text{-}of\text{-}set(ids\ g)) + 1$

export-code *get-fresh-id*

value *get-fresh-id eg2-sq*
value $get\text{-}fresh\text{-}id\ (add\text{-}node\ 6\ (ParameterNode\ 2,\ default\text{-}stamp)\ eg2\text{-}sq)$

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:
$[\![find\text{-}node\text{-}and\text{-}stamp\ g\ (ConstantNode\ c,\ constantAsStamp\ c) = Some\ n]\!]$
$\Longrightarrow g \triangleleft (ConstantExpr\ c) \rightsquigarrow (g,\ n)$ |

ConstantNodeNew:
$[\![find\text{-}node\text{-}and\text{-}stamp\ g\ (ConstantNode\ c,\ constantAsStamp\ c) = None;$
$n = get\text{-}fresh\text{-}id\ g;$
$g' = add\text{-}node\ n\ (ConstantNode\ c,\ constantAsStamp\ c)\ g\ ]\!]$
$\Longrightarrow g \triangleleft (ConstantExpr\ c) \rightsquigarrow (g',\ n)$ |

ParameterNodeSame:
$[\![find\text{-}node\text{-}and\text{-}stamp\ g\ (ParameterNode\ i,\ s) = Some\ n]\!]$
$\Longrightarrow g \triangleleft (ParameterExpr\ i\ s) \rightsquigarrow (g,\ n)$ |

ParameterNodeNew:
$[\![find\text{-}node\text{-}and\text{-}stamp\ g\ (ParameterNode\ i,\ s) = None;$
$n = get\text{-}fresh\text{-}id\ g;$
$g' = add\text{-}node\ n\ (ParameterNode\ i,\ s)\ g]\!]$
$\Longrightarrow g \triangleleft (ParameterExpr\ i\ s) \rightsquigarrow (g',\ n)$ |

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

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

$\Longrightarrow g \triangleleft (ConditionalExpr\ ce\ te\ fe) \leadsto (g',\ n)\ |$

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

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

BinaryNodeSame:
$[\![g \triangleleft_L [xe,\ ye] \leadsto (g2,\ [x,\ y]);$
$s' = stamp\text{-}binary\ op\ (stamp\ g2\ x)\ (stamp\ g2\ y);$
$find\text{-}node\text{-}and\text{-}stamp\ g2\ (bin\text{-}node\ op\ x\ y,\ s') = Some\ n]\!]$
$\Longrightarrow g \triangleleft (BinaryExpr\ op\ xe\ ye) \leadsto (g2,\ n)\ |$

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

AllLeafNodes:
$stamp\ g\ n = s$
$\Longrightarrow g \triangleleft (LeafExpr\ n\ s) \leadsto (g,\ n)\ |$

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

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

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

unrepRules

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}g \triangleleft xe \leadsto (g2,\ x) \qquad s' = \textit{stamp-unary}\ op\ (\textit{stamp}\ g2\ x) \\ \textit{find-node-and-stamp}\ g2\ (\textit{unary-node}\ op\ x,\ s') = \textit{Some}\ n\end{array}}{g \triangleleft \textit{UnaryExpr}\ op\ xe \leadsto (g2,\ n)}$$

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

$$\frac{\textit{stamp}\ g\ n = s}{g \triangleleft \textit{LeafExpr}\ n\ s \leadsto (g,\ n)}$$

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

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

definition *encodeeval* :: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *ID* ⇒ *Value* ⇒ *bool*
([-,-,-] ⊢ - ↦ - *50*)
where
$encodeeval\ g\ m\ p\ n\ v = (\exists\ e.\ (g \vdash n \simeq e) \wedge ([m,p] \vdash e \mapsto v))$

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

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

definition *graph-refinement* :: *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
graph-refinement g1 g2 =
$(\forall\ n\ .\ n \in ids\ g1 \longrightarrow (\forall\, e1.\ (g1 \vdash n \simeq e1) \longrightarrow (\exists\, e2.\ (g2 \vdash n \simeq e2) \wedge e1 \geq e2)))$

lemma *graph-refinement*:
$graph\text{-}refinement\ g1\ g2 \Longrightarrow (\forall\, n\ m\ p\ v.\ n \in ids\ g1 \longrightarrow ([g1,\ m,\ p] \vdash n \mapsto v) \longrightarrow ([g2,\ m,\ p] \vdash n \mapsto v))$
by (*meson encodeeval-def graph-refinement-def le-expr-def*)

definition *graph-represents-expression* :: *IRGraph* ⇒ *ID* ⇒ *IRExpr* ⇒ *bool*
(- ⊢ - ⊴ - *50*)
where
$graph\text{-}represents\text{-}expression\ g\ n\ e = (\forall\ \ m\ p\ v\ .\ ([m,p] \vdash e \mapsto v) \longrightarrow ([g,m,p] \vdash n \mapsto v))$

