

Veriopt Theories

September 15, 2022

Contents

1	Data-flow Semantics	1
1.1	Data-flow Tree Representation	1
1.2	Functions for re-calculating stamps	3
1.3	Data-flow Tree Evaluation	4
1.4	Data-flow Tree Refinement	6
1.5	Stamp Masks	7
2	Tree to Graph	8
2.1	Subgraph to Data-flow Tree	8
2.2	Data-flow Tree to Subgraph	12
2.3	Lift Data-flow Tree Semantics	17
2.4	Graph Refinement	17
2.5	Maximal Sharing	17
2.6	Formedness Properties	17
2.7	Dynamic Frames	19
3	Control-flow Semantics	31
3.1	Object Heap	31
3.2	Intraprocedural Semantics	32
3.3	Interprocedural Semantics	34
3.4	Big-step Execution	36
3.4.1	Heap Testing	37
3.5	Data-flow Tree Theorems	38
3.5.1	Deterministic Data-flow Evaluation	38
3.5.2	Typing Properties for Integer Evaluation Functions	38
3.5.3	Evaluation Results are Valid	40
3.5.4	Example Data-flow Optimisations	42
3.5.5	Monotonicity of Expression Refinement	42
3.6	Unfolding rules for evaltree quadruples down to bin-eval level	43
3.7	Lemmas about <i>new_int</i> and integer eval results.	44
3.8	Tree to Graph Theorems	49

3.8.1	Extraction and Evaluation of Expression Trees is Deterministic.	50
3.8.2	Monotonicity of Graph Refinement	56
3.8.3	Lift Data-flow Tree Refinement to Graph Refinement	59
3.8.4	Term Graph Reconstruction	75
3.8.5	Data-flow Tree to Subgraph Preserves Maximal Sharing	83
3.9	Control-flow Semantics Theorems	97
3.9.1	Control-flow Step is Deterministic	98

1 Data-flow Semantics

```

theory IRTreeEval
  imports
    Graph.Stamp
begin

```

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

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

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

The method state also stores a mapping of node ids to values. The contents of this mapping is calculated during the traversal of the control flow graph.

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

```

type-synonym ID = nat
type-synonym MapState = ID  $\Rightarrow$  Value
type-synonym Params = Value list

```

```

definition new-map-state :: MapState where
  new-map-state = ( $\lambda x.$  UndefVal)

```

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
| BinShortCircuitOr
| 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 Functions for re-calculating stamps

Note: in Java all integer calculations are done as 32 or 64 bit calculations. However, here we generalise the operators to allow any size calculations. Many operators have the same output bits as their inputs. However, the unary integer operators that are not *normal_unary* are narrowing or widening operators, so the result bits is specified by the operator. The binary integer operators are divided into three groups: (1) *binary_fixed_32* operators always output 32 bits, (2) *binary_shift_ops* operators output size is determined by their left argument, and (3) other operators output the same number of bits as both their inputs.

abbreviation *binary-fixed-32-ops* :: *IRBinaryOp* set **where**

binary-fixed-32-ops ≡ {*BinShortCircuitOr*, *BinIntegerEquals*, *BinIntegerLessThan*, *BinIntegerBelow*}

abbreviation *binary-shift-ops* :: *IRBinaryOp* set **where**

binary-shift-ops ≡ {*BinLeftShift*, *BinRightShift*, *BinURightShift*}

abbreviation *normal-unary* :: *IRUnaryOp* set **where**

normal-unary ≡ {*UnaryAbs*, *UnaryNeg*, *UnaryNot*, *UnaryLogicNegation*}

fun *stamp-unary* :: *IRUnaryOp* ⇒ *Stamp* ⇒ *Stamp* **where**

stamp-unary *op* (*IntegerStamp* *b* *lo* *hi*) =
unrestricted-stamp (*IntegerStamp* (if *op* ∈ *normal-unary* then *b* else (*ir-resultBits* *op*)) *lo* *hi*) |

stamp-unary *op* - = *IllegalStamp*

fun *stamp-binary* :: *IRBinaryOp* ⇒ *Stamp* ⇒ *Stamp* ⇒ *Stamp* **where**

stamp-binary *op* (*IntegerStamp* *b1* *lo1* *hi1*) (*IntegerStamp* *b2* *lo2* *hi2*) =
 (if *op* ∈ *binary-shift-ops* then *unrestricted-stamp* (*IntegerStamp* *b1* *lo1* *hi1*)
 else if *b1* ≠ *b2* then *IllegalStamp* else
 (if *op* ∈ *binary-fixed-32-ops*
 then *unrestricted-stamp* (*IntegerStamp* 32 *lo1* *hi1*)
 else *unrestricted-stamp* (*IntegerStamp* *b1* *lo1* *hi1*))) |

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*

1.3 Data-flow Tree Evaluation

fun *unary-eval* :: *IRUnaryOp* \Rightarrow *Value* \Rightarrow *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* *v* = *intval-logic-negation* *v* |
unary-eval (*UnaryNarrow* *inBits* *outBits*) *v* = *intval-narrow* *inBits* *outBits* *v* |
unary-eval (*UnarySignExtend* *inBits* *outBits*) *v* = *intval-sign-extend* *inBits* *outBits* *v* |
unary-eval (*UnaryZeroExtend* *inBits* *outBits*) *v* = *intval-zero-extend* *inBits* *outBits* *v*

fun *bin-eval* :: *IRBinaryOp* \Rightarrow *Value* \Rightarrow *Value* \Rightarrow *Value* **where**
bin-eval *BinAdd* *v1* *v2* = *intval-add* *v1* *v2* |
bin-eval *BinMul* *v1* *v2* = *intval-mul* *v1* *v2* |
bin-eval *BinSub* *v1* *v2* = *intval-sub* *v1* *v2* |
bin-eval *BinAnd* *v1* *v2* = *intval-and* *v1* *v2* |
bin-eval *BinOr* *v1* *v2* = *intval-or* *v1* *v2* |
bin-eval *BinXor* *v1* *v2* = *intval-xor* *v1* *v2* |
bin-eval *BinShortCircuitOr* *v1* *v2* = *intval-short-circuit-or* *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*

lemmas *eval-thms* =
intval-abs.simps *intval-negate.simps* *intval-not.simps*
intval-logic-negation.simps *intval-narrow.simps*
intval-sign-extend.simps *intval-zero-extend.simps*
intval-add.simps *intval-mul.simps* *intval-sub.simps*
intval-and.simps *intval-or.simps* *intval-xor.simps*
intval-left-shift.simps *intval-right-shift.simps*
intval-uright-shift.simps *intval-equals.simps*
intval-less-than.simps *intval-below.simps*

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

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

inductive

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

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

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

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

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

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

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

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

inductive

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

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

```

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

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

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

values {v. evaltree new-map-state [IntVal 32 5] sq-param0 v}

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

```

1.4 Data-flow Tree Refinement

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

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

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

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

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

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

notation *less-eq* (**infix** \sqsubseteq 65)

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*]:
 $(e1 < e2) \longleftrightarrow (e1 \leq e2 \wedge \neg (e1 \doteq e2))$

```

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

end

abbreviation (output) Refines :: IRExpr  $\Rightarrow$  IRExpr  $\Rightarrow$  bool (infix  $\sqsupseteq$  64)
  where e1  $\sqsupseteq$  e2  $\equiv$  (e2  $\leq$  e1)

```

1.5 Stamp Masks

A stamp can contain additional range information in the form of masks. A stamp has an up mask and a down mask, corresponding to the bits that may be set and the bits that must be set.

Examples: A stamp where no range information is known will have; an up mask of -1 as all bits may be set, and a down mask of 0 as no bits must be set.

A stamp known to be one should have; an up mask of 1 as only the first bit may be set, no others, and a down mask of 1 as the first bit must be set and no others.

We currently don't carry mask information in stamps, and instead assume correct masks to prove optimizations.

```

locale stamp-mask =
  fixes up :: IRExpr  $\Rightarrow$  int64 ( $\uparrow$ )
  fixes down :: IRExpr  $\Rightarrow$  int64 ( $\downarrow$ )
  assumes up-spec: [m, p]  $\vdash$  e  $\mapsto$  IntVal b v  $\implies$  (and v (not ((ucast ( $\uparrow$ e)))))) = 0
    and down-spec: [m, p]  $\vdash$  e  $\mapsto$  IntVal b v  $\implies$  (and (not v) (ucast ( $\downarrow$ e))) = 0
begin

```

```

lemma may-implies-either:
  [m, p]  $\vdash$  e  $\mapsto$  IntVal b v  $\implies$  bit ( $\uparrow$ e) n  $\implies$  bit v n = False  $\vee$  bit v n = True
  by simp

```

```

lemma not-may-implies-false:
  [m, p]  $\vdash$  e  $\mapsto$  IntVal b v  $\implies$   $\neg$ (bit ( $\uparrow$ e) n)  $\implies$  bit v n = False
  using up-spec
  using bit-and-iff bit-eq-iff bit-not-iff bit-unsigned-iff down-spec
  by (smt (verit, best) bit.double-compl)

```

```

lemma must-implies-true:
  [m, p]  $\vdash$  e  $\mapsto$  IntVal b v  $\implies$  bit ( $\downarrow$ e) n  $\implies$  bit v n = True
  using down-spec
  by (metis bit.compl-one bit-and-iff bit-minus-1-iff bit-not-iff impossible-bit ucast-id)

```


lemma *not-must-implies-either*:

$[m, p] \vdash e \mapsto \text{IntVal } b \ v \implies \neg(\text{bit } (\downarrow e) \ n) \implies \text{bit } v \ n = \text{False} \vee \text{bit } v \ n = \text{True}$
by *simp*

lemma *must-implies-may*:

$[m, p] \vdash e \mapsto \text{IntVal } b \ v \implies n < 32 \implies \text{bit } (\downarrow e) \ n \implies \text{bit } (\uparrow e) \ n$
by (*meson must-implies-true not-may-implies-false*)

lemma *up-mask-and-zero-implies-zero*:

assumes *and* $(\uparrow x) (\uparrow y) = 0$
assumes $[m, p] \vdash x \mapsto \text{IntVal } b \ xv$
assumes $[m, p] \vdash y \mapsto \text{IntVal } b \ yv$
shows *and* $xv \ yv = 0$
using *assms*
by (*smt (z3) and.commute and.right-neutral and-zero-eq bit.compl-zero bit.conj-cancel-right bit.conj-disj-distrib(1) ucast-id up-spec word-bw-assocs(1) word-not-dist(2)*)

lemma *not-down-up-mask-and-zero-implies-zero*:

assumes *and* $(\text{not } (\downarrow x)) (\uparrow y) = 0$
assumes $[m, p] \vdash x \mapsto \text{IntVal } b \ xv$
assumes $[m, p] \vdash y \mapsto \text{IntVal } b \ yv$
shows *and* $xv \ yv = yv$
using *assms*
by (*smt (z3) and-zero-eq bit.conj-cancel-left bit.conj-disj-distrib(1) bit.conj-disj-distrib(2) bit.de-Morgan-disj down-spec or-eq-not-not-and ucast-id up-spec word-ao-absorbs(2) word-ao-absorbs(8) word-bw-lcs(1) word-not-dist(2)*)

end

end

2 Tree to Graph

theory *TreeToGraph*

imports

Semantics.IRTreeEval

Graph.IRGraph

begin

2.1 Subgraph to Data-flow Tree

fun *find-node-and-stamp* :: *IRGraph* \Rightarrow (*IRNode* \times *Stamp*) \Rightarrow *ID option* **where**

find-node-and-stamp *g* (*n,s*) =

find ($\lambda i. \text{kind } g \ i = n \wedge \text{stamp } g \ i = s$) (*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]]
  ⇒ g ⊢ n ≃ (UnaryExpr UnaryNeg xe) |

```

LogicNegationNode:

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

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

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

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

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

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

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

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

LeftShiftNode:
 $\llbracket \text{kind } g \ n = \text{LeftShiftNode } x \ y;$

$$\begin{aligned}
&g \vdash x \simeq xe; \\
&g \vdash y \simeq ye \\
\implies &g \vdash n \simeq (\text{BinaryExpr BinLeftShift } xe \ ye) \mid
\end{aligned}$$

RightShiftNode:

$$\begin{aligned}
&[[\text{kind } g \ n = \text{RightShiftNode } x \ y; \\
&\quad g \vdash x \simeq xe; \\
&\quad g \vdash y \simeq ye]] \\
\implies &g \vdash n \simeq (\text{BinaryExpr BinRightShift } xe \ ye) \mid
\end{aligned}$$

UnsignedRightShiftNode:

$$\begin{aligned}
&[[\text{kind } g \ n = \text{UnsignedRightShiftNode } x \ y; \\
&\quad g \vdash x \simeq xe; \\
&\quad g \vdash y \simeq ye]] \\
\implies &g \vdash n \simeq (\text{BinaryExpr BinURightShift } xe \ ye) \mid
\end{aligned}$$

IntegerBelowNode:

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

IntegerEqualsNode:

$$\begin{aligned}
&[[\text{kind } g \ n = \text{IntegerEqualsNode } x \ y; \\
&\quad g \vdash x \simeq xe; \\
&\quad g \vdash y \simeq ye]] \\
\implies &g \vdash n \simeq (\text{BinaryExpr BinIntegerEquals } xe \ ye) \mid
\end{aligned}$$

IntegerLessThanNode:

$$\begin{aligned}
&[[\text{kind } g \ n = \text{IntegerLessThanNode } x \ y; \\
&\quad g \vdash x \simeq xe; \\
&\quad g \vdash y \simeq ye]] \\
\implies &g \vdash n \simeq (\text{BinaryExpr BinIntegerLessThan } xe \ ye) \mid
\end{aligned}$$

NarrowNode:

$$\begin{aligned}
&[[\text{kind } g \ n = \text{NarrowNode } \text{inputBits } \text{resultBits } x; \\
&\quad g \vdash x \simeq xe]] \\
\implies &g \vdash n \simeq (\text{UnaryExpr } (\text{UnaryNarrow } \text{inputBits } \text{resultBits}) \ xe) \mid
\end{aligned}$$

SignExtendNode:

$$\begin{aligned}
&[[\text{kind } g \ n = \text{SignExtendNode } \text{inputBits } \text{resultBits } x; \\
&\quad g \vdash x \simeq xe]] \\
\implies &g \vdash n \simeq (\text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \ xe) \mid
\end{aligned}$$

ZeroExtendNode:

$$\begin{aligned}
&[[\text{kind } g \ n = \text{ZeroExtendNode } \text{inputBits } \text{resultBits } x; \\
&\quad g \vdash x \simeq xe]] \\
\implies &g \vdash n \simeq (\text{UnaryExpr } (\text{UnaryZeroExtend } \text{inputBits } \text{resultBits}) \ xe) \mid
\end{aligned}$$

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

RefNode:
 $\llbracket \text{kind } g \ n = \text{RefNode } n';$
 $g \vdash n' \simeq e \rrbracket$
 $\implies g \vdash n \simeq e$

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

inductive

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

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

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

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

definition *wf-term-graph* :: $\text{MapState} \Rightarrow \text{Params} \Rightarrow \text{IRGraph} \Rightarrow \text{ID} \Rightarrow \text{bool}$ **where**
 $\text{wf-term-graph } m \ p \ g \ n = (\exists \ e. (g \vdash n \simeq e) \wedge (\exists \ v. ([m, p] \vdash e \mapsto v)))$

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

2.2 Data-flow Tree to Subgraph

fun *unary-node* :: $\text{IRUnaryOp} \Rightarrow \text{ID} \Rightarrow \text{IRNode}$ **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  $\Rightarrow$  ID  $\Rightarrow$  ID  $\Rightarrow$  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 BinShortCircuitOr x y = ShortCircuitOrNode 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

```

```

inductive fresh-id :: IRGraph  $\Rightarrow$  ID  $\Rightarrow$  bool where
   $n \notin \text{ids } g \implies \text{fresh-id } g \ n$ 

```

```

code-pred fresh-id .

```

```

fun get-fresh-id :: IRGraph  $\Rightarrow$  ID where

```

```

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

```

```

export-code get-fresh-id

```

```

value get-fresh-id eg2-sq

```

```

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

```

```

inductive

```

```

  unrep :: IRGraph  $\Rightarrow$  IRExpr  $\Rightarrow$  (IRGraph  $\times$  ID)  $\Rightarrow$  bool (-  $\oplus$  -  $\rightsquigarrow$  - 55)
  where

```

```

  ConstantNodeSame:

```

```

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

```

```

  ConstantNodeNew:

```

```

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

```

```

  ParameterNodeSame:

```

```

   $\llbracket \text{find-node-and-stamp } g \text{ (ParameterNode } i, s) = \text{Some } n \rrbracket$ 

```

$$\implies g \oplus (\text{ParameterExpr } i \ s) \rightsquigarrow (g, n) \mid$$

ParameterNodeNew:

$$\begin{aligned} & \llbracket \text{find-node-and-stamp } g \ (\text{ParameterNode } i, s) = \text{None}; \\ & \quad n = \text{get-fresh-id } g; \\ & \quad g' = \text{add-node } n \ (\text{ParameterNode } i, s) \ g \rrbracket \\ & \implies g \oplus (\text{ParameterExpr } i \ s) \rightsquigarrow (g', n) \mid \end{aligned}$$

ConditionalNodeSame:

$$\begin{aligned} & \llbracket g \oplus ce \rightsquigarrow (g2, c); \\ & \quad g2 \oplus te \rightsquigarrow (g3, t); \\ & \quad g3 \oplus fe \rightsquigarrow (g4, f); \\ & \quad s' = \text{meet } (\text{stamp } g4 \ t) \ (\text{stamp } g4 \ f); \\ & \quad \text{find-node-and-stamp } g4 \ (\text{ConditionalNode } c \ t \ f, s') = \text{Some } n \rrbracket \\ & \implies g \oplus (\text{ConditionalExpr } ce \ te \ fe) \rightsquigarrow (g4, n) \mid \end{aligned}$$

ConditionalNodeNew:

$$\begin{aligned} & \llbracket g \oplus ce \rightsquigarrow (g2, c); \\ & \quad g2 \oplus te \rightsquigarrow (g3, t); \\ & \quad g3 \oplus fe \rightsquigarrow (g4, f); \\ & \quad s' = \text{meet } (\text{stamp } g4 \ t) \ (\text{stamp } g4 \ f); \\ & \quad \text{find-node-and-stamp } g4 \ (\text{ConditionalNode } c \ t \ f, s') = \text{None}; \\ & \quad n = \text{get-fresh-id } g4; \\ & \quad g' = \text{add-node } n \ (\text{ConditionalNode } c \ t \ f, s') \ g4 \rrbracket \\ & \implies g \oplus (\text{ConditionalExpr } ce \ te \ fe) \rightsquigarrow (g', n) \mid \end{aligned}$$

UnaryNodeSame:

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

UnaryNodeNew:

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

BinaryNodeSame:

$$\begin{aligned} & \llbracket g \oplus xe \rightsquigarrow (g2, x); \\ & \quad g2 \oplus ye \rightsquigarrow (g3, y); \\ & \quad s' = \text{stamp-binary } op \ (\text{stamp } g3 \ x) \ (\text{stamp } g3 \ y); \\ & \quad \text{find-node-and-stamp } g3 \ (\text{bin-node } op \ x \ y, s') = \text{Some } n \rrbracket \\ & \implies g \oplus (\text{BinaryExpr } op \ xe \ ye) \rightsquigarrow (g3, n) \mid \end{aligned}$$

BinaryNodeNew:

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

```

g2  $\oplus$  ye  $\rightsquigarrow$  (g3, y);
s' = stamp-binary op (stamp g3 x) (stamp g3 y);
find-node-and-stamp g3 (bin-node op x y, s') = None;
n = get-fresh-id g3;
g' = add-node n (bin-node op x y, s') g3
 $\implies g \oplus (\textit{BinaryExpr op xe ye}) \rightsquigarrow (g', n) \mid$ 

```

AllLeafNodes:

```

 $\llbracket$  stamp g n = s;
   is-preevaluated (kind g n)  $\rrbracket$ 
 $\implies g \oplus (\textit{LeafExpr n s}) \rightsquigarrow (g, n)$ 

```

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

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

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

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

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

$$\frac{\begin{array}{l} g \oplus ce \rightsquigarrow (g2, c) \quad g2 \oplus te \rightsquigarrow (g3, t) \\ g3 \oplus fe \rightsquigarrow (g4, f) \quad s' = \text{meet (stamp } g4 \text{ } t) \text{ (stamp } g4 \text{ } f) \\ \text{find-node-and-stamp } g4 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{Some } n \end{array}}{g \oplus \text{ConditionalExpr } ce \text{ } te \text{ } fe \rightsquigarrow (g4, n)}$$

$$\frac{\begin{array}{l} g \oplus ce \rightsquigarrow (g2, c) \quad g2 \oplus te \rightsquigarrow (g3, t) \\ g3 \oplus fe \rightsquigarrow (g4, f) \quad s' = \text{meet (stamp } g4 \text{ } t) \text{ (stamp } g4 \text{ } f) \\ \text{find-node-and-stamp } g4 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{None} \\ n = \text{get-fresh-id } g4 \quad g' = \text{add-node } n \text{ (ConditionalNode } c \text{ } t \text{ } f, s') \end{array}}{g \oplus \text{ConditionalExpr } ce \text{ } te \text{ } fe \rightsquigarrow (g', n)}$$

$$\frac{\begin{array}{l} g \oplus xe \rightsquigarrow (g2, x) \\ g2 \oplus ye \rightsquigarrow (g3, y) \quad s' = \text{stamp-binary op (stamp } g3 \text{ } x) \text{ (stamp } g3 \text{ } y) \\ \text{find-node-and-stamp } g3 \text{ (bin-node op } x \text{ } y, s') = \text{Some } n \end{array}}{g \oplus \text{BinaryExpr op } xe \text{ } ye \rightsquigarrow (g3, n)}$$

$$\frac{\begin{array}{l} g \oplus xe \rightsquigarrow (g2, x) \\ g2 \oplus ye \rightsquigarrow (g3, y) \quad s' = \text{stamp-binary op (stamp } g3 \text{ } x) \text{ (stamp } g3 \text{ } y) \\ \text{find-node-and-stamp } g3 \text{ (bin-node op } x \text{ } y, s') = \text{None} \\ n = \text{get-fresh-id } g3 \quad g' = \text{add-node } n \text{ (bin-node op } x \text{ } y, s') \end{array}}{g \oplus \text{BinaryExpr op } xe \text{ } ye \rightsquigarrow (g', n)}$$

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

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

$$\frac{\text{stamp } g \text{ } n = s \quad \text{is-preevaluated (kind } g \text{ } n)}{g \oplus \text{LeafExpr } n \text{ } s \rightsquigarrow (g, n)}$$

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

2.3 Lift Data-flow Tree Semantics

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

2.4 Graph Refinement

definition *graph-represents-expression* :: *IRGraph* \Rightarrow *ID* \Rightarrow *IRExpr* \Rightarrow *bool*
 $(\vdash \cdot \leq \cdot \ 50)$
where
 $(g \vdash n \leq e) = (\exists e'. (g \vdash n \simeq e') \wedge (e' \leq e))$

definition *graph-refinement* :: *IRGraph* \Rightarrow *IRGraph* \Rightarrow *bool* **where**
graph-refinement *g1 g2* =
 $((ids\ g_1 \subseteq ids\ g_2) \wedge$
 $(\forall n . n \in ids\ g_1 \longrightarrow (\forall e. (g_1 \vdash n \simeq e) \longrightarrow (g_2 \vdash n \leq e))))$

lemma *graph-refinement*:

graph-refinement *g1 g2* $\implies (\forall n\ m\ p\ v. n \in ids\ g1 \longrightarrow ([g1, m, p] \vdash n \mapsto v) \longrightarrow$
 $([g2, m, p] \vdash n \mapsto v))$
by (*meson encodeeval-def graph-refinement-def graph-represents-expression-def le-expr-def*)

2.5 Maximal Sharing

definition *maximal-sharing*:

maximal-sharing *g* = $(\forall n_1\ n_2 . n_1 \in true\text{-}ids\ g \wedge n_2 \in true\text{-}ids\ g \longrightarrow$
 $(\forall e. (g \vdash n_1 \simeq e) \wedge (g \vdash n_2 \simeq e) \wedge (stamp\ g\ n_1 = stamp\ g\ n_2) \longrightarrow n_1 =$
 $n_2))$

end

2.6 Formedness Properties

theory *Form*

imports

Semantics.TreeToGraph

begin

definition *wf-start* **where**

wf-start *g* = $(0 \in ids\ g \wedge$
 $is\text{-}StartNode\ (kind\ g\ 0))$

definition *wf-closed* **where**

wf-closed *g* =
 $(\forall n \in ids\ g .$

$$\begin{aligned} &inputs\ g\ n \subseteq ids\ g \wedge \\ &succ\ g\ n \subseteq ids\ g \wedge \\ &kind\ g\ n \neq NoNode) \end{aligned}$$

definition *wf-phs* **where**

$$\begin{aligned} wf-phs\ g = & \\ &(\forall\ n \in ids\ g. \\ &\quad is-PhiNode\ (kind\ g\ n) \longrightarrow \\ &\quad length\ (ir-values\ (kind\ g\ n)) \\ &= length\ (ir-ends \\ &\quad (kind\ g\ (ir-merge\ (kind\ g\ n)))))) \end{aligned}$$

definition *wf-ends* **where**

$$\begin{aligned} wf-ends\ g = & \\ &(\forall\ n \in ids\ g . \\ &\quad is-AbstractEndNode\ (kind\ g\ n) \longrightarrow \\ &\quad card\ (usages\ g\ n) > 0) \end{aligned}$$

fun *wf-graph* :: *IRGraph* \Rightarrow *bool* **where**

$$wf-graph\ g = (wf-start\ g \wedge wf-closed\ g \wedge wf-phs\ g \wedge wf-ends\ g)$$

lemmas *wf-folds* =

$$\begin{aligned} &wf-graph.simps \\ &wf-start-def \\ &wf-closed-def \\ &wf-phs-def \\ &wf-ends-def \end{aligned}$$

fun *wf-stamps* :: *IRGraph* \Rightarrow *bool* **where**

$$\begin{aligned} wf-stamps\ g = &(\forall\ n \in ids\ g . \\ &(\forall\ v\ m\ p\ e . (g \vdash n \simeq e) \wedge ([m, p] \vdash e \mapsto v) \longrightarrow valid-value\ v\ (stamp-expr\ e))) \end{aligned}$$

fun *wf-stamp* :: *IRGraph* \Rightarrow (*ID* \Rightarrow *Stamp*) \Rightarrow *bool* **where**

$$\begin{aligned} wf-stamp\ g\ s = &(\forall\ n \in ids\ g . \\ &(\forall\ v\ m\ p\ e . (g \vdash n \simeq e) \wedge ([m, p] \vdash e \mapsto v) \longrightarrow valid-value\ v\ (s\ n))) \end{aligned}$$

lemma *wf-empty*: *wf-graph start-end-graph*

unfolding *start-end-graph-def wf-folds by simp*

lemma *wf-eg2-sq*: *wf-graph eg2-sq*

unfolding *eg2-sq-def wf-folds by simp*

fun *wf-logic-node-inputs* :: *IRGraph* \Rightarrow *ID* \Rightarrow *bool* **where**

$$\begin{aligned} wf-logic-node-inputs\ g\ n = & \\ &(\forall\ inp \in set\ (inputs-of\ (kind\ g\ n)) . (\forall\ v\ m\ p . ([g, m, p] \vdash inp \mapsto v) \longrightarrow wf-bool \\ &v)) \end{aligned}$$

fun *wf-values* :: *IRGraph* \Rightarrow *bool* **where**

$$wf-values\ g = (\forall\ n \in ids\ g .$$

$$(\forall v m p . ([g, m, p] \vdash n \mapsto v) \longrightarrow \\ (is-LogicNode (kind g n) \longrightarrow \\ wf-bool v \wedge wf-logic-node-inputs g n)))$$

end

2.7 Dynamic Frames

This theory defines two operators, 'unchanged' and 'changeonly', that are useful for specifying which nodes in an IRGraph can change. The dynamic framing idea originates from 'Dynamic Frames' in software verification, started by Ioannis T. Kassios in "Dynamic frames: Support for framing, dependencies and sharing without restrictions", In FM 2006.

theory *IRGraphFrames*

imports

Form

begin

fun *unchanged* :: *ID set* \Rightarrow *IRGraph* \Rightarrow *IRGraph* \Rightarrow *bool* **where**

unchanged ns g1 g2 = $(\forall n . n \in ns \longrightarrow$
 $(n \in ids\ g1 \wedge n \in ids\ g2 \wedge kind\ g1\ n = kind\ g2\ n \wedge stamp\ g1\ n = stamp\ g2\ n))$

fun *changeonly* :: *ID set* \Rightarrow *IRGraph* \Rightarrow *IRGraph* \Rightarrow *bool* **where**

changeonly ns g1 g2 = $(\forall n . n \in ids\ g1 \wedge n \notin ns \longrightarrow$
 $(n \in ids\ g1 \wedge n \in ids\ g2 \wedge kind\ g1\ n = kind\ g2\ n \wedge stamp\ g1\ n = stamp\ g2\ n))$

lemma *node-unchanged:*

assumes *unchanged ns g1 g2*

assumes *nid* \in *ns*

shows *kind g1 nid* = *kind g2 nid*

using *assms* **by** *auto*

lemma *other-node-unchanged:*

assumes *changeonly ns g1 g2*

assumes *nid* \in *ids g1*

assumes *nid* \notin *ns*

shows *kind g1 nid* = *kind g2 nid*

using *assms*

using *changeonly.simps* **by** *blast*

Some notation for input nodes used

inductive *eval-uses*:: *IRGraph* \Rightarrow *ID* \Rightarrow *ID* \Rightarrow *bool*

for *g* **where**

use0: *nid* \in *ids g*

```

     $\implies \text{eval-uses } g \text{ nid nid} \mid$ 

    use-inp:  $\text{nid}' \in \text{inputs } g \text{ n}$ 
     $\implies \text{eval-uses } g \text{ nid nid}' \mid$ 

    use-trans:  $\llbracket \text{eval-uses } g \text{ nid nid}';$ 
                $\text{eval-uses } g \text{ nid}' \text{ nid}'' \rrbracket$ 
     $\implies \text{eval-uses } g \text{ nid nid}''$ 

fun eval-usages :: IRGraph  $\Rightarrow$  ID  $\Rightarrow$  ID set where
    eval-usages g nid = {n  $\in$  ids g . eval-uses g nid n}

lemma eval-usages-self:
    assumes nid  $\in$  ids g
    shows nid  $\in$  eval-usages g nid
    using assms eval-usages.simps eval-uses.intros(1)
    by (simp add: ids.rep-eq)

lemma not-in-g-inputs:
    assumes nid  $\notin$  ids g
    shows inputs g nid = {}
proof –
    have k: kind g nid = NoNode using assms not-in-g by blast
    then show ?thesis by (simp add: k)
qed

lemma child-member:
    assumes n = kind g nid
    assumes n  $\neq$  NoNode
    assumes List.member (inputs-of n) child
    shows child  $\in$  inputs g nid
    unfolding inputs.simps using assms
    by (metis in-set-member)

lemma child-member-in:
    assumes nid  $\in$  ids g
    assumes List.member (inputs-of (kind g nid)) child
    shows child  $\in$  inputs g nid
    unfolding inputs.simps using assms
    by (metis child-member ids-some inputs.elims)

lemma inp-in-g:
    assumes n  $\in$  inputs g nid
    shows nid  $\in$  ids g
proof –
    have inputs g nid  $\neq$  {}

```

```

    using assms
    by (metis empty-iff empty-set)
  then have kind g nid  $\neq$  NoNode
    using not-in-g-inputs
    using ids-some by blast
  then show ?thesis
    using not-in-g
    by metis
qed

```

```

lemma inp-in-g-wf:
  assumes wf-graph g
  assumes  $n \in \text{inputs } g \text{ nid}$ 
  shows  $n \in \text{ids } g$ 
  using assms unfolding wf-folds
  using inp-in-g by blast

```

```

lemma kind-unchanged:
  assumes  $\text{nid} \in \text{ids } g1$ 
  assumes unchanged (eval-usages g1 nid) g1 g2
  shows  $\text{kind } g1 \text{ nid} = \text{kind } g2 \text{ nid}$ 
proof -
  show ?thesis
    using assms eval-usages-self
    using unchanged.simps by blast
qed

```

```

lemma stamp-unchanged:
  assumes  $\text{nid} \in \text{ids } g1$ 
  assumes unchanged (eval-usages g1 nid) g1 g2
  shows  $\text{stamp } g1 \text{ nid} = \text{stamp } g2 \text{ nid}$ 
  by (meson assms(1) assms(2) eval-usages-self unchanged.elims(2))

```

```

lemma child-unchanged:
  assumes  $\text{child} \in \text{inputs } g1 \text{ nid}$ 
  assumes unchanged (eval-usages g1 nid) g1 g2
  shows unchanged (eval-usages g1 child) g1 g2
  by (smt assms(1) assms(2) eval-usages.simps mem-Collect-eq
    unchanged.simps use-inp use-trans)

```

```

lemma eval-usages:
  assumes  $us = \text{eval-usages } g \text{ nid}$ 
  assumes  $\text{nid}' \in \text{ids } g$ 
  shows  $\text{eval-uses } g \text{ nid } \text{nid}' \longleftrightarrow \text{nid}' \in us$  (is ?P  $\longleftrightarrow$  ?Q)
  using assms eval-usages.simps
  by (simp add: ids.rep-eq)

```

lemma *inputs-are-uses*:
 assumes $nid' \in \text{inputs } g \text{ } nid$
 shows $\text{eval-uses } g \text{ } nid \text{ } nid'$
 by (metis *assms use-inp*)

lemma *inputs-are-usages*:
 assumes $nid' \in \text{inputs } g \text{ } nid$
 assumes $nid' \in \text{ids } g$
 shows $nid' \in \text{eval-usages } g \text{ } nid$
 using *assms(1) assms(2) eval-usages inputs-are-uses* by blast

lemma *inputs-of-are-usages*:
 assumes $\text{List.member } (\text{inputs-of } (\text{kind } g \text{ } nid)) \text{ } nid'$
 assumes $nid' \in \text{ids } g$
 shows $nid' \in \text{eval-usages } g \text{ } nid$
 by (metis *assms(1) assms(2) in-set-member inputs.elims inputs-are-usages*)

lemma *usage-includes-inputs*:
 assumes $us = \text{eval-usages } g \text{ } nid$
 assumes $ls = \text{inputs } g \text{ } nid$
 assumes $ls \subseteq \text{ids } g$
 shows $ls \subseteq us$
 using *inputs-are-usages eval-usages*
 using *assms(1) assms(2) assms(3)* by blast

lemma *elim-inp-set*:
 assumes $k = \text{kind } g \text{ } nid$
 assumes $k \neq \text{NoNode}$
 assumes $\text{child} \in \text{set } (\text{inputs-of } k)$
 shows $\text{child} \in \text{inputs } g \text{ } nid$
 using *assms* by auto

lemma *encode-in-ids*:
 assumes $g \vdash nid \simeq e$
 shows $nid \in \text{ids } g$
 using *assms*
 apply (induction rule: *rep.induct*)
 apply *simp+*
 by *fastforce+*

lemma *eval-in-ids*:
 assumes $[g, m, p] \vdash nid \mapsto v$
 shows $nid \in \text{ids } g$
 using *assms* using *encodeeval-def encode-in-ids*
 by auto

lemma *transitive-kind-same*:
 assumes *unchanged* (*eval-usages* $g1 \text{ } nid$) $g1 \text{ } g2$
 shows $\forall \text{ } nid' \in (\text{eval-usages } g1 \text{ } nid) . \text{kind } g1 \text{ } nid' = \text{kind } g2 \text{ } nid'$

```

using assms
by (meson unchanged.elims(1))

theorem stay-same-encoding:
  assumes nc: unchanged (eval-usages g1 nid) g1 g2
  assumes g1: g1  $\vdash$  nid  $\simeq$  e
  assumes wf: wf-graph g1
  shows g2  $\vdash$  nid  $\simeq$  e
proof –
  have dom: nid  $\in$  ids g1
    using g1 encode-in-ids by simp
  show ?thesis
using g1 nc wf dom proof (induction e rule: rep.induct)
  case (ConstantNode n c)
    then have kind g2 n = ConstantNode c
      using dom nc kind-unchanged
      by metis
    then show ?case using rep.ConstantNode
      by presburger
  next
    case (ParameterNode n i s)
    then have kind g2 n = ParameterNode i
      by (metis kind-unchanged)
    then show ?case
      by (metis ParameterNode.hyps(2) ParameterNode.prems(1) ParameterNode.prems(3)
rep.ParameterNode stamp-unchanged)
  next
    case (ConditionalNode n c t f ce te fe)
    then have kind g2 n = ConditionalNode c t f
      by (metis kind-unchanged)
    have c  $\in$  eval-usages g1 n  $\wedge$  t  $\in$  eval-usages g1 n  $\wedge$  f  $\in$  eval-usages g1 n
      using inputs-of-ConditionalNode
      by (metis ConditionalNode.hyps(1) ConditionalNode.hyps(2) ConditionalNode.hyps(3) ConditionalNode.hyps(4) encode-in-ids inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons subset-code(1))
    then show ?case using transitive-kind-same
      by (metis ConditionalNode.hyps(1) ConditionalNode.prems(1) IRNodes.inputs-of-ConditionalNode
 $\langle$ kind g2 n = ConditionalNode c t f $\rangle$  child-unchanged inputs.simps list.set-intros(1)
local.ConditionalNode(5) local.ConditionalNode(6) local.ConditionalNode(7) local.ConditionalNode(9)
rep.ConditionalNode set-subset-Cons subset-code(1) unchanged.elims(2))
  next
    case (AbsNode n x xe)
    then have kind g2 n = AbsNode x
      using kind-unchanged
      by metis
    then have x  $\in$  eval-usages g1 n
      using inputs-of-AbsNode
      by (metis AbsNode.hyps(1) AbsNode.hyps(2) encode-in-ids inputs.simps inputs-are-usages list.set-intros(1))

