Veriopt Theories

April 17, 2024

Contents

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 calculates during the traversal of the control flow graph.

As a concrete example, as the *SignedDivNode*::$'a$ 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*::$'a$ calculates the value of a node and maps the node identifier to the value within the method state. The data-flow semantics then just reads the value stored in the method state for the node.

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

definition *new-map-state* :: *MapState* **where**
  new-map-state = (λ*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*)
| *UnaryIsNull*
| *UnaryReverseBytes*
| *UnaryBitCount*

datatype *IRBinaryOp* =
BinAdd
| *BinSub*
| *BinMul*
| *BinDiv*
| *BinMod*
| *BinAnd*
| *BinOr*
| *BinXor*
| *BinShortCircuitOr*
| *BinLeftShift*
| *BinRightShift*
| *BinURightShift*
| *BinIntegerEquals*
| *BinIntegerLessThan*
| *BinIntegerBelow*
| *BinIntegerTest*
| *BinIntegerNormalizeCompare*
| *BinIntegerMulHigh*

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.literal*)
| *VariableExpr* (*ir-name*: *String.literal*) (*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-normal* :: *IRBinaryOp set* **where**
binary-normal ≡ {*BinAdd*, *BinMul*, *BinDiv*, *BinMod*, *BinSub*, *BinAnd*, *BinOr*, *BinXor*}

abbreviation *binary-fixed-32-ops* :: *IRBinaryOp set* **where**
binary-fixed-32-ops ≡ {*BinShortCircuitOr*, *BinIntegerEquals*, *BinIntegerLessThan*, *BinIntegerBelow*, *BinIntegerTest*, *BinIntegerNormalizeCompare*}

abbreviation *binary-shift-ops* :: *IRBinaryOp set* **where**
binary-shift-ops ≡ {*BinLeftShift*, *BinRightShift*, *BinURightShift*}

abbreviation *binary-fixed-ops* :: *IRBinaryOp set* **where**
binary-fixed-ops ≡ {*BinIntegerMulHigh*}

abbreviation *normal-unary* :: *IRUnaryOp set* **where**
normal-unary ≡ {*UnaryAbs*, *UnaryNeg*, *UnaryNot*, *UnaryLogicNegation*, *UnaryReverseBytes*}

abbreviation *unary-fixed-32-ops* :: *IRUnaryOp set* **where**
unary-fixed-32-ops ≡ {*UnaryBitCount*}

abbreviation *boolean-unary* :: *IRUnaryOp set* **where**
boolean-unary ≡ {*UnaryIsNull*}

lemma *binary-ops-all*:
shows $op \in$ *binary-normal* $\vee$ $op \in$ *binary-fixed-32-ops* $\vee$ $op \in$ *binary-fixed-ops* $\vee$ $op \in$ *binary-shift-ops*
by (*cases op*; *auto*)

lemma *binary-ops-distinct-normal*:
shows $op \in$ *binary-normal* $\Longrightarrow$ $op \notin$ *binary-fixed-32-ops* $\wedge$ $op \notin$ *binary-fixed-ops* $\wedge$ $op \notin$ *binary-shift-ops*
by *auto*

lemma *binary-ops-distinct-fixed-32*:
shows $op \in$ *binary-fixed-32-ops* $\Longrightarrow$ $op \notin$ *binary-normal* $\wedge$ $op \notin$ *binary-fixed-ops* $\wedge$ $op \notin$ *binary-shift-ops*
by *auto*

lemma *binary-ops-distinct-fixed*:
shows $op \in$ *binary-fixed-ops* $\Longrightarrow$ $op \notin$ *binary-fixed-32-ops* $\wedge$ $op \notin$ *binary-normal* $\wedge$ $op \notin$ *binary-shift-ops*
by *auto*

lemma *binary-ops-distinct-shift*:
shows $op \in$ *binary-shift-ops* $\Longrightarrow$ $op \notin$ *binary-fixed-32-ops* $\wedge$ $op \notin$ *binary-fixed-ops* $\wedge$ $op \notin$ *binary-normal*
by *auto*

lemma *unary-ops-distinct*:
shows $op \in$ *normal-unary* $\Longrightarrow$ $op \notin$ *boolean-unary* $\wedge$ $op \notin$ *unary-fixed-32-ops*
and $op \in$ *boolean-unary* $\Longrightarrow$ $op \notin$ *normal-unary* $\wedge$ $op \notin$ *unary-fixed-32-ops*
and $op \in$ *unary-fixed-32-ops* $\Longrightarrow$ $op \notin$ *boolean-unary* $\wedge$ $op \notin$ *normal-unary*
by *auto*

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

stamp-unary UnaryIsNull - = (*IntegerStamp 32 0 1*) |
stamp-unary op (*IntegerStamp b lo hi*) =
unrestricted-stamp (*IntegerStamp*
(*if op* $\in$ *normal-unary then b else*
if op $\in$ *boolean-unary then 32 else*
if op $\in$ *unary-fixed-32-ops then 32 else*
(*ir-resultBits op*)) *lo hi*) |

stamp-unary op - = IllegalStamp

fun *stamp-binary* :: *IRBinaryOp* $\Rightarrow$ *Stamp* $\Rightarrow$ *Stamp* $\Rightarrow$ *Stamp* **where**
stamp-binary op (*IntegerStamp b1 lo1 hi1*) (*IntegerStamp b2 lo2 hi2*) =
(*if op* $\in$ *binary-shift-ops then unrestricted-stamp* (*IntegerStamp b1 lo1 hi1*)
else if b1 $\neq$ *b2 then IllegalStamp else*
(*if op* $\in$ *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* ⇒ *Value* ⇒ *Value* **where**
unary-eval UnaryAbs v = *intval-abs v* |
unary-eval UnaryNeg v = *intval-negate v* |
unary-eval UnaryNot v = *intval-not v* |
unary-eval UnaryLogicNegation 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* |
unary-eval UnaryIsNull v = *intval-is-null v* |
unary-eval UnaryReverseBytes v = *intval-reverse-bytes v* |
unary-eval UnaryBitCount v = *intval-bit-count v*

fun *bin-eval* :: *IRBinaryOp* ⇒ *Value* ⇒ *Value* ⇒ *Value* **where**
bin-eval BinAdd v1 v2 = *intval-add v1 v2* |
bin-eval BinSub v1 v2 = *intval-sub v1 v2* |
bin-eval BinMul v1 v2 = *intval-mul v1 v2* |
bin-eval BinDiv v1 v2 = *intval-div v1 v2* |
bin-eval BinMod v1 v2 = *intval-mod 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* |

bin-eval BinIntegerTest v1 v2 = *intval-test v1 v2* |
bin-eval BinIntegerNormalizeCompare v1 v2 = *intval-normalize-compare v1 v2* |
bin-eval BinIntegerMulHigh v1 v2 = *intval-mul-high v1 v2*

lemma *defined-eval-is-intval*:
shows *bin-eval op x y* ≠ *UndefVal* ⟹ (*is-IntVal x* ∧ *is-IntVal y*)
by (*cases op*; *cases x*; *cases y*; *auto*)

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

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

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

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

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

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

[*m*,*p*] ⊢ *te* ↦ *true*; *true* ≠ *UndefVal*;
[*m*,*p*] ⊢ *fe* ↦ *false*; *false* ≠ *UndefVal*⟧
⟹ [*m*,*p*] ⊢ (*ConditionalExpr ce te fe*) ↦ *result* |

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

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

LeafExpr:
⟦*val* = *m n*;
valid-value val s⟧
⟹ [*m*,*p*] ⊢ *LeafExpr n s* ↦ *val*

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

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

EvalNil:
[*m*,*p*] ⊢ [] [↦] [] |

EvalCons:
⟦[*m*,*p*] ⊢ *x* ↦ *xval*;
[*m*,*p*] ⊢ *yy* [↦] *yyval*⟧
⟹ [*m*,*p*] ⊢ (*x*#*yy*) [↦] (*xval*#*yyval*)

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

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

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

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

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

notation *less-eq* (**infix** ⊑ *65*)

definition
  le-expr-def [*simp*]:
    $(e_2 \leq e_1) \longleftrightarrow (\forall\ m\ p\ v.\ (([m,p] \vdash e_1 \mapsto v) \longrightarrow ([m,p] \vdash e_2 \mapsto v)))$

definition
  lt-expr-def [*simp*]:
    $(e_1 < e_2) \longleftrightarrow (e_1 \leq e_2 \wedge \neg\ (e_1 \doteq e_2))$

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

end

abbreviation (**output**) *Refines* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* (**infix** ⊒ *64*)
  where $e_1 \sqsupseteq e_2 \equiv (e_2 \leq e_1)$

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 a 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 ⇒ int64 (↑)
  fixes down :: IRExpr ⇒ int64 (↓)
  assumes up-spec: [m, p] ⊢ e ↦ IntVal b v ⟹ (and v (not ((ucast (↑e))))) = 0
      and down-spec: [m, p] ⊢ e ↦ IntVal b v ⟹ (and (not v) (ucast (↓e))) = 0
begin

lemma may-implies-either:
  [m, p] ⊢ e ↦ IntVal b v ⟹ bit (↑e) n ⟹ bit v n = False ∨ bit v n = True
  by simp

lemma not-may-implies-false:
  [m, p] ⊢ e ↦ IntVal b v ⟹ ¬(bit (↑e) n) ⟹ bit v n = False
 by (metis (no-types, lifting) bit.double-compl up-spec bit-and-iff bit-not-iff bit-unsigned-iff

    down-spec)

lemma must-implies-true:
  [m, p] ⊢ e ↦ IntVal b v ⟹ bit (↓e) n ⟹ bit v n = True
 by (metis bit.compl-one bit-and-iff bit-minus-1-iff bit-not-iff impossible-bit ucast-id
down-spec)

lemma not-must-implies-either:
  [m, p] ⊢ e ↦ IntVal b v ⟹ ¬(bit (↓e) n) ⟹ bit v n = False ∨ bit v n = True
  by simp

lemma must-implies-may:
  [m, p] ⊢ e ↦ IntVal b v ⟹ n < 32 ⟹ bit (↓e) n ⟹ bit (↑e) n
  by (meson must-implies-true not-may-implies-false)