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

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

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

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

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

lemma *rep-negate* [*rep*]:
  $g \vdash n \simeq e \Longrightarrow$
   $kind\ g\ n = NegateNode\ x \Longrightarrow$

$(\exists\, xe.\ e = \mathit{UnaryExpr}\ \mathit{UnaryNeg}\ xe)$
by $(\mathit{induction}\ \mathit{rule}\colon \mathit{rep.induct};\ \mathit{auto})$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

method *solve-det* **uses** *node* =
(*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*⟩⟩⟩ |
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*. - = *node x y* ⟹ -) ⟹ - ⇒

‹*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 $(g \vdash n \simeq e1) \Longrightarrow (g \vdash n \simeq e2) \Longrightarrow e1 = e2$
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 ⊢ xs ≃L e1 ⟹
```

$g \vdash xs \simeq_L e2 \Longrightarrow$
$e1 = e2$
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*:
$[m,p] \vdash e \mapsto v1 \Longrightarrow$
$[m,p] \vdash e \mapsto v2 \Longrightarrow$
$v1 = v2$
apply (*induction arbitrary*: *v2 rule*: *evaltree.induct*)
by (*elim EvalTreeE*; *auto*)+

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

lemma *encodeEvalDet*:
$[g,m,p] \vdash e \mapsto v1 \Longrightarrow$
$[g,m,p] \vdash e \mapsto v2 \Longrightarrow$
$v1 = v2$
by (*metis encodeeval-def evalDet repDet*)

lemma *graphDet*: $([g,m,p] \vdash nid \mapsto v1) \wedge ([g,m,p] \vdash nid \mapsto v2) \Longrightarrow v1 = v2$
using *encodeEvalDet* **by** *blast*

A valid value cannot be $UndefVal$.

lemma *valid-not-undef*:
assumes *a1*: *valid-value s val*
assumes *a2*: $s \neq$ *VoidStamp*
shows *val* $\neq$ *UndefVal*
apply (*rule valid-value.elims*(*1*)[*of s val True*])
using *a1 a2* **by** *auto*

lemma *valid-VoidStamp*[*elim*]:
shows *valid-value VoidStamp val* $\Longrightarrow$
val = *UndefVal*

using *valid-value.simps* **by** (*metis IRTreeEval.val-to-bool.cases*)

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

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

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

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

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

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

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

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

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

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

using *assms valid-int32* **by** *force*

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

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

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

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

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 $e \geq e'$
shows (*UnaryExpr op e*) ≥ (*UnaryExpr op e′*)
using *UnaryExpr assms* **by** *auto*

lemma *mono-binary*:
assumes $x \geq x'$

```
  assumes y ≥ y′
  shows (BinaryExpr op x y) ≥ (BinaryExpr op x′ y′)
  using BinaryExpr assms by auto

lemma mono-conditional:
  assumes ce ≥ ce′
  assumes te ≥ te′
  assumes fe ≥ fe′
  shows (ConditionalExpr ce te fe) ≥ (ConditionalExpr ce′ te′ fe′)
proof (simp only: le-expr-def; (rule allI)+; rule impI)
  fix m p v
  assume a: [m,p] ⊢ ConditionalExpr ce te fe ↦ v
  then obtain cond where ce: [m,p] ⊢ ce ↦ cond by auto
  then have ce′: [m,p] ⊢ ce′ ↦ cond using assms by auto
  define branch  where b:  branch  = (if val-to-bool cond then te else fe)
  define branch′ where b′: branch′ = (if val-to-bool cond then te′ else fe′)
  then have [m,p] ⊢ branch ↦ v using a b ce evalDet by blast
  then have [m,p] ⊢ branch′ ↦ v using assms b b′ by auto
  then show [m,p] ⊢ ConditionalExpr ce′ te′ fe′ ↦ 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 ∧ kind g2 n = AbsNode x
  assumes (g1 ⊢ x ≃ xe1) ∧ (g2 ⊢ x ≃ xe2)
  assumes xe1 ≥ xe2
  assumes (g1 ⊢ n ≃ e1) ∧ (g2 ⊢ n ≃ e2)
  shows e1 ≥ e2
  by (metis AbsNode assms(1) assms(2) assms(3) assms(4) mono-unary repDet)