```



```

then show ?case
  by (metis AbsNode.IH AbsNode.hyps(1) AbsNode.prem(1) AbsNode.prem(3)
    IRNodes.inputs-of-AbsNode ⟨kind g2 n = AbsNode x⟩ child-member-in child-unchanged
    local.wf member-rec(1) rep.AbsNode unchanged.simps)
next
  case (NotNode n x xe)
  then have kind g2 n = NotNode x
  using kind-unchanged
  by metis
  then have x ∈ eval-usages g1 n
  using inputs-of-NotNode
  by (metis NotNode.hyps(1) NotNode.hyps(2) encode-in-ids inputs.simps in-
    puts-are-usages list.set-intros(1))
  then show ?case
  by (metis NotNode.IH NotNode.hyps(1) NotNode.prem(1) NotNode.prem(3)
    IRNodes.inputs-of-NotNode ⟨kind g2 n = NotNode x⟩ child-member-in child-unchanged
    local.wf member-rec(1) rep.NotNode unchanged.simps)
next
  case (NegateNode n x xe)
  then have kind g2 n = NegateNode x
  using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
  using inputs-of-NegateNode
  by (metis NegateNode.hyps(1) NegateNode.hyps(2) encode-in-ids inputs.simps
    inputs-are-usages list.set-intros(1))
  then show ?case
  by (metis IRNodes.inputs-of-NegateNode NegateNode.IH NegateNode.hyps(1)
    NegateNode.prem(1) NegateNode.prem(3) ⟨kind g2 n = NegateNode x⟩ child-member-in
    child-unchanged local.wf member-rec(1) rep.NegateNode unchanged.elims(1))
next
  case (LogicNegationNode n x xe)
  then have kind g2 n = LogicNegationNode x
  using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
  using inputs-of-LogicNegationNode inputs-of-are-usages
  by (metis LogicNegationNode.hyps(1) LogicNegationNode.hyps(2) encode-in-ids
    member-rec(1))
  then show ?case
  by (metis IRNodes.inputs-of-LogicNegationNode LogicNegationNode.IH Log-
    icNegationNode.hyps(1) LogicNegationNode.hyps(2) LogicNegationNode.prem(1)
    ⟨kind g2 n = LogicNegationNode x⟩ child-unchanged encode-in-ids inputs.simps
    list.set-intros(1) local.wf rep.LogicNegationNode)
next
  case (AddNode n x y xe ye)
  then have kind g2 n = AddNode x y
  using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
  using inputs-of-LogicNegationNode inputs-of-are-usages
  by (metis AddNode.hyps(1) AddNode.hyps(2) AddNode.hyps(3) IRNodes.inputs-of-AddNode

```

```

encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case
    by (metis AddNode.IH(1) AddNode.IH(2) AddNode.hyps(1) AddNode.hyps(2)
      AddNode.hyps(3) AddNode.premis(1) IRNodes.inputs-of-AddNode ⟨kind g2 n = AddNode
        x y⟩ child-unchanged encode-in-ids in-set-member inputs.simps local.wf member-rec(1)
        rep.AddNode)
  next
    case (MulNode n x y xe ye)
    then have kind g2 n = MulNode x y
      using kind-unchanged by metis
    then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
      using inputs-of-LogicNegationNode inputs-of-are-usages
    by (metis MulNode.hyps(1) MulNode.hyps(2) MulNode.hyps(3) IRNodes.inputs-of-MulNode
      encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
    then show ?case using MulNode inputs-of-MulNode
      by (metis ⟨kind g2 n = MulNode x y⟩ child-unchanged inputs.simps list.set-intros(1)
        rep.MulNode set-subset-Cons subset-iff unchanged.elims(2))
  next
    case (SubNode n x y xe ye)
    then have kind g2 n = SubNode x y
      using kind-unchanged by metis
    then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
      using inputs-of-LogicNegationNode inputs-of-are-usages
    by (metis SubNode.hyps(1) SubNode.hyps(2) SubNode.hyps(3) IRNodes.inputs-of-SubNode
      encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
    then show ?case using SubNode inputs-of-SubNode
      by (metis ⟨kind g2 n = SubNode x y⟩ child-member child-unchanged encode-in-ids
        ids-some member-rec(1) rep.SubNode)
  next
    case (AndNode n x y xe ye)
    then have kind g2 n = AndNode x y
      using kind-unchanged by metis
    then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
      using inputs-of-LogicNegationNode inputs-of-are-usages
    by (metis AndNode.hyps(1) AndNode.hyps(2) AndNode.hyps(3) IRNodes.inputs-of-AndNode
      encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
    then show ?case using AndNode inputs-of-AndNode
      by (metis ⟨kind g2 n = AndNode x y⟩ child-unchanged inputs.simps list.set-intros(1)
        rep.AndNode set-subset-Cons subset-iff unchanged.elims(2))
  next
    case (OrNode n x y xe ye)
    then have kind g2 n = OrNode x y
      using kind-unchanged by metis
    then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
      using inputs-of-OrNode inputs-of-are-usages
    by (metis OrNode.hyps(1) OrNode.hyps(2) OrNode.hyps(3) IRNodes.inputs-of-OrNode
      encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
    then show ?case using OrNode inputs-of-OrNode
      by (metis ⟨kind g2 n = OrNode x y⟩ child-member child-unchanged encode-in-ids

```

```

ids-some member-rec(1) rep.OrNode)
next
case (XorNode n x y xe ye)
then have kind g2 n = XorNode x y
using kind-unchanged by metis
then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
using inputs-of-XorNode inputs-of-are-usages
by (metis XorNode.hyps(1) XorNode.hyps(2) XorNode.hyps(3) IRNodes.inputs-of-XorNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
then show ?case using XorNode inputs-of-XorNode
by (metis ⟨kind g2 n = XorNode x y⟩ child-member child-unchanged en-
code-in-ids ids-some member-rec(1) rep.XorNode)
next
case (ShortCircuitOrNode n x y xe ye)
then have kind g2 n = ShortCircuitOrNode x y
using kind-unchanged by metis
then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
using inputs-of-XorNode inputs-of-are-usages
by (metis ShortCircuitOrNode.hyps(1) ShortCircuitOrNode.hyps(2) ShortCir-
cuitOrNode.hyps(3) IRNodes.inputs-of-ShortCircuitOrNode encode-in-ids in-mono
inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
then show ?case using ShortCircuitOrNode inputs-of-ShortCircuitOrNode
by (metis ⟨kind g2 n = ShortCircuitOrNode x y⟩ child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.ShortCircuitOrNode)
next
case (LeftShiftNode n x y xe ye)
then have kind g2 n = LeftShiftNode x y
using kind-unchanged by metis
then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
using inputs-of-XorNode inputs-of-are-usages
by (metis LeftShiftNode.hyps(1) LeftShiftNode.hyps(2) LeftShiftNode.hyps(3)
IRNodes.inputs-of-LeftShiftNode encode-in-ids in-mono inputs.simps inputs-are-usages
list.set-intros(1) set-subset-Cons)
then show ?case using LeftShiftNode inputs-of-LeftShiftNode
by (metis ⟨kind g2 n = LeftShiftNode x y⟩ child-member child-unchanged en-
code-in-ids ids-some member-rec(1) rep.LeftShiftNode)
next
case (RightShiftNode n x y xe ye)
then have kind g2 n = RightShiftNode x y
using kind-unchanged by metis
then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
using inputs-of-RightShiftNode inputs-of-are-usages
by (metis RightShiftNode.hyps(1) RightShiftNode.hyps(2) RightShiftNode.hyps(3)
IRNodes.inputs-of-RightShiftNode encode-in-ids in-mono inputs.simps inputs-are-usages
list.set-intros(1) set-subset-Cons)
then show ?case using RightShiftNode inputs-of-RightShiftNode
by (metis ⟨kind g2 n = RightShiftNode x y⟩ child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.RightShiftNode)
next

```

```

case (UnsignedRightShiftNode n x y xe ye)
  then have kind g2 n = UnsignedRightShiftNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-UnsignedRightShiftNode inputs-of-are-usages
    by (metis UnsignedRightShiftNode.hyps(1) UnsignedRightShiftNode.hyps(2) Un-
signedRightShiftNode.hyps(3) IRNodes.inputs-of-UnsignedRightShiftNode encode-in-ids
in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using UnsignedRightShiftNode inputs-of-UnsignedRightShiftNode
    by (metis ⟨kind g2 n = UnsignedRightShiftNode x y⟩ child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.UnsignedRightShiftNode)
next
  case (IntegerBelowNode n x y xe ye)
  then have kind g2 n = IntegerBelowNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-IntegerBelowNode inputs-of-are-usages
    by (metis IntegerBelowNode.hyps(1) IntegerBelowNode.hyps(2) IntegerBelowN-
ode.hyps(3) IRNodes.inputs-of-IntegerBelowNode encode-in-ids in-mono inputs.simps
inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using IntegerBelowNode inputs-of-IntegerBelowNode
    by (metis ⟨kind g2 n = IntegerBelowNode x y⟩ child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.IntegerBelowNode)
next
  case (IntegerEqualsNode n x y xe ye)
  then have kind g2 n = IntegerEqualsNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-IntegerEqualsNode inputs-of-are-usages
    by (metis IntegerEqualsNode.hyps(1) IntegerEqualsNode.hyps(2) IntegerEqual-
sNode.hyps(3) IRNodes.inputs-of-IntegerEqualsNode encode-in-ids in-mono inputs.simps
inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using IntegerEqualsNode inputs-of-IntegerEqualsNode
    by (metis ⟨kind g2 n = IntegerEqualsNode x y⟩ child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.IntegerEqualsNode)
next
  case (IntegerLessThanNode n x y xe ye)
  then have kind g2 n = IntegerLessThanNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-IntegerLessThanNode inputs-of-are-usages
    by (metis IntegerLessThanNode.hyps(1) IntegerLessThanNode.hyps(2) Inte-
gerLessThanNode.hyps(3) IRNodes.inputs-of-IntegerLessThanNode encode-in-ids in-mono
inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using IntegerLessThanNode inputs-of-IntegerLessThanNode
    by (metis ⟨kind g2 n = IntegerLessThanNode x y⟩ child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.IntegerLessThanNode)
next
  case (NarrowNode n ib rb x xe)

```

```

then have kind g2 n = NarrowNode ib rb x
  using kind-unchanged by metis
then have x ∈ eval-usages g1 n
  using inputs-of-NarrowNode inputs-of-are-usages
  by (metis NarrowNode.hyps(1) NarrowNode.hyps(2) IRNodes.inputs-of-NarrowNode
    encode-in-ids inputs.simps inputs-are-usages list.set-intros(1))
  then show ?case using NarrowNode inputs-of-NarrowNode
    by (metis ⟨kind g2 n = NarrowNode ib rb x⟩ child-unchanged inputs.elims
      list.set-intros(1) rep.NarrowNode unchanged.simps)
next
  case (SignExtendNode n ib rb x xe)
  then have kind g2 n = SignExtendNode ib rb x
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-SignExtendNode inputs-of-are-usages
    by (metis SignExtendNode.hyps(1) SignExtendNode.hyps(2) encode-in-ids in-
      puts.simps inputs-are-usages list.set-intros(1))
    then show ?case using SignExtendNode inputs-of-SignExtendNode
      by (metis ⟨kind g2 n = SignExtendNode ib rb x⟩ child-member-in child-unchanged
        in-set-member list.set-intros(1) rep.SignExtendNode unchanged.elims(2))
  next
    case (ZeroExtendNode n ib rb x xe)
    then have kind g2 n = ZeroExtendNode ib rb x
      using kind-unchanged by metis
    then have x ∈ eval-usages g1 n
      using inputs-of-ZeroExtendNode inputs-of-are-usages
      by (metis ZeroExtendNode.hyps(1) ZeroExtendNode.hyps(2) IRNodes.inputs-of-ZeroExtendNode
        encode-in-ids inputs.simps inputs-are-usages list.set-intros(1))
      then show ?case using ZeroExtendNode inputs-of-ZeroExtendNode
        by (metis ⟨kind g2 n = ZeroExtendNode ib rb x⟩ child-member-in child-unchanged
          member-rec(1) rep.ZeroExtendNode unchanged.simps)
    next
      case (LeafNode n s)
      then show ?case
        by (metis kind-unchanged rep.LeafNode stamp-unchanged)
    next
      case (RefNode n n')
      then have kind g2 n = RefNode n'
        using kind-unchanged by metis
      then have n' ∈ eval-usages g1 n
        by (metis IRNodes.inputs-of-RefNode RefNode.hyps(1) RefNode.hyps(2) en-
          code-in-ids inputs.elims inputs-are-usages list.set-intros(1))
      then show ?case
        by (metis IRNodes.inputs-of-RefNode RefNode.IH RefNode.hyps(1) RefNode.hyps(2)
          RefNode.premis(1) ⟨kind g2 n = RefNode n'⟩ child-unchanged encode-in-ids in-
          puts.elims list.set-intros(1) local.wf rep.RefNode)
qed
qed

```

```

theorem stay-same:
  assumes nc: unchanged (eval-usages g1 nid) g1 g2
  assumes g1: [g1, m, p] ⊢ nid ↦ v1
  assumes wf: wf-graph g1
  shows [g2, m, p] ⊢ nid ↦ v1
proof –
  have nid: nid ∈ ids g1
    using g1 eval-in-ids by simp
  then have nid ∈ eval-usages g1 nid
    using eval-usages-self by blast
  then have kind-same: kind g1 nid = kind g2 nid
    using nc node-unchanged by blast
  obtain e where e: (g1 ⊢ nid ≃ e) ∧ ([m,p] ⊢ e ↦ v1)
    using encodeeval-def g1
    by auto
  then have val: [m,p] ⊢ e ↦ v1
    using g1 encodeeval-def
    by simp
  then show ?thesis using e nid nc
    unfolding encodeeval-def
  proof (induct e v1 arbitrary: nid rule: evaltree.induct)
    case (ConstantExpr c)
      then show ?case
        by (meson local.wf stay-same-encoding)
    next
      case (ParameterExpr i s)
        have g2 ⊢ nid ≃ ParameterExpr i s
          using stay-same-encoding ParameterExpr
          by (meson local.wf)
        then show ?case using evaltree.ParameterExpr
          by (meson ParameterExpr.hyps)
    next
      case (ConditionalExpr ce cond branch te fe v)
        then have g2 ⊢ nid ≃ ConditionalExpr ce te fe
          using ConditionalExpr.prems(1) ConditionalExpr.prems(3) local.wf stay-same-encoding
          by presburger
        then show ?case
          by (meson ConditionalExpr.prems(1) ConditionalExpr.prems(3) local.wf
stay-same-encoding)
    next
      case (UnaryExpr xe v op)
        then show ?case
          using local.wf stay-same-encoding by blast
    next
      case (BinaryExpr xe x ye y op)
        then show ?case
          using local.wf stay-same-encoding by blast

```

```

next
  case (LeafExpr val nid s)
  then show ?case
    by (metis local.wf stay-same-encoding)
qed
qed

lemma add-changed:
  assumes gup = add-node new k g
  shows changeonly {new} g gup
  using assms unfolding add-node-def changeonly.simps
  using add-node.rep-eq add-node-def kind.rep-eq stamp.rep-eq by simp

lemma disjoint-change:
  assumes changeonly change g gup
  assumes nochange = ids g - change
  shows unchanged nochange g gup
  using assms unfolding changeonly.simps unchanged.simps
  by blast

lemma add-node-unchanged:
  assumes new  $\notin$  ids g
  assumes nid  $\in$  ids g
  assumes gup = add-node new k g
  assumes wf-graph g
  shows unchanged (eval-usages g nid) g gup
proof -
  have new  $\notin$  (eval-usages g nid) using assms
    using eval-usages.simps by blast
  then have changeonly {new} g gup
    using assms add-changed by blast
  then show ?thesis using assms add-node-def disjoint-change
    using Diff-insert-absorb by auto
qed

lemma eval-uses-imp:
  ((nid'  $\in$  ids g  $\wedge$  nid = nid')
   $\vee$  nid'  $\in$  inputs g nid
   $\vee$  ( $\exists$  nid'' . eval-uses g nid nid''  $\wedge$  eval-uses g nid'' nid'))
 $\longleftrightarrow$  eval-uses g nid nid'
  using use0 use-inp use-trans
  by (meson eval-uses.simps)

lemma wf-use-ids:
  assumes wf-graph g
  assumes nid  $\in$  ids g
  assumes eval-uses g nid nid'
  shows nid'  $\in$  ids g

```

```

    using assms(3)
  proof (induction rule: eval-uses.induct)
    case use0
    then show ?case by simp
  next
    case use-inp
    then show ?case
      using assms(1) inp-in-g-wf by blast
  next
    case use-trans
    then show ?case by blast
  qed

lemma no-external-use:
  assumes wf-graph g
  assumes nid'  $\notin$  ids g
  assumes nid  $\in$  ids g
  shows  $\neg$ (eval-uses g nid nid')
proof -
  have 0: nid  $\neq$  nid'
  using assms by blast
  have inp: nid'  $\notin$  inputs g nid
  using assms
  using inp-in-g-wf by blast
  have rec-0:  $\nexists n . n \in \text{ids } g \wedge n = \text{nid}'$ 
  using assms by blast
  have rec-inp:  $\nexists n . n \in \text{ids } g \wedge n \in \text{inputs } g \text{ nid}'$ 
  using assms(2) inp-in-g by blast
  have rec:  $\nexists \text{nid}'' . \text{eval-uses } g \text{ nid nid}'' \wedge \text{eval-uses } g \text{ nid}'' \text{ nid}'$ 
  using wf-use-ids assms(1) assms(2) assms(3) by blast
  from 0 rec show ?thesis
  using eval-uses-imp by blast
qed

end

```

3 Control-flow Semantics

```

theory IRStepObj
  imports
    TreeToGraph
begin

```

3.1 Object 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 refer-

ences 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 = ((λ f. λ p. UndefVal), 0)

3.2 Intraprocedural Semantics

```

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

fun phi-list :: IRGraph  $\Rightarrow$  ID  $\Rightarrow$  ID list where
  phi-list g n =
    (filter ( $\lambda$ x.(is-PhiNode (kind g x)))
     (sorted-list-of-set (usages g n)))

fun input-index :: IRGraph  $\Rightarrow$  ID  $\Rightarrow$  ID  $\Rightarrow$  nat where
  input-index g n n' = find-index n' (inputs-of (kind g n))

fun phi-inputs :: IRGraph  $\Rightarrow$  nat  $\Rightarrow$  ID list  $\Rightarrow$  ID list where
  phi-inputs g i nodes = (map ( $\lambda$ n. (inputs-of (kind g n))!(i + 1)) nodes)

fun set-phis :: ID list  $\Rightarrow$  Value list  $\Rightarrow$  MapState  $\Rightarrow$  MapState where
  set-phis [] [] m = m |
  set-phis (n # xs) (v # vs) m = (set-phis xs vs (m(n := v))) |
  set-phis [] (v # vs) m = m |
  set-phis (x # xs) [] m = m

```

Intraprocedural semantics are given as a small-step semantics.