lemma up-mask-and-zero-implies-zero:
  assumes and (↑x) (↑y) = 0
  assumes [m, p] ⊢ x ↦ IntVal b xv
  assumes [m, p] ⊢ y ↦ IntVal b yv
  shows and xv yv = 0
 by (smt (z3) assms and.commute and.right-neutral bit.compl-zero bit.conj-cancel-right
ucast-id
   bit.conj-disj-distribs(1) up-spec word-bw-assocs(1) word-not-dist(2) word-ao-absorbs(8)
    and-eq-not-not-or)
```

lemma *not-down-up-mask-and-zero-implies-zero*:
assumes *and* (*not* (↓*x*)) (↑*y*) = *0*
assumes [*m*, *p*] ⊢ *x* ↦ *IntVal b xv*
assumes [*m*, *p*] ⊢ *y* ↦ *IntVal b yv*
shows *and xv yv* = *yv*
by (*metis* (*no-types*, *opaque-lifting*) *assms bit.conj-cancel-left bit.conj-disj-distribs*(*1*,*2*) *bit.de-Morgan-disj ucast-id down-spec or-eq-not-not-and up-spec word-ao-absorbs*(*2*,*8*) *word-bw-lcs*(*1*) *word-not-dist*(*2*))

end

definition *IRExpr-up* :: *IRExpr* ⇒ *int64* **where**
IRExpr-up e = *not 0*

definition *IRExpr-down* :: *IRExpr* ⇒ *int64* **where**
IRExpr-down e = *0*

lemma *ucast-zero*: (*ucast* (*0*::*int64*)::*int32*) = *0*
by *simp*

lemma *ucast-minus-one*: (*ucast* (−*1*::*int64*)::*int32*) = −*1*
apply *transfer* **by** *auto*

interpretation *simple-mask*: *stamp-mask*
IRExpr-up :: *IRExpr* ⇒ *int64*
IRExpr-down :: *IRExpr* ⇒ *int64*
apply *unfold-locales*
by (*simp add*: *ucast-minus-one IRExpr-up-def IRExpr-down-def*)+

end

2 Tree to Graph

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

2.1 Subgraph to Data-flow Tree

fun *find-node-and-stamp* :: *IRGraph* ⇒ (*IRNode* × *Stamp*) ⇒ *ID option* **where**
find-node-and-stamp g (*n*,*s*) =
find (λ*i*. *kind g i* = *n* ∧ *stamp g i* = *s*) (*sorted-list-of-set*(*ids g*))

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

fun *is-preevaluated* :: *IRNode* ⇒ *bool* **where**
is-preevaluated (*InvokeNode n - - - - -*) = *True* |
is-preevaluated (*InvokeWithExceptionNode n - - - - - -*) = *True* |
is-preevaluated (*NewInstanceNode n - - -*) = *True* |
is-preevaluated (*LoadFieldNode n - - -*) = *True* |
is-preevaluated (*SignedDivNode n - - - - -*) = *True* |
is-preevaluated (*SignedRemNode n - - - - -*) = *True* |
is-preevaluated (*ValuePhiNode n - -*) = *True* |
is-preevaluated (*BytecodeExceptionNode n - -*) = *True* |
is-preevaluated (*NewArrayNode n - -*) = *True* |
is-preevaluated (*ArrayLengthNode n -*) = *True* |
is-preevaluated (*LoadIndexedNode n - - -*) = *True* |
is-preevaluated (*StoreIndexedNode 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*) |

ReverseBytesNode:
⟦*kind g n* = *ReverseBytesNode x*;
g ⊢ *x* ≃ *xe*⟧
⟹ *g* ⊢ *n* ≃ (*UnaryExpr UnaryReverseBytes xe*) |

BitCountNode:
⟦*kind g n* = *BitCountNode x*;
g ⊢ *x* ≃ *xe*⟧

$\Longrightarrow g \vdash n \simeq (\mathit{UnaryExpr}\ \mathit{UnaryBitCount}\ \mathit{xe})$ |

NotNode:
$[\![\mathit{kind}\ g\ n = \mathit{NotNode}\ x;$
$g \vdash x \simeq \mathit{xe}]\!]$
$\Longrightarrow g \vdash n \simeq (\mathit{UnaryExpr}\ \mathit{UnaryNot}\ \mathit{xe})$ |

NegateNode:
$[\![\mathit{kind}\ g\ n = \mathit{NegateNode}\ x;$
$g \vdash x \simeq \mathit{xe}]\!]$
$\Longrightarrow g \vdash n \simeq (\mathit{UnaryExpr}\ \mathit{UnaryNeg}\ \mathit{xe})$ |

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

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

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

DivNode:
$[\![\mathit{kind}\ g\ n = \mathit{SignedFloatingIntegerDivNode}\ x\ y;$
$g \vdash x \simeq \mathit{xe};$
$g \vdash y \simeq \mathit{ye}]\!]$
$\Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr}\ \mathit{BinDiv}\ \mathit{xe}\ \mathit{ye})$ |

ModNode:
$[\![\mathit{kind}\ g\ n = \mathit{SignedFloatingIntegerRemNode}\ x\ y;$
$g \vdash x \simeq \mathit{xe};$
$g \vdash y \simeq \mathit{ye}]\!]$
$\Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr}\ \mathit{BinMod}\ \mathit{xe}\ \mathit{ye})$ |

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

AndNode:

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

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

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

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

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

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

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

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

IntegerEqualsNode:
$[\![kind\ g\ n = IntegerEqualsNode\ x\ y;$

$g \vdash x \simeq xe$;
$g \vdash y \simeq ye$]]
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerEquals\ xe\ ye)$ |

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

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

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

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

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

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

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

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

PiNode:
$[\![kind\ g\ n = PiNode\ n'\ guard;$
$g \vdash n' \simeq e]\!]$
$\Longrightarrow g \vdash n \simeq e\ |$

RefNode:
$[\![kind\ g\ n = RefNode\ n';$
$g \vdash n' \simeq e]\!]$
$\Longrightarrow g \vdash n \simeq e\ |$

IsNullNode:
$[\![kind\ g\ n = IsNullNode\ v;$
$g \vdash v \simeq lfn]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryIsNull\ lfn)$

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

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

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

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

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

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

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

2.2 Data-flow Tree to Subgraph

fun *unary-node* :: *IRUnaryOp* $\Rightarrow$ *ID* $\Rightarrow$ *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* |
unary-node UnaryIsNull v = *IsNullNode v* |
unary-node UnaryReverseBytes v = *ReverseBytesNode v* |
unary-node UnaryBitCount v = *BitCountNode v*

fun *bin-node* :: *IRBinaryOp* ⇒ *ID* ⇒ *ID* ⇒ *IRNode* **where**
bin-node BinAdd x y = *AddNode x y* |
bin-node BinMul x y = *MulNode x y* |
bin-node BinDiv x y = *SignedFloatingIntegerDivNode x y* |
bin-node BinMod x y = *SignedFloatingIntegerRemNode 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* |
bin-node BinIntegerTest x y = *IntegerTestNode x y* |
bin-node BinIntegerNormalizeCompare x y = *IntegerNormalizeCompareNode x y*
|
bin-node BinIntegerMulHigh x y = *IntegerMulHighNode x y*

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

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

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

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

export-code *get-fresh-id*

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

inductive *unique* :: *IRGraph* ⇒ (*IRNode* × *Stamp*) ⇒ (*IRGraph* × *ID*) ⇒ *bool*
where
Exists:
⟦*find-node-and-stamp g node* = *Some n*⟧
⟹ *unique g node* (*g*, *n*) |
New:

$[\![$*find-node-and-stamp g node = None*;
n = get-fresh-id g;
$g' =$ *add-node n node g*$]\!]$
$\Longrightarrow$ *unique g node* (g', n)

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

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

UnrepConstantNode:
$[\![$*unique g* (*ConstantNode c*, *constantAsStamp c*) $(g_1, n)]\!]$
$\Longrightarrow g \oplus$ (*ConstantExpr c*) $\rightsquigarrow (g_1, n)$ |

UnrepParameterNode:
$[\![$*unique g* (*ParameterNode i*, *s*) $(g_1, n)]\!]$
$\Longrightarrow g \oplus$ (*ParameterExpr i s*) $\rightsquigarrow (g_1, n)$ |

UnrepConditionalNode:
$[\![g \oplus ce \rightsquigarrow (g_1, c)$;
$g_1 \oplus te \rightsquigarrow (g_2, t)$;
$g_2 \oplus fe \rightsquigarrow (g_3, f)$;
$s' =$ *meet* (*stamp* g_3 *t*) (*stamp* g_3 *f*);
unique g_3 (*ConditionalNode c t f*, s') $(g_4, n)]\!]$
$\Longrightarrow g \oplus$ (*ConditionalExpr ce te fe*) $\rightsquigarrow (g_4, n)$ |

UnrepUnaryNode:
$[\![g \oplus xe \rightsquigarrow (g_1, x)$;
$s' =$ *stamp-unary op* (*stamp* g_1 *x*);
unique g_1 (*unary-node op x*, s') $(g_2, n)]\!]$
$\Longrightarrow g \oplus$ (*UnaryExpr op xe*) $\rightsquigarrow (g_2, n)$ |

UnrepBinaryNode:
$[\![g \oplus xe \rightsquigarrow (g_1, x)$;
$g_1 \oplus ye \rightsquigarrow (g_2, y)$;
$s' =$ *stamp-binary op* (*stamp* g_2 *x*) (*stamp* g_2 *y*);
unique g_2 (*bin-node op x y*, s') $(g_3, n)]\!]$
$\Longrightarrow g \oplus$ (*BinaryExpr op xe ye*) $\rightsquigarrow (g_3, n)$ |

AllLeafNodes:
$[\![$*stamp g n = s*;
is-preevaluated (*kind g n*)$]\!]$
$\Longrightarrow g \oplus$ (*LeafExpr n s*) $\rightsquigarrow (g, n)$

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

$$\frac{\textit{find-node-and-stamp}\ (g{::}\textit{IRGraph})\ (\textit{node}{::}\textit{IRNode} \times \textit{Stamp}) = \textit{Some}\ (n{::}\textit{nat})}{\textit{unique}\ g\ \textit{node}\ (g,\ n)}$$

$$\frac{\textit{find-node-and-stamp}\ (g{::}\textit{IRGraph})\ (\textit{node}{::}\textit{IRNode} \times \textit{Stamp}) = \textit{None} \quad (n{::}\textit{nat}) = \textit{get-fresh-id}\ g \quad (g'{::}\textit{IRGraph}) = \textit{add-node}\ n\ \textit{node}\ g}{\textit{unique}\ g\ \textit{node}\ (g',\ n)}$$

$$\frac{\textit{unique}\ (g{::}\textit{IRGraph})\ (\textit{ConstantNode}\ (c{::}\textit{Value}),\ \textit{constantAsStamp}\ c)\ (g_1{::}\textit{IRGraph},\ n{::}\textit{nat})}{g \oplus \textit{ConstantExpr}\ c \rightsquigarrow (g_1,\ n)}$$

$$\frac{\textit{unique}\ (g{::}\textit{IRGraph})\ (\textit{ParameterNode}\ (i{::}\textit{nat}),\ s{::}\textit{Stamp})\ (g_1{::}\textit{IRGraph},\ n{::}\textit{nat})}{g \oplus \textit{ParameterExpr}\ i\ s \rightsquigarrow (g_1,\ n)}$$

$$\frac{\begin{array}{c} g{::}\textit{IRGraph} \oplus \textit{ce}{::}\textit{IRExpr} \rightsquigarrow (g_1{::}\textit{IRGraph},\ c{::}\textit{nat}) \\ g_1 \oplus \textit{te}{::}\textit{IRExpr} \rightsquigarrow (g_2{::}\textit{IRGraph},\ t{::}\textit{nat}) \\ g_2 \oplus \textit{fe}{::}\textit{IRExpr} \rightsquigarrow (g_3{::}\textit{IRGraph},\ f{::}\textit{nat}) \\ (s'{::}\textit{Stamp}) = \textit{meet}\ (\textit{stamp}\ g_3\ t)\ (\textit{stamp}\ g_3\ f) \\ \textit{unique}\ g_3\ (\textit{ConditionalNode}\ c\ t\ f,\ s')\ (g_4{::}\textit{IRGraph},\ n{::}\textit{nat}) \end{array}}{g \oplus \textit{ConditionalExpr}\ \textit{ce}\ \textit{te}\ \textit{fe} \rightsquigarrow (g_4,\ n)}$$

$$\frac{\begin{array}{c} g{::}\textit{IRGraph} \oplus \textit{xe}{::}\textit{IRExpr} \rightsquigarrow (g_1{::}\textit{IRGraph},\ x{::}\textit{nat}) \\ g_1 \oplus \textit{ye}{::}\textit{IRExpr} \rightsquigarrow (g_2{::}\textit{IRGraph},\ y{::}\textit{nat}) \\ (s'{::}\textit{Stamp}) = \textit{stamp-binary}\ (\textit{op}{::}\textit{IRBinaryOp})\ (\textit{stamp}\ g_2\ x)\ (\textit{stamp}\ g_2\ y) \\ \textit{unique}\ g_2\ (\textit{bin-node}\ \textit{op}\ x\ y,\ s')\ (g_3{::}\textit{IRGraph},\ n{::}\textit{nat}) \end{array}}{g \oplus \textit{BinaryExpr}\ \textit{op}\ \textit{xe}\ \textit{ye} \rightsquigarrow (g_3,\ n)}$$

$$\frac{\begin{array}{c} g{::}\textit{IRGraph} \oplus \textit{xe}{::}\textit{IRExpr} \rightsquigarrow (g_1{::}\textit{IRGraph},\ x{::}\textit{nat}) \\ (s'{::}\textit{Stamp}) = \textit{stamp-unary}\ (\textit{op}{::}\textit{IRUnaryOp})\ (\textit{stamp}\ g_1\ x) \\ \textit{unique}\ g_1\ (\textit{unary-node}\ \textit{op}\ x,\ s')\ (g_2{::}\textit{IRGraph},\ n{::}\textit{nat}) \end{array}}{g \oplus \textit{UnaryExpr}\ \textit{op}\ \textit{xe} \rightsquigarrow (g_2,\ n)}$$

$$\frac{\begin{array}{c} \textit{stamp}\ (g{::}\textit{IRGraph})\ (n{::}\textit{nat}) = (s{::}\textit{Stamp}) \\ \textit{is-preevaluated}\ (\textit{kind}\ g\ n) \end{array}}{g \oplus \textit{LeafExpr}\ n\ s \rightsquigarrow (g,\ n)}$$

2.3 Lift Data-flow Tree Semantics

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

([-,-,-] ⊢ - ↦ - *50*)
where
$(g \vdash n \simeq e) \wedge ([m,p] \vdash e \mapsto v) \Longrightarrow [g, m, p] \vdash n \mapsto v$

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

inductive *encodeEvalAll* :: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *ID list* ⇒ *Value list* ⇒ *bool*
([-,-,-] ⊢ - [↦] - *60*) **where**
$(g \vdash nids\ [\simeq]\ es) \wedge ([m, p] \vdash es\ [\mapsto]\ vs) \Longrightarrow ([g, m, p] \vdash nids\ [\mapsto]\ vs)$

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

2.4 Graph Refinement

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

definition *graph-refinement* :: *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
graph-refinement $g_1\ g_2$ =
$((ids\ g_1 \subseteq ids\ g_2) \wedge$
$(\forall\ n\ .\ n \in ids\ g_1 \longrightarrow (\forall\, e.\ (g_1 \vdash n \simeq e) \longrightarrow (g_2 \vdash n \trianglelefteq e))))$

lemma *graph-refinement*:
graph-refinement g1 g2 ⟹
$(\forall\, n\ m\ p\ v.\ n \in ids\ g1 \longrightarrow ([g1, m, p] \vdash n \mapsto v) \longrightarrow ([g2, m, p] \vdash n \mapsto v))$
by (*meson encodeeval.simps 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-StartNode (*kind g 0*))

definition *wf-closed* **where**
wf-closed g =
($\forall$ *n* $\in$ *ids g* .
inputs g n $\subseteq$ *ids g* $\wedge$
succ g n $\subseteq$ *ids g* $\wedge$
kind g n $\neq$ *NoNode*)

definition *wf-phis* **where**
wf-phis g =
($\forall$ *n* $\in$ *ids g*.
is-PhiNode (*kind g n*) $\longrightarrow$
length (*ir-values* (*kind g n*))
= *length* (*ir-ends*
(*kind g* (*ir-merge* (*kind g n*)))))

definition *wf-ends* **where**
wf-ends g =
($\forall$ *n* $\in$ *ids g* .
is-AbstractEndNode (*kind g n*) $\longrightarrow$
card (*usages g n*) > *0*)

fun *wf-graph* :: *IRGraph* $\Rightarrow$ *bool* **where**
wf-graph g = (*wf-start g* $\wedge$ *wf-closed g* $\wedge$ *wf-phis g* $\wedge$ *wf-ends g*)

lemmas *wf-folds* =
wf-graph.simps
wf-start-def
wf-closed-def
wf-phis-def
wf-ends-def

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

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

lemma *wf-empty*: *wf-graph start-end-graph*
unfolding *wf-folds* **by** (*simp add*: *start-end-graph-def*)

lemma *wf-eg2-sq*: *wf-graph eg2-sq*
unfolding *wf-folds* **by** (*simp add*: *eg2-sq-def*)

fun *wf-logic-node-inputs* :: *IRGraph* $\Rightarrow$ *ID* $\Rightarrow$ *bool* **where**
wf-logic-node-inputs g n =

(∀ *inp* ∈ *set* (*inputs-of* (*kind g n*)) . (∀ *v m p* . ([*g*, *m*, *p*] ⊢ *inp* ↦ *v*) ⟶ *wf-bool v*))

fun *wf-values* :: *IRGraph* ⇒ *bool* **where**
wf-values g = (∀ *n* ∈ *ids g* .
(∀ *v m p* . ([*g*, *m*, *p*] ⊢ *n* ↦ *v*) ⟶
(*is-LogicNode* (*kind g n*) ⟶
wf-bool v ∧ *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* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
unchanged ns g1 g2 = (∀ *n* . *n* ∈ *ns* ⟶
(*n* ∈ *ids g1* ∧ *n* ∈ *ids g2* ∧ *kind g1 n* = *kind g2 n* ∧ *stamp g1 n* = *stamp g2 n*))

fun *changeonly* :: *ID set* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
changeonly ns g1 g2 = (∀ *n* . *n* ∈ *ids g1* ∧ *n* ∉ *ns* ⟶
(*n* ∈ *ids g1* ∧ *n* ∈ *ids g2* ∧ *kind g1 n* = *kind g2 n* ∧ *stamp g1 n* = *stamp g2 n*))

lemma *node-unchanged*:
assumes *unchanged ns g1 g2*
assumes *nid* ∈ *ns*
shows *kind g1 nid* = *kind g2 nid*
using *assms* **by** *simp*

lemma *other-node-unchanged*:
assumes *changeonly ns g1 g2*
assumes *nid* ∈ *ids g1*
assumes *nid* ∉ *ns*
shows *kind g1 nid* = *kind g2 nid*
using *assms* **by** *simp*

Some notation for input nodes used

```
inductive eval-uses:: IRGraph ⇒ ID ⇒ ID ⇒ bool
  for g where

  use0: nid ∈ ids g
    ⟹ eval-uses g nid nid |

  use-inp: nid' ∈ inputs g n
    ⟹ eval-uses g nid nid' |

  use-trans: ⟦eval-uses g nid nid';
    eval-uses g nid' nid''⟧
    ⟹ eval-uses g nid nid''

fun eval-usages :: IRGraph ⇒ ID ⇒ ID set where
  eval-usages g nid = {n ∈ ids g . eval-uses g nid n}

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

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

lemma child-member:
  assumes n = kind g nid
  assumes n ≠ NoNode
  assumes List.member (inputs-of n) child
  shows child ∈ inputs g nid
  by (metis in-set-member inputs.simps assms(1,3))

lemma child-member-in:
  assumes nid ∈ ids g
  assumes List.member (inputs-of (kind g nid)) child
  shows child ∈ inputs g nid
  by (metis child-member ids-some assms)