lemma mono-not:
  assumes kind g1 n = NotNode x ∧ kind g2 n = NotNode x
  assumes (g1 ⊢ x ≃ xe1) ∧ (g2 ⊢ x ≃ xe2)
  assumes xe1 ≥ xe2
  assumes (g1 ⊢ n ≃ e1) ∧ (g2 ⊢ n ≃ e2)
  shows e1 ≥ e2
```

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

lemma *mono-negate*:
assumes *kind g1 n* = *NegateNode x* ∧ *kind g2 n* = *NegateNode x*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes *xe1* ≥ *xe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis NegateNode assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *mono-unary repDet*)

lemma *mono-logic-negation*:
assumes *kind g1 n* = *LogicNegationNode x* ∧ *kind g2 n* = *LogicNegationNode x*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes *xe1* ≥ *xe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis LogicNegationNode assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *mono-unary repDet*)

lemma *mono-narrow*:
assumes *kind g1 n* = *NarrowNode ib rb x* ∧ *kind g2 n* = *NarrowNode ib rb x*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes *xe1* ≥ *xe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
using *assms mono-unary repDet NarrowNode*
by *metis*

lemma *mono-sign-extend*:
assumes *kind g1 n* = *SignExtendNode ib rb x* ∧ *kind g2 n* = *SignExtendNode ib rb x*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes *xe1* ≥ *xe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis SignExtendNode assms*(*1*) *assms*(*2*) *assms*(*3*) *assms*(*4*) *mono-unary repDet*)

lemma *mono-zero-extend*:
assumes *kind g1 n* = *ZeroExtendNode ib rb x* ∧ *kind g2 n* = *ZeroExtendNode ib rb x*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes *xe1* ≥ *xe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
using *assms mono-unary repDet ZeroExtendNode*
by *metis*

lemma *mono-conditional-graph*:

assumes *kind g1 n = ConditionalNode c t f* ∧ *kind g2 n = ConditionalNode c t f*
assumes (*g1* ⊢ *c* ≃ *ce1*) ∧ (*g2* ⊢ *c* ≃ *ce2*)
assumes (*g1* ⊢ *t* ≃ *te1*) ∧ (*g2* ⊢ *t* ≃ *te2*)
assumes (*g1* ⊢ *f* ≃ *fe1*) ∧ (*g2* ⊢ *f* ≃ *fe2*)
assumes *ce1* ≥ *ce2* ∧ *te1* ≥ *te2* ∧ *fe1* ≥ *fe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *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 *kind g1 n = AddNode x y* ∧ *kind g2 n = AddNode x y*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes (*g1* ⊢ *y* ≃ *ye1*) ∧ (*g2* ⊢ *y* ≃ *ye2*)
assumes *xe1* ≥ *xe2* ∧ *ye1* ≥ *ye2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
using *mono-binary assms*
by (*metis AddNodeE IRNode.inject*(*2*) *repDet rep-add*)

lemma *mono-mul*:
assumes *kind g1 n = MulNode x y* ∧ *kind g2 n = MulNode x y*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes (*g1* ⊢ *y* ≃ *ye1*) ∧ (*g2* ⊢ *y* ≃ *ye2*)
assumes *xe1* ≥ *xe2* ∧ *ye1* ≥ *ye2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
using *mono-binary assms*
by (*metis IRNode.inject*(*27*) *MulNodeE repDet rep-mul*)

lemma *encodes-contains*:
g ⊢ *n* ≃ *e* ⟹
kind g n ≠ *NoNode*
apply (*induction rule*: *rep.induct*)
apply (*match IRNode.distinct* **in** *e*: *?n* ≠ *NoNode* ⇒
‹*presburger add*: *e*›)+
by *fastforce*

lemma *no-encoding*:
assumes *n* ∉ *ids g*
shows ¬(*g* ⊢ *n* ≃ *e*)
using *assms* **apply** *simp* **apply** (*rule notI*) **by** (*induction e*; *simp add*: *encodes-contains*)

lemma *not-excluded-keep-type*:
assumes *n* ∈ *ids g1*
assumes *n* ∉ *excluded*
assumes (*excluded* ⊴ *as-set g1*) ⊆ *as-set g2*

shows *kind g1 n* = *kind g2 n* ∧ *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* ⇒ *IRNode* =
(*match IRNode.inject* **in** *i*: (*node* - = *node* -) = - ⇒
‹*metis i*›)
method *metis-node-eq-binary* **for** *node* :: ′*a* ⇒ ′*a* ⇒ *IRNode* =
(*match IRNode.inject* **in** *i*: (*node* - - = *node* - -) = - ⇒
‹*metis i*›)
method *metis-node-eq-ternary* **for** *node* :: ′*a* ⇒ ′*a* ⇒ ′*a* ⇒ *IRNode* =
(*match IRNode.inject* **in** *i*: (*node* - - - = *node* - - -) = - ⇒
‹*metis i*›)

lemma *graph-semantics-preservation*:
assumes *a*: *e1*′ ≥ *e2*′
assumes *b*: ({*n*′} ⊴ *as-set g1*) ⊆ *as-set g2*
assumes *c*: *g1* ⊢ *n*′ ≃ *e1*′
assumes *d*: *g2* ⊢ *n*′ ≃ *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* ∈ *ids g1*
assume *f*: (*g1* ⊢ *n* ≃ *e1*)