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

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

SequentialNode:

$\llbracket is_sequential_node \ (kind \ g \ nid);$
 $\quad nid' = (successors_of \ (kind \ g \ nid))!0 \rrbracket$
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m, h) \mid$

IfNode:

$\llbracket kind \ g \ nid = (IfNode \ cond \ tb \ fb);$
 $\quad g \vdash cond \simeq condE;$
 $\quad [m, p] \vdash condE \mapsto val;$
 $\quad nid' = (if \ val_to_bool \ val \ then \ tb \ else \ fb) \rrbracket$
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m, h) \mid$

EndNodes:

$\llbracket is_AbstractEndNode \ (kind \ g \ nid);$
 $\quad merge = any_usage \ g \ nid;$
 $\quad is_AbstractMergeNode \ (kind \ g \ merge);$

 $\quad i = find_index \ nid \ (inputs_of \ (kind \ g \ merge));$
 $\quad phis = (phi_list \ g \ merge);$
 $\quad inps = (phi_inputs \ g \ i \ phis);$
 $\quad g \vdash inps \simeq_L inpsE;$
 $\quad [m, p] \vdash inpsE \mapsto_L vs;$

 $\quad m' = set_phis \ phis \ vs \ m \rrbracket$
 $\implies g, p \vdash (nid, m, h) \rightarrow (merge, m', h) \mid$

NewInstanceNode:

$\llbracket kind \ g \ nid = (NewInstanceNode \ nid \ f \ obj \ nid');$
 $\quad (h', ref) = h_new_inst \ h;$
 $\quad m' = m(nid := ref) \rrbracket$
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m', h') \mid$

LoadFieldNode:

$\llbracket kind \ g \ nid = (LoadFieldNode \ nid \ f \ (Some \ obj) \ nid');$
 $\quad g \vdash obj \simeq objE;$
 $\quad [m, p] \vdash objE \mapsto ObjRef \ ref;$
 $\quad h_load_field \ f \ ref \ h = v;$
 $\quad m' = m(nid := v) \rrbracket$
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m', h) \mid$

SignedDivNode:

$\llbracket kind \ g \ nid = (SignedDivNode \ nid \ x \ y \ zero \ sb \ nxt);$
 $\quad g \vdash x \simeq xe;$
 $\quad g \vdash y \simeq ye;$

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

SignedRemNode:

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

StaticLoadFieldNode:

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

StoreFieldNode:

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

StaticStoreFieldNode:

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

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

3.3 Interprocedural Semantics

type-synonym *Signature* = *string*

type-synonym *Program* = *Signature* \rightarrow *IRGraph*

inductive *step-top* :: *Program* \Rightarrow (*IRGraph* \times *ID* \times *MapState* \times *Params*) *list* \times

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

$(- \vdash - \longrightarrow - \ 55)$

for P where

Lift:

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

InvokeNodeStep:

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

$callTarget = ir-callTarget \ (kind \ g \ nid);$

$kind \ g \ callTarget = (MethodCallTargetNode \ targetMethod \ arguments);$

$Some \ targetGraph = P \ targetMethod;$

$m' = new-map-state;$

$g \vdash arguments \simeq_L argsE;$

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

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

ReturnNode:

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

$g \vdash expr \simeq e;$

$[m, p] \vdash e \mapsto v;$

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

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

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

ReturnNodeVoid:

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

$cm' = cm(cnid := (ObjRef \ (Some \ (2048))));$

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

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

UnwindNode:

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

$g \vdash exception \simeq exceptionE;$

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

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

$cm' = cm(cnid := e)$

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

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

3.4 Big-step Execution

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

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

inductive *exec* :: *Program*
 \Rightarrow (*IRGraph* \times *ID* \times *MapState* \times *Params*) *list* \times *FieldRefHeap*
 \Rightarrow *Trace*
 \Rightarrow (*IRGraph* \times *ID* \times *MapState* \times *Params*) *list* \times *FieldRefHeap*
 \Rightarrow *Trace*
 \Rightarrow *bool*
 (- \vdash - | - \longrightarrow^* - | -)
for *P*
where
 $\llbracket P \vdash (((g, \text{id}, m, p) \# xs), h) \longrightarrow (((g', \text{id}', m', p') \# ys), h'); \neg(\text{has-return } m') \rrbracket$
 $l' = (l @ [(g, \text{id}, m, p)]);$
 $\text{exec } P (((g', \text{id}', m', p') \# ys), h') \text{ } l' \text{ next-state } l''$
 $\implies \text{exec } P (((g, \text{id}, m, p) \# xs), h) \text{ } l \text{ next-state } l''$
 |
 $\llbracket P \vdash (((g, \text{id}, m, p) \# xs), h) \longrightarrow (((g', \text{id}', m', p') \# ys), h'); \text{has-return } m';$
 $l' = (l @ [(g, \text{id}, m, p)]);$
 $\implies \text{exec } P (((g, \text{id}, m, p) \# xs), h) \text{ } l (((g', \text{id}', m', p') \# ys), h') \text{ } l'$
code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow \text{bool}$ as *Exec*) *exec* .

inductive *exec-debug* :: *Program*
 \Rightarrow (*IRGraph* \times *ID* \times *MapState* \times *Params*) *list* \times *FieldRefHeap*
 \Rightarrow *nat*
 \Rightarrow (*IRGraph* \times *ID* \times *MapState* \times *Params*) *list* \times *FieldRefHeap*
 \Rightarrow *bool*
 (- $\vdash \longrightarrow^* \text{--}^* \text{--}$ -)
where
 $\llbracket n > 0;$
 $p \vdash s \longrightarrow s';$
 $\text{exec-debug } p \text{ } s' \text{ } (n - 1) \text{ } s'' \rrbracket$
 $\implies \text{exec-debug } p \text{ } s \text{ } n \text{ } s'' \mid$
 $\llbracket n = 0 \rrbracket$

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

3.4.1 Heap Testing

definition *p3* :: *Params* **where**
p3 = [*IntVal* 32 3]

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

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

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

values {*h-load-field* *field-sq* *None* (*prod.snd* *res*)
| *res*. (λx . *Some* *eg3-sq*) \vdash [(*eg3-sq*, 0, *new-map-state*, *p3*), (*eg3-sq*, 0,
new-map-state, *p3*)], *new-heap*) $\rightarrow^* 3^* \text{res}$ }

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*) \vdash [(*eg4-sq*, 0, *new-map-state*, *p3*), (*eg4-sq*, 0, *new-map-state*,
p3)], *new-heap*) $\rightarrow^* 3^* \text{res}$ }

end

3.5 Data-flow Tree Theorems

```

theory IRTreeEvalThms
  imports
    Graph.ValueThms
    IRTreeEval
begin

```

3.5.1 Deterministic Data-flow Evaluation

```

lemma evalDet:
   $[m,p] \vdash e \mapsto v_1 \implies$ 
   $[m,p] \vdash e \mapsto v_2 \implies$ 
   $v_1 = v_2$ 
  apply (induction arbitrary: v2 rule: evaltree.induct)
  by (elim EvalTreeE; auto)+

```

```

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

```

3.5.2 Typing Properties for Integer Evaluation Functions

We use three simple typing properties on integer values: *isIntVal32*, *isIntVal64* and the more general *isIntVal*.

```

lemma unary-eval-not-obj-ref:
  shows unary-eval op x  $\neq$  ObjRef v
  by (cases op; cases x; auto)

```

```

lemma unary-eval-not-obj-str:
  shows unary-eval op x  $\neq$  ObjStr v
  by (cases op; cases x; auto)

```

```

lemma unary-eval-int:
  assumes def: unary-eval op x  $\neq$  UndefVal
  shows isIntVal (unary-eval op x)
  unfolding isIntVal-def using def
  apply (cases unary-eval op x; auto)
  using unary-eval-not-obj-ref unary-eval-not-obj-str by simp+

```

```

lemma bin-eval-int:
  assumes def: bin-eval op x y ≠ UndefVal
  shows is-IntVal (bin-eval op x y)
  apply (cases op; cases x; cases y)
  unfolding is-IntVal-def using def apply auto
    apply presburger+
    apply (meson bool-to-val.elims)
    apply (meson bool-to-val.elims)
    apply (smt (verit) new-int.simps)+
  by (meson bool-to-val.elims)+

lemma IntVal0:
  (IntVal 32 0) = (new-int 32 0)
  unfolding new-int.simps
  by auto

lemma IntVal1:
  (IntVal 32 1) = (new-int 32 1)
  unfolding new-int.simps
  by auto

lemma bin-eval-new-int:
  assumes def: bin-eval op x y ≠ UndefVal
  shows ∃ b v. (bin-eval op x y) = new-int b v ∧
    b = (if op ∈ binary-fixed-32-ops then 32 else intval-bits x)
  apply (cases op; cases x; cases y)
  unfolding is-IntVal-def using def apply auto
  apply presburger+
  apply (metis take-bit-and)
  apply presburger
  apply (metis take-bit-or)
  apply presburger
  apply (metis take-bit-xor)
  apply presburger
  using IntVal0 IntVal1
  apply (metis bool-to-val.elims new-int.simps)
  apply presburger
  apply (smt (verit) new-int.elims)
  apply (smt (verit, best) new-int.elims)
  apply (metis IntVal0 IntVal1 bool-to-val.elims new-int.simps)
  apply presburger
  apply (metis IntVal0 IntVal1 bool-to-val.elims new-int.simps)
  apply presburger
  apply (metis IntVal0 IntVal1 bool-to-val.elims new-int.simps)
  by meson

lemma int-stamp:

```



```

assumes i: is-IntVal v
shows is-IntegerStamp (constantAsStamp v)
using i unfolding is-IntegerStamp-def is-IntVal-def by auto

```

```

lemma validStampIntConst:
  assumes v = IntVal b ival
  assumes  $0 < b \wedge b \leq 64$ 
  shows valid-stamp (constantAsStamp v)
proof –
  have bnds: fst (bit-bounds b)  $\leq$  int-signed-value b ival  $\wedge$  int-signed-value b ival
 $\leq$  snd (bit-bounds b)
    using assms int-signed-value-bounds
    by presburger
  have s: constantAsStamp v = IntegerStamp b (int-signed-value b ival) (int-signed-value
b ival)
    using assms(1) constantAsStamp.simps(1) by blast
  then show ?thesis
    unfolding s valid-stamp.simps
    using assms(2) assms bnds by linarith
qed

```

```

lemma validDefIntConst:
  assumes v: v = IntVal b ival
  assumes  $0 < b \wedge b \leq 64$ 
  assumes take-bit b ival = ival
  shows valid-value v (constantAsStamp v)
proof –
  have bnds: fst (bit-bounds b)  $\leq$  int-signed-value b ival  $\wedge$  int-signed-value b ival
 $\leq$  snd (bit-bounds b)
    using assms int-signed-value-bounds
    by presburger
  have s: constantAsStamp v = IntegerStamp b (int-signed-value b ival) (int-signed-value
b ival)
    using assms(1) constantAsStamp.simps(1) by blast
  then show ?thesis
    unfolding s unfolding v unfolding valid-value.simps
    using assms validStampIntConst
    by simp
qed

```

3.5.3 Evaluation Results are Valid

A valid value cannot be *UndefVal*.

```

lemma valid-not-undef:
  assumes a1: valid-value val s
  assumes a2: s  $\neq$  VoidStamp
  shows val  $\neq$  UndefVal

```

```

apply (rule valid-value.elims(1)[of val s True])
using a1 a2 by auto

lemma valid-VoidStamp[elim]:
  shows valid-value val VoidStamp  $\implies$ 
    val = UndefVal
  using valid-value.simps by metis

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

lemma valid-int[elim]:
  shows valid-value val (IntegerStamp b lo hi)  $\implies$ 
    ( $\exists v. \text{val} = \text{IntVal } b \ v$ )
  using valid-value.elims(2) by fastforce

lemmas valid-value-elim =
  valid-VoidStamp
  valid-ObjStamp
  valid-int

lemma evaltree-not-undef:
  fixes m p e v
  shows ( $[m,p] \vdash e \mapsto v$ )  $\implies v \neq \text{UndefVal}$ 
  apply (induction rule: evaltree.induct)
  using valid-not-undef by auto

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

proof –
  have valid-value val (IntegerStamp b lo hi)
  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 valid-value-signed-int-range [simp]:
  assumes valid-value val (IntegerStamp b lo hi)

```

```

assumes  $lo < 0$ 
shows  $\exists v. (val = IntVal\ b\ v \wedge$ 
          $lo \leq int\text{-}signed\text{-}value\ b\ v \wedge$ 
          $int\text{-}signed\text{-}value\ b\ v \leq hi)$ 
using assms valid-int
by (metis valid-value.simps(1))

```

3.5.4 Example Data-flow Optimisations

3.5.5 Monotonicity of Expression Refinement

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  $(UnaryExpr\ op\ e) \geq (UnaryExpr\ op\ e')$ 
  using UnaryExpr assms by auto

```

```

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

```

```

lemma never-void:
  assumes  $[m, p] \vdash x \mapsto xv$ 
  assumes valid-value xv (stamp-expr xe)
  shows  $stamp\text{-}expr\ xe \neq VoidStamp$ 
  using valid-value.simps
  using assms(2) by force

```

```

lemma compatible-trans:
  compatible x y  $\wedge$  compatible y z  $\implies$  compatible x z
  by (cases x; cases y; cases z; simp del: valid-stamp.simps)

```

```

lemma compatible-refl:
  compatible x y  $\implies$  compatible y x
  using compatible.elims(2) by fastforce

```

```

lemma mono-conditional:
  assumes  $ce \geq ce'$ 
  assumes  $te \geq te'$ 
  assumes  $fe \geq fe'$ 
  shows  $(ConditionalExpr\ ce\ te\ fe) \geq (ConditionalExpr\ ce'\ te'\ fe')$ 
proof (simp only: le-expr-def; (rule allI)+; rule impI)
  fix  $m\ p\ v$ 
  assume  $a: [m,p] \vdash ConditionalExpr\ ce\ te\ fe \mapsto v$ 
  then obtain  $cond$  where  $ce: [m,p] \vdash ce \mapsto cond$  by auto
  then have  $ce': [m,p] \vdash ce' \mapsto cond$  using assms by auto

  define  $branch$  where  $b: branch = (if\ val\text{-}to\text{-}bool\ cond\ then\ te\ else\ fe)$ 
  define  $branch'$  where  $b': branch' = (if\ val\text{-}to\text{-}bool\ cond\ then\ te'\ else\ fe')$ 
  then have  $beval: [m,p] \vdash branch \mapsto v$  using  $a\ b\ ce\ evalDet$  by blast

  from  $beval$  have  $[m,p] \vdash branch' \mapsto v$  using assms  $b\ b'$  by auto
  then show  $[m,p] \vdash ConditionalExpr\ ce'\ te'\ fe' \mapsto v$ 
    using  $ConditionalExpr\ ce'\ b'$ 
    using  $a$  by blast
qed

```

3.6 Unfolding rules for evaltree quadruples down to bin-eval level

These rewrite rules can be useful when proving optimizations. They support top-down rewriting of each level of the tree into the lower-level *bin_eeval* / *unary_eeval* level, simply by saying *unfoldingunfold_eevaltree*.

```

lemma unfold-const:
  shows  $([m,p] \vdash ConstantExpr\ c \mapsto v) = (valid\text{-}value\ v\ (constantAsStamp\ c) \wedge v = c)$ 
  by blast

```

```

lemma unfold-binary:
  shows  $([m,p] \vdash BinaryExpr\ op\ xe\ ye \mapsto val) = (\exists\ x\ y. (([m,p] \vdash xe \mapsto x) \wedge ([m,p] \vdash ye \mapsto y) \wedge (val = bin\text{-}eval\ op\ x\ y) \wedge (val \neq UndefinedVal)))$ 
  proof (intro iffI)
    assume  $?L: ?L$ 
    show  $?R$  by (rule evaltree.cases[OF ?L]; blast+)
  next
    assume  $?R$ 

```

```

then obtain  $x\ y$  where  $[m,p] \vdash xe \mapsto x$ 
  and  $[m,p] \vdash ye \mapsto y$ 
  and  $val = \text{bin-eval } op\ x\ y$ 
  and  $val \neq \text{UndefVal}$ 
by auto
then show ?L
  by (rule BinaryExpr)
qed

```

```

lemma unfold-unary:
shows  $([m,p] \vdash \text{UnaryExpr } op\ xe \mapsto val)$ 
  =  $(\exists x.$ 
     $(([m,p] \vdash xe \mapsto x) \wedge$ 
     $(val = \text{unary-eval } op\ x) \wedge$ 
     $(val \neq \text{UndefVal})$ 
   $))$  (is ?L = ?R)
by auto

```

```

lemmas unfold-evaltree =
  unfold-binary
  unfold-unary

```

3.7 Lemmas about *new_int* and integer eval results.