lemma inp-in-g:
  assumes n ∈ inputs g nid
  shows nid ∈ ids g
proof −
```

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

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

lemma *kind-unchanged*:
assumes *nid* $\in$ *ids g1*
assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
shows *kind g1 nid* = *kind g2 nid*
proof −
show *?thesis*
using *assms eval-usages-self* **by** *simp*
qed

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

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

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

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

lemma *inputs-are-usages*:
assumes *nid'* $\in$ *inputs g nid*
assumes *nid'* $\in$ *ids g*

shows *nid'* ∈ *eval-usages g nid*
using *assms* **by** (*simp add*: *inputs-are-uses*)

lemma *inputs-of-are-usages*:
assumes *List.member* (*inputs-of* (*kind g nid*)) *nid'*
assumes *nid'* ∈ *ids g*
shows *nid'* ∈ *eval-usages g nid*
by (*metis assms in-set-member inputs.elims inputs-are-usages*)

lemma *usage-includes-inputs*:
assumes *us* = *eval-usages g nid*
assumes *ls* = *inputs g nid*
assumes *ls* ⊆ *ids g*
shows *ls* ⊆ *us*
using *inputs-are-usages assms* **by** *blast*

lemma *elim-inp-set*:
assumes *k* = *kind g nid*
assumes *k* ≠ *NoNode*
assumes *child* ∈ *set* (*inputs-of k*)
shows *child* ∈ *inputs g nid*
using *assms* **by** *simp*

lemma *encode-in-ids*:
assumes *g* ⊢ *nid* ≃ *e*
shows *nid* ∈ *ids g*
using *assms* **apply** (*induction rule*: *rep.induct*) **by** *fastforce+*

lemma *eval-in-ids*:
assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
shows *nid* ∈ *ids g*
using *assms encode-in-ids* **by** (*auto simp add*: *encodeeval.simps*)

lemma *transitive-kind-same*:
assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
shows ∀ *nid'* ∈ (*eval-usages g1 nid*) . *kind g1 nid'* = *kind g2 nid'*
by (*meson unchanged.elims*(*1*) *assms*)

theorem *stay-same-encoding*:
assumes *nc*: *unchanged* (*eval-usages g1 nid*) *g1 g2*
assumes *g1*: *g1* ⊢ *nid* ≃ *e*
assumes *wf*: *wf-graph g1*
shows *g2* ⊢ *nid* ≃ *e*
proof −
have *dom*: *nid* ∈ *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*
by (*metis kind-unchanged*)
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*,*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 ∈ eval-usages g1 n ∧ t ∈ eval-usages g1 n ∧ f ∈ eval-usages g1 n*
by (*metis inputs-of-ConditionalNode ConditionalNode.hyps*(*1*,*2*,*3*,*4*) *encode-in-ids inputs.simps*
inputs-are-usages list.set-intros(*1*) *set-subset-Cons subset-code*(*1*))
then show *?case*
by (*metis ConditionalNode.hyps*(*1*) *ConditionalNode.prems*(*1*) *IRNodes.inputs-of-ConditionalNode*

‹*kind g2 n = ConditionalNode c t f*› *child-unchanged inputs.simps list.set-intros*(*1*)

local.ConditionalNode(*5*,*6*,*7*,*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*
by (*metis kind-unchanged*)
then have *x ∈ eval-usages g1 n*
by (*metis inputs-of-AbsNode AbsNode.hyps*(*1*,*2*) *encode-in-ids inputs.simps inputs-are-usages*
list.set-intros(*1*))
then show *?case*
by (*metis AbsNode.IH AbsNode.hyps*(*1*) *AbsNode.prems*(*1*,*3*) *IRNodes.inputs-of-AbsNode rep.AbsNode*
‹*kind g2 n = AbsNode x*› *child-member-in child-unchanged local.wf member-rec*(*1*)
unchanged.simps)
next
case (*ReverseBytesNode n x xe*)
then have *kind g2 n = ReverseBytesNode x*
by (*metis kind-unchanged*)
then have *x ∈ eval-usages g1 n*
by (*metis IRNodes.inputs-of-ReverseBytesNode ReverseBytesNode.hyps*(*1*,*2*) *encode-in-ids*

inputs.simps inputs-are-usages list.set-intros(*1*))
then show *?case*
by (*metis IRNodes.inputs-of-ReverseBytesNode ReverseBytesNode.IH Reverse-BytesNode.hyps*(*1*,*2*)
ReverseBytesNode.prems(*1*) *child-member-in child-unchanged local.wf member-rec*(*1*)
‹*kind g2 n* = *ReverseBytesNode x*› *encode-in-ids rep.ReverseBytesNode*)
next
case (*BitCountNode n x xe*)
then have *kind g2 n* = *BitCountNode x*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n*
by (*metis BitCountNode.hyps*(*1*,*2*) *IRNodes.inputs-of-BitCountNode encode-in-ids inputs.simps*
inputs-are-usages list.set-intros(*1*))
then show *?case*
by (*metis BitCountNode.IH BitCountNode.hyps*(*1*,*2*) *BitCountNode.prems*(*1*) *member-rec*(*1*) *local.wf*
IRNodes.inputs-of-BitCountNode ‹*kind g2 n* = *BitCountNode x*› *encode-in-ids rep.BitCountNode*
child-member-in child-unchanged)
next
case (*NotNode n x xe*)
then have *kind g2 n* = *NotNode x*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n*
by (*metis inputs-of-NotNode NotNode.hyps*(*1*,*2*) *encode-in-ids inputs.simps inputs-are-usages*
list.set-intros(*1*))
then show *?case*
by (*metis NotNode.IH NotNode.hyps*(*1*) *NotNode.prems*(*1*,*3*) *IRNodes.inputs-of-NotNode rep.NotNode*
‹*kind g2 n* = *NotNode x*› *child-member-in child-unchanged local.wf member-rec*(*1*)
unchanged.simps)
next
case (*NegateNode n x xe*)
then have *kind g2 n* = *NegateNode x*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n*
by (*metis inputs-of-NegateNode NegateNode.hyps*(*1*,*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.prems*(*1*,*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*
    by (*metis kind-unchanged*)
  then have *x* ∈ *eval-usages g1 n*
    by (*metis inputs-of-LogicNegationNode inputs-of-are-usages LogicNegationNode.hyps*(*1*,*2*)
      encode-in-ids member-rec(*1*))
  then show *?case*
    by (*metis IRNodes.inputs-of-LogicNegationNode LogicNegationNode.IH LogicNegationNode.hyps*(*1*,*2*)
      LogicNegationNode.prems(*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*
    by (*metis kind-unchanged*)
  then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
    by (*metis AddNode.hyps*(*1*,*2*,*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*,*2*) *AddNode.hyps*(*1*,*2*,*3*) *AddNode.prems*(*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*
    by (*metis kind-unchanged*)
  then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
    by (*metis MulNode.hyps*(*1*,*2*,*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*
    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*) *inputs-of-MulNode MulNode*(*1*,*4*,*5*,*6*,*7*))
next
  case (*DivNode n x y xe ye*)
  then have *kind g2 n* = *SignedFloatingIntegerDivNode x y*
    by (*metis kind-unchanged*)
  then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
    by (*metis DivNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-SignedFloatingIntegerDivNode encode-in-ids in-mono inputs.simps*
      inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)

then show *?case*
by (*metis* ‹*kind g2 n* = *SignedFloatingIntegerDivNode x y*› *child-unchanged inputs.simps list.set-intros*(*1*) *rep.DivNode*
set-subset-Cons subset-iff unchanged.elims(*2*) *inputs-of-SignedFloatingIntegerDivNode DivNode*(*1*,*4*,*5*,*6*,*7*))
next
case (*ModNode n x y xe ye*)
then have *kind g2 n* = *SignedFloatingIntegerRemNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis ModNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-SignedFloatingIntegerRemNode encode-in-ids in-mono inputs.simps*
inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis* ‹*kind g2 n* = *SignedFloatingIntegerRemNode x y*› *child-unchanged inputs.simps list.set-intros*(*1*) *rep.ModNode*
set-subset-Cons subset-iff unchanged.elims(*2*) *inputs-of-SignedFloatingIntegerRemNode ModNode*(*1*,*4*,*5*,*6*,*7*))
next
case (*SubNode n x y xe ye*)
then have *kind g2 n* = *SubNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis SubNode.hyps*(*1*,*2*,*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*
by (*metis* ‹*kind g2 n* = *SubNode x y*› *child-member child-unchanged encode-in-ids ids-some SubNode*
member-rec(*1*) *rep.SubNode inputs-of-SubNode*)
next
case (*AndNode n x y xe ye*)
then have *kind g2 n* = *AndNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis AndNode.hyps*(*1*,*2*,*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*
by (*metis AndNode*(*1*,*4*,*5*,*6*,*7*) *inputs-of-AndNode* ‹*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*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis OrNode.hyps*(*1*,*2*,*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*
by (*metis inputs-of-OrNode* ‹*kind g2 n* = *OrNode x y*› *child-unchanged encode-in-ids rep.OrNode*
child-member ids-some member-rec(*1*) *OrNode*)
next
case (*XorNode n x y xe ye*)
then have *kind g2 n* = *XorNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis XorNode.hyps*(*1*,*2*,*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*
by (*metis inputs-of-XorNode* ‹*kind g2 n* = *XorNode x y*› *child-member child-unchanged rep.XorNode*
encode-in-ids ids-some member-rec(*1*) *XorNode*)
next
case (*ShortCircuitOrNode n x y xe ye*)
then have *kind g2 n* = *ShortCircuitOrNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis ShortCircuitOrNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-ShortCircuitOrNode inputs-are-usages*
in-mono inputs.simps list.set-intros(*1*) *set-subset-Cons encode-in-ids*)
then show *?case*
by (*metis ShortCircuitOrNode inputs-of-ShortCircuitOrNode* ‹*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*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis LeftShiftNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-LeftShiftNode encode-in-ids inputs.simps*
inputs-are-usages list.set-intros(*1*) *set-subset-Cons in-mono*)
then show *?case*
by (*metis LeftShiftNode inputs-of-LeftShiftNode* ‹*kind g2 n* = *LeftShiftNode x y*› *child-unchanged*
encode-in-ids ids-some member-rec(*1*) *rep.LeftShiftNode child-member*)
next
case (*RightShiftNode n x y xe ye*)
then have *kind g2 n* = *RightShiftNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis RightShiftNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-RightShiftNode encode-in-ids inputs.simps*

inputs-are-usages list.set-intros(*1*) *set-subset-Cons in-mono*)
then show *?case*
by (*metis RightShiftNode inputs-of-RightShiftNode* ‹*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*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
by (*metis UnsignedRightShiftNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-UnsignedRightShiftNode in-mono*
encode-in-ids inputs.simps inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis UnsignedRightShiftNode inputs-of-UnsignedRightShiftNode child-member child-unchanged*
‹*kind g2 n* = *UnsignedRightShiftNode x y*› *encode-in-ids ids-some rep.UnsignedRightShiftNode*
member-rec(*1*))
next
case (*IntegerBelowNode n x y xe ye*)
then have *kind g2 n* = *IntegerBelowNode x y*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
by (*metis IntegerBelowNode.hyps*(*1*,*2*,*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*
by (*metis inputs-of-IntegerBelowNode* ‹*kind g2 n* = *IntegerBelowNode x y*› *rep.IntegerBelowNode*
child-member child-unchanged encode-in-ids ids-some member-rec(*1*) *IntegerBelowNode*)
next
case (*IntegerEqualsNode n x y xe ye*)
then have *kind g2 n* = *IntegerEqualsNode x y*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
by (*metis IntegerEqualsNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-IntegerEqualsNode inputs-are-usages*
in-mono inputs.simps encode-in-ids list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis inputs-of-IntegerEqualsNode* ‹*kind g2 n* = *IntegerEqualsNode x y*› *rep.IntegerEqualsNode*
child-member child-unchanged encode-in-ids ids-some member-rec(*1*) *IntegerEqualsNode*)
next
case (*IntegerLessThanNode n x y xe ye*)
then have *kind g2 n* = *IntegerLessThanNode x y*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*

by (*metis IntegerLessThanNode.hyps*(*1*,*2*,*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*
by (*metis rep.IntegerLessThanNode inputs-of-IntegerLessThanNode child-unchanged encode-in-ids*
‹*kind g2 n* = *IntegerLessThanNode x y*› *child-member member-rec*(*1*) *IntegerLessThanNode*
ids-some)
next
case (*IntegerTestNode n x y xe ye*)
then have *kind g2 n* = *IntegerTestNode x y*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
by (*metis IntegerTestNode.hyps IRNodes.inputs-of-IntegerTestNode encode-in-ids*
in-mono inputs.simps inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis rep.IntegerTestNode inputs-of-IntegerTestNode child-unchanged encode-in-ids*
‹*kind g2 n* = *IntegerTestNode x y*› *child-member member-rec*(*1*) *IntegerTestNode ids-some*)
next
case (*IntegerNormalizeCompareNode n x y xe ye*)
then have *kind g2 n* = *IntegerNormalizeCompareNode x y*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
by (*metis IRNodes.inputs-of-IntegerNormalizeCompareNode IntegerNormalizeCompareNode.hyps*(*1*,*2*,*3*)
encode-in-ids in-set-member inputs.simps inputs-are-usages member-rec(*1*))
then show *?case*
by (*metis IRNodes.inputs-of-IntegerNormalizeCompareNode IntegerNormalizeCompareNode.IH*(*1*,*2*)
IntegerNormalizeCompareNode.hyps(*1*,*2*,*3*) *IntegerNormalizeCompareNode.prems*(*1*) *inputs.simps*
‹*kind* (*g2*::*IRGraph*) (*n*::*nat*) = *IntegerNormalizeCompareNode* (*x*::*nat*) (*y*::*nat*)› *local.wf*
encode-in-ids list.set-intros(*1*) *rep.IntegerNormalizeCompareNode set-subset-Cons in-mono*
child-unchanged)
next
case (*IntegerMulHighNode n x y xe ye*)
then have *kind g2 n* = *IntegerMulHighNode x y*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n*
by (*metis IRNodes.inputs-of-IntegerMulHighNode IntegerMulHighNode.hyps*(*1*,*2*) *encode-in-ids*
inputs-of-are-usages member-rec(*1*))
then show *?case*
by (*metis inputs-of-IntegerMulHighNode IntegerMulHighNode.IH*(*1*,*2*) *Inte-*

gerMulHighNode.hyps(1,2,3)
IntegerMulHighNode.prems(1) child-unchanged encode-in-ids inputs.simps list.set-intros(1,2)
‹kind (g2::IRGraph) (n::nat) = IntegerMulHighNode (x::nat) (y::nat)› rep.IntegerMulHighNode
local.wf)
next
case (*NarrowNode n ib rb x xe*)
then have *kind g2 n = NarrowNode ib rb x*
by (*metis kind-unchanged*)
then have *x ∈ eval-usages g1 n*
by (*metis NarrowNode.hyps(1,2) IRNodes.inputs-of-NarrowNode inputs-are-usages encode-in-ids*
list.set-intros(1) inputs.simps)
then show *?case*
by (*metis NarrowNode(1,3,4,5) inputs-of-NarrowNode ‹kind g2 n = NarrowNode ib rb x› inputs.elims*
child-unchanged 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*
by (*metis kind-unchanged*)
then have *x ∈ eval-usages g1 n*
by (*metis inputs-of-SignExtendNode SignExtendNode.hyps(1,2) inputs-are-usages encode-in-ids*
list.set-intros(1) inputs.simps)
then show *?case*
by (*metis SignExtendNode(1,3,4,5,6) inputs-of-SignExtendNode in-set-member list.set-intros(1)*
‹kind g2 n = SignExtendNode ib rb x› child-member-in child-unchanged rep.SignExtendNode
unchanged.elims(2))
next
case (*ZeroExtendNode n ib rb x xe*)
then have *kind g2 n = ZeroExtendNode ib rb x*
by (*metis kind-unchanged*)
then have *x ∈ eval-usages g1 n*
by (*metis ZeroExtendNode.hyps(1,2) IRNodes.inputs-of-ZeroExtendNode encode-in-ids inputs.simps*
inputs-are-usages list.set-intros(1))
then show *?case*
by (*metis ZeroExtendNode(1,3,4,5,6) inputs-of-ZeroExtendNode child-unchanged unchanged.simps*
‹kind g2 n = ZeroExtendNode ib rb x› child-member-in rep.ZeroExtendNode member-rec(1))
next
case (*LeafNode n s*)
then show *?case*
by (*metis kind-unchanged rep.LeafNode stamp-unchanged*)

next
  case (*PiNode n n′ gu*)
  then have *kind g2 n = PiNode n′ gu*
    by (*metis kind-unchanged*)
  then show *?case*
    by (*metis PiNode.IH* ‹*kind* (*g2*) (*n*) = *PiNode* (*n′*) (*gu*)› *child-unchanged encode-in-ids rep.PiNode*
      inputs.elims list.set-intros(*1*)*PiNode.hyps PiNode.prems*(*1*,*2*) *IRNodes.inputs-of-PiNode*)
next
  case (*RefNode n n′*)
  then have *kind g2 n = RefNode n′*
    by (*metis kind-unchanged*)
  then have *n′* ∈ *eval-usages g1 n*
   by (*metis IRNodes.inputs-of-RefNode RefNode.hyps*(*1*,*2*) *inputs-are-usages list.set-intros*(*1*)
      inputs.elims encode-in-ids)
  then show *?case*
    by (*metis IRNodes.inputs-of-RefNode RefNode.IH RefNode.hyps*(*1*,*2*) *RefNode.prems*(*1*) *inputs.elims*
      ‹*kind g2 n = RefNode n′*› *child-unchanged encode-in-ids list.set-intros*(*1*) *rep.RefNode*
      local.wf)
next
  case (*IsNullNode n v*)
  then have *kind g2 n = IsNullNode v*
    by (*metis kind-unchanged*)
  then show *?case*
    by (*metis IRNodes.inputs-of-IsNullNode IsNullNode.IH IsNullNode.hyps*(*1*,*2*) *IsNullNode.prems*(*1*)
      ‹*kind g2 n = IsNullNode v*› *child-unchanged encode-in-ids inputs.simps list.set-intros*(*1*)
      local.wf rep.IsNullNode)
 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** *simp*
  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 *g1* **by** (*auto simp add*: *encodeeval.simps*)