show ∃ *e2*. (*g2* ⊢ *n* ≃ *e2*) ∧ *e1* ≥ *e2*
proof (*cases n* = *n*′)
case *True*
have *g*: *e1* = *e1*′ **using** *c f True repDet* **by** *simp*
have *h*: (*g2* ⊢ *n* ≃ *e2*′) ∧ *e1*′ ≥ *e2*′
using *True a d* **by** *blast*
then show *?thesis*
using *g* **by** *blast*
next
case *False*
have *n* ∉ {*n*′}
using *False* **by** *simp*
then have *i*: *kind g1 n* = *kind g2 n* ∧ *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.prems 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* ⊢ *n* ≃ *UnaryExpr UnaryAbs e2′* **using** *AbsNode.hyps*(*1*) *l m n*
using *AbsNode.prems True d rep.AbsNode* **by** *simp*
then have *r*: *UnaryExpr UnaryAbs e1′* ≥ *UnaryExpr UnaryAbs e2′*
by (*meson a mono-unary*)
then show *?thesis* **using** *ev r*
by (*metis n*)
next
case *False*
have *g1* ⊢ *xn* ≃ *xe1* **using** *m* **by** *simp*
have ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *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 ∃ *xe2*. (*g2* ⊢ *n* ≃ *UnaryExpr UnaryAbs xe2*) ∧ *UnaryExpr UnaryAbs xe1* ≥ *UnaryExpr UnaryAbs xe2*
by (*metis AbsNode.prems l mono-unary rep.AbsNode*)
then show *?thesis*
by *meson*
qed
next
case (*NotNode n x xe1*)
have *k*: *g1* ⊢ *n* ≃ *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* ⊢ *xn* ≃ *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* ⊢ *n* ≃ *UnaryExpr UnaryNot e2′* **using** *NotNode.hyps*(*1*) *l m n*
using *NotNode.prems True d rep.NotNode* **by** *simp*
then have *r*: *UnaryExpr UnaryNot e1′* ≥ *UnaryExpr UnaryNot e2′*
by (*meson a mono-unary*)
then show *?thesis* **using** *ev r*
by (*metis n*)
next
case *False*
have *g1* ⊢ *xn* ≃ *xe1* **using** *m* **by** *simp*
have ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *NotNode*
using *False i b l not-excluded-keep-type singletonD no-encoding*
by (*metis-node-eq-unary NotNode*)
then have ∃ *xe2*. (*g2* ⊢ *n* ≃ *UnaryExpr UnaryNot xe2*) ∧ *UnaryExpr UnaryNot xe1* ≥ *UnaryExpr UnaryNot xe2*

by (*metis NotNode.prems l mono-unary rep.NotNode*)
then show *?thesis*
by *meson*
qed
next
case (*NegateNode n x xe1*)
have *k*: $g1 \vdash n \simeq$ *UnaryExpr UnaryNeg xe1* **using** *f NegateNode*
by (*simp add*: *NegateNode.hyps*(*2*) *rep.NegateNode*)
obtain *xn* **where** *l*: *kind g1 n* = *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*: $g2 \vdash n \simeq$ *UnaryExpr UnaryNeg* $e2'$ **using** *NegateNode.hyps*(*1*) *l m n*
using *NegateNode.prems True d rep.NegateNode* **by** *simp*
then have *r*: *UnaryExpr UnaryNeg* $e1' \geq$ *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 $\exists\ xe2.\ (g2 \vdash n \simeq$ *UnaryExpr UnaryNeg xe2*) $\wedge$ *UnaryExpr UnaryNeg xe1* $\geq$ *UnaryExpr UnaryNeg xe2*
by (*metis NegateNode.prems l mono-unary rep.NegateNode*)
then show *?thesis*
by *meson*
qed
next
case (*LogicNegationNode n x xe1*)
have *k*: $g1 \vdash n \simeq$ *UnaryExpr UnaryLogicNegation xe1* **using** *f LogicNegationNode*
by (*simp add*: *LogicNegationNode.hyps*(*2*) *rep.LogicNegationNode*)
obtain *xn* **where** *l*: *kind g1 n* = *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*: $g2 \vdash n \simeq$ *UnaryExpr UnaryLogicNegation* $e2'$ **using** *LogicNegationNode.hyps*(*1*) *l m n*
using *LogicNegationNode.prems True d rep.LogicNegationNode* **by** *simp*
then have *r*: *UnaryExpr UnaryLogicNegation* $e1' \geq$ *UnaryExpr UnaryLogicNegation* $e2'$
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*. $(g2 \vdash n \simeq$ *UnaryExpr UnaryLogicNegation* $xe2) \wedge$ *UnaryExpr UnaryLogicNegation* $xe1 \geq$ *UnaryExpr UnaryLogicNegation xe2*
by (*metis LogicNegationNode.prems l mono-unary rep.LogicNegationNode*)
then show *?thesis*
by *meson*
qed
next
case (*AddNode n x y xe1 ye1*)
have *k*: $g1 \vdash n \simeq$ *BinaryExpr BinAdd xe1 ye1* **using** *f AddNode*
by (*simp add*: *AddNode.hyps*(*2*) *rep.AddNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *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*. $(g2 \vdash n \simeq$ *BinaryExpr BinAdd xe2 ye2*$) \wedge$ *BinaryExpr BinAdd xe1 ye1* $\geq$ *BinaryExpr BinAdd xe2 ye2*
by (*metis AddNode.prems l mono-binary rep.AddNode xer*)
then show *?thesis*
by *meson*
qed