```

lemma unary-eval-new-int:
assumes def:  $\text{unary-eval } op\ x \neq \text{UndefVal}$ 
shows  $\exists b\ v. \text{unary-eval } op\ x = \text{new-int } b\ v \wedge$ 
       $b = (\text{if } op \in \text{normal-unary then intval-bits } x \text{ else ir-resultBits } op)$ 
proof (cases  $op \in \text{normal-unary}$ )
case True
  then show ?thesis
    by (metis def empty-iff insert-iff intval-abs.elims intval-bits.simps intval-logic-negation.elims
      intval-negate.elims intval-not.elims unary-eval.simps(1) unary-eval.simps(2) unary-eval.simps(3)
      unary-eval.simps(4))
next
case False
  consider  $ib\ ob$  where  $op = \text{UnaryNarrow } ib\ ob \mid$ 
     $ib\ ob$  where  $op = \text{UnaryZeroExtend } ib\ ob \mid$ 
     $ib\ ob$  where  $op = \text{UnarySignExtend } ib\ ob$ 
  by (metis False IRUnaryOp.exhaust insert-iff)
  then show ?thesis
proof (cases)
case 1
  then show ?thesis
    by (metis False IRUnaryOp.sel(4) def intval-narrow.elims unary-eval.simps(5))
next
case 2

```

```

    then show ?thesis
    by (metis False IRUnaryOp.sel(6) def intval-zero-extend.elims unary-eval.simps(7))
next
  case 3
  then show ?thesis
  by (metis False IRUnaryOp.sel(5) def intval-sign-extend.elims unary-eval.simps(6))
qed
qed

lemma new-int-unused-bits-zero:
  assumes IntVal b ival = new-int b ival0
  shows take-bit b ival = ival
  using assms(1) new-int-take-bits by blast

lemma unary-eval-unused-bits-zero:
  assumes unary-eval op x = IntVal b ival
  shows take-bit b ival = ival
  using assms unary-eval-new-int
  by (metis Value.inject(1) Value.simps(5) new-int.elims new-int-unused-bits-zero)

lemma bin-eval-unused-bits-zero:
  assumes bin-eval op x y = (IntVal b ival)
  shows take-bit b ival = ival
  using assms bin-eval-new-int
  by (metis Value.distinct(1) Value.inject(1) new-int.elims new-int-take-bits)

lemma eval-unused-bits-zero:
  [m,p] ⊢ xe ↦ (IntVal b ix) ⇒ take-bit b ix = ix
proof (induction xe)
  case (UnaryExpr x1 xe)
  then show ?case
    using unary-eval-unused-bits-zero by force
next
  case (BinaryExpr x1 xe1 xe2)
  then show ?case
    using bin-eval-unused-bits-zero by force
next
  case (ConditionalExpr xe1 xe2 xe3)
  then show ?case
    by (metis (full-types) EvalTreeE(3))
next
  case (ParameterExpr i s)
  then have valid-value (p!i) s
    by fastforce
  then show ?case
    by (metis ParameterExprE Value.distinct(7) intval-bits.simps intval-word.simps
    local.ParameterExpr valid-value.elims(2))
next
  case (LeafExpr x1 x2)

```

```

then show ?case
  by (smt (z3) EvalTreeE(6) Value.simps(11) valid-value.elims(1) valid-value.simps(1))

next
  case (ConstantExpr x)
  then show ?case
    by (metis EvalTreeE(1) constantAsStamp.simps(1) valid-value.simps(1))
next
  case (ConstantVar x)
  then show ?case
    by fastforce
next
  case (VariableExpr x1 x2)
  then show ?case
    by fastforce
qed

```

```

lemma unary-normal-bitsize:
  assumes unary-eval op x = IntVal b ival
  assumes op ∈ normal-unary
  shows ∃ ix. x = IntVal b ix
  apply (cases op)
    prefer 7 using assms apply blast
    prefer 6 using assms apply blast
    prefer 5 using assms apply blast
  using Value.distinct(1) Value.sel(1) assms(1) new-int.simps unary-eval.simps
    intval-abs.elims intval-negate.elims intval-not.elims intval-logic-negation.elims
  apply metis+
done

```

```

lemma unary-not-normal-bitsize:
  assumes unary-eval op x = IntVal b ival
  assumes op ∉ normal-unary
  shows b = ir-resultBits op ∧ 0 < b ∧ b ≤ 64
  apply (cases op)
  using assms apply blast+
    apply (metis IRUnaryOp.sel(4) Value.distinct(1) Value.sel(1) assms(1) int-
      val-narrow.elims intval-narrow-ok new-int.simps unary-eval.simps(5))
    apply (smt (verit) IRUnaryOp.sel(5) Value.distinct(1) Value.sel(1) assms(1)
      intval-sign-extend.elims new-int.simps order-less-le-trans unary-eval.simps(6))
    apply (metis IRUnaryOp.sel(6) Value.distinct(1) assms(1) intval-bits.simps int-
      val-zero-extend.elims linorder-not-less neq0-conv new-int.simps unary-eval.simps(7))
done

```

```

lemma unary-eval-bitsize:
  assumes unary-eval op x = IntVal b ival
  assumes 2: x = IntVal bx ix

```

```

    assumes  $0 < bx \wedge bx \leq 64$ 
    shows  $0 < b \wedge b \leq 64$ 
  proof (cases  $op \in \text{normal-unary}$ )
    case True
    then obtain tmp where unary-eval  $op\ x = \text{new-int}\ bx\ tmp$ 
      by (cases  $op$ ; simp; auto simp: 2)
    then show ?thesis
      using assms by simp
  next
    case False
    then obtain tmp where unary-eval  $op\ x = \text{new-int}\ b\ tmp \wedge 0 < b \wedge b \leq 64$ 
      apply (cases  $op$ ; simp; auto simp: 2)
      apply (metis 2 Value.inject(1) Value.simps(5) assms(1) intval-narrow.simps(1)
        intval-narrow-ok new-int.simps unary-eval.simps(5))
      apply (metis 2 Value.distinct(1) Value.inject(1) assms(1) bot-nat-0.not-eq-extremum
        diff-is-0-eq intval-sign-extend.elims new-int.simps unary-eval.simps(6) zero-less-diff)
      by (smt (verit, del-insts) 2 Value.simps(5) assms(1) intval-bits.simps int-
        val-zero-extend.simps(1) new-int.simps order-less-le-trans unary-eval.simps(7))
    then show ?thesis
      by blast
  qed

```

```

lemma bin-eval-inputs-are-ints:
  assumes bin-eval  $op\ x\ y = \text{IntVal}\ b\ ix$ 
  obtains  $xb\ yb\ xi\ yi$  where  $x = \text{IntVal}\ xb\ xi \wedge y = \text{IntVal}\ yb\ yi$ 
proof -
  have bin-eval  $op\ x\ y \neq \text{UndefVal}$ 
    by (simp add: assms)
  then show ?thesis
    using assms apply (cases  $op$ ; cases  $x$ ; cases  $y$ ; simp)
    using that by blast+
qed

```

```

lemma eval-bits-1-64:
   $[m,p] \vdash xe \mapsto (\text{IntVal}\ b\ ix) \implies 0 < b \wedge b \leq 64$ 
proof (induction  $xe$  arbitrary:  $b\ ix$ )
  case (UnaryExpr  $op\ x2$ )
  then obtain  $xv$  where
     $xv: ([m,p] \vdash x2 \mapsto xv) \wedge$ 
     $\text{IntVal}\ b\ ix = \text{unary-eval}\ op\ xv$ 
    using unfold-binary by auto
  then have  $b = (\text{if } op \in \text{normal-unary} \text{ then intval-bits } xv \text{ else ir-resultBits } op)$ 
    using unary-eval-new-int
    by (metis Value.disc(1) Value.discI(1) Value.sel(1) new-int.simps)
  then show ?case

```



```

    by (metis xv UnaryExpr.IH unary-normal-bitsize unary-not-normal-bitsize)
next
  case (BinaryExpr op x y)
  then obtain xv yv where
    xy: ([m,p] ⊢ x ↦ xv) ∧
        ([m,p] ⊢ y ↦ yv) ∧
        IntVal b ix = bin-eval op xv yv
    using unfold-binary by auto
  then have def: bin-eval op xv yv ≠ UndefVal and xv: xv ≠ UndefVal and yv ≠
UndefVal
    using evaltree-not-undef xy by (force, blast, blast)
  then have b = (if op ∈ binary-fixed-32-ops then 32 else intval-bits xv)
    by (metis xy intval-bits.simps new-int.simps bin-eval-new-int)
  then show ?case
    by (metis BinaryExpr.IH(1) Value.distinct(7) Value.distinct(9) xv bin-eval-inputs-are-ints
intval-bits.elims le-add-same-cancel1 less-or-eq-imp-le numeral-Bit0 xy zero-less-numeral)
next
  case (ConditionalExpr xe1 xe2 xe3)
  then show ?case
    by (metis (full-types) EvalTreeE(3))
next
  case (ParameterExpr x1 x2)
  then show ?case
    using ParameterExprE intval-bits.simps valid-stamp.simps(1) valid-value.elims(2)
valid-value.simps(17)
    by (metis (no-types, lifting))
next
  case (LeafExpr x1 x2)
  then show ?case
    by (smt (z3) EvalTreeE(6) Value.distinct(7) Value.inject(1) valid-stamp.simps(1)
valid-value.elims(1))
next
  case (ConstantExpr x)
  then show ?case
    by (metis EvalTreeE(1) constantAsStamp.simps(1) valid-stamp.simps(1) valid-value.simps(1))

next
  case (ConstantVar x)
  then show ?case
    by blast
next
  case (VariableExpr x1 x2)
  then show ?case
    by blast
qed

```

lemma *unfold-binary-width*:

```

assumes  $op \notin \text{binary-fixed-32-ops} \wedge op \notin \text{binary-shift-ops}$ 
shows  $([m,p] \vdash \text{BinaryExpr } op \ x \ y \mapsto \text{IntVal } b \ \text{val}) = (\exists \ x \ y. \\
  ([m,p] \vdash x \mapsto \text{IntVal } b \ x) \wedge \\
  ([m,p] \vdash y \mapsto \text{IntVal } b \ y) \wedge \\
  (\text{IntVal } b \ \text{val} = \text{bin-eval } op \ (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)) \wedge \\
  (\text{IntVal } b \ \text{val} \neq \text{UndefVal}) \\
  )) \text{ (is } ?L = ?R)$ 
proof (intro iffI)
  assume  $?L$ 
  show  $?R$  apply (rule evaltree.cases[OF ?L])
    apply force+ apply auto[1]
    using assms apply (cases op; auto)
      apply (smt (verit) intval-add.elims Value.inject(1))
    using intval-mul.elims Value.inject(1)
      apply (smt (verit) new-int.simps new-int-bin.simps)
    using intval-sub.elims Value.inject(1)
      apply (smt (verit) new-int.simps new-int-bin.simps)
    using intval-and.elims Value.inject(1)
      apply (smt (verit) new-int.simps new-int-bin.simps take-bit-and)
    using intval-or.elims Value.inject(1)
      apply (smt (verit) new-int.simps new-int-bin.simps take-bit-or)
    using intval-xor.elims Value.inject(1)
      apply (smt (verit) new-int.simps new-int-bin.simps take-bit-xor)
  by blast

next
  assume  $R$ :  $?R$ 
  then obtain  $x \ y$  where  $[m,p] \vdash x \mapsto \text{IntVal } b \ x$ 
    and  $[m,p] \vdash y \mapsto \text{IntVal } b \ y$ 
    and  $\text{new-int } b \ \text{val} = \text{bin-eval } op \ (\text{IntVal } b \ x) \ (\text{IntVal } b \ y)$ 
    and  $\text{new-int } b \ \text{val} \neq \text{UndefVal}$ 
    using bin-eval-unused-bits-zero by force
  then show  $?L$ 
    using  $R$  by blast
qed

end

```

3.8 Tree to Graph Theorems

```

theory TreeToGraphThms
imports
  IRTreeEvalThms
  IRGraphFrames
  HOL-Eisbach.Eisbach
  HOL-Eisbach.Eisbach-Tools
begin

```

3.8.1 Extraction and Evaluation of Expression Trees is Deterministic.

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

named-theorems *rep*

lemma *rep-constant* [*rep*]:

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

lemma *rep-parameter* [*rep*]:

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

lemma *rep-conditional* [*rep*]:

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

lemma *rep-abs* [*rep*]:

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

lemma *rep-not* [*rep*]:

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

lemma *rep-negate* [*rep*]:

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

lemma *rep-logicnegation* [*rep*]:

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

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

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

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

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

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

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

lemma *rep-short-circuit-or* [*rep*]:
 $g \vdash n \simeq e \implies$
 $kind\ g\ n = ShortCircuitOrNode\ x\ y \implies$
 $(\exists\ xe\ ye. e = BinaryExpr\ BinShortCircuitOr\ xe\ ye)$
by (*induction rule: rep.induct; auto*)

lemma *rep-left-shift* [*rep*]:
 $g \vdash n \simeq e \implies$
 $kind\ g\ n = LeftShiftNode\ x\ y \implies$
 $(\exists\ xe\ ye. e = BinaryExpr\ BinLeftShift\ xe\ ye)$
by (*induction rule: rep.induct; auto*)

lemma *rep-right-shift* [rep]:

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

lemma *rep-unsigned-right-shift* [rep]:

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

lemma *rep-integer-below* [rep]:

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

lemma *rep-integer-equals* [rep]:

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

lemma *rep-integer-less-than* [rep]:

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

lemma *rep-narrow* [rep]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = NarrowNode\ ib\ rb\ x \implies$
 $(\exists\ x. e = UnaryExpr\ (UnaryNarrow\ ib\ rb)\ x)$
by (induction rule: *rep.induct*; *auto*)

lemma *rep-sign-extend* [rep]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = SignExtendNode\ ib\ rb\ x \implies$
 $(\exists\ x. e = UnaryExpr\ (UnarySignExtend\ ib\ rb)\ x)$
by (induction rule: *rep.induct*; *auto*)

lemma *rep-zero-extend* [rep]:

$g \vdash n \simeq e \implies$
 $kind\ g\ n = ZeroExtendNode\ ib\ rb\ x \implies$
 $(\exists\ x. e = UnaryExpr\ (UnaryZeroExtend\ ib\ rb)\ x)$
by (induction rule: *rep.induct*; *auto*)

lemma *rep-load-field* [rep]:

```

g ⊢ n ≃ e ⇒
is-preevaluated (kind g n) ⇒
(∃ s. e = LeafExpr n s)
by (induction rule: rep.induct; auto)

```

lemma *rep-ref* [*rep*]:

```

g ⊢ n ≃ e ⇒
kind g n = RefNode n' ⇒
g ⊢ n' ≃ e
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 ⇒ -) ⇒ - ⇒
        ⟨match IRNode.distinct in d: node - ≠ RefNode - ⇒
          ⟨metis i e r d⟩⟩⟩⟩ |
  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 ⇒ -) ⇒ - ⇒
          ⟨match IRNode.distinct in d: node - - ≠ RefNode - ⇒
            ⟨metis i e r d⟩⟩⟩⟩ |
  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 ⇒ -) ⇒ - ⇒
          ⟨match IRNode.distinct in d: node - - - ≠ RefNode - ⇒
            ⟨metis i e r d⟩⟩⟩⟩ |
  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 ⇒ -) ⇒ - ⇒
          ⟨match IRNode.distinct in d: node - - - ≠ RefNode - ⇒
            ⟨metis i e r d⟩⟩⟩⟩)

```

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

lemma *repDet*:

shows $(g ⊢ n ≃ e_1) ⇒ (g ⊢ n ≃ e_2) ⇒ e_1 = e_2$

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*

by (*metis IRNode.disc(2685) ParameterNodeE is-RefNode-def rep-parameter*)

```

next
  case (ConditionalNode n c t f ce te fe)
  then show ?case
    using IRNode.distinct(593)
    using IRNode.inject(6) ConditionalNodeE rep-conditional
    by metis
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 (ShortCircuitOrNode n x y xe ye)
  then show ?case

```

```

    by (solve-det node: ShortCircuitOrNode)
next
  case (LeftShiftNode n x y xe ye)
  then show ?case
    by (solve-det node: LeftShiftNode)
next
  case (RightShiftNode n x y xe ye)
  then show ?case
    by (solve-det node: RightShiftNode)
next
  case (UnsignedRightShiftNode n x y xe ye)
  then show ?case
    by (solve-det node: UnsignedRightShiftNode)
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.distinct(2203) IRNode.inject(28) NarrowNodeE rep-narrow)
next
  case (SignExtendNode n x xe)
  then show ?case
    by (metis IRNode.distinct(2599) IRNode.inject(39) SignExtendNodeE rep-sign-extend)
next
  case (ZeroExtendNode n x xe)
  then show ?case
    by (metis IRNode.distinct(2753) IRNode.inject(50) ZeroExtendNodeE rep-zero-extend)
next
  case (LeafNode n s)
  then show ?case using rep-load-field LeafNodeE
    by (metis is-preevaluated.simps(53))
next
  case (RefNode n')
  then show ?case
    using rep-ref by blast
qed

```

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


```

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

```

```

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

```

```

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

```

3.8.2 Monotonicity of Graph Refinement

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

lemma *mono-narrow*:

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

lemma *mono-sign-extend*:

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

lemma *mono-zero-extend*:

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

lemma *mono-conditional-graph*:

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

by (*smt* (*verit*, *best*) *ConditionalNode*)

lemma *mono-add*:

assumes *kind g1 n = AddNode x y* \wedge *kind g2 n = AddNode x y*
assumes (*g1* \vdash *x* \simeq *xe1*) \wedge (*g2* \vdash *x* \simeq *xe2*)
assumes (*g1* \vdash *y* \simeq *ye1*) \wedge (*g2* \vdash *y* \simeq *ye2*)
assumes *xe1* \geq *xe2* \wedge *ye1* \geq *ye2*
assumes (*g1* \vdash *n* \simeq *e1*) \wedge (*g2* \vdash *n* \simeq *e2*)
shows *e1* \geq *e2*
using *mono-binary assms AddNodeE IRNode.inject(2) repDet rep-add*
by (*metis IRNode.distinct(205)*)

lemma *mono-mul*:

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

lemma *term-graph-evaluation*:

(*g* \vdash *n* \leq *e*) \implies (\forall *m p v* . (*[m,p]* \vdash *e* \mapsto *v*) \longrightarrow (*[g,m,p]* \vdash *n* \mapsto *v*))
unfolding *graph-represents-expression-def* **apply** *auto*
by (*meson encodeeval-def*)

lemma *encodes-contains*:

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

lemma *no-encoding*:

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

lemma *not-excluded-keep-type*:

assumes *n* \in *ids g1*
assumes *n* \notin *excluded*
assumes (*excluded* \leq *as-set g1*) \subseteq *as-set g2*
shows *kind g1 n = kind g2 n* \wedge *stamp g1 n = stamp g2 n*
using *assms* **unfolding** *as-set-def domain-subtraction-def* **by** *blast*

```

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

```

3.8.3 Lift Data-flow Tree Refinement to Graph Refinement

```

theorem graph-semantics-preservation:
  assumes a:  $e1' \geq e2'$ 
  assumes b:  $(\{n'\} \trianglelefteq \text{as-set } g1) \subseteq \text{as-set } g2$ 
  assumes c:  $g1 \vdash n' \simeq e1'$ 
  assumes d:  $g2 \vdash n' \simeq e2'$ 
  shows graph-refinement  $g1 \ g2$ 
  unfolding graph-refinement-def apply rule
  apply (metis b d ids-some no-encoding not-excluded-keep-type singleton-iff sub-
    setI)
  apply (rule allI) apply (rule impI) apply (rule allI) apply (rule impI)
  unfolding graph-represents-expression-def
proof -
  fix n e1
  assume e:  $n \in \text{ids } g1$ 
  assume f:  $(g1 \vdash n \simeq e1)$ 

  show  $\exists e2. (g2 \vdash n \simeq e2) \wedge e1 \geq e2$ 
  proof (cases  $n = n'$ )
    case True
      have g:  $e1 = e1'$  using c f True repDet by simp
      have h:  $(g2 \vdash n \simeq e2') \wedge e1' \geq e2'$ 
        using True a d by blast
      then show ?thesis
        using g by blast
    next
      case False
      have  $n \notin \{n'\}$ 
        using False by simp
      then have i:  $\text{kind } g1 \ n = \text{kind } g2 \ n \wedge \text{stamp } g1 \ n = \text{stamp } g2 \ n$ 
        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:  $g1 \vdash n \simeq \text{ConditionalExpr } ce1 \ te1 \ fe1$  using f ConditionalNode
    by (simp add: ConditionalNode.hyps(2) rep.ConditionalNode)
  obtain cn tn fn where l:  $\text{kind } g1 \ n = \text{ConditionalNode } cn \ tn \ fn$ 
    using ConditionalNode.hyps(1) by blast
  then have mc:  $g1 \vdash cn \simeq ce1$ 
    using ConditionalNode.hyps(1) ConditionalNode.hyps(2) by fastforce
  from l have mt:  $g1 \vdash tn \simeq te1$ 
    using ConditionalNode.hyps(1) ConditionalNode.hyps(3) by fastforce
  from l have mf:  $g1 \vdash fn \simeq fe1$ 
    using ConditionalNode.hyps(1) ConditionalNode.hyps(4) by fastforce
  then show ?case
  proof -
    have  $g1 \vdash cn \simeq ce1$  using mc by simp
    have  $g1 \vdash tn \simeq te1$  using mt by simp
    have  $g1 \vdash fn \simeq fe1$  using mf by simp
    have cer:  $\exists \ ce2. (g2 \vdash cn \simeq ce2) \wedge ce1 \geq ce2$ 
      using ConditionalNode
      using a b c d l no-encoding not-excluded-keep-type repDet singletonD
      by (metis-node-eq-ternary ConditionalNode)
    have ter:  $\exists \ te2. (g2 \vdash tn \simeq te2) \wedge te1 \geq te2$ 
      using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
      singletonD
      by (metis-node-eq-ternary ConditionalNode)
    have  $\exists \ fe2. (g2 \vdash fn \simeq fe2) \wedge fe1 \geq fe2$ 
      using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
      singletonD
      by (metis-node-eq-ternary ConditionalNode)
    then have  $\exists \ ce2 \ te2 \ fe2. (g2 \vdash n \simeq \text{ConditionalExpr } ce2 \ te2 \ fe2) \wedge$ 
       $\text{ConditionalExpr } ce1 \ te1 \ fe1 \geq \text{ConditionalExpr } ce2 \ te2 \ fe2$ 
      using ConditionalNode.premis l rep.ConditionalNode cer ter
      by (smt (verit) mono-conditional)
    then show ?thesis
      by meson
  qed
next
  case (AbsNode n x xe1)
  have k:  $g1 \vdash n \simeq \text{UnaryExpr } \text{UnaryAbs } xe1$  using f AbsNode
    by (simp add: AbsNode.hyps(2) rep.AbsNode)
  obtain xn where l:  $\text{kind } g1 \ n = \text{AbsNode } xn$ 
    using AbsNode.hyps(1) by blast
  then have m:  $g1 \vdash xn \simeq xe1$ 
    using AbsNode.hyps(1) AbsNode.hyps(2) by fastforce
  then show ?case