```
  then have val: [m,p] ⊢ e ↦ v1
    by (simp add: g1 encodeeval.simps)
  then show ?thesis
    using e nc unfolding encodeeval.simps
  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
      by (meson local.wf stay-same-encoding ParameterExpr)
    then show ?case
      by (meson ParameterExpr.hyps evaltree.ParameterExpr)
  next
    case (ConditionalExpr ce cond branch te fe v)
    then have g2 ⊢ nid ≃ ConditionalExpr ce te fe
      using local.wf stay-same-encoding by presburger
    then show ?case
      by (meson ConditionalExpr.prems(1))
  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
  by (simp add: assms add-node.rep-eq kind.rep-eq stamp.rep-eq)

lemma disjoint-change:
  assumes changeonly change g gup
  assumes nochange = ids g − change
  shows unchanged nochange g gup
  using assms by simp

lemma add-node-unchanged:
  assumes new ∉ ids g
  assumes nid ∈ ids g
```

assumes *gup* = *add-node new k g*
assumes *wf-graph g*
shows *unchanged* (*eval-usages g nid*) *g gup*
proof −
have *new* ∉ (*eval-usages g nid*)
using *assms* **by** *simp*
then have *changeonly* {*new*} *g gup*
using *assms add-changed* **by** *simp*
then show *?thesis*
using *assms* **by** *auto*
qed

lemma *eval-uses-imp*:
((*nid′* ∈ *ids g* ∧ *nid* = *nid′*)
∨ *nid′* ∈ *inputs g nid*
∨ (∃ *nid″* . *eval-uses g nid nid″* ∧ *eval-uses g nid″ nid′*))
⟷ *eval-uses g nid nid′*
by (*meson eval-uses.simps*)

lemma *wf-use-ids*:
assumes *wf-graph g*
assumes *nid* ∈ *ids g*
assumes *eval-uses g nid nid′*
shows *nid′* ∈ *ids g*
using *assms*(*3*) **apply** (*induction rule*: *eval-uses.induct*) **using** *assms*(*1*) *inp-in-g-wf*
by *auto*

lemma *no-external-use*:
assumes *wf-graph g*
assumes *nid′* ∉ *ids g*
assumes *nid* ∈ *ids g*
shows ¬(*eval-uses g nid nid′*)
proof −
have *0*: *nid* ≠ *nid′*
using *assms* **by** *auto*
have *inp*: *nid′* ∉ *inputs g nid*
using *assms inp-in-g-wf* **by** *auto*
have *rec-0*: ∄ *n* . *n* ∈ *ids g* ∧ *n* = *nid′*
using *assms* **by** *simp*
have *rec-inp*: ∄ *n* . *n* ∈ *ids g* ∧ *n* ∈ *inputs g nid′*
using *assms*(*2*) **by** (*simp add*: *inp-in-g*)
have *rec*: ∄ *nid″* . *eval-uses g nid nid″* ∧ *eval-uses g nid″ nid′*
using *wf-use-ids assms* **by** *blast*
from *inp 0 rec* **show** *?thesis*
using *eval-uses-imp* **by** *blast*
qed

end

3 Control-flow Semantics

theory *IRStepObj*
  imports
    TreeToGraph
    Graph.Class
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 references sequentially storing the next free object reference as 'Free'.

type-synonym ('*a*, '*b*) *Heap* = '*a* ⇒ '*b* ⇒ *Value*
type-synonym *Free* = *nat*
type-synonym ('*a*, '*b*) *DynamicHeap* = ('*a*, '*b*) *Heap* × *Free*

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

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

fun *h-new-inst* :: (*string*, *objref*) *DynamicHeap* ⇒ *string* ⇒ (*string*, *objref*) *DynamicHeap* × *Value* **where**
  h-new-inst (*h*, *n*) *className* = (*h-store-field* ''*class*'' (*Some n*) (*ObjStr className*) (*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* ⇒ '*a list* ⇒ *nat* **where**
  find-index - [] = *0* |
  find-index v (*x* # *xs*) = (*if* (*x*=*v*) *then 0 else find-index v xs* + *1*)

inductive *indexof* :: '*a list* ⇒ *nat* ⇒ '*a* ⇒ *bool* **where**
  find-index x xs = *i* ⟹ *indexof xs i x*

lemma *indexof-det*:
  indexof xs i x ⟹ *indexof xs i*' *x* ⟹ *i* = *i*'

apply (*induction rule*: *indexof.induct*)
by (*simp add*: *indexof.simps*)

code-pred (*modes*: *i* ⇒ *o* ⇒ *i* ⇒ *bool*) *indexof* **.**

notation (*latex* **output**)
indexof (-!- = -)

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

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

definition
fun-add :: (′*a* ⇒ ′*b*) ⇒ (′*a* ⇀ ′*b*) ⇒ (′*a* ⇒ ′*b*) (**infixl** $++_f$ *100*) **where**
f1 $++_f$ *f2* = (λ*x*. *case f2 x of None* ⇒ *f1 x* | *Some y* ⇒ *y*)

definition *upds* :: (′*a* ⇒ ′*b*) ⇒ ′*a list* ⇒ ′*b list* ⇒ (′*a* ⇒ ′*b*) (-/′(- [→] -/′) *900*)
where
upds m ns vs = *m* $++_f$ (*map-of* (*rev* (*zip ns vs*)))

lemma *fun-add-empty*:
xs $++_f$ (*map-of* []) = *xs*
unfolding *fun-add-def* **by** *simp*

lemma *upds-inc*:
m(*a*#*as* [→] *b*#*bs*) = (*m*(*a*:=*b*))(*as*[→]*bs*)
unfolding *upds-def fun-add-def* **apply** *simp* **sorry**

lemma *upds-compose*:
a $++_f$ *map-of* (*rev* (*zip* (*n* # *ns*) (*v* # *vs*))) = *a*(*n* := *v*) $++_f$ *map-of* (*rev* (*zip*
ns vs))
using *upds-inc*
by (*metis upds-def*)

lemma *set-phis ns vs* = (λ*m*. *upds m ns vs*)
proof (*induction rule*: *set-phis.induct*)
case (*1 m*)
then show *?case* **unfolding** *set-phis.simps upds-def*
by (*metis Nil-eq-zip-iff Nil-is-rev-conv fun-add-empty*)
next

case (*2 n xs v vs m*)
then show *?case* **unfolding** *set-phis.simps upds-def*
by (*metis upds-compose*)
next
case (*3 v vs m*)
then show *?case*
by (*metis fun-add-empty rev.simps*(*1*) *upds-def set-phis.simps*(*3*) *zip-Nil*)
next
case (*4 x xs m*)
then show *?case*
by (*metis Nil-eq-zip-iff fun-add-empty rev.simps*(*1*) *upds-def set-phis.simps*(*4*))
qed

fun *is-PhiKind* :: *IRGraph* ⇒ *ID* ⇒ *bool* **where**
is-PhiKind g nid = *is-PhiNode* (*kind g nid*)

definition *filter-phis* :: *IRGraph* ⇒ *ID* ⇒ *ID list* **where**
filter-phis g merge = (*filter* (*is-PhiKind g*) (*sorted-list-of-set* (*usages g merge*)))

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

Intraprocedural semantics are given as a small-step semantics.

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

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

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

FixedGuardNode:
⟦(*kind g nid*) = (*FixedGuardNode cond before next*);
[*g*, *m*, *p*] ⊢ *cond* ↦ *val*;

¬(*val-to-bool val*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*next*, *m*, *h*) |

BytecodeExceptionNode:
⟦(*kind g nid*) = (*BytecodeExceptionNode args st nid′*);
exceptionType = *stp-type* (*stamp g nid*);
(*h′*, *ref*) = *h-new-inst h exceptionType*;
m′ = *m*(*nid* := *ref*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m′*, *h′*) |

IfNode:
$[\![$ *kind g nid* $=$ (*IfNode cond tb fb*);
  $[g, m, p] \vdash$ *cond* $\mapsto$ *val*;
  $nid' =$ (*if val-to-bool val then tb else fb*)$]\!]$
  $\Longrightarrow g, p \vdash (nid, m, h) \to (nid', m, h)$ |

EndNodes:
$[\![$*is-AbstractEndNode* (*kind g nid*);
  merge $=$ *any-usage g nid*;
  is-AbstractMergeNode (*kind g merge*);

  indexof (*inputs-of* (*kind g merge*)) *i nid*;
  phis $=$ *filter-phis g merge*;
  inps $=$ *phi-inputs g phis i*;
  $[g, m, p] \vdash$ *inps* $[\mapsto]$ *vs*;

  $m' = (m(phis[\to]vs))]\!]$
  $\Longrightarrow g, p \vdash (nid, m, h) \to (merge, m', h)$ |

NewArrayNode:
  $[\![$*kind g nid* $=$ (*NewArrayNode len st nid'*);
    $[g, m, p] \vdash$ *len* $\mapsto$ *length'*;

    arrayType $=$ *stp-type* (*stamp g nid*);
    $(h', ref) =$ *h-new-inst h arrayType*;
    ref $=$ *ObjRef refNo*;
    $h'' =$ *h-store-field* '''' *refNo* (*intval-new-array length' arrayType*) h';

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

ArrayLengthNode:
  $[\![$*kind g nid* $=$ (*ArrayLengthNode x nid'*);
    $[g, m, p] \vdash x \mapsto$ *ObjRef ref*;

    h-load-field '''' *ref h* $=$ *arrayVal*;
    length' $=$ *array-length* (*arrayVal*);

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

LoadIndexedNode:
  $[\![$*kind g nid* $=$ (*LoadIndexedNode index guard array nid'*);
    $[g, m, p] \vdash$ *index* $\mapsto$ *indexVal*;
    $[g, m, p] \vdash$ *array* $\mapsto$ *ObjRef ref*;

    h-load-field '''' *ref h* $=$ *arrayVal*;
    loaded $=$ *intval-load-index arrayVal indexVal*;

$m' = m(nid := loaded)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h)\ |$

StoreIndexedNode:
$[\![kind\ g\ nid = (StoreIndexedNode\ check\ val\ st\ index\ guard\ array\ nid');$
$[g,\ m,\ p] \vdash index \mapsto indexVal;$
$[g,\ m,\ p] \vdash array \mapsto ObjRef\ ref;$
$[g,\ m,\ p] \vdash val \mapsto value;$

$h\text{-}load\text{-}field\ ''''\ ref\ h = arrayVal;$
$updated = intval\text{-}store\text{-}index\ arrayVal\ indexVal\ value;$
$h' = h\text{-}store\text{-}field\ ''''\ ref\ updated\ h;$
$m' = m(nid := updated)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h')\ |$

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

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

SignedDivNode:
$[\![kind\ g\ nid = (SignedDivNode\ nid\ x\ y\ zero\ sb\ next);$
$[g,\ m,\ p] \vdash x \mapsto v1;$
$[g,\ m,\ p] \vdash y \mapsto v2;$
$m' = m(nid := intval\text{-}div\ v1\ v2)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (next,\ m',\ h)\ |$

SignedRemNode:
$[\![kind\ g\ nid = (SignedRemNode\ nid\ x\ y\ zero\ sb\ next);$
$[g,\ m,\ p] \vdash x \mapsto v1;$
$[g,\ m,\ p] \vdash y \mapsto v2;$
$m' = m(nid := intval\text{-}mod\ v1\ v2)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (next,\ m',\ h)\ |$

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

StoreFieldNode:
$[\![kind\ g\ nid = (StoreFieldNode\ nid\ f\ newval\ \text{-}\ (Some\ obj)\ nid');$

$[g, m, p] \vdash newval \mapsto val;$
$[g, m, p] \vdash obj \mapsto ObjRef\ ref;$
$h' = h\text{-}store\text{-}field\ f\ ref\ val\ h;$
$m' = m(nid := val)]\!]$
$\Longrightarrow g, p \vdash (nid, m, h) \rightarrow (nid', m', h')$ |

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

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

3.3 Interprocedural Semantics

type-synonym *Signature* = *string*
type-synonym *Program* = *Signature* ⇀ *IRGraph*
type-synonym *System* = *Program* × *Classes*

function *dynamic-lookup* :: *System* ⇒ *string* ⇒ *string* ⇒ *string list* ⇒ *IRGraph option* **where**
dynamic-lookup (*P*,*cl*) *cn mn path* = (
if (*cn* = ''*None*'' ∨ *cn* ∉ *set* (*Class.mapJVMFunc class-name cl*) ∨ *path* = [])
then (*P mn*)
else (

let method-index = (*find-index* (*get-simple-signature mn*) (*CLsimple-signatures cn cl*)) *in*
let parent = *hd path in*

if (*method-index* = *length* (*CLsimple-signatures cn cl*))
then (*dynamic-lookup* (*P*, *cl*) *parent mn* (*tl path*))
else (*P* (*nth* (*map method-unique-name* (*CLget-Methods cn cl*)) *method-index*))
)
)

by *auto*
termination *dynamic-lookup* **apply** (*relation measure* (λ(*S*,*cn*,*mn*,*path*). (*length path*))) **by** *auto*

inductive *step-top* :: *System* ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap* ⇒
(*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap* ⇒ *bool*
(- ⊢ - ⟶ - *55*)
for *S* **where**

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

InvokeNodeStepStatic:
$[\![$*is-Invoke* (*kind g nid*);
callTarget = *ir-callTarget* (*kind g nid*);
kind g callTarget = (*MethodCallTargetNode targetMethod actuals invoke-kind*);
$\neg$(*hasReceiver invoke-kind*);
Some targetGraph = (*dynamic-lookup S ''None'' targetMethod* []);
$[g,\ m,\ p] \vdash$ *actuals* $[\mapsto]$ $p']\!]$
$\Longrightarrow (S) \vdash ((g,nid,m,p)\#stk,\ h) \longrightarrow ((targetGraph,0,new\text{-}map\text{-}state,p')\#(g,nid,m,p)\#stk,$
$h)$ |

InvokeNodeStep:
$[\![$*is-Invoke* (*kind g nid*);
callTarget = *ir-callTarget* (*kind g nid*);
kind g callTarget = (*MethodCallTargetNode targetMethod arguments invoke-kind*);
hasReceiver invoke-kind;
$[g,\ m,\ p] \vdash$ *arguments* $[\mapsto]$ p';
ObjRef self = *hd* p';
ObjStr cname = (*h-load-field ''class'' self h*);
S = (*P*,*cl*);
Some targetGraph = *dynamic-lookup S cname targetMethod* (*class-parents*
(*CLget-JVMClass cname cl*))$]\!]$
$\Longrightarrow (S) \vdash ((g,nid,m,p)\#stk,\ h) \longrightarrow ((targetGraph,0,new\text{-}map\text{-}state,p')\#(g,nid,m,p)\#stk,$
$h)$ |

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

$m'_c = m_c(nid_c := v)$;
$nid'_c = (successors\text{-}of\ (kind\ g_c\ nid_c))!0]\!]$
$\Longrightarrow (S) \vdash ((g,nid,m,p)\#(g_c,nid_c,m_c,p_c)\#stk,\ h) \longrightarrow ((g_c,nid'_c,m'_c,p_c)\#stk,\ h)$
|

ReturnNodeVoid:
$[\![$*kind g nid* = (*ReturnNode None* -);

$nid'_c = (successors\text{-}of\ (kind\ g_c\ nid_c))!0]\!]$
$\Longrightarrow (S) \vdash ((g,nid,m,p)\#(g_c,nid_c,m_c,p_c)\#stk,\ h) \longrightarrow ((g_c,nid'_c,m_c,p_c)\#stk,\ h)$
|

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

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

$kind\ g_c\ nid_c = (InvokeWithExceptionNode\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ exEdge)$;

$m'_c = m_c(nid_c := e)]\!]$
$\Longrightarrow (S) \vdash ((g,nid,m,p)\#(g_c,nid_c,m_c,p_c)\#stk,\ h) \longrightarrow ((g_c,exEdge,m'_c,p_c)\#stk,\ h)$

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

3.4 Big-step Execution

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

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

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

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

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

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

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

inductive *exec-debug* :: *System*
$\Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times FieldRefHeap$
$\Rightarrow nat$
$\Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times FieldRefHeap$
$\Rightarrow bool$
$(\text{-}\vdash\text{-}\rightarrow*\text{-}*\ \text{-})$
where
$[\![n > 0$;

$p \vdash s \longrightarrow s'$;
exec-debug p s' $(n - 1)$ s'']]
$\Longrightarrow$ *exec-debug p s n s''* |

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

3.4.1 Heap Testing

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

fun *graphToSystem* :: *IRGraph* ⇒ *System* **where**
graphToSystem graph = ((λ*x. Some graph*), *JVMClasses* [])

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

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

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

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

definition *eg4-sq* :: *IRGraph* **where**
eg4-sq = *irgraph* [
(*0*, *StartNode None 4*, *VoidStamp*),
(*1*, *ParameterNode 0*, *default-stamp*),
(*3*, *MulNode 1 1*, *default-stamp*),
(*4*, *NewInstanceNode 4* ''*obj-class*'' *None 5*, *ObjectStamp* ''*obj-class*'' *True True False*),
(*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*. (*graphToSystem* (*eg4-sq*)) ⊢ ([(*eg4-sq*, *0*, *new-map-state*, *p3*), (*eg4-sq*, *0*, *new-map-state*, *p3*)], *new-heap*) →∗*3*∗ *res*}

end

3.5 Data-flow Tree Theorems

theory *IRTreeEvalThms*
  imports
    Graph.ValueThms
    IRTreeEval
begin

3.5.1 Deterministic Data-flow Evaluation

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

lemma *evalAllDet*:
  $[m,p] \vdash e\ [\mapsto]\ v1 \Longrightarrow$
   $[m,p] \vdash e\ [\mapsto]\ v2 \Longrightarrow$
   $v1 = v2$
  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: $is_IntVal32, is_IntVal64$ and the more general is_IntVal.

lemma *unary-eval-not-obj-ref*:
  shows *unary-eval op x* ≠ *ObjRef v*