```
  next
    case (MulNode n x y xe1 ye1)
    have k: g1 ⊢ n ≃ BinaryExpr BinMul xe1 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 ⊢ xn ≃ xe1
      using MulNode.hyps(1) MulNode.hyps(2) by fastforce
    from l have my: g1 ⊢ yn ≃ ye1
      using MulNode.hyps(1) MulNode.hyps(3) by fastforce
    then show ?case
    proof -
      have g1 ⊢ xn ≃ xe1 using mx by simp
      have g1 ⊢ yn ≃ ye1 using my by simp
      have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
        using MulNode
        using a b c d l no-encoding not-excluded-keep-type repDet singletonD
        by (metis-node-eq-binary MulNode)
      have ∃ ye2. (g2 ⊢ yn ≃ ye2) ∧ ye1 ≥ 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 ∃ xe2 ye2. (g2 ⊢ n ≃ BinaryExpr BinMul xe2 ye2) ∧ BinaryExpr
BinMul xe1 ye1 ≥ BinaryExpr BinMul xe2 ye2
        by (metis MulNode.prems l mono-binary rep.MulNode xer)
      then show ?thesis
        by meson
    qed
  next
    case (SubNode n x y xe1 ye1)
    have k: g1 ⊢ n ≃ BinaryExpr BinSub xe1 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 ⊢ xn ≃ xe1
      using SubNode.hyps(1) SubNode.hyps(2) by fastforce
    from l have my: g1 ⊢ yn ≃ ye1
      using SubNode.hyps(1) SubNode.hyps(3) by fastforce
    then show ?case
    proof -
      have g1 ⊢ xn ≃ xe1 using mx by simp
      have g1 ⊢ yn ≃ ye1 using my by simp
      have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
        using SubNode
        using a b c d l no-encoding not-excluded-keep-type repDet singletonD
        by (metis-node-eq-binary SubNode)
      have ∃ ye2. (g2 ⊢ yn ≃ ye2) ∧ ye1 ≥ ye2
      using SubNode a b c d l no-encoding not-excluded-keep-type repDet singletonD
        by (metis-node-eq-binary SubNode)
```

then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinSub xe2 ye2*) ∧ *BinaryExpr BinSub xe1 ye1* ≥ *BinaryExpr BinSub xe2 ye2*
by (*metis SubNode.prems l mono-binary rep.SubNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*AndNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *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* ⊢ *xn* ≃ *xe1*
using *AndNode.hyps*(*1*) *AndNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *AndNode.hyps*(*1*) *AndNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1* **using** *mx* **by** *simp*
have *g1* ⊢ *yn* ≃ *ye1* **using** *my* **by** *simp*
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *AndNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary AndNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *AndNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary AndNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinAnd xe2 ye2*) ∧ *BinaryExpr BinAnd xe1 ye1* ≥ *BinaryExpr BinAnd xe2 ye2*
by (*metis AndNode.prems l mono-binary rep.AndNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*OrNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *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* ⊢ *xn* ≃ *xe1*
using *OrNode.hyps*(*1*) *OrNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *OrNode.hyps*(*1*) *OrNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1* **using** *mx* **by** *simp*
have *g1* ⊢ *yn* ≃ *ye1* **using** *my* **by** *simp*
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *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.prems 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.prems 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 IntegerBelowNode*
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.prems 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 IntegerEqualsNode*
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.prems l mono-binary rep.IntegerEqualsNode xer*)
then show *?thesis*

by *meson*
qed
next
case (*IntegerLessThanNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinIntegerLessThan xe1 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* ⊢ *xn* ≃ *xe1*
using *IntegerLessThanNode.hyps*(*1*) *IntegerLessThanNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *IntegerLessThanNode.hyps*(*1*) *IntegerLessThanNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1* **using** *mx* **by** *simp*
have *g1* ⊢ *yn* ≃ *ye1* **using** *my* **by** *simp*
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *IntegerLessThanNode*