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

    then have  $\exists xe2 ye2. (g2 \vdash n \simeq BinaryExpr\ BinIntegerLessThan\ xe2\ ye2)$ 
 $\wedge BinaryExpr\ BinIntegerLessThan\ xe1\ ye1 \geq BinaryExpr\ BinIntegerLessThan\ xe2\ ye2$ 
    by (metis IntegerLessThanNode.prem1 mono-binary rep.IntegerLessThanNode
xer)
    then show ?thesis
    by meson
  qed
next
  case (NarrowNode n inputBits resultBits x xe1)
  have k:  $g1 \vdash n \simeq UnaryExpr\ (UnaryNarrow\ inputBits\ resultBits)\ xe1$  using
f NarrowNode
    by (simp add: NarrowNode.hyps(2) rep.NarrowNode)
  obtain xn where l: kind g1 n = NarrowNode inputBits resultBits xn
    using NarrowNode.hyps(1) by blast
  then have m:  $g1 \vdash xn \simeq xe1$ 
    using NarrowNode.hyps(1) NarrowNode.hyps(2)
    by auto
  then show ?case
  proof (cases xn = n')
    case True
    then have n:  $xe1 = e1'$  using c m repDet by simp
    then have ev:  $g2 \vdash n \simeq UnaryExpr\ (UnaryNarrow\ inputBits\ resultBits)$ 
e2' using NarrowNode.hyps(1) l m n
      using NarrowNode.prem1 True d rep.NarrowNode by simp
    then have r:  $UnaryExpr\ (UnaryNarrow\ inputBits\ resultBits)\ e1' \geq Unary-$ 
Expr (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  $\exists xe2. (g2 \vdash n \simeq UnaryExpr\ (UnaryNarrow\ inputBits\ result-$ 
Bits) xe2)  $\wedge UnaryExpr\ (UnaryNarrow\ inputBits\ resultBits)\ xe1 \geq UnaryExpr$ 
(UnaryNarrow inputBits resultBits) xe2
    by (metis NarrowNode.prem1 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  $l$ :  $\text{kind } g1 \ n = \text{SignExtendNode } \text{inputBits } \text{resultBits } xn$ 
using  $\text{SignExtendNode.hyps}(1)$  by blast
then have  $m$ :  $g1 \vdash xn \simeq xe1$ 
using  $\text{SignExtendNode.hyps}(1)$   $\text{SignExtendNode.hyps}(2)$ 
by auto
then show  $?case$ 
proof ( $\text{cases } xn = n'$ )
  case True
    then have  $n$ :  $xe1 = e1'$  using  $c \ m \ \text{repDet}$  by simp
    then have  $ev$ :  $g2 \vdash n \simeq \text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits})$ 
 $e2'$  using  $\text{SignExtendNode.hyps}(1)$   $l \ m \ n$ 
      using  $\text{SignExtendNode.premis } \text{True } d \ \text{rep.SignExtendNode}$  by simp
    then have  $r$ :  $\text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \ e1' \geq$ 
 $\text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \ e2'$ 
      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  $\text{SignExtendNode}$ 
        using  $\text{False } b \ \text{encodes-contains } l \ \text{not-excluded-keep-type not-in-g singleton-iff}$ 
        by (metis-node-eq-ternary SignExtendNode)
      then have  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{result-}$ 
 $\text{Bits}) \ xe2) \wedge \text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \ xe1 \geq \text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \ xe2$ 
        by (metis SignExtendNode.premis l mono-unary rep.SignExtendNode)
      then show  $?thesis$ 
        by meson
    qed
  next
    case ( $\text{ZeroExtendNode } n \ \text{inputBits } \text{resultBits } x \ xe1$ )
      have  $k$ :  $g1 \vdash n \simeq \text{UnaryExpr } (\text{UnaryZeroExtend } \text{inputBits } \text{resultBits}) \ xe1$ 
using  $f \ \text{ZeroExtendNode}$ 
      by (simp add: ZeroExtendNode.hyps(2) rep.ZeroExtendNode)
    obtain  $xn$  where  $l$ :  $\text{kind } g1 \ n = \text{ZeroExtendNode } \text{inputBits } \text{resultBits } xn$ 
using  $\text{ZeroExtendNode.hyps}(1)$  by blast
then have  $m$ :  $g1 \vdash xn \simeq xe1$ 
using  $\text{ZeroExtendNode.hyps}(1)$   $\text{ZeroExtendNode.hyps}(2)$ 
by auto
then show  $?case$ 
proof ( $\text{cases } xn = n'$ )
  case True
    then have  $n$ :  $xe1 = e1'$  using  $c \ m \ \text{repDet}$  by simp
    then have  $ev$ :  $g2 \vdash n \simeq \text{UnaryExpr } (\text{UnaryZeroExtend } \text{inputBits } \text{resultBits})$ 
 $e2'$  using  $\text{ZeroExtendNode.hyps}(1)$   $l \ m \ n$ 
      using  $\text{ZeroExtendNode.premis } \text{True } d \ \text{rep.ZeroExtendNode}$  by simp
    then have  $r$ :  $\text{UnaryExpr } (\text{UnaryZeroExtend } \text{inputBits } \text{resultBits}) \ e1' \geq$ 

```

```

UnaryExpr (UnaryZeroExtend inputBits resultBits) e2'
  by (meson a mono-unary)
  then show ?thesis using ev r
  by (metis n)
next
case False
have g1 ⊢ xn ≃ xe1 using m by simp
have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ 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 ∃ xe2. (g2 ⊢ n ≃ UnaryExpr (UnaryZeroExtend inputBits result-
Bits) xe2) ∧ UnaryExpr (UnaryZeroExtend inputBits resultBits) xe1 ≥ UnaryExpr
(UnaryZeroExtend inputBits resultBits) xe2
  by (metis ZeroExtendNode.premis l mono-unary rep.ZeroExtendNode)
  then show ?thesis
  by meson
qed
next
case (LeafNode n s)
then show ?case
  by (metis eq-refl rep.LeafNode)
next
case (RefNode n')
then show ?case
  by (metis a b c d no-encoding not-excluded-keep-type rep.RefNode repDet
singletonD)
qed
qed
qed

```

lemma *graph-antics-preservation-subscript:*
assumes $a: e_1' \geq e_2'$
assumes $b: (\{n\} \trianglelefteq \text{as-set } g_1) \subseteq \text{as-set } g_2$
assumes $c: g_1 \vdash n \simeq e_1'$
assumes $d: g_2 \vdash n \simeq e_2'$
shows *graph-refinement* $g_1 \ g_2$
using *graph-antics-preservation* *assms* **by** *simp*

lemma *tree-to-graph-rewriting:*
 $e_1 \geq e_2$
 $\wedge (g_1 \vdash n \simeq e_1) \wedge \text{maximal-sharing } g_1$
 $\wedge (\{n\} \trianglelefteq \text{as-set } g_1) \subseteq \text{as-set } g_2$
 $\wedge (g_2 \vdash n \simeq e_2) \wedge \text{maximal-sharing } g_2$
 $\implies \text{graph-refinement } g_1 \ g_2$
using *graph-antics-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 eval-contains-id[simp]: g1 ⊢ n ≃ e ⇒ n ∈ ids g1
  using no-encoding by blast

lemma subset-kind[simp]: as-set g1 ⊆ as-set g2 ⇒ g1 ⊢ n ≃ e ⇒ kind g1 n =
kind g2 n
  using eval-contains-id unfolding as-set-def
  by blast

lemma subset-stamp[simp]: as-set g1 ⊆ as-set g2 ⇒ g1 ⊢ n ≃ e ⇒ stamp g1
n = stamp g2 n
  using eval-contains-id unfolding as-set-def
  by blast

method solve-subset-eval uses as-set eval =
  (metis eval as-set subset-kind subset-stamp |
   metis eval as-set subset-kind)

lemma subset-implies-evals:
  assumes as-set g1 ⊆ as-set g2
  assumes (g1 ⊢ n ≃ e)
  shows (g2 ⊢ n ≃ e)
  using assms(2)
  apply (induction e)
    apply (solve-subset-eval as-set: assms(1) eval: ConstantNode)
    apply (solve-subset-eval as-set: assms(1) eval: ParameterNode)
    apply (solve-subset-eval as-set: assms(1) eval: ConditionalNode)
    apply (solve-subset-eval as-set: assms(1) eval: AbsNode)
    apply (solve-subset-eval as-set: assms(1) eval: NotNode)
    apply (solve-subset-eval as-set: assms(1) eval: NegateNode)
    apply (solve-subset-eval as-set: assms(1) eval: LogicNegationNode)
    apply (solve-subset-eval as-set: assms(1) eval: AddNode)
    apply (solve-subset-eval as-set: assms(1) eval: MulNode)
    apply (solve-subset-eval as-set: assms(1) eval: SubNode)
    apply (solve-subset-eval as-set: assms(1) eval: AndNode)
    apply (solve-subset-eval as-set: assms(1) eval: OrNode)
    apply (solve-subset-eval as-set: assms(1) eval: XorNode)
    apply (solve-subset-eval as-set: assms(1) eval: ShortCircuitOrNode)

```

```

    apply (solve-subset-eval as-set: assms(1) eval: LeftShiftNode)
    apply (solve-subset-eval as-set: assms(1) eval: RightShiftNode)
    apply (solve-subset-eval as-set: assms(1) eval: UnsignedRightShiftNode)
    apply (solve-subset-eval as-set: assms(1) eval: IntegerBelowNode)
    apply (solve-subset-eval as-set: assms(1) eval: IntegerEqualsNode)
    apply (solve-subset-eval as-set: assms(1) eval: IntegerLessThanNode)
    apply (solve-subset-eval as-set: assms(1) eval: NarrowNode)
    apply (solve-subset-eval as-set: assms(1) eval: SignExtendNode)
    apply (solve-subset-eval as-set: assms(1) eval: ZeroExtendNode)
    apply (solve-subset-eval as-set: assms(1) eval: LeafNode)
  by (solve-subset-eval as-set: assms(1) eval: RefNode)

```

lemma *subset-refines*:

assumes $as\text{-}set\ g1 \subseteq as\text{-}set\ g2$

shows *graph-refinement* $g1\ g2$

proof –

have $ids\ g1 \subseteq ids\ g2$ **using** *assms* **unfolding** *as-set-def*

by *blast*

then show *?thesis* **unfolding** *graph-refinement-def* **apply** *rule*

apply (*rule allI*) **apply** (*rule impI*) **apply** (*rule allI*) **apply** (*rule impI*)

unfolding *graph-represents-expression-def*

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*:

$e1 \geq e2$

$\wedge as\text{-}set\ g1 \subseteq as\text{-}set\ g2$

$\wedge (g2 \vdash n \simeq e2)$

$\implies (g2 \vdash n \trianglelefteq e1) \wedge graph\text{-}refinement\ g1\ g2$

using *subset-refines*

by (*meson encodeeval-def graph-represents-expression-def le-expr-def*)

3.8.4 Term Graph Reconstruction

lemma *find-exists-kind*:

assumes *find-node-and-stamp* $g\ (node, s) = Some\ nid$

shows *kind* $g\ nid = node$

using *assms* **unfolding** *find-node-and-stamp.simps*

by (*metis* (*mono-tags*, *lifting*) *find-Some-iff*)

lemma *find-exists-stamp*:

```

assumes find-node-and-stamp g (node, s) = Some nid
shows stamp g nid = s
using assms unfolding find-node-and-stamp.simps
by (metis (mono-tags, lifting) find-Some-iff)

```

```

lemma find-new-kind:
  assumes g' = add-node nid (node, s) g
  assumes node ≠ NoNode
  shows kind g' nid = node
  using assms
  using add-node-lookup by presburger

```

```

lemma find-new-stamp:
  assumes g' = add-node nid (node, s) g
  assumes node ≠ NoNode
  shows stamp g' nid = s
  using assms
  using add-node-lookup by presburger

```

```

lemma sorted-bottom:
  assumes finite xs
  assumes x ∈ xs
  shows x ≤ last(sorted-list-of-set(xs::nat set))
  using assms
  using sorted2-simps(2) sorted-list-of-set(2)
  by (smt (verit, del-insts) Diff-iff Max-ge Max-in empty-iff list.set(1) snoc-eq-iff-butlast
sorted-insort-is-snoc sorted-list-of-set(1) sorted-list-of-set.fold-insort-key.infinite sorted-list-of-set.fold-insort-k)

```

```

lemma fresh: finite xs ⇒ last(sorted-list-of-set(xs::nat set)) + 1 ∉ xs
  using sorted-bottom
  using not-le by auto

```

```

lemma fresh-ids:
  assumes n = get-fresh-id g
  shows n ∉ ids g
proof –
  have finite (ids g) using Rep-IRGraph by auto
  then show ?thesis
    using assms fresh unfolding get-fresh-id.simps
    by blast
qed

```

```

lemma graph-unchanged-rep-unchanged:
  assumes ∀ n ∈ ids g. kind g n = kind g' n
  assumes ∀ n ∈ ids g. stamp g n = stamp g' n
  shows (g ⊢ n ≃ e) ⇒ (g' ⊢ n ≃ e)
  apply (rule impI) subgoal premises e using e assms
    apply (induction n e)
      apply (metis no-encoding rep.ConstantNode)

```

```

    apply (metis no-encoding rep.ParameterNode)
    apply (metis no-encoding rep.ConditionalNode)
    apply (metis no-encoding rep.AbsNode)
    apply (metis no-encoding rep.NotNode)
    apply (metis no-encoding rep.NegateNode)
    apply (metis no-encoding rep.LogicNegationNode)
    apply (metis no-encoding rep.AddNode)
    apply (metis no-encoding rep.MulNode)
    apply (metis no-encoding rep.SubNode)
    apply (metis no-encoding rep.AndNode)
    apply (metis no-encoding rep.OrNode)
    apply (metis no-encoding rep.XorNode)
    apply (metis no-encoding rep.ShortCircuitOrNode)
    apply (metis no-encoding rep.LeftShiftNode)
    apply (metis no-encoding rep.RightShiftNode)
    apply (metis no-encoding rep.UnsignedRightShiftNode)
    apply (metis no-encoding rep.IntegerBelowNode)
    apply (metis no-encoding rep.IntegerEqualsNode)
    apply (metis no-encoding rep.IntegerLessThanNode)
    apply (metis no-encoding rep.NarrowNode)
    apply (metis no-encoding rep.SignExtendNode)
    apply (metis no-encoding rep.ZeroExtendNode)
    apply (metis no-encoding rep.LeafNode)
    by (metis no-encoding rep.RefNode)
done

```

lemma *fresh-node-subset*:

```

  assumes  $n \notin \text{ids } g$ 
  assumes  $g' = \text{add-node } n \ (k, s) \ g$ 
  shows  $\text{as-set } g \subseteq \text{as-set } g'$ 
  using assms
  by (smt (verit, del-Insts) Collect-mono-iff Diff-idemp Diff-insert-absorb add-changed
    as-set-def disjoint-change unchanged.simps)

```

lemma *unrep-subset*:

```

  assumes  $(g \oplus e \rightsquigarrow (g', n))$ 
  shows  $\text{as-set } g \subseteq \text{as-set } g'$ 
  using assms proof (induction  $g \ e \ (g', n)$  arbitrary:  $g' \ n$ )
  case (ConstantNodeSame  $g \ c \ n$ )
  then show ?case by blast
next
  case (ConstantNodeNew  $g \ c \ n \ g'$ )
  then show ?case using fresh-ids fresh-node-subset
    by presburger
next
  case (ParameterNodeSame  $g \ i \ s \ n$ )
  then show ?case by blast
next
  case (ParameterNodeNew  $g \ i \ s \ n \ g'$ )

```

```

    then show ?case using fresh-ids fresh-node-subset
      by presburger
  next
    case (ConditionalNodeSame g ce g2 c te g3 t fe g4 f s' n)
    then show ?case by blast
  next
    case (ConditionalNodeNew g ce g2 c te g3 t fe g4 f s' n g')
    then show ?case using fresh-ids fresh-node-subset
      by (meson subset-trans)
  next
    case (UnaryNodeSame g xe g2 x s' op n)
    then show ?case by blast
  next
    case (UnaryNodeNew g xe g2 x s' op n g')
    then show ?case using fresh-ids fresh-node-subset
      by (meson subset-trans)
  next
    case (BinaryNodeSame g xe g2 x ye g3 y s' op n)
    then show ?case by blast
  next
    case (BinaryNodeNew g xe g2 x ye g3 y s' op n g')
    then show ?case using fresh-ids fresh-node-subset
      by (meson subset-trans)
  next
    case (AllLeafNodes g n s)
    then show ?case by blast
qed

lemma fresh-node-preserves-other-nodes:
  assumes n' = get-fresh-id g
  assumes g' = add-node n' (k, s) g
  shows  $\forall n \in \text{ids } g. (g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$ 
  using assms
  by (smt (verit, ccfv-SIG) Diff-idemp Diff-insert-absorb add-changed disjoint-change
    fresh-ids graph-unchanged-rep-unchanged unchanged.elims(2))

lemma found-node-preserves-other-nodes:
  assumes find-node-and-stamp g (k, s) = Some n
  shows  $\forall n \in \text{ids } g. (g \vdash n \simeq e) \longleftrightarrow (g \vdash n \simeq e)$ 
  using assms
  by blast

lemma unrep-ids-subset[simp]:
  assumes  $g \oplus e \rightsquigarrow (g', n)$ 
  shows  $\text{ids } g \subseteq \text{ids } g'$ 
  using assms unrep-subset
  by (meson graph-refinement-def subset-refines)

lemma unrep-unchanged:

```

assumes $g \oplus e \rightsquigarrow (g', n)$
shows $\forall n \in \text{ids } g . \forall e . (g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$
using *assms unrep-subset fresh-node-preserves-other-nodes*
by (*meson subset-implies-evals*)

theorem *term-graph-reconstruction:*

$g \oplus e \rightsquigarrow (g', n) \implies (g' \vdash n \simeq e) \wedge \text{as-set } g \subseteq \text{as-set } g'$
subgoal premises e **apply** (*rule conjI*) **defer**
 using e *unrep-subset* **apply** *blast* **using** e
proof (*induction g e (g', n) arbitrary: g' n*)
 case (*ConstantNodeSame g' c n*)
 then have $\text{kind } g' n = \text{ConstantNode } c$
 using *find-exists-kind local.ConstantNodeSame* **by** *blast*
 then show $?case$ **using** *ConstantNode* **by** *blast*
next
 case (*ConstantNodeNew g c*)
 then show $?case$
 using *ConstantNode IRNode.distinct(683) add-node-lookup* **by** *presburger*
next
 case (*ParameterNodeSame i s*)
 then show $?case$
 by (*metis ParameterNode find-exists-kind find-exists-stamp*)
next
 case (*ParameterNodeNew g i s*)
 then show $?case$
 by (*metis IRNode.distinct(2447) ParameterNode add-node-lookup*)
next
 case (*ConditionalNodeSame g ce g2 c te g3 t fe g4 f s' n*)
 then have $k: \text{kind } g4 n = \text{ConditionalNode } c t f$
 using *find-exists-kind* **by** *blast*
 have $c: g4 \vdash c \simeq ce$ **using** *local.ConditionalNodeSame unrep-unchanged*
 using *no-encoding* **by** *blast*
 have $t: g4 \vdash t \simeq te$ **using** *local.ConditionalNodeSame unrep-unchanged*
 using *no-encoding* **by** *blast*
 have $f: g4 \vdash f \simeq fe$ **using** *local.ConditionalNodeSame unrep-unchanged*
 using *no-encoding* **by** *blast*
 then show $?case$ **using** $c t f$
 using *ConditionalNode k* **by** *blast*
next
 case (*ConditionalNodeNew g ce g2 c te g3 t fe g4 f s' n g'*)
 moreover have $\text{ConditionalNode } c t f \neq \text{NoNode}$
 using *unary-node.elims* **by** *blast*
 ultimately have $k: \text{kind } g' n = \text{ConditionalNode } c t f$
 using *find-new-kind local.ConditionalNodeNew*
 by *presburger*
 then have $c: g' \vdash c \simeq ce$ **using** *local.ConditionalNodeNew unrep-unchanged*
 using *no-encoding*
 by (*metis ConditionalNodeNew.hyps(9) fresh-node-preserves-other-nodes*)
 then have $t: g' \vdash t \simeq te$ **using** *local.ConditionalNodeNew unrep-unchanged*


```

    using no-encoding fresh-node-preserves-other-nodes
    by metis
  then have  $f: g' \vdash f \simeq fe$  using local.ConditionalNodeNew unrep-unchanged
    using no-encoding fresh-node-preserves-other-nodes
    by metis
  then show  $?case$  using  $c \ t \ f$ 
    using ConditionalNode k by blast
next
case (UnaryNodeSame  $g \ xe \ g' \ x \ s' \ op \ n$ )
then have  $k: kind \ g' \ n = unary-node \ op \ x$ 
  using find-exists-kind local.UnaryNodeSame by blast
then have  $g' \vdash x \simeq xe$  using local.UnaryNodeSame by blast
then show  $?case$  using  $k$ 
  apply (cases op)
  using AbsNode unary-node.simps(1) apply presburger
  using NegateNode unary-node.simps(3) apply presburger
  using NotNode unary-node.simps(2) apply presburger
  using LogicNegationNode unary-node.simps(4) apply presburger
  using NarrowNode unary-node.simps(5) apply presburger
  using SignExtendNode unary-node.simps(6) apply presburger
  using ZeroExtendNode unary-node.simps(7) by presburger
next
case (UnaryNodeNew  $g \ xe \ g2 \ x \ s' \ op \ n \ g'$ )
moreover have  $unary-node \ op \ x \neq NoNode$ 
  using unary-node.elims by blast
ultimately have  $k: kind \ g' \ n = unary-node \ op \ x$ 
  using find-new-kind local.UnaryNodeNew
  by presburger
have  $x \in ids \ g2$  using local.UnaryNodeNew
  using eval-contains-id by blast
then have  $x \neq n$  using local.UnaryNodeNew(5) fresh-ids by blast
have  $g' \vdash x \simeq xe$  using local.UnaryNodeNew fresh-node-preserves-other-nodes
  using  $\langle x \in ids \ g2 \rangle$  by blast
then show  $?case$  using  $k$ 
  apply (cases op)
  using AbsNode unary-node.simps(1) apply presburger
  using NegateNode unary-node.simps(3) apply presburger
  using NotNode unary-node.simps(2) apply presburger
  using LogicNegationNode unary-node.simps(4) apply presburger
  using NarrowNode unary-node.simps(5) apply presburger
  using SignExtendNode unary-node.simps(6) apply presburger
  using ZeroExtendNode unary-node.simps(7) by presburger
next
case (BinaryNodeSame  $g \ xe \ g2 \ x \ ye \ g3 \ y \ s' \ op \ n$ )
then have  $k: kind \ g3 \ n = bin-node \ op \ x \ y$ 
  using find-exists-kind by blast
have  $x: g3 \vdash x \simeq xe$  using local.BinaryNodeSame unrep-unchanged
  using no-encoding by blast
have  $y: g3 \vdash y \simeq ye$  using local.BinaryNodeSame unrep-unchanged

```

```

    using no-encoding by blast
  then show ?case using x y k apply (cases op)
    using AddNode bin-node.simps(1) apply presburger
    using MulNode bin-node.simps(2) apply presburger
    using SubNode bin-node.simps(3) apply presburger
    using AndNode bin-node.simps(4) apply presburger
    using OrNode bin-node.simps(5) apply presburger
    using XorNode bin-node.simps(6) apply presburger
    using ShortCircuitOrNode bin-node.simps(7) apply presburger
    using LeftShiftNode bin-node.simps(8) apply presburger
    using RightShiftNode bin-node.simps(9) apply presburger
    using UnsignedRightShiftNode bin-node.simps(10) apply presburger
    using IntegerEqualsNode bin-node.simps(11) apply presburger
    using IntegerLessThanNode bin-node.simps(12) apply presburger
    using IntegerBelowNode bin-node.simps(13) by presburger
next
  case (BinaryNodeNew g xe g2 x ye g3 y s' op n g')
  moreover have bin-node op x y ≠ NoNode
    using bin-node.elims by blast
  ultimately have k: kind g' n = bin-node op x y
    using find-new-kind local.BinaryNodeNew
    by presburger
  then have k: kind g' n = bin-node op x y
    using find-exists-kind by blast
  have x: g' ⊢ x ≃ xe using local.BinaryNodeNew unrep-unchanged
    using no-encoding
    by (meson fresh-node-preserves-other-nodes)
  have y: g' ⊢ y ≃ ye using local.BinaryNodeNew unrep-unchanged
    using no-encoding
    by (meson fresh-node-preserves-other-nodes)
  then show ?case using x y k apply (cases op)
    using AddNode bin-node.simps(1) apply presburger
    using MulNode bin-node.simps(2) apply presburger
    using SubNode bin-node.simps(3) apply presburger
    using AndNode bin-node.simps(4) apply presburger
    using OrNode bin-node.simps(5) apply presburger
    using XorNode bin-node.simps(6) apply presburger
    using ShortCircuitOrNode bin-node.simps(7) apply presburger
    using LeftShiftNode bin-node.simps(8) apply presburger
    using RightShiftNode bin-node.simps(9) apply presburger
    using UnsignedRightShiftNode bin-node.simps(10) apply presburger
    using IntegerEqualsNode bin-node.simps(11) apply presburger
    using IntegerLessThanNode bin-node.simps(12) apply presburger
    using IntegerBelowNode bin-node.simps(13) by presburger
next
  case (AllLeafNodes g n s)
  then show ?case using rep.LeafNode by blast
qed
done

```

lemma *ref-refinement*:

assumes $g \vdash n \simeq e_1$
assumes $\text{kind } g \ n' = \text{RefNode } n$
shows $g \vdash n' \sqsubseteq e_1$
using *assms RefNode*
by (*meson equal-refines graph-represents-expression-def*)

lemma *unrep-refines*:

assumes $g \oplus e \rightsquigarrow (g', n)$
shows *graph-refinement* $g \ g'$
using *assms*
using *graph-refinement-def subset-refines unrep-subset* **by** *blast*

lemma *add-new-node-refines*:

assumes $n \notin \text{ids } g$
assumes $g' = \text{add-node } n \ (k, s) \ g$
shows *graph-refinement* $g \ g'$
using *assms unfolding graph-refinement*
using *fresh-node-subset subset-refines* **by** *presburger*

lemma *add-node-as-set*:

assumes $g' = \text{add-node } n \ (k, s) \ g$
shows $\{\{n\} \sqsubseteq \text{as-set } g\} \subseteq \text{as-set } g'$
using *assms unfolding as-set-def domain-subtraction-def*
using *add-changed*
by (*smt (z3) case-prodE changeonly.simps mem-Collect-eq prod.sel(1) subsetI*)

theorem *refined-insert*:

assumes $e_1 \geq e_2$
assumes $g_1 \oplus e_2 \rightsquigarrow (g_2, n')$
shows $(g_2 \vdash n' \sqsubseteq e_1) \wedge \text{graph-refinement } g_1 \ g_2$
using *assms*
using *graph-construction term-graph-reconstruction* **by** *blast*

lemma *ids-finite*: *finite* (*ids* g)

using *Rep-IRGraph ids.rep-eq* **by** *simp*

lemma *unwrap-sorted*: *set* (*sorted-list-of-set* (*ids* g)) = *ids* g

using *Rep-IRGraph set-sorted-list-of-set ids-finite*
by *blast*

lemma *find-none*:

assumes *find-node-and-stamp* $g \ (k, s) = \text{None}$
shows $\forall n \in \text{ids } g. \text{kind } g \ n \neq k \vee \text{stamp } g \ n \neq s$

proof –

have $(\nexists n. n \in \text{ids } g \wedge (\text{kind } g \ n = k \wedge \text{stamp } g \ n = s))$
using *assms unfolding find-node-and-stamp.simps* **using** *find-None-iff un-*

```

wrap-sorted
  by (metis (mono-tags, lifting))
then show ?thesis
  by blast
qed

```

```

method ref-represents uses node =
  (metis IRNode.distinct(2755) RefNode dual-order.refl find-new-kind fresh-node-subset
  node subset-implies-evals)

```

3.8.5 Data-flow Tree to Subgraph Preserves Maximal Sharing

```

lemma same-kind-stamp-encodes-equal:
  assumes kind g n = kind g n'
  assumes stamp g n = stamp g n'
  assumes  $\neg(\text{is-preevaluated } (\text{kind } g \ n))$ 
  shows  $\forall e. (g \vdash n \simeq e) \longrightarrow (g \vdash n' \simeq e)$ 
  apply (rule allI)
  subgoal for e
    apply (rule impI)
    subgoal premises eval using eval assms
      apply (induction e)
    using ConstantNode apply presburger
    using ParameterNode apply presburger
      apply (metis ConditionalNode)
      apply (metis AbsNode)
      apply (metis NotNode)
      apply (metis NegateNode)
      apply (metis LogicNegationNode)
      apply (metis AddNode)
      apply (metis MulNode)
      apply (metis SubNode)
      apply (metis AndNode)
      apply (metis OrNode)
      apply (metis XorNode)
      apply (metis ShortCircuitOrNode)
      apply (metis LeftShiftNode)

```

```

      apply (metis RightShiftNode)
      apply (metis UnsignedRightShiftNode)
      apply (metis IntegerBelowNode)
      apply (metis IntegerEqualsNode)
      apply (metis IntegerLessThanNode)
      apply (metis NarrowNode)
      apply (metis SignExtendNode)
      apply (metis ZeroExtendNode)
    defer
      apply (metis RefNode)
    by blast
  done
done

```

lemma *new-node-not-present*:

```

  assumes find-node-and-stamp  $g$  (node, s) = None
  assumes  $n = \text{get-fresh-id } g$ 
  assumes  $g' = \text{add-node } n \text{ (node, s) } g$ 
  shows  $\forall n' \in \text{true-ids } g. (\forall e. ((g \vdash n \simeq e) \wedge (g \vdash n' \simeq e)) \longrightarrow n = n')$ 
  using assms
  using encode-in-ids fresh-ids by blast

```

lemma *true-ids-def*:

```

  true-ids  $g = \{n \in \text{ids } g. \neg(\text{is-RefNode (kind } g \text{ } n)) \wedge ((\text{kind } g \text{ } n) \neq \text{NoNode})\}$ 
  unfolding true-ids-def ids-def
  using ids-def is-RefNode-def by fastforce

```

lemma *add-node-some-node-def*:

```

  assumes  $k \neq \text{NoNode}$ 
  assumes  $g' = \text{add-node } nid \text{ (k, s) } g$ 
  shows  $g' = \text{Abs-IRGraph } ((\text{Rep-IRGraph } g)(nid \mapsto (k, s)))$ 
  using assms
  by (metis Rep-IRGraph-inverse add-node.rep-eq fst-conv)

```

lemma *ids-add-update-v1*:

```

  assumes  $g' = \text{add-node } nid \text{ (k, s) } g$ 
  assumes  $k \neq \text{NoNode}$ 
  shows  $\text{dom } (\text{Rep-IRGraph } g') = \text{dom } (\text{Rep-IRGraph } g) \cup \{nid\}$ 
  using assms ids.rep-eq add-node-some-node-def
  by (simp add: add-node.rep-eq)

```

lemma *ids-add-update-v2*:

```

  assumes  $g' = \text{add-node } nid \text{ (k, s) } g$ 
  assumes  $k \neq \text{NoNode}$ 
  shows  $nid \in \text{ids } g'$ 
  using assms
  using find-new-kind ids-some by presburger

```

lemma *add-node-ids-subset*:

```

assumes  $n \in \text{ids } g$ 
assumes  $g' = \text{add-node } n \text{ node } g$ 
shows  $\text{ids } g' = \text{ids } g \cup \{n\}$ 
using assms unfolding add-node-def
apply (cases fst node = NoNode)
using ids.rep-eq replace-node.rep-eq replace-node-def apply auto[1]
unfolding ids-def
by (smt (verit, best) Collect-cong Un-insert-right dom-fun-upd fst-conv fun-upd-apply
ids.rep-eq ids-def insert-absorb mem-Collect-eq option.inject option.simps(3) re-
place-node.rep-eq replace-node-def sup-bot.right-neutral)

```

lemma *convert-maximal*:

```

assumes  $\forall n \ n'. \ n \in \text{true-ids } g \wedge n' \in \text{true-ids } g \longrightarrow (\forall e \ e'. \ (g \vdash n \simeq e) \wedge (g$ 
 $\vdash n' \simeq e') \longrightarrow e \neq e')$ 
shows maximal-sharing  $g$ 
using assms
using maximal-sharing by blast

```

lemma *add-node-set-eq*:

```

assumes  $k \neq \text{NoNode}$ 
assumes  $n \notin \text{ids } g$ 
shows  $\text{as-set } (\text{add-node } n \ (k, s) \ g) = \text{as-set } g \cup \{(n, (k, s))\}$ 
using assms unfolding as-set-def add-node-def apply transfer apply simp
by blast

```

lemma *add-node-as-set-eq*:

```

assumes  $g' = \text{add-node } n \ (k, s) \ g$ 
assumes  $n \notin \text{ids } g$ 
shows  $(\{n\} \trianglelefteq \text{as-set } g') = \text{as-set } g$ 
using assms unfolding domain-subtraction-def
using add-node-set-eq
by (smt (z3) Collect-cong Rep-IRGraph-inverse UnCI UnE add-node.rep-eq as-set-def
case-prodE2 case-prodI2 le-boolE le-boolI' mem-Collect-eq prod.sel(1) singletonD
singletonI)

```

lemma *true-ids*:

```

 $\text{true-ids } g = \text{ids } g - \{n \in \text{ids } g. \text{ is-RefNode } (\text{kind } g \ n)\}$ 
unfolding true-ids-def
by fastforce

```

lemma *as-set-ids*:

```

assumes  $\text{as-set } g = \text{as-set } g'$ 
shows  $\text{ids } g = \text{ids } g'$ 
using assms
by (metis antisym equalityD1 graph-refinement-def subset-refines)

```

lemma *ids-add-update*:

```

assumes  $k \neq \text{NoNode}$ 
assumes  $n \notin \text{ids } g$ 

```

```

assumes  $g' = \text{add-node } n \ (k, s) \ g$ 
shows  $\text{ids } g' = \text{ids } g \cup \{n\}$ 
using assms apply (subst assms(3)) using add-node-set-eq as-set-ids
by (smt (verit, del-insts) Collect-cong Diff-idemp Diff-insert-absorb Un-commute
add-node.rep-eq add-node-def ids.rep-eq ids-add-update-v1 ids-add-update-v2 insertE
insert-Collect insert-is-Un map-upd-Some-unfold mem-Collect-eq replace-node-def
replace-node-unchanged)

```

lemma *true-ids-add-update*:

```

assumes  $k \neq \text{NoNode}$ 
assumes  $n \notin \text{ids } g$ 
assumes  $g' = \text{add-node } n \ (k, s) \ g$ 
assumes  $\neg(\text{is-RefNode } k)$ 
shows  $\text{true-ids } g' = \text{true-ids } g \cup \{n\}$ 
using assms using true-ids ids-add-update
by (smt (z3) Collect-cong Diff-iff Diff-insert-absorb Un-commute add-node-def
find-new-kind insert-Diff-if insert-is-Un mem-Collect-eq replace-node-def replace-node-unchanged)

```

lemma *new-def*:

```

assumes  $(\text{new} \sqsubseteq \text{as-set } g') = \text{as-set } g$ 
shows  $n \in \text{ids } g \longrightarrow n \notin \text{new}$ 
using assms
by (smt (z3) as-set-def case-prodD domain-subtraction-def mem-Collect-eq)