  by (*cases op*; *cases x*; *auto*)

lemma *unary-eval-not-obj-str*:
  shows *unary-eval op x* ≠ *ObjStr v*
  by (*cases op*; *cases x*; *auto*)

lemma *unary-eval-not-array*:
  shows *unary-eval op x* ≠ *ArrayVal len v*
  by (*cases op*; *cases x*; *auto*)

lemma *unary-eval-int*:
  assumes *unary-eval op x* ≠ *UndefVal*
  shows *is-IntVal* (*unary-eval op x*)
 by (*cases unary-eval op x*; *auto simp add*: *assms unary-eval-not-obj-ref unary-eval-not-obj-str unary-eval-not-array*)

lemma *bin-eval-int*:
  assumes *bin-eval op x y* ≠ *UndefVal*
  shows *is-IntVal* (*bin-eval op x y*)
  using *assms*
  apply (*cases op*; *cases x*; *cases y*; *auto simp add*: *is-IntVal-def*)
  apply *presburger*+
  prefer *3* **prefer** *4*
    apply (*smt* (*verit*, *del-insts*) *new-int.simps*)
         apply (*smt* (*verit*, *del-insts*) *new-int.simps*)
         apply (*meson new-int-bin.simps*)+
         apply (*meson bool-to-val.elims*)
         apply (*meson bool-to-val.elims*)
        apply (*smt* (*verit*, *del-insts*) *new-int.simps*)+
  by (*metis bool-to-val.elims*)+

lemma *IntVal0*:
  (*IntVal 32 0*) = (*new-int 32 0*)
  by *auto*

lemma *IntVal1*:
  (*IntVal 32 1*) = (*new-int 32 1*)
  by *auto*

lemma *bin-eval-new-int*:
  assumes *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*)
  using *is-IntVal-def assms*
proof (*cases op*)
  case *BinAdd*
  then show *?thesis*
    using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** *presburger*
next
  case *BinMul*
  then show *?thesis*
    using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** *presburger*
next
  case *BinDiv*

then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*)
by (*meson new-int-bin.simps*)
next
case *BinMod*
then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*)
by (*meson new-int-bin.simps*)
next
case *BinSub*
then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** *presburger*
next
case *BinAnd*
then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** (*metis take-bit-and*)+
next
case *BinOr*
then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** (*metis take-bit-or*)+
next
case *BinXor*
then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** (*metis take-bit-xor*)+
next
case *BinShortCircuitOr*
then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*)
by (*metis IntVal1 bits-mod-0 bool-to-val.elims new-int.simps take-bit-eq-mod*)+
next
case *BinLeftShift*
then show *?thesis*
using *assms* **by** (*cases x*; *cases y*; *auto*)
next
case *BinRightShift*
then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** (*smt* (*verit*, *del-insts*) *new-int.simps*)+
next
case *BinURightShift*
then show *?thesis*
using *assms* **by** (*cases x*; *cases y*; *auto*)
next
case *BinIntegerEquals*
then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*)
apply (*metis* (*full-types*) *IntVal0 IntVal1 bool-to-val.simps*(*1*,*2*) *new-int.elims*)
by *presburger*
next
case *BinIntegerLessThan*

then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*)
apply (*metis* (*no-types*, *opaque-lifting*) *bool-to-val.simps*(*1* ,*2*) *bool-to-val.elims* *new-int.simps*
IntVal1 take-bit-of-0)
by *presburger*
next
case *BinIntegerBelow*
then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*)
apply (*metis bool-to-val.simps*(*1* ,*2*) *bool-to-val.elims new-int.simps IntVal0 Int-Val1*)
by *presburger*
next
case *BinIntegerTest*
then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*)
apply (*metis bool-to-val.simps*(*1* ,*2*) *bool-to-val.elims new-int.simps IntVal0 Int-Val1*)
by *presburger*
next
case *BinIntegerNormalizeCompare*
then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*) **using** *take-bit-of-0* **apply** *blast*
by (*metis IntVal1 intval-word.simps new-int.elims take-bit-minus-one-eq-mask*)+
next
case *BinIntegerMulHigh*
then show *?thesis*
using *assms* **apply** (*cases x*; *cases y*; *auto*)
prefer *2* **prefer** *5* **prefer** *8*
apply *presburger+*
by *metis+*
qed

lemma *int-stamp*:
assumes *is-IntVal v*
shows *is-IntegerStamp* (*constantAsStamp v*)
using *assms 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*) ≤ *int-signed-value b ival* ∧
int-signed-value b ival ≤ *snd* (*bit-bounds b*)
using *assms*(*2*) *int-signed-value-bounds* **by** *simp*
have *s*: *constantAsStamp v* = *IntegerStamp b* (*int-signed-value b ival*) (*int-signed-value b ival*)

using *assms*(*1*) **by** *simp*
then show *?thesis*
unfolding *s valid-stamp.simps* **using** *assms*(*2*) *bnds* **by** *linarith*
qed

lemma *validDefIntConst*:
assumes *v*: *v* = *IntVal b ival*
assumes *0* < *b* ∧ *b* ≤ *64*
assumes *take-bit b ival* = *ival*
shows *valid-value v* (*constantAsStamp v*)
proof −
have *bnds*: *fst* (*bit-bounds b*) ≤ *int-signed-value b ival* ∧
int-signed-value b ival ≤ *snd* (*bit-bounds b*)
using *assms*(*2*) *int-signed-value-bounds* **by** *simp*
have *s*: *constantAsStamp v* = *IntegerStamp b* (*int-signed-value b ival*) (*int-signed-value b ival*)
using *assms*(*1*) **by** *simp*
then show *?thesis*
using *assms validStampIntConst* **by** *simp*
qed

3.5.3 Evaluation Results are Valid

A valid value cannot be $UndefVal$.

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

lemma *valid-VoidStamp*[*elim*]:
shows *valid-value val VoidStamp* ⟹ *val* = *UndefVal*
by *simp*

lemma *valid-ObjStamp*[*elim*]:
shows *valid-value val* (*ObjectStamp klass exact nonNull alwaysNull*) ⟹ (∃ *v*. *val* = *ObjRef v*)
by (*metis Value.exhaust valid-value.simps*(*3*,*11*,*12*,*18*))

lemma *valid-int*[*elim*]:
shows *valid-value val* (*IntegerStamp b lo hi*) ⟹ (∃ *v*. *val* = *IntVal b v*)
using *valid-value.elims*(*2*) **by** *fastforce*

lemmas *valid-value-elims* =
valid-VoidStamp
valid-ObjStamp
valid-int

lemma *evaltree-not-undef*:
  fixes *m p e v*
  shows $([m,p] \vdash e \mapsto v) \Longrightarrow v \neq \mathit{UndefVal}$
  apply (*induction rule*: *evaltree.induct*) **by** (*auto simp add*: *wf-value-def*)

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

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

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

lemma *valid-value-signed-int-range* [*simp*]:
  assumes *valid-value val* (*IntegerStamp b lo hi*)
  assumes $lo < 0$
  shows $\exists\, v.\ (val = \mathit{IntVal}\ b\ v\ \wedge$
            $lo \leq \mathit{int\text{-}signed\text{-}value}\ b\ v\ \wedge$
            $\mathit{int\text{-}signed\text{-}value}\ b\ v \leq hi)$
  by (*metis valid-value.simps*(*1*) *assms*(*1*) *valid-int*)

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 $mono(UnaryExprop)$, but it is not obvious how to do this for both arguments of the binary expressions.

lemma *mono-unary*:
  assumes $x \geq x'$
  shows $(\mathit{UnaryExpr}\ op\ x) \geq (\mathit{UnaryExpr}\ op\ x')$
  using *assms* **by** *auto*

lemma *mono-binary*:
  assumes $x \geq x'$
  assumes $y \geq y'$
  shows $(\mathit{BinaryExpr}\ op\ x\ y) \geq (\mathit{BinaryExpr}\ op\ x'\ y')$

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

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

lemma *compatible-trans*:
compatible x y $\wedge$ *compatible y z* $\Longrightarrow$ *compatible x z*
by (*cases x*; *cases y*; *cases z*; *auto*)

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

lemma *mono-conditional*:
assumes $c \geq c'$
assumes $t \geq t'$
assumes $f \geq f'$
shows (*ConditionalExpr c t f*) $\geq$ (*ConditionalExpr* c' t' f')
proof (*simp only*: *le-expr-def*; (*rule allI*)+; *rule impI*)
fix *m p v*
assume *a*: $[m,p] \vdash$ *ConditionalExpr c t f* $\mapsto v$
then obtain *cond* **where** *c*: $[m,p] \vdash c \mapsto$ *cond*
by *auto*
then have c': $[m,p] \vdash c' \mapsto$ *cond*
using *assms* **by** *simp*

then obtain *tr* **where** *tr*: $[m,p] \vdash t \mapsto tr$
using *a* **by** *auto*
then have tr': $[m,p] \vdash t' \mapsto tr$
using *assms*(*2*) **by** *auto*
then obtain *fa* **where** *fa*: $[m,p] \vdash f \mapsto fa$
using *a* **by** *blast*
then have fa': $[m,p] \vdash f' \mapsto fa$
using *assms*(*3*) **by** *auto*
define *branch* **where** *b*: *branch* = (*if val-to-bool cond then t else f*)
define $branch'$ **where** b': $branch'$ = (*if val-to-bool cond then* t' *else* f')
then have *beval*: $[m,p] \vdash$ *branch* $\mapsto v$
using *a b c evalDet* **by** *blast*

from *beval* **have** $[m,p] \vdash branch' \mapsto v$

using *assms* **by** (*auto simp add*: *b b′*)
then show [*m*,*p*] ⊢ *ConditionalExpr c′ t′ f′* ↦ *v*
using *c′ fa′ tr′* **by** (*simp add*: *evaltree-not-undef b′ ConditionalExpr*)
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_eval / $unary_eval$ level, simply by saying $unfoldingunfold_evaltree$.

lemma *unfold-const*:
([*m*,*p*] ⊢ *ConstantExpr c* ↦ *v*) = (*wf-value v* ∧ *v* = *c*)
by *auto*

lemma *unfold-binary*:
shows ([*m*,*p*] ⊢ *BinaryExpr op xe ye* ↦ *val*) = (∃ *x y*.
(([*m*,*p*] ⊢ *xe* ↦ *x*) ∧
([*m*,*p*] ⊢ *ye* ↦ *y*) ∧
(*val* = *bin-eval op x y*) ∧
(*val* ≠ *UndefVal*)
)) (**is** *?L* = *?R*)
proof (*intro iffI*)
assume *3*: *?L*
show *?R* **by** (*rule evaltree.cases*[*OF 3*]; *blast*+)
next
assume *?R*
then obtain *x y* **where** [*m*,*p*] ⊢ *xe* ↦ *x*
and [*m*,*p*] ⊢ *ye* ↦ *y*
and *val* = *bin-eval op x y*
and *val* ≠ *UndefVal*
by *auto*
then show *?L*
by (*rule BinaryExpr*)
qed

lemma *unfold-unary*:
shows ([*m*,*p*] ⊢ *UnaryExpr op xe* ↦ *val*)
= (∃ *x*.
(([*m*,*p*] ⊢ *xe* ↦ *x*) ∧
(*val* = *unary-eval op x*) ∧
(*val* ≠ *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*: *unary-eval op x* ≠ *UndefVal*
shows ∃ *b v*. (*unary-eval op x* = *new-int b v* ∧

b = (*if op* ∈ *normal-unary* *then intval-bits x* *else*
if op ∈ *boolean-unary* *then 32* *else*
if op ∈ *unary-fixed-32-ops then 32* *else*
ir-resultBits op))
proof (*cases op*)
case *UnaryAbs*
then show *?thesis*
apply *auto*
by (*metis intval-bits.simps intval-abs.simps*(*1*) *UnaryAbs def new-int.elims*
unary-eval.simps(*1*)
intval-abs.elims)
next
case *UnaryNeg*
then show *?thesis*
apply *auto*
by (*metis def intval-bits.simps intval-negate.elims new-int.elims unary-eval.simps*(*2*))
next
case *UnaryNot*
then show *?thesis*
apply *auto*
by (*metis intval-bits.simps intval-not.elims new-int.simps unary-eval.simps*(*3*)
def)
next
case *UnaryLogicNegation*
then show *?thesis*
apply *auto*
by (*metis intval-bits.simps UnaryLogicNegation intval-logic-negation.elims new-int.elims*
def
unary-eval.simps(*4*))
next
case (*UnaryNarrow x51 x52*)
then show *?thesis*
using *assms* **apply** *auto*
subgoal premises *p*
proof −
obtain *xb xvv* **where** *xvv*: *x* = *IntVal xb xvv*
by (*metis UnaryNarrow def intval-logic-negation.cases intval-narrow.simps*(*2*,*3*,*4*,*5*)
unary-eval.simps(*5*))

then have *evalNotUndef*: *intval-narrow x51 x52 x* ≠ *UndefVal*
using *p* **by** *fast*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *new-int.elims intval-narrow.simps*(*1*) *xvv*)
qed done
next
case (*UnarySignExtend x61 x62*)
then show *?thesis*
using *assms* **apply** *auto*
subgoal premises *p*
proof −
obtain *xb xvv* **where** *xvv*: *x* = *IntVal xb xvv*
by (*metis Value.exhaust intval-sign-extend.simps*(*2*,*3*,*4*,*5*) *p*(*2*))
then have *evalNotUndef*: *intval-sign-extend x61 x62 x* ≠ *UndefVal*
using *p* **by** *fast*
then show *?thesis*
by (*metis intval-sign-extend.simps*(*1*) *new-int.elims xvv*)
qed done
next
case (*UnaryZeroExtend x71 x72*)
then show *?thesis*
using *assms* **apply** *auto*
subgoal premises *p*
proof −
obtain *xb xvv* **where** *xvv*: *x* = *IntVal xb xvv*
by (*metis Value.exhaust intval-zero-extend.simps*(*2*,*3*,*4*,*5*) *p*(*2*))
then have *evalNotUndef*: *intval-zero-extend x71 x72 x* ≠ *UndefVal*
using *p* **by** *fast*
then show *?thesis*
by (*metis intval-zero-extend.simps*(*1*) *new-int.elims xvv*)
qed done
next
case *UnaryIsNull*
then show *?thesis*
apply *auto*
by (*metis bool-to-val.simps*(*1*) *new-int.simps IntVal0 IntVal1 unary-eval.simps*(*8*)
assms def
intval-is-null.elims bool-to-val.elims)
next
case *UnaryReverseBytes*
then show *?thesis*
apply *auto*
by (*metis intval-bits.simps intval-reverse-bytes.elims new-int.elims unary-eval.simps*(*9*)
def)
next
case *UnaryBitCount*
then show *?thesis*
apply *auto*
by (*metis intval-bit-count.elims new-int.simps unary-eval.simps*(*10*) *intval-bit-count.simps*(*1*)

def)
qed

lemma *new-int-unused-bits-zero*:
assumes *IntVal b ival* = *new-int b ival0*
shows *take-bit b ival* = *ival*
by (*simp add*: *new-int-take-bits assms*)

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

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

assms)

lemma *eval-unused-bits-zero*:
$[m,p] \vdash xe \mapsto (IntVal\ b\ ix) \Longrightarrow take\text{-}bit\ b\ ix = ix$
proof (*induction xe*)
case (*UnaryExpr x1 xe*)
then show *?case*
by (*auto simp add*: *unary-eval-unused-bits-zero*)
next
case (*BinaryExpr x1 xe1 xe2*)
then show *?case*
by (*auto simp add*: *bin-eval-unused-bits-zero*)
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* (*no-types*, *opaque-lifting*) *Value.distinct*(*9*) *intval-bits.simps valid-value.elims*(*2*)
local.ParameterExpr ParameterExprE intval-word.simps)
next
case (*LeafExpr x1 x2*)
then show *?case*
apply *auto*
by (*metis* (*no-types*, *opaque-lifting*) *intval-bits.simps intval-word.simps valid-value.elims*(*2*)
valid-value.simps(*18*))

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

lemma unary-normal-bitsize:
  assumes unary-eval op x = IntVal b ival
  assumes op ∈ normal-unary
  shows ∃ ix. x = IntVal b ix
  using assms apply (cases op; auto) prefer 5
 apply (smt (verit, ccfv-threshold) Value.distinct(1) Value.inject(1) intval-reverse-bytes.elims
     new-int.simps)
 by (metis Value.distinct(1) Value.inject(1) intval-logic-negation.elims new-int.simps
     intval-not.elims intval-negate.elims intval-abs.elims)+

lemma unary-not-normal-bitsize:
  assumes unary-eval op x = IntVal b ival
  assumes op ∉ normal-unary ∧ op ∉ boolean-unary ∧ op ∉ unary-fixed-32-ops
  shows b = ir-resultBits op ∧ 0 < b ∧ b ≤ 64
  apply (cases op) prefer 8 prefer 10 prefer 10 using assms apply blast+
 by (smt(verit, ccfv-SIG) Value.distinct(1) assms(1) intval-bits.simps intval-narrow.elims
    intval-narrow-ok intval-zero-extend.elims linorder-not-less neq0-conv new-int.simps
     unary-eval.simps(5,6,7) IRUnaryOp.sel(4,5,6) intval-sign-extend.elims)+

lemma unary-eval-bitsize:
  assumes unary-eval op x = IntVal b ival
  assumes 2: x = IntVal bx ix
  assumes 0 < bx ∧ bx ≤ 64
  shows 0 < b ∧ b ≤ 64
  using assms apply (cases op; simp)
 by (metis Value.distinct(1) Value.inject(1) intval-narrow.simps(1) le-zero-eq int-
val-narrow-ok
     new-int.simps le-zero-eq gr-zeroI)+

lemma bin-eval-inputs-are-ints:
  assumes bin-eval op x y = IntVal b ix
  obtains xb yb xi yi where x = IntVal xb xi ∧ y = IntVal yb yi
proof −
```