using *a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary IntegerLessThanNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *IntegerLessThanNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary IntegerLessThanNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinIntegerLessThan xe2 ye2*) ∧ *BinaryExpr BinIntegerLessThan xe1 ye1* ≥ *BinaryExpr BinIntegerLessThan xe2 ye2*
by (*metis IntegerLessThanNode.prems l mono-binary rep.IntegerLessThanNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*NarrowNode n inputBits resultBits x xe1*)
have *k*: *g1* ⊢ *n* ≃ *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* ⊢ *xn* ≃ *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*: $g2 \vdash n \simeq$ *UnaryExpr* (*UnaryNarrow inputBits resultBits*) $e2'$ **using** *NarrowNode.hyps*(*1*) *l m n*
using *NarrowNode.prems True d rep.NarrowNode* **by** *simp*
then have *r*: *UnaryExpr* (*UnaryNarrow inputBits resultBits*) $e1' \geq$ *UnaryExpr* (*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*. ($g2 \vdash n \simeq$ *UnaryExpr* (*UnaryNarrow inputBits resultBits*) *xe2*) $\wedge$ *UnaryExpr* (*UnaryNarrow inputBits resultBits*) $xe1 \geq$ *UnaryExpr* (*UnaryNarrow inputBits resultBits*) *xe2*
by (*metis NarrowNode.prems l mono-unary rep.NarrowNode*)
then show *?thesis*
by *meson*
qed
next
case (*SignExtendNode n inputBits resultBits x xe1*)
have *k*: $g1 \vdash n \simeq$ *UnaryExpr* (*UnarySignExtend inputBits resultBits*) *xe1* **using** *f SignExtendNode*
by (*simp add*: *SignExtendNode.hyps*(*2*) *rep.SignExtendNode*)
obtain *xn* **where** *l*: *kind g1 n* = *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*: $g2 \vdash n \simeq$ *UnaryExpr* (*UnarySignExtend inputBits resultBits*) $e2'$ **using** *SignExtendNode.hyps*(*1*) *l m n*
using *SignExtendNode.prems True d rep.SignExtendNode* **by** *simp*
then have *r*: *UnaryExpr* (*UnarySignExtend inputBits resultBits*) $e1' \geq$ *UnaryExpr* (*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* $\simeq$ *UnaryExpr* (*UnarySignExtend inputBits resultBits*) *xe2*) $\wedge$ *UnaryExpr* (*UnarySignExtend inputBits resultBits*) *xe1* $\geq$ *UnaryExpr* (*UnarySignExtend inputBits resultBits*) *xe2*
by (*metis SignExtendNode.prems l mono-unary rep.SignExtendNode*)
then show *?thesis*
by *meson*
qed
next
case (*ZeroExtendNode n inputBits resultBits x xe1*)
have *k*: *g1* $\vdash$ *n* $\simeq$ *UnaryExpr* (*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* $\simeq$ *UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *e2'* **using** *ZeroExtendNode.hyps*(*1*) *l m n*
using *ZeroExtendNode.prems True d rep.ZeroExtendNode* **by** *simp*
then have *r*: *UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *e1'* $\geq$ *UnaryExpr* (*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* $\simeq$ *UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *xe2*) $\wedge$ *UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *xe1* $\geq$ *UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *xe2*
by (*metis ZeroExtendNode.prems 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 = (∀ n1 n2 . n1 ∈ ids g ∧ n2 ∈ ids g ⟶
      (∀ e. (g ⊢ n1 ≃ e) ∧ (g ⊢ n2 ≃ e) ⟶ n1 = n2))

lemma tree-to-graph-rewriting:
  e1 ≥ e2
  ∧ (g1 ⊢ n ≃ e1) ∧ maximal-sharing g1
  ∧ ({n} ⊴ as-set g1) ⊆ as-set g2
  ∧ (g2 ⊢ n ≃ e2) ∧ maximal-sharing g2
  ⟹ 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 ≥ e2
  using assms
  by simp
declare [[simp-trace=false]]

lemma subset-implies-evals:
  assumes as-set g1 ⊆ as-set g2
  shows (g1 ⊢ n ≃ e) ⟹ (g2 ⊢ n ≃ e)
proof (induction e arbitrary: n)
  case (UnaryExpr op e)
  then have n ∈ 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 NegateN-
ode NotNode SignExtendNode ZeroExtendNode)
next
  case (BinaryExpr op e1 e2)
  then have n ∈ 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 ∈ 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 ConditionalExpr ConditionalExprE
    by (smt (verit, best) ConditionalNode ConditionalNodeE)
next
  case (ConstantExpr x)
  then have n ∈ 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 ConstantExpr ConstantExprE
    by (metis ConstantNode ConstantNodeE)
next
  case (ParameterExpr x1 x2)
  then have in-g1: n ∈ ids g1
    using no-encoding by force
  then have kinds: kind g1 n = kind g2 n