```

lemma *add-preserves-rep*:

```

assumes unchanged:  $(\text{new} \sqsubseteq \text{as-set } g') = \text{as-set } g$ 
assumes closed: wf-closed g
assumes existed:  $n \in \text{ids } g$ 
assumes  $g' \vdash n \simeq e$ 
shows  $g \vdash n \simeq e$ 
proof (cases  $n \in \text{new}$ )
  case True
    have  $n \notin \text{ids } g$ 
      using unchanged True unfolding as-set-def domain-subtraction-def
      by blast
    then show ?thesis using existed by simp
  next
    case False
      then have kind-eq:  $\forall n'. n' \notin \text{new} \longrightarrow \text{kind } g \ n' = \text{kind } g' \ n'$ 
        — can be more general than stamp_eq because NoNode default is equal
      using unchanged not-excluded-keep-type
      by (smt (z3) case-prodE domain-subtraction-def ids-some mem-Collect-eq sub-setI)
      from False have stamp-eq:  $\forall n' \in \text{ids } g'. n' \notin \text{new} \longrightarrow \text{stamp } g \ n' = \text{stamp } g' \ n'$ 
      using unchanged not-excluded-keep-type
      by (metis equalityE)

```

```

show ?thesis using assms(4) kind-eq stamp-eq False
proof (induction n e rule: rep.induct)
  case (ConstantNode n c)
  then show ?case
    using rep.ConstantNode kind-eq by presburger
next
  case (ParameterNode n i s)
  then show ?case
    using rep.ParameterNode
    by (metis no-encoding)
next
  case (ConditionalNode n c t f ce te fe)
  have kind: kind g n = ConditionalNode c t f
    using ConditionalNode.hyps(1) ConditionalNode.premis(3) kind-eq by pres-
    burger
  then have isin:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{c, t, f\} = \text{inputs } g \ n$ 
    using kind unfolding inputs.simps using inputs-of-ConditionalNode by simp
  have  $c \in \text{ids } g \wedge t \in \text{ids } g \wedge f \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have  $c \notin \text{new} \wedge t \notin \text{new} \wedge f \notin \text{new}$ 
    using new-def unchanged by blast
  then show ?case using ConditionalNode apply simp
    using rep.ConditionalNode by presburger
next
  case (AbsNode n x xe)
  then have kind: kind g n = AbsNode x
    by simp
  then have isin:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{x\} = \text{inputs } g \ n$ 
    using kind unfolding inputs.simps by simp
  have  $x \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have  $x \notin \text{new}$ 
    using new-def unchanged by blast
  then show ?case
    using AbsNode
    using rep.AbsNode by presburger
next
  case (NotNode n x xe)
  then have kind: kind g n = NotNode x
    by simp
  then have isin:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{x\} = \text{inputs } g \ n$ 

```



```

    using kind unfolding inputs.simps by simp
  have  $x \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have  $x \notin \text{new}$ 
    using new-def unchanged by blast
  then show ?case using NotNode
    using rep.NotNode by presburger
next
case (NegateNode n x xe)
  then have kind:  $\text{kind } g \ n = \text{NegateNode } x$ 
    by simp
  then have isin:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{x\} = \text{inputs } g \ n$ 
    using kind unfolding inputs.simps by simp
  have  $x \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have  $x \notin \text{new}$ 
    using new-def unchanged by blast
  then show ?case using NegateNode
    using rep.NegateNode by presburger
next
case (LogicNegationNode n x xe)
  then have kind:  $\text{kind } g \ n = \text{LogicNegationNode } x$ 
    by simp
  then have isin:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{x\} = \text{inputs } g \ n$ 
    using kind unfolding inputs.simps by simp
  have  $x \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have  $x \notin \text{new}$ 
    using new-def unchanged by blast
  then show ?case using LogicNegationNode
    using rep.LogicNegationNode by presburger
next
case (AddNode n x y xe ye)
  then have kind:  $\text{kind } g \ n = \text{AddNode } x \ y$ 
    by simp
  then have isin:  $n \in \text{ids } g$ 
    by simp
  have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
    using kind unfolding inputs.simps by simp
  have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
    using closed unfolding wf-closed-def
    using isin inputs by blast

```

```

then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using AddNode
  using rep.AddNode by presburger
next
case (MulNode  $n\ x\ y\ xe\ ye$ )
then have kind:  $\text{kind } g\ n = \text{MulNode } x\ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g\ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using MulNode
  using rep.MulNode by presburger
next
case (SubNode  $n\ x\ y\ xe\ ye$ )
then have kind:  $\text{kind } g\ n = \text{SubNode } x\ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g\ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using SubNode
  using rep.SubNode by presburger
next
case (AndNode  $n\ x\ y\ xe\ ye$ )
then have kind:  $\text{kind } g\ n = \text{AndNode } x\ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g\ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using AndNode
  using rep.AndNode by presburger

```

```

next
  case (OrNode n x y xe ye)
  then have kind: kind g n = OrNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using OrNode
    using rep.OrNode by presburger
next
  case (XorNode n x y xe ye)
  then have kind: kind g n = XorNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using XorNode
    using rep.XorNode by presburger
next
  case (ShortCircuitOrNode n x y xe ye)
  then have kind: kind g n = ShortCircuitOrNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using ShortCircuitOrNode
    using rep.ShortCircuitOrNode by presburger
next
  case (LeftShiftNode n x y xe ye)
  then have kind: kind g n = LeftShiftNode x y
    by simp

```

```

then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using LeftShiftNode
  using rep.LeftShiftNode by presburger
next
case (RightShiftNode  $n \ x \ y \ x_e \ y_e$ )
then have kind:  $\text{kind } g \ n = \text{RightShiftNode } x \ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using RightShiftNode
  using rep.RightShiftNode by presburger
next
case (UnsignedRightShiftNode  $n \ x \ y \ x_e \ y_e$ )
then have kind:  $\text{kind } g \ n = \text{UnsignedRightShiftNode } x \ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using UnsignedRightShiftNode
  using rep.UnsignedRightShiftNode by presburger
next
case (IntegerBelowNode  $n \ x \ y \ x_e \ y_e$ )
then have kind:  $\text{kind } g \ n = \text{IntegerBelowNode } x \ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp

```

```

have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using IntegerBelowNode
  using rep.IntegerBelowNode by presburger
next
case (IntegerEqualsNode  $n \ x \ y \ xe \ ye$ )
then have kind:  $\text{kind } g \ n = \text{IntegerEqualsNode } x \ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using IntegerEqualsNode
  using rep.IntegerEqualsNode by presburger
next
case (IntegerLessThanNode  $n \ x \ y \ xe \ ye$ )
then have kind:  $\text{kind } g \ n = \text{IntegerLessThanNode } x \ y$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x, y\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g \wedge y \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new} \wedge y \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using IntegerLessThanNode
  using rep.IntegerLessThanNode by presburger
next
case (NarrowNode  $n \ \text{inputBits} \ \text{resultBits} \ x \ xe$ )
then have kind:  $\text{kind } g \ n = \text{NarrowNode } \text{inputBits} \ \text{resultBits} \ x$ 
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new}$ 

```

```

    using new-def unchanged by blast
  then show ?case using NarrowNode
    using rep.NarrowNode by presburger
next
case (SignExtendNode n inputBits resultBits x xe)
then have kind: kind g n = SignExtendNode inputBits resultBits x
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using SignExtendNode
  using rep.SignExtendNode by presburger
next
case (ZeroExtendNode n inputBits resultBits x xe)
then have kind: kind g n = ZeroExtendNode inputBits resultBits x
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{x\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $x \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $x \notin \text{new}$ 
  using new-def unchanged by blast
then show ?case using ZeroExtendNode
  using rep.ZeroExtendNode by presburger
next
case (LeafNode n s)
then show ?case
  by (metis no-encoding rep.LeafNode)
next
case (RefNode n n' e)
then have kind: kind g n = RefNode n'
  by simp
then have isin:  $n \in \text{ids } g$ 
  by simp
have inputs:  $\{n'\} = \text{inputs } g \ n$ 
  using kind unfolding inputs.simps by simp
have  $n' \in \text{ids } g$ 
  using closed unfolding wf-closed-def
  using isin inputs by blast
then have  $n' \notin \text{new}$ 

```

```

      using new-def unchanged by blast
    then show ?case
      using RefNode
      using rep.RefNode by presburger
  qed
qed

```

```

lemma not-in-no-rep:
   $n \notin \text{ids } g \implies \forall e. \neg(g \vdash n \simeq e)$ 
  using eval-contains-id by blast

```

```

lemma unary-inputs:
  assumes kind g n = unary-node op x
  shows inputs g n = {x}
  using assms by (cases op; auto)

```

```

lemma unary-succ:
  assumes kind g n = unary-node op x
  shows succ g n = {}
  using assms by (cases op; auto)

```

```

lemma binary-inputs:
  assumes kind g n = bin-node op x y
  shows inputs g n = {x, y}
  using assms by (cases op; auto)

```

```

lemma binary-succ:
  assumes kind g n = bin-node op x y
  shows succ g n = {}
  using assms by (cases op; auto)

```

```

lemma unrep-contains:
  assumes  $g \oplus e \rightsquigarrow (g', n)$ 
  shows  $n \in \text{ids } g'$ 
  using assms
  using not-in-no-rep term-graph-reconstruction by blast

```

```

lemma unrep-preserves-contains:
  assumes  $n \in \text{ids } g$ 
  assumes  $g \oplus e \rightsquigarrow (g', n')$ 
  shows  $n \in \text{ids } g'$ 
  using assms
  by (meson subsetD unrep-ids-subset)

```

```

lemma unrep-preserves-closure:
  assumes wf-closed g
  assumes  $g \oplus e \rightsquigarrow (g', n)$ 

```

```

shows wf-closed g'
using assms(2,1) unfolding wf-closed-def
proof (induction g e (g', n) arbitrary: g' n)
  case (ConstantNodeSame g c n)
  then show ?case
    by blast
next
  case (ConstantNodeNew g c n g')
  then have dom:  $ids\ g' = ids\ g \cup \{n\}$ 
    by (meson IRNode.distinct(683) add-node-ids-subset ids-add-update)
  have k:  $kind\ g'\ n = ConstantNode\ c$ 
    using ConstantNodeNew add-node-lookup by simp
  then have inp:  $\{\} = inputs\ g'\ n$ 
    unfolding inputs.simps by simp
  from k have suc:  $\{\} = succ\ g'\ n$ 
    unfolding succ.simps by simp
  have  $inputs\ g'\ n \subseteq ids\ g' \wedge succ\ g'\ n \subseteq ids\ g' \wedge kind\ g'\ n \neq NoNode$ 
    using inp suc k by simp
  then show ?case
    by (smt (verit) ConstantNodeNew.hyps(3) ConstantNodeNew.prem Un-insert-right
    add-changed changeonly.elims(2) dom inputs.simps insert-iff singleton-iff subset-insertI
    subset-trans succ.simps sup-bot-right)
next
  case (ParameterNodeSame g i s n)
  then show ?case by blast
next
  case (ParameterNodeNew g i s n g')
  then have dom:  $ids\ g' = ids\ g \cup \{n\}$ 
    using IRNode.distinct(2447) fresh-ids ids-add-update by presburger
  have k:  $kind\ g'\ n = ParameterNode\ i$ 
    using ParameterNodeNew add-node-lookup by simp
  then have inp:  $\{\} = inputs\ g'\ n$ 
    unfolding inputs.simps by simp
  from k have suc:  $\{\} = succ\ g'\ n$ 
    unfolding succ.simps by simp
  have  $inputs\ g'\ n \subseteq ids\ g' \wedge succ\ g'\ n \subseteq ids\ g' \wedge kind\ g'\ n \neq NoNode$ 
    using k inp suc by simp
  then show ?case
    by (smt (verit) ParameterNodeNew.hyps(3) ParameterNodeNew.prem Un-insert-right
    add-node-as-set dom inputs.elims insertE not-excluded-keep-type order-trans single-
    tonD subset-insertI succ.elims sup-bot-right)
next
  case (ConditionalNodeSame g ce g2 c te g3 t fe g4 f s' n)
  then show ?case by blast
next
  case (ConditionalNodeNew g ce g2 c te g3 t fe g4 f s' n g')
  then have dom:  $ids\ g' = ids\ g4 \cup \{n\}$ 
    by (meson IRNode.distinct(591) add-node-ids-subset ids-add-update)
  have k:  $kind\ g'\ n = ConditionalNode\ c\ t\ f$ 

```



```

    using ConditionalNodeNew add-node-lookup by simp
  then have inp:  $\{c, t, f\} = \text{inputs } g' n$ 
    unfolding inputs.simps by simp
  from k have suc:  $\{\} = \text{succ } g' n$ 
    unfolding succ.simps by simp
  have inputs  $g' n \subseteq \text{ids } g' \wedge \text{succ } g' n \subseteq \text{ids } g' \wedge \text{kind } g' n \neq \text{NoNode}$ 
    using k inp suc unrep-contains unrep-preserves-contains
    using ConditionalNodeNew(1,3,5,10)
    by (smt (verit) IRNode.simps(643) Un-insert-right bot.extremum dom insert-absorb insert-subset subset-insertI sup-bot-right)
  then show ?case using dom
    by (smt (z3) ConditionalNodeNew.hyps(10) ConditionalNodeNew.hyps(2)
        ConditionalNodeNew.hyps(4) ConditionalNodeNew.hyps(6) ConditionalNodeNew.prem
        Diff-eq-empty-iff Diff-iff Un-insert-right Un-upper1 add-node-def inputs.simps insertE
        replace-node-def replace-node-unchanged subset-trans succ.simps sup-bot-right)
  next
    case (UnaryNodeSame g xe g2 x s' op n)
    then show ?case by blast
  next
    case (UnaryNodeNew g xe g2 x s' op n g')
    then have dom:  $\text{ids } g' = \text{ids } g2 \cup \{n\}$ 
      by (metis add-node-ids-subset add-node-lookup ids-add-update ids-some unrep.UnaryNodeNew
        unrep-contains)
    have k:  $\text{kind } g' n = \text{unary-node op } x$ 
      using UnaryNodeNew add-node-lookup
      by (metis fresh-ids ids-some)
    then have inp:  $\{x\} = \text{inputs } g' n$ 
      using unary-inputs by simp
    from k have suc:  $\{\} = \text{succ } g' n$ 
      using unary-succ by simp
    have inputs  $g' n \subseteq \text{ids } g' \wedge \text{succ } g' n \subseteq \text{ids } g' \wedge \text{kind } g' n \neq \text{NoNode}$ 
      using k inp suc unrep-contains unrep-preserves-contains
      using UnaryNodeNew(1,6)
    by (metis Un-upper1 dom empty-subsetI ids-some insert-not-empty insert-subsetI
        not-in-g-inputs subset-iff)
    then show ?case
      by (smt (verit) Un-insert-right UnaryNodeNew.hyps(2) UnaryNodeNew.hyps(6)
          UnaryNodeNew.prem
          add-changed changeonly.elims(2) dom inputs.simps insert-iff
          singleton-iff subset-insertI subset-trans succ.simps sup-bot-right)
  next
    case (BinaryNodeSame g xe g2 x ye g3 y s' op n)
    then show ?case by blast
  next
    case (BinaryNodeNew g xe g2 x ye g3 y s' op n g')
    then have dom:  $\text{ids } g' = \text{ids } g3 \cup \{n\}$ 
      by (metis binary-inputs fresh-ids ids-add-update ids-some insert-not-empty
        not-in-g-inputs)
    have k:  $\text{kind } g' n = \text{bin-node op } x y$ 
      using BinaryNodeNew add-node-lookup

```

```

    by (metis fresh-ids ids-some)
  then have inp:  $\{x, y\} = \text{inputs } g' \ n$ 
    using binary-inputs by simp
  from k have suc:  $\{\} = \text{succ } g' \ n$ 
    using binary-succ by simp
  have inputs  $g' \ n \subseteq \text{ids } g' \wedge \text{succ } g' \ n \subseteq \text{ids } g' \wedge \text{kind } g' \ n \neq \text{NoNode}$ 
    using k inp suc unrep-contains unrep-preserves-contains
    using BinaryNodeNew(1,3,6)
  by (metis Un-upper1 dom empty-subsetI ids-some insert-not-empty insert-subsetI
not-in-g-inputs subset-iff)
  then show ?case using dom BinaryNodeNew
    by (smt (verit, del-Insts) Diff-eq-empty-iff Diff-iff Un-insert-right Un-upper1
add-node-def inputs.simps insertE replace-node-def replace-node-unchanged subset-trans
succ.simps sup-bot-right)
  next
    case (AllLeafNodes g n s)
    then show ?case
      by blast
  qed

```

inductive-cases *ConstUnrepE*: $g \oplus (\text{ConstantExpr } x) \rightsquigarrow (g', n)$

definition *constant-value* **where**

constant-value = (IntVal 32 0)

definition *bad-graph* **where**

bad-graph = irgraph [
 (0, AbsNode 1, constantAsStamp constant-value),
 (1, RefNode 2, constantAsStamp constant-value),
 (2, ConstantNode constant-value, constantAsStamp constant-value)
]

end

3.9 Control-flow Semantics Theorems

theory *IRStepThms*

imports

IRStepObj

TreeToGraphThms

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.

3.9.1 Control-flow Step is Deterministic

theorem *stepDet*:

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

proof (*induction rule: step.induct*)

case (*SequentialNode nid next m h*)

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

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

by (*metis is-IfNode-def*)

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

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

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

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

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

by (*metis is-NewInstanceNode-def*)

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

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

by (*metis is-LoadFieldNode-def*)

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

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

by (*metis is-StoreFieldNode-def*)

have *notdivrem*: $\neg(is_IntegerDivRemNode\ (kind\ g\ nid))$

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

is-SignedRemNode-def

by (*metis is-IntegerDivRemNode.simps*)

from *notif notend notnew notload notstore notdivrem*

show *?case using SequentialNode step.cases*

by (*smt (z3) IRNode.disc(1028) IRNode.disc(2270) IRNode.discI(31) Pair-inject*

is-sequential-node.simps(18) is-sequential-node.simps(43) is-sequential-node.simps(44))

next

case (*IfNode nid cond tb fb m val next h*)

then have *notseq*: $\neg(is_sequential-node\ (kind\ g\ nid))$

using *is-sequential-node.simps is-AbstractMergeNode.simps*

by (*simp add: IfNode.hyps(1)*)

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

using *is-AbstractEndNode.simps*

by (*simp add: IfNode.hyps(1)*)

have *notdivrem*: $\neg(is_IntegerDivRemNode\ (kind\ g\ nid))$

using *is-AbstractEndNode.simps*

by (*simp add: IfNode.hyps(1)*)

from *notseq notend notdivrem show ?case using IfNode repDet evalDet 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*: $\neg(is_sequential-node\ (kind\ g\ nid))$

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

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

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

```

    using EndNodes.hyps(1) is-IfNode-def is-AbstractEndNode.elims
    by (metis IRNode.distinct-disc(1058) is-EndNode.simps(12))
  have notref: ¬(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.distinct(1028) IRNode.distinct(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(2)
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(2)
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 Stat-
icStoreFieldNode.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:

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

using SequentialNode

by (metis IRNodes.successors-of-RefNode is-sequential-node.simps(7) nth-Cons-0)

```

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

lemma IfNodeSeq:
  shows  $\text{kind } g \text{ nid} = \text{IfNode cond tb fb} \longrightarrow \neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 
  unfolding is-sequential-node.simps
  using is-sequential-node.simps(18) by presburger

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

lemma step-in-ids:
  assumes  $g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m', h')$ 
  shows  $\text{nid} \in \text{ids } g$ 
  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

```