have *bin-eval op x y* ≠ *UndefVal*
by (*simp add*: *assms*)
then show *?thesis*
using *assms that* **by** (*cases op*; *cases x*; *cases y*; *auto*)
qed

lemma *eval-bits-1-64*:
[*m*,*p*] ⊢ *xe* ↦ (*IntVal b ix*) ⟹ *0* < *b* ∧ *b* ≤ *64*
proof (*induction xe arbitrary*: *b ix*)
case (*UnaryExpr op x2*)
then obtain *xv* **where**
xv: ([*m*,*p*] ⊢ *x2* ↦ *xv*) ∧
IntVal b ix = *unary-eval op xv*
by (*auto simp add*: *unfold-binary*)
then have *b* = (*if op* ∈ *normal-unary then intval-bits xv else*
if op ∈ *unary-fixed-32-ops then 32 else*
if op ∈ *boolean-unary then 32 else*
ir-resultBits op)
by (*metis Value.disc*(*1*) *Value.discI*(*1*) *Value.sel*(*1*) *new-int.simps unary-eval-new-int*)
then show *?case*
by (*metis xv linorder-le-cases linorder-not-less numeral-less-iff semiring-norm*(*76*,*78*) *gr0I*
unary-normal-bitsize unary-not-normal-bitsize UnaryExpr.IH)
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*
by (*auto simp add*: *unfold-binary*)
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 (*smt* (*verit*, *best*) *Value.distinct*(*9*,*11*,*13*) *BinaryExpr.IH*(*1*) *xv bin-eval-inputs-are-ints xy*
intval-bits.elims le-add-same-cancel1 less-or-eq-imp-le numeral-Bit0 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*
apply *auto*
using *valid-value.elims*(*2*)
by (*metis valid-stamp.simps*(*1*) *intval-bits.simps valid-value.simps*(*18*))+

```
next
  case (LeafExpr x1 x2)
  then show ?case
    apply auto
    using valid-value.elims(1,2)
   by (metis Value.inject(1) valid-stamp.simps(1) valid-value.simps(18) Value.distinct(9))+
next
  case (ConstantExpr x)
  then show ?case
   by (metis wf-value-def constantAsStamp.simps(1) valid-stamp.simps(1) valid-value.simps(1)
       EvalTreeE(1))
next
  case (ConstantVar x)
  then show ?case
    by auto
next
  case (VariableExpr x1 x2)
  then show ?case
    by auto
qed


lemma bin-eval-normal-bits:
  assumes op ∈ binary-normal
  assumes bin-eval op x y = xy
  assumes xy ≠ UndefVal
  shows ∃ xv yv xyv b. (x = IntVal b xv ∧ y = IntVal b yv ∧ xy = IntVal b xyv)
  using assms apply simp
  proof (cases op ∈ binary-normal)
  case True
  then show ?thesis
    proof −
      have operator: xy = bin-eval op x y
        by (simp add: assms(2))
      obtain xv xb where xv: x = IntVal xb xv
      by (metis assms(3) bin-eval-inputs-are-ints bin-eval-int is-IntVal-def operator)
      obtain yv yb where yv: y = IntVal yb yv
      by (metis assms(3) bin-eval-inputs-are-ints bin-eval-int is-IntVal-def operator)
       then have notUndefMeansWidthSame: bin-eval op x y ≠ UndefVal ⟹ (xb
= yb)
        using assms apply (cases op; auto)
          by (metis intval-xor.simps(1) intval-or.simps(1) intval-div.simps(1) int-
val-mod.simps(1) intval-and.simps(1) intval-sub.simps(1)
            intval-mul.simps(1) intval-add.simps(1) new-int-bin.elims xv)+
      then have inWidthsSame: xb = yb
        using assms(3) operator by auto
      obtain ob xyv where out: xy = IntVal ob xyv
        by (metis Value.collapse(1) assms(3) bin-eval-int operator)
      then have yb = ob
```

using *assms* **apply** (*cases op*; *auto*)
apply (*simp add*: *inWidthsSame xv yv*)+
apply (*metis assms*(*3*) *intval-bits.simps new-int.simps new-int-bin.elims*)
apply (*metis xv yv Value.distinct*(*1*) *intval-mod.simps*(*1*) *new-int.simps new-int-bin.elims*)
by (*simp add*: *inWidthsSame xv yv*)+
then show *?thesis*
using *xv yv inWidthsSame assms out* **by** *blast*
qed
next
case *False*
then show *?thesis*
using *assms* **by** *simp*
qed

lemma *unfold-binary-width-bin-normal*:
assumes $op \in binary\text{-}normal$
shows $\bigwedge xv\ yv.$
$IntVal\ b\ val = bin\text{-}eval\ op\ xv\ yv \Longrightarrow$
$[m,p] \vdash xe \mapsto xv \Longrightarrow$
$[m,p] \vdash ye \mapsto yv \Longrightarrow$
$bin\text{-}eval\ op\ xv\ yv \neq UndefVal \Longrightarrow$
$\exists\, xa.$
$(([m,p] \vdash xe \mapsto IntVal\ b\ xa) \wedge$
$(\exists\, ya.\ (([m,p] \vdash ye \mapsto IntVal\ b\ ya) \wedge$
$bin\text{-}eval\ op\ xv\ yv = bin\text{-}eval\ op\ (IntVal\ b\ xa)\ (IntVal\ b\ ya))))$
using *assms* **apply** *simp*
subgoal premises *p* **for** *x y*
proof −
obtain *xv yv* **where** *eval*: $([m,p] \vdash xe \mapsto xv) \wedge ([m,p] \vdash ye \mapsto yv)$
using *p*(*2*,*3*) **by** *blast*
then obtain *xa bb* **where** *xa*: *xv* = *IntVal bb xa*
by (*metis bin-eval-inputs-are-ints evalDet p*(*1*,*2*))
then obtain *ya yb* **where** *ya*: *yv* = *IntVal yb ya*
by (*metis bin-eval-inputs-are-ints evalDet p*(*1*,*3*) *eval*)
then have *eqWidth*: *bb* = *b*
by (*metis intval-bits.simps p*(*1*,*2*,*4*) *assms eval xa bin-eval-normal-bits evalDet*)
then obtain *xy* **where** *eval0*: *bin-eval op x y* = *IntVal b xy*
by (*metis p*(*1*))
then have *sameVals*: *bin-eval op x y* = *bin-eval op xv yv*
by (*metis evalDet p*(*2*,*3*) *eval*)
then have *notUndefMeansSameWidth*: *bin-eval op xv yv* ≠ *UndefVal* ⟹ (*bb* = *yb*)
using *assms* **apply** (*cases op*; *auto*)
by (*metis intval-add.simps*(*1*) *intval-mul.simps*(*1*) *intval-div.simps*(*1*) *intval-mod.simps*(*1*) *intval-sub.simps*(*1*) *intval-and.simps*(*1*)
intval-or.simps(*1*) *intval-xor.simps*(*1*) *new-int-bin.simps xa ya*)+
have *unfoldVal*: *bin-eval op x y* = *bin-eval op* (*IntVal bb xa*) (*IntVal yb ya*)
unfolding *sameVals xa ya* **by** *simp*

```
    then have sameWidth: b = yb
      using eqWidth notUndefMeansSameWidth p(4) sameVals by force
    then show ?thesis
      using eqWidth eval xa ya unfoldVal by blast
  qed
  done

lemma unfold-binary-width:
  assumes op ∈ binary-normal
  shows ([m,p] ⊢ BinaryExpr op xe ye ↦ IntVal b val) = (∃ x y.
          (([m,p] ⊢ xe ↦ IntVal b x) ∧
           ([m,p] ⊢ ye ↦ IntVal b y) ∧
           (IntVal b val = bin-eval op (IntVal b x) (IntVal b y)) ∧
           (IntVal b val ≠ UndefVal)
        )) (is ?L = ?R)
proof (intro iffI)
  assume 3: ?L
  show ?R
    apply (rule evaltree.cases[OF 3]) apply auto
    apply (cases op ∈ binary-normal)
    using unfold-binary-width-bin-normal assms by force+
next
  assume R: ?R
  then obtain x y where [m,p] ⊢ xe ↦ IntVal b x
        and [m,p] ⊢ ye ↦ IntVal b y
        and new-int b val = bin-eval op (IntVal b x) (IntVal b y)
        and new-int b val ≠ 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*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = ConstantNode\ c \Longrightarrow$
$e = ConstantExpr\ c$
by (*induction rule*: *rep.induct*; *auto*)

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

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

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

lemma *rep-reverse-bytes* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = ReverseBytesNode\ x \Longrightarrow$
$(\exists xe.\ e = UnaryExpr\ UnaryReverseBytes\ xe)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-bit-count* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = BitCountNode\ x \Longrightarrow$
$(\exists xe.\ e = UnaryExpr\ UnaryBitCount\ xe)$
by (*induction rule*: *rep.induct*; *auto*)

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

lemma *rep-negate* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = NegateNode\ x \Longrightarrow$
$(\exists xe.\ e = UnaryExpr\ UnaryNeg\ xe)$

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *rep-unsigned-right-shift* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *UnsignedRightShiftNode x y* $\Longrightarrow$
($\exists$ *xe ye*. *e* = *BinaryExpr BinURightShift xe ye*)
by (*induction rule*: *rep.induct*; *auto*)

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

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

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

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

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

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

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

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

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

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

lemma *rep-bytecode-exception* [*rep*]:
  $g \vdash n \simeq e \Longrightarrow$
   (*kind g n*) = *BytecodeExceptionNode gu st n'* $\Longrightarrow$
   ($\exists$ *s*. *e* = *LeafExpr n s*)
  by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-new-array* [*rep*]:

$g \vdash n \simeq e \Longrightarrow$
$(kind\ g\ n) = NewArrayNode\ len\ st\ n' \Longrightarrow$
$(\exists\, s.\ e = LeafExpr\ n\ s)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-array-length* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$(kind\ g\ n) = ArrayLengthNode\ x\ n' \Longrightarrow$
$(\exists\, s.\ e = LeafExpr\ n\ s)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-load-index* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$(kind\ g\ n) = LoadIndexedNode\ index\ guard\ x\ n' \Longrightarrow$
$(\exists\, s.\ e = LeafExpr\ n\ s)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-store-index* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$(kind\ g\ n) = StoreIndexedNode\ check\ val\ st\ index\ guard\ x\ n' \Longrightarrow$
$(\exists\, s.\ e = LeafExpr\ n\ s)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-ref* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = RefNode\ n' \Longrightarrow$
$g \vdash n' \simeq e$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-pi* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = PiNode\ n'\ gu \Longrightarrow$
$g \vdash n' \simeq e$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-is-null* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = IsNullNode\ x \Longrightarrow$
$(\exists\, xe.\ e = (UnaryExpr\ UnaryIsNull\ xe))$
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* - ⇒
‹*match IRNode.distinct in f*: *node* - ≠ *PiNode* - - ⇒
‹*metis i e r d f*››››› |

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* - ⇒
‹*match IRNode.distinct in f*: *node* - - ≠ *PiNode* - - ⇒
‹*metis i e r d f*›››››› |
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* - ⇒
‹*match IRNode.distinct in f*: *node* - - - ≠ *PiNode* - - ⇒
‹*metis i e r d f*›››››› |
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* - ⇒
‹*match IRNode.distinct in f*: *node* - - - ≠ *PiNode* - - ⇒
‹*metis i e r d f*››››››)

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

lemma *repDet*:
shows $(g \vdash n \simeq e_1) \Longrightarrow (g \vdash n \simeq e_2) \Longrightarrow e_1 = e_2$
proof (*induction arbitrary*: e_2 *rule*: *rep.induct*)
case (*ConstantNode n c*)
then show *?case*
using *rep-constant* **by** *simp*
next
case (*ParameterNode n i s*)
then show *?case*
by (*metis IRNode.distinct*(*3655*) *IRNode.distinct*(*3697*) *ParameterNodeE rep-parameter*)
next
case (*ConditionalNode n c t f ce te fe*)
then show *?case*
by (*metis ConditionalNodeE IRNode.distinct*(*925*) *IRNode.distinct*(*967*) *IRNode.sel*(*90*) *IRNode.sel*(*93*) *IRNode.sel*(*94*) *rep-conditional*)
next
case (*AbsNode n x xe*)
then show *?case*
by (*solve-det node*: *AbsNode*)
next
case (*ReverseBytesNode n x xe*)
then show *?case*
by (*solve-det node*: *ReverseBytesNode*)
next
case (*BitCountNode n x xe*)