    using assms unfolding as-set-def
    by blast
  from in-g1 have stamps: stamp g1 n = stamp g2 n
    using assms unfolding as-set-def
    by blast
  from kinds stamps show ?case using ParameterExpr ParameterExprE
    by (metis ParameterNode ParameterNodeE)
next
  case (LeafExpr nid s)
  then have in-g1: n ∈ ids g1
    using no-encoding by force
  then have kinds: kind g1 n = kind g2 n
    using assms unfolding as-set-def
    by blast
  from in-g1 have stamps: stamp g1 n = stamp g2 n
    using assms unfolding as-set-def
    by blast
  from 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*: $n \in$ *ids g1*
  using *no-encoding* **by** *force*
then have *kinds*: *kind g1 n* = *kind g2 n*
  using *assms* **unfolding** *as-set-def*
  by *blast*
from *in-g1* **have** *stamps*: *stamp g1 n* = *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*: $n \in$ *ids g1*
  using *no-encoding* **by** *force*
then have *kinds*: *kind g1 n* = *kind g2 n*
  using *assms* **unfolding** *as-set-def*
  by *blast*
from *in-g1* **have** *stamps*: *stamp g1 n* = *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 *as-set g1* $\subseteq$ *as-set g2*
shows *graph-refinement g1 g2*
proof –
have *ids g1* $\subseteq$ *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*:$n \in$ *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*
$\wedge$ *as-set g1* $\subseteq$ *as-set g2* $\wedge$ *maximal-sharing g1*
$\wedge$ (*g2* $\vdash$ $n \simeq$ *e2*) $\wedge$ *maximal-sharing g2*
$\Longrightarrow$ (*g2* $\vdash$ $n \trianglelefteq$ *e1*) $\wedge$ *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 = $((\lambda f.\ \lambda p.$ *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* (λx.(*is-PhiNode* (*kind g x*)))
    (*sorted-list-of-set* (*usages g n*)))

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

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

fun *set-phis* :: *ID list* ⇒ *Value list* ⇒ *MapState* ⇒ *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* ⇒ *Params* ⇒ (*ID* × *MapState* × *FieldRefHeap*) ⇒ (*ID* × *MapState* × *FieldRefHeap*) ⇒ *bool*
  (-, - ⊢ - → - *55*) **for** *g p* **where**

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

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

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

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

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

NewInstanceNode:
$[\![kind\ g\ nid = (NewInstanceNode\ nid\ f\ obj\ nid');$
$(h', ref) = h\text{-}new\text{-}inst\ h;$
$m' = m(nid := ref)]\!]$
$\Longrightarrow g, p \vdash (nid, m, h) \to (nid', m', h') \mid$

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

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

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

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

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

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

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

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* = $Signature \rightharpoonup IRGraph$

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

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

InvokeNodeStep:
$[\![is\text{-}Invoke\ (kind\ g\ nid);$

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

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

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

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

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

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

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

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

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

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

4.4 Big-step Execution

type-synonym *Trace* = (*IRGraph* × *ID* × *MapState* × *Params*) *list*

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

inductive *exec* :: *Program*
    ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap*
    ⇒ *Trace*
    ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap*
    ⇒ *Trace*
    ⇒ *bool*
  (- ⊢ - | - ⟶* - | -)
  for *P*
  where
  $\llbracket P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h')$;