```
  then show ?case
    by (solve-det node: BitCountNode)
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 (DivNode n x y xe ye)
  then show ?case
    by (solve-det node: DivNode)
next
  case (ModNode n x y xe ye)
  then show ?case
    by (solve-det node: ModNode)
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 (IntegerTestNode n x y xe ye)
  then show ?case
    by (solve-det node: IntegerTestNode)
next
  case (IntegerNormalizeCompareNode n x y xe ye)
  then show ?case
    by (solve-det node: IntegerNormalizeCompareNode)
next
  case (IntegerMulHighNode n x xe)
  then show ?case
    by (solve-det node: IntegerMulHighNode)
next
  case (NarrowNode n x xe)
  then show ?case
    using NarrowNodeE rep-narrow
    by (metis IRNode.distinct(3361) IRNode.distinct(3403) IRNode.inject(36))
next
  case (SignExtendNode n x xe)
  then show ?case
    using SignExtendNodeE rep-sign-extend
    by (metis IRNode.distinct(3707) IRNode.distinct(3919) IRNode.inject(48))
next
  case (ZeroExtendNode n x xe)
```

then show *?case*
using *ZeroExtendNodeE rep-zero-extend*
by (*metis IRNode.distinct*(*3735*) *IRNode.distinct*(*4157*) *IRNode.inject*(*62*))
next
case (*LeafNode n s*)
then show *?case*
using *rep-load-field LeafNodeE*
by (*metis is-preevaluated.simps*(*48*) *is-preevaluated.simps*(*65*))
next
case (*RefNode n'*)
then show *?case*
using *rep-ref* **by** *blast*
next
case (*PiNode n v*)
then show *?case*
using *rep-pi* **by** *blast*
next
case (*IsNullNode n v*)
then show *?case*
using *IsNullNodeE rep-is-null*
by (*metis IRNode.distinct*(*2557*) *IRNode.distinct*(*2599*) *IRNode.inject*(*24*))
qed

lemma *repAllDet*:
$g \vdash xs\ [\simeq]\ e1 \Longrightarrow$
$g \vdash xs\ [\simeq]\ e2 \Longrightarrow$
$e1 = e2$
proof (*induction arbitrary*: *e2 rule*: *replist.induct*)
case *RepNil*
then show *?case*
using *replist.cases* **by** *auto*
next
case (*RepCons x xe xs xse*)
then show *?case*
by (*metis list.distinct*(*1*) *list.sel*(*1*,*3*) *repDet replist.cases*)
qed

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

lemma *graphDet*: $([g,m,p] \vdash n \mapsto v_1) \wedge ([g,m,p] \vdash n \mapsto v_2) \Longrightarrow v_1 = v_2$
by (*auto simp add*: *encodeEvalDet*)

lemma *encodeEvalAllDet*:
$[g, m, p] \vdash nids\ [\mapsto]\ vs \Longrightarrow [g, m, p] \vdash nids\ [\mapsto]\ vs' \Longrightarrow vs = vs'$
using *repAllDet evalAllDet*

by (*metis encodeEvalAll.simps*)

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 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 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 mono-unary repDet*)

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

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

lemma *mono-sign-extend*:
assumes *kind g1 n = SignExtendNode ib rb x* ∧ *kind g2 n = 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 mono-unary repDet*)

lemma *mono-zero-extend*:
assumes *kind g1 n = ZeroExtendNode ib rb x* ∧ *kind g2 n = 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 $e1 \geq e2$
by (*metis ZeroExtendNode assms mono-unary repDet*)

lemma *mono-conditional-graph*:
assumes *kind g1 n = ConditionalNode c t f* ∧ *kind g2 n = ConditionalNode c t f*
assumes $(g1 \vdash c \simeq ce1) \wedge (g2 \vdash c \simeq ce2)$
assumes $(g1 \vdash t \simeq te1) \wedge (g2 \vdash t \simeq te2)$
assumes $(g1 \vdash f \simeq fe1) \wedge (g2 \vdash f \simeq fe2)$
assumes $ce1 \geq ce2 \wedge te1 \geq te2 \wedge fe1 \geq fe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
by (*smt* (*verit*, *ccfv-SIG*) *ConditionalNode assms mono-conditional repDet le-expr-def*)

lemma *mono-add*:
assumes *kind g1 n = AddNode x y* ∧ *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 $e1 \geq e2$
by (*metis* (*no-types*, *lifting*) *AddNode mono-binary assms repDet*)

lemma *mono-mul*:
assumes *kind g1 n = MulNode x y* ∧ *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 $e1 \geq e2$
by (*metis* (*no-types*, *lifting*) *MulNode assms mono-binary repDet*)

lemma *mono-div*:
assumes *kind g1 n = SignedFloatingIntegerDivNode x y* ∧ *kind g2 n = SignedFloatingIntegerDivNode 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* ≥ *xe2* ∧ *ye1* ≥ *ye2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis* (*no-types*, *lifting*) *DivNode assms mono-binary repDet*)

lemma *mono-mod*:
assumes *kind g1 n* = *SignedFloatingIntegerRemNode x y* ∧ *kind g2 n* = *SignedFloatingIntegerRemNode x y*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes (*g1* ⊢ *y* ≃ *ye1*) ∧ (*g2* ⊢ *y* ≃ *ye2*)
assumes *xe1* ≥ *xe2* ∧ *ye1* ≥ *ye2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis* (*no-types*, *lifting*) *ModNode assms mono-binary repDet*)

lemma *term-graph-evaluation*:
(*g* ⊢ *n* ⊴ *e*) ⟹ (∀ *m p v* . ([*m*,*p*] ⊢ *e* ↦ *v*) ⟶ ([*g*,*m*,*p*] ⊢ *n* ↦ *v*))
using *graph-represents-expression-def encodeeval.simps* **by** (*auto*; *meson*)

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

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

lemma *not-excluded-keep-type*:
assumes *n* ∈ *ids g1*
assumes *n* ∉ *excluded*
assumes (*excluded* ⊴ *as-set g1*) ⊆ *as-set g2*
shows *kind g1 n* = *kind g2 n* ∧ *stamp g1 n* = *stamp g2 n*
using *assms* **by** (*auto simp add*: *domain-subtraction-def as-set-def*)

method *metis-node-eq-unary* **for** *node* :: *'a* ⇒ *IRNode* =
(*match IRNode.inject* **in** *i*: (*node* - = *node* -) = - ⇒
‹*metis i*›)
method *metis-node-eq-binary* **for** *node* :: *'a* ⇒ *'a* ⇒ *IRNode* =
(*match IRNode.inject* **in** *i*: (*node* - - = *node* - -) = - ⇒
‹*metis i*›)
method *metis-node-eq-ternary* **for** *node* :: *'a* ⇒ *'a* ⇒ *'a* ⇒ *IRNode* =
(*match IRNode.inject* **in** *i*: (*node* - - - = *node* - - -) = - ⇒
‹*metis i*›)

3.8.3 Lift Data-flow Tree Refinement to Graph Refinement

theorem *graph-semantics-preservation*:
  assumes *a*: $e1' \geq e2'$
  assumes *b*: $(\{n'\} \trianglelefteq as\text{-}set\ g1) \subseteq as\text{-}set\ g2$
  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*: $n \in ids\ g1$
  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 *f* **by** (*simp add*: *repDet True c*)
    have *h*: $(g2 \vdash n \simeq e2') \wedge e1' \geq e2'$
      using *a* **by** (*simp add*: *d True*)
    then show *?thesis*
      by (*auto simp add*: *g*)
  next
    case *False*
    have $n \notin \{n'\}$
      by (*simp add*: *False*)
    then have *i*: $kind\ g1\ n = kind\ g2\ n \wedge stamp\ g1\ n = stamp\ g2\ n$
      using *not-excluded-keep-type 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*: $g1 \vdash n \simeq ConditionalExpr\ ce1\ te1\ fe1$
      using *ConditionalNode* **by** (*simp add*: *ConditionalNode.hyps*(*2*) *rep.ConditionalNode f*)
      obtain *cn tn fn* **where** *l*: $kind\ g1\ n = ConditionalNode\ cn\ tn\ fn$
        by (*auto simp add*: *ConditionalNode.hyps*(*1*))
      then have *mc*: $g1 \vdash cn \simeq ce1$

```
      using ConditionalNode.hyps(1,2) by simp
    from l have mt: g1 ⊢ tn ≃ te1
      using ConditionalNode.hyps(1,3) by simp
    from l have mf: g1 ⊢ fn ≃ fe1
      using ConditionalNode.hyps(1,4) by simp
    then show ?case
    proof −
      have g1 ⊢ cn ≃ ce1
        by (simp add: mc)
      have g1 ⊢ tn ≃ te1
        by (simp add: mt)
      have g1 ⊢ fn ≃ fe1
        by (simp add: mf)
      have cer: ∃ ce2. (g2 ⊢ cn ≃ ce2) ∧ ce1 ≥ ce2
       using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
        by (metis-node-eq-ternary ConditionalNode)
      have ter: ∃ te2. (g2 ⊢ tn ≃ te2) ∧ te1 ≥ te2
       using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
        by (metis-node-eq-ternary ConditionalNode)
      have ∃ fe2. (g2 ⊢ fn ≃ fe2) ∧ fe1 ≥ fe2
       using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
        by (metis-node-eq-ternary ConditionalNode)
      then have ∃ ce2 te2 fe2. (g2 ⊢ n ≃ ConditionalExpr ce2 te2 fe2) ∧
             ConditionalExpr ce1 te1 fe1 ≥ ConditionalExpr ce2 te2 fe2
        apply meson
     by (smt (verit, best) mono-conditional ConditionalNode.prems l rep.ConditionalNode
cer ter)
      then show ?thesis
        by meson
    qed
  next
    case (AbsNode n x xe1)
    have k: g1 ⊢ n ≃ UnaryExpr UnaryAbs xe1
      using AbsNode by (simp add: AbsNode.hyps(2) rep.AbsNode f)
    obtain xn where l: kind g1 n = AbsNode xn
      by (auto simp add: AbsNode.hyps(1))
    then have m: g1 ⊢ xn ≃ xe1
      using AbsNode.hyps(1,2) by simp
    then show ?case
    proof (cases xn = n′)
      case True
      then have n: xe1 = e1′
        using m by (simp add: repDet c)
      then have ev: g2 ⊢ n ≃ UnaryExpr UnaryAbs e2′
        using l d by (simp add: rep.AbsNode True AbsNode.prems)
      then have r: UnaryExpr UnaryAbs e1′ ≥ UnaryExpr UnaryAbs e2′
```

by (*meson a mono-unary*)
then show *?thesis*
by (*metis n ev*)
next
case *False*
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *m*)
have ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *AbsNode False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff*
by (*metis-node-eq-unary AbsNode*)
then have ∃ *xe2*. (*g2* ⊢ *n* ≃ *UnaryExpr UnaryAbs xe2*) ∧
UnaryExpr UnaryAbs xe1 ≥ *UnaryExpr UnaryAbs xe2*
by (*metis AbsNode.prems l mono-unary rep.AbsNode*)
then show *?thesis*
by *meson*
qed
next
case (*ReverseBytesNode n x xe1*)
have *k*: *g1* ⊢ *n* ≃ *UnaryExpr UnaryReverseBytes xe1*
by (*simp add*: *ReverseBytesNode.hyps*(*1*,*2*) *rep.ReverseBytesNode*)
obtain *xn* **where** *l*: *kind g1 n* = *ReverseBytesNode xn*
by (*simp add*: *ReverseBytesNode.hyps*(*1*))
then have *m*: *g1* ⊢ *xn* ≃ *xe1*
by (*metis IRNode.inject*(*45*) *ReverseBytesNode.hyps*(*1*,*2*))
then show *?case*
proof (*cases xn* = *n′*)
case *True*
then have *n*: *xe1* = *e1′*
using *m* **by** (*simp add*: *repDet c*)
then have *ev*: *g2* ⊢ *n* ≃ *UnaryExpr UnaryReverseBytes e2′*
using *ReverseBytesNode.prems True d l rep.ReverseBytesNode* **by** *presburger*
then have *r*: *UnaryExpr UnaryReverseBytes e1′* ≥ *UnaryExpr UnaryReverseBytes e2′*
by (*meson a mono-unary*)
then show *?thesis*
by (*metis n ev*)
next
case *False*
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *m*)
have ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
by (*metis False IRNode.inject*(*45*) *ReverseBytesNode.IH ReverseBytesNode.hyps*(*1*,*2*) *b l*
encodes-contains ids-some not-excluded-keep-type singleton-iff)
then have ∃ *xe2*. (*g2* ⊢ *n* ≃ *UnaryExpr UnaryReverseBytes xe2*) ∧
UnaryExpr UnaryReverseBytes xe1 ≥ *UnaryExpr UnaryReverseBytes xe2*
by (*metis ReverseBytesNode.prems l mono-unary rep.ReverseBytesNode*)
then show *?thesis*

```
        by meson
      qed
    next
      case (BitCountNode n x xe1)
      have k: g1 ⊢ n ≃ UnaryExpr UnaryBitCount xe1
        by (simp add: BitCountNode.hyps(1,2) rep.BitCountNode)
      obtain xn where l: kind g1 n = BitCountNode xn
        by (simp add: BitCountNode.hyps(1))
      then have m: g1 ⊢ xn ≃ xe1
        by (metis BitCountNode.hyps(1,2) IRNode.inject(6))
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1'
          using m by (simp add: repDet c)
        then have ev: g2 ⊢ n ≃ UnaryExpr UnaryBitCount e2'
          using BitCountNode.prems True d l rep.BitCountNode by presburger
        then have r: UnaryExpr UnaryBitCount e1' ≥ UnaryExpr UnaryBitCount
e2'
          by (meson a mono-unary)
        then show ?thesis
          by (metis n ev)
      next
        case False
        have g1 ⊢ xn ≃ xe1
          by (simp add: m)
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          by (metis BitCountNode.IH BitCountNode.hyps(1) False IRNode.inject(6)
b emptyE insertE l m
            no-encoding not-excluded-keep-type)
        then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr UnaryBitCount xe2) ∧
      UnaryExpr UnaryBitCount xe1 ≥ UnaryExpr UnaryBitCount xe2
          by (metis BitCountNode.prems l mono-unary rep.BitCountNode)
        then show ?thesis
          by meson
      qed
    next
      case (NotNode n x xe1)
      have k: g1 ⊢ n ≃ UnaryExpr UnaryNot xe1
        using NotNode by (simp add: NotNode.hyps(2) rep.NotNode f)
      obtain xn where l: kind g1 n = NotNode xn
        by (auto simp add: NotNode.hyps(1))
      then have m: g1 ⊢ xn ≃ xe1
        using NotNode.hyps(1,2) by simp
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1'
          using m by (simp add: repDet c)
```

```
    then have ev: g2 ⊢ n ≃ UnaryExpr UnaryNot e2'
      using l by (simp add: rep.NotNode d True NotNode.prems)
    then have r: UnaryExpr UnaryNot e1' ≥ UnaryExpr UnaryNot e2'
      by (meson a mono-unary)
    then show ?thesis
      by (metis n ev)
  next
    case False
    have g1 ⊢ xn ≃ xe1
      by (simp add: m)
    have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
      using NotNode False b l not-excluded-keep-type singletonD no-encoding
      by (metis-node-eq-unary NotNode)
    then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr UnaryNot xe2) ∧
       UnaryExpr UnaryNot xe1 ≥ UnaryExpr UnaryNot xe2
      by (metis NotNode.prems l mono-unary rep.NotNode)
    then show ?thesis
      by meson
  qed
next
  case (NegateNode n x xe1)
  have k: g1 ⊢ n ≃ UnaryExpr UnaryNeg xe1
    using NegateNode by (simp add: NegateNode.hyps(2) rep.NegateNode f)
  obtain xn where l: kind g1 n = NegateNode xn
    by (auto simp add: NegateNode.hyps(1))
  then have m: g1 ⊢ xn ≃ xe1
    using NegateNode.hyps(1,2) by simp
  then show ?case
  proof (cases xn = n')
    case True
    then have n: xe1 = e1'
      using m by (simp add: c repDet)
    then have ev: g2 ⊢ n ≃ UnaryExpr UnaryNeg e2'
      using l by (simp add: rep.NegateNode True NegateNode.prems d)
    then have r: UnaryExpr UnaryNeg e1' ≥ UnaryExpr UnaryNeg e2'
      by (meson a mono-unary)
    then show ?thesis
      by (metis n ev)
  next
    case False
    have g1 ⊢ xn ≃ xe1
      by (simp add: m)
    have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
      using NegateNode False b l not-excluded-keep-type singletonD no-encoding
      by (metis-node-eq-unary NegateNode)
    then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr UnaryNeg xe2) ∧
       UnaryExpr UnaryNeg xe1 ≥ UnaryExpr UnaryNeg xe2
      by (metis NegateNode.prems l mono-unary rep.NegateNode)
    then show ?thesis
```

```
        by meson
      qed
    next
      case (LogicNegationNode n x xe1)
      have k: g1 ⊢ n ≃ UnaryExpr UnaryLogicNegation xe1
      using LogicNegationNode by (simp add: LogicNegationNode.hyps(2) rep.LogicNegationNode)
      obtain xn where l: kind g1 n = LogicNegationNode xn
        by (simp add: LogicNegationNode.hyps(1))
      then have m: g1 ⊢ xn ≃ xe1
        using LogicNegationNode.hyps(1,2) by simp
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1'
          using m by (simp add: c repDet)
        then have ev: g2 ⊢ n ≃ UnaryExpr UnaryLogicNegation e2'
        using l by (simp add: rep.LogicNegationNode True LogicNegationNode.prems
d
                                LogicNegationNode.hyps(1))
        then have r: UnaryExpr UnaryLogicNegation e1' ≥ UnaryExpr UnaryLog-
icNegation e2'
          by (meson a mono-unary)
        then show ?thesis
          by (metis n ev)
      next
        case False
        have g1 ⊢ xn ≃ xe1
          by (simp add: m)
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
            using LogicNegationNode False b l not-excluded-keep-type singletonD
no-encoding
          by (metis-node-eq-unary LogicNegationNode)
        then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr UnaryLogicNegation xe2) ∧
 UnaryExpr UnaryLogicNegation xe1 ≥ UnaryExpr UnaryLogicNegation xe2
          by (metis LogicNegationNode.prems l mono-unary rep.LogicNegationNode)
        then show ?thesis
          by meson
      qed
    next
      case (AddNode n x y xe1 ye1)
      have k: g1 ⊢ n ≃ BinaryExpr BinAdd xe1 ye1
        using AddNode by (simp add: AddNode.hyps(2) rep.AddNode f)
      obtain xn yn where l: kind g1 n = AddNode xn yn
        by (simp add: AddNode.hyps(1))
      then have mx: g1 ⊢ xn ≃ xe1
        using AddNode.hyps(1,2) by simp
      from l have my: g1 ⊢ yn ≃ ye1
        using AddNode.hyps(1,3) by simp
      then show ?case
```

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