  $\neg(has\text{-}return\ m')$;

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

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

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

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

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

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

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

4.4.1 Heap Testing

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

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

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

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

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

definition *eg4-sq* :: *IRGraph* **where**
eg4-sq = *irgraph* [

(*0*, *StartNode None 4*, *VoidStamp*),
(*1*, *ParameterNode 0*, *default-stamp*),
(*3*, *MulNode 1 1*, *default-stamp*),
(*4*, *NewInstanceNode 4 ''obj-class'' None 5*, *ObjectStamp ''obj-class'' True True True*),
(*5*, *StoreFieldNode 5 field-sq 3 None* (*Some 4*) *6*, *VoidStamp*),
(*6*, *ReturnNode* (*Some 3*) *None*, *default-stamp*)
]

values {*h-load-field field-sq* (*Some 0*) (*prod.snd res*) | *res*.
(*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* ⊢ (*nid*,*m*,*h*) → *next*) ⟹
(∀ *next′*. ((*g*, *p* ⊢ (*nid*,*m*,*h*) → *next′*) ⟶ *next* = *next′*))

proof (*induction rule*: *step.induct*)
case (*SequentialNode nid next m h*)
have *notif*: ¬(*is-IfNode* (*kind g nid*))
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-IfNode-def*)
have *notend*: ¬(*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*: ¬(*is-NewInstanceNode* (*kind g nid*))
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-NewInstanceNode-def*)
have *notload*: ¬(*is-LoadFieldNode* (*kind g nid*))
using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-LoadFieldNode-def*)
have *notstore*: ¬(*is-StoreFieldNode* (*kind g nid*))

using *SequentialNode.hyps*(*1*) *is-sequential-node.simps*
by (*metis is-StoreFieldNode-def*)
have *notdivrem*: ¬(*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*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *IfNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
using *is-AbstractEndNode.simps*
by (*simp add*: *IfNode.hyps*(*1*))
have *notdivrem*: ¬(*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 IRNode.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 i phis inputs m vs m′ h*)
have *notseq*: ¬(*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*: ¬(*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*: ¬(*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*: ¬(*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*: ¬(*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*: ¬(*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*: ¬(*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 nxt 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 nxt 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 nxt 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 StaticStoreFieldNode.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*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *SignedDivNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g 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*: ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps is-AbstractMergeNode.simps*
by (*simp add*: *SignedRemNode.hyps*(*1*))
have *notend*: ¬(*is-AbstractEndNode* (*kind g 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*:
⟦*kind g nid* = *RefNode nid′*⟧ ⟹ *g*, *p* ⊢ (*nid*,*m*,*h*) → (*nid′*,*m*,*h*)
by (*simp add*: *SequentialNode*)

lemma *IfNodeStepCases*:
assumes *kind g nid* = *IfNode cond tb fb*
assumes *g* ⊢ *cond* ≃ *condE*
assumes [*m*, *p*] ⊢ *condE* ↦ *v*
assumes *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m*, *h*)
shows *nid′* ∈ {*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* ⟶ ¬(*is-sequential-node* (*kind g nid*))
unfolding *is-sequential-node.simps* **by** *simp*

lemma *IfNodeCond*:
assumes *kind g nid* = *IfNode cond tb fb*
assumes *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m*, *h*)
shows ∃ *condE v*. ((*g* ⊢ *cond* ≃ *condE*) ∧ ([*m*, *p*] ⊢ *condE* ↦ *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