```
  next
    case (DivNode n x y xe1 ye1)
    have k: g1 ⊢ n ≃ BinaryExpr BinDiv xe1 ye1
      using DivNode by (simp add: DivNode.hyps(2) rep.DivNode f)
    obtain xn yn where l: kind g1 n = SignedFloatingIntegerDivNode xn yn
      by (simp add: DivNode.hyps(1))
    then have mx: g1 ⊢ xn ≃ xe1
      using DivNode.hyps(1,2) by simp
    from l have my: g1 ⊢ yn ≃ ye1
      using DivNode.hyps(1,3) by simp
    then show ?case
    proof –
      have g1 ⊢ xn ≃ xe1
        by (simp add: mx)
      have g1 ⊢ yn ≃ ye1
        by (simp add: my)
      have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
           using DivNode  a  b  c  d  l  no-encoding  not-excluded-keep-type  repDet
singletonD
        by (metis-node-eq-binary SignedFloatingIntegerDivNode)
      have ∃ ye2. (g2 ⊢ yn ≃ ye2) ∧ ye1 ≥ ye2
      using DivNode a b c d l no-encoding not-excluded-keep-type repDet singletonD
        by (metis-node-eq-binary SignedFloatingIntegerDivNode)
      then have ∃ xe2 ye2. (g2 ⊢ n ≃ BinaryExpr BinDiv xe2 ye2) ∧
          BinaryExpr BinDiv xe1 ye1 ≥ BinaryExpr BinDiv xe2 ye2
        by (metis DivNode.prems l mono-binary rep.DivNode xer)
      then show ?thesis
        by meson
    qed
  next
    case (ModNode n x y xe1 ye1)
    have k: g1 ⊢ n ≃ BinaryExpr BinMod xe1 ye1
      using ModNode by (simp add: ModNode.hyps(2) rep.ModNode f)
    obtain xn yn where l: kind g1 n = SignedFloatingIntegerRemNode xn yn
      by (simp add: ModNode.hyps(1))
    then have mx: g1 ⊢ xn ≃ xe1
      using ModNode.hyps(1,2) by simp
    from l have my: g1 ⊢ yn ≃ ye1
      using ModNode.hyps(1,3) by simp
    then show ?case
    proof –
      have g1 ⊢ xn ≃ xe1
        by (simp add: mx)
      have g1 ⊢ yn ≃ ye1
        by (simp add: my)
      have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using ModNode  a  b  c  d  l  no-encoding  not-excluded-keep-type  repDet
singletonD
        by (metis-node-eq-binary SignedFloatingIntegerRemNode)
```

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

```
      proof -
        have g1 ⊢ xn ≃ xe1
          by (simp add: mx)
        have g1 ⊢ yn ≃ ye1
          by (simp add: my)
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
             using AndNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
          by (metis-node-eq-binary AndNode)
        have ∃ ye2. (g2 ⊢ yn ≃ ye2) ∧ ye1 ≥ ye2
             using AndNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
          by (metis-node-eq-binary AndNode)
        then have ∃ xe2 ye2. (g2 ⊢ n ≃ BinaryExpr BinAnd xe2 ye2) ∧
            BinaryExpr BinAnd xe1 ye1 ≥ BinaryExpr BinAnd xe2 ye2
          by (metis AndNode.prems l mono-binary rep.AndNode xer)
        then show ?thesis
          by meson
      qed
    next
      case (OrNode n x y xe1 ye1)
      have k: g1 ⊢ n ≃ BinaryExpr BinOr xe1 ye1
        using OrNode by (simp add: OrNode.hyps(2) rep.OrNode f)
      obtain xn yn where l: kind g1 n = OrNode xn yn
        using OrNode.hyps(1) by simp
      then have mx: g1 ⊢ xn ≃ xe1
        using OrNode.hyps(1,2) by simp
      from l have my: g1 ⊢ yn ≃ ye1
        using OrNode.hyps(1,3) by simp
      then show ?case
      proof -
        have g1 ⊢ xn ≃ xe1
          by (simp add: mx)
        have g1 ⊢ yn ≃ ye1
          by (simp add: my)
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
        using OrNode a b c d l no-encoding not-excluded-keep-type repDet singletonD
          by (metis-node-eq-binary OrNode)
        have ∃ ye2. (g2 ⊢ yn ≃ ye2) ∧ ye1 ≥ ye2
        using OrNode a b c d l no-encoding not-excluded-keep-type repDet singletonD
          by (metis-node-eq-binary OrNode)
        then have ∃ xe2 ye2. (g2 ⊢ n ≃ BinaryExpr BinOr xe2 ye2) ∧
            BinaryExpr BinOr xe1 ye1 ≥ BinaryExpr BinOr xe2 ye2
          by (metis OrNode.prems l mono-binary rep.OrNode xer)
        then show ?thesis
          by meson
      qed
    next
      case (XorNode n x y xe1 ye1)
```

have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinXor xe1 ye1*
  using *XorNode* **by** (*simp add*: *XorNode.hyps*(*2*) *rep.XorNode f*)
obtain *xn yn* **where** *l*: *kind g1 n* = *XorNode xn yn*
  using *XorNode.hyps*(*1*) **by** *simp*
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
  using *XorNode.hyps*(*1*,*2*) **by** *simp*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
  using *XorNode.hyps*(*1*,*3*) **by** *simp*
then show *?case*
proof −
  have *g1* ⊢ *xn* ≃ *xe1*
    by (*simp add*: *mx*)
  have *g1* ⊢ *yn* ≃ *ye1*
    by (*simp add*: *my*)
  have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
      using *XorNode a b c d l no-encoding not-excluded-keep-type repDet*
singletonD
    by (*metis-node-eq-binary XorNode*)
  have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
      using *XorNode a b c d l no-encoding not-excluded-keep-type repDet*
singletonD
    by (*metis-node-eq-binary XorNode*)
  then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinXor xe2 ye2*) ∧
    BinaryExpr BinXor xe1 ye1 ≥ *BinaryExpr BinXor xe2 ye2*
    by (*metis XorNode.prems l mono-binary rep.XorNode xer*)
  then show *?thesis*
    by *meson*
qed
next
case (*ShortCircuitOrNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinShortCircuitOr xe1 ye1*
using *ShortCircuitOrNode* **by** (*simp add*: *ShortCircuitOrNode.hyps*(*2*) *rep.ShortCircuitOrNode*
f)
obtain *xn yn* **where** *l*: *kind g1 n* = *ShortCircuitOrNode xn yn*
  using *ShortCircuitOrNode.hyps*(*1*) **by** *simp*
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
  using *ShortCircuitOrNode.hyps*(*1*,*2*) **by** *simp*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
  using *ShortCircuitOrNode.hyps*(*1*,*3*) **by** *simp*
then show *?case*
proof −
  have *g1* ⊢ *xn* ≃ *xe1*
    by (*simp add*: *mx*)
  have *g1* ⊢ *yn* ≃ *ye1*
    by (*simp add*: *my*)
  have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
    using *ShortCircuitOrNode a b c d l no-encoding not-excluded-keep-type*
repDet singletonD
    by (*metis-node-eq-binary ShortCircuitOrNode*)

have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *ShortCircuitOrNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary ShortCircuitOrNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinShortCircuitOr xe2 ye2*) ∧
BinaryExpr BinShortCircuitOr xe1 ye1 ≥ *BinaryExpr BinShortCircuitOr xe2 ye2*
by (*metis ShortCircuitOrNode.prems l mono-binary rep.ShortCircuitOrNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*LeftShiftNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinLeftShift xe1 ye1*
using *LeftShiftNode* **by** (*simp add*: *LeftShiftNode.hyps*(*2*) *rep.LeftShiftNode f*)
obtain *xn yn* **where** *l*: *kind g1 n* = *LeftShiftNode xn yn*
using *LeftShiftNode.hyps*(*1*) **by** *simp*
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *LeftShiftNode.hyps*(*1*,*2*) **by** *simp*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *LeftShiftNode.hyps*(*1*,*3*) **by** *simp*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *mx*)
have *g1* ⊢ *yn* ≃ *ye1*
by (*simp add*: *my*)
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *LeftShiftNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary LeftShiftNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *LeftShiftNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary LeftShiftNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinLeftShift xe2 ye2*) ∧
BinaryExpr BinLeftShift xe1 ye1 ≥ *BinaryExpr BinLeftShift xe2 ye2*
by (*metis LeftShiftNode.prems l mono-binary rep.LeftShiftNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*RightShiftNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinRightShift xe1 ye1*
using *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** *simp*

then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *RightShiftNode.hyps*(*1*,*2*) **by** *simp*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *RightShiftNode.hyps*(*1*,*3*) **by** *simp*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *mx*)
have *g1* ⊢ *yn* ≃ *ye1*
by (*simp add*: *my*)
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *RightShiftNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary RightShiftNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *RightShiftNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary RightShiftNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinRightShift xe2 ye2*) ∧ *BinaryExpr BinRightShift xe1 ye1* ≥ *BinaryExpr BinRightShift xe2 ye2*
by (*metis RightShiftNode.prems l mono-binary rep.RightShiftNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*UnsignedRightShiftNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinURightShift xe1 ye1*
using *UnsignedRightShiftNode* **by** (*simp add*: *UnsignedRightShiftNode.hyps*(*2*) *rep.UnsignedRightShiftNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *UnsignedRightShiftNode xn yn*
using *UnsignedRightShiftNode.hyps*(*1*) **by** *simp*
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *UnsignedRightShiftNode.hyps*(*1*,*2*) **by** *simp*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *UnsignedRightShiftNode.hyps*(*1*,*3*) **by** *simp*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *mx*)
have *g1* ⊢ *yn* ≃ *ye1*
by (*simp add*: *my*)
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *UnsignedRightShiftNode a b c d no-encoding not-excluded-keep-type repDet singletonD l*
by (*metis-node-eq-binary UnsignedRightShiftNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *UnsignedRightShiftNode a b c d no-encoding not-excluded-keep-type*

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

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

∧
BinaryExpr BinIntegerLessThan xe1 ye1 ≥ *BinaryExpr BinIntegerLessThan xe2 ye2*
by (*metis IntegerLessThanNode.prems l mono-binary rep.IntegerLessThanNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*IntegerTestNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinIntegerTest xe1 ye1*
using *IntegerTestNode* **by** (*meson rep.IntegerTestNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *IntegerTestNode xn yn*
by (*simp add*: *IntegerTestNode.hyps*(*1*))
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *IRNode.inject*(*21*) *IntegerTestNode.hyps*(*1*,*2*) **by** *presburger*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
by (*metis IRNode.inject*(*21*) *IntegerTestNode.hyps*(*1*,*3*))
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *mx*)
have *g1* ⊢ *yn* ≃ *ye1*
by (*simp add*: *my*)
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *IntegerTestNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis IRNode.inject*(*21*))
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *IntegerLessThanNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis IRNode.inject*(*21*) *IntegerTestNode.IH*(*2*) *IntegerTestNode.hyps*(*1*) *my*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinIntegerTest xe2 ye2*) ∧
BinaryExpr BinIntegerTest xe1 ye1 ≥ *BinaryExpr BinIntegerTest xe2 ye2*
by (*metis IntegerTestNode.prems l mono-binary xer rep.IntegerTestNode*)
then show *?thesis*
by *meson*
qed
next
case (*IntegerNormalizeCompareNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinIntegerNormalizeCompare xe1 ye1*
by (*simp add*: *IntegerNormalizeCompareNode.hyps*(*1*,*2*,*3*) *rep.IntegerNormalizeCompareNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *IntegerNormalizeCompareNode xn yn*
by (*simp add*: *IntegerNormalizeCompareNode.hyps*(*1*))
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *IRNode.inject*(*20*) *IntegerNormalizeCompareNode.hyps*(*1*,*2*) **by** *presburger*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*

using *IRNode.inject(20) IntegerNormalizeCompareNode.hyps(1,3)* **by** *presburger*
then show *?case*
proof −
have *g1 ⊢ xn ≃ xe1*
by (*simp add: mx*)
have *g1 ⊢ yn ≃ ye1*
by (*simp add: my*)
have *xer*: ∃ *xe2*. (*g2 ⊢ xn ≃ xe2*) ∧ *xe1 ≥ xe2*
by (*metis IRNode.inject(20) IntegerNormalizeCompareNode.IH(1) l mx no-encoding a b c d*
IntegerNormalizeCompareNode.hyps(1) emptyE insertE not-excluded-keep-type repDet)
have ∃ *ye2*. (*g2 ⊢ yn ≃ ye2*) ∧ *ye1 ≥ ye2*
by (*metis IRNode.inject(20) IntegerNormalizeCompareNode.IH(2) my no-encoding a b c d l*
IntegerNormalizeCompareNode.hyps(1) emptyE insertE not-excluded-keep-type repDet)
then have ∃ *xe2 ye2*. (*g2 ⊢ n ≃ BinaryExpr BinIntegerNormalizeCompare xe2 ye2*) ∧
BinaryExpr BinIntegerNormalizeCompare xe1 ye1 ≥ BinaryExpr BinIntegerNormalizeCompare xe2 ye2
by (*metis IntegerNormalizeCompareNode.prems l mono-binary rep.IntegerNormalizeCompareNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*IntegerMulHighNode n x y xe1 ye1*)
have *k*: *g1 ⊢ n ≃ BinaryExpr BinIntegerMulHigh xe1 ye1*
by (*simp add: IntegerMulHighNode.hyps(1,2,3) rep.IntegerMulHighNode*)
obtain *xn yn* **where** *l*: *kind g1 n = IntegerMulHighNode xn yn*
by (*simp add: IntegerMulHighNode.hyps(1)*)
then have *mx*: *g1 ⊢ xn ≃ xe1*
using *IRNode.inject(19) IntegerMulHighNode.hyps(1,2)* **by** *presburger*
from *l* **have** *my*: *g1 ⊢ yn ≃ ye1*
using *IRNode.inject(19) IntegerMulHighNode.hyps(1,3)* **by** *presburger*
then show *?case*
proof −
have *g1 ⊢ xn ≃ xe1*
by (*simp add: mx*)
have *g1 ⊢ yn ≃ ye1*
by (*simp add: my*)
have *xer*: ∃ *xe2*. (*g2 ⊢ xn ≃ xe2*) ∧ *xe1 ≥ xe2*
by (*metis IRNode.inject(19) IntegerMulHighNode.IH(1) IntegerMulHighNode.hyps(1) a b c d*
emptyE insertE l mx no-encoding not-excluded-keep-type repDet)
have ∃ *ye2*. (*g2 ⊢ yn ≃ ye2*) ∧ *ye1 ≥ ye2*
by (*metis IRNode.inject(19) IntegerMulHighNode.IH(2) IntegerMulHigh-*

```
Node.hyps(1) a b c d
            emptyE insertE l my no-encoding not-excluded-keep-type repDet)
      then have ∃ xe2 ye2. (g2 ⊢ n ≃ BinaryExpr BinIntegerMulHigh xe2 ye2)
∧
 BinaryExpr BinIntegerMulHigh xe1 ye1 ≥ BinaryExpr BinIntegerMulHigh xe2 ye2
      by (metis IntegerMulHighNode.prems l mono-binary rep.IntegerMulHighNode
xer)
      then show ?thesis
        by meson
    qed
  next
    case (NarrowNode n inputBits resultBits x xe1)
    have k: g1 ⊢ n ≃ UnaryExpr (UnaryNarrow inputBits resultBits) xe1
      using 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 simp
    then have m: g1 ⊢ xn ≃ xe1
      using NarrowNode.hyps(1,2) by simp
    then show ?case
    proof (cases xn = n')
      case True
      then have n: xe1 = e1'
        using m by (simp add: repDet c)
       then have ev: g2 ⊢ n ≃ UnaryExpr (UnaryNarrow inputBits resultBits)
e2'
        using l by (simp add: rep.NarrowNode d True NarrowNode.prems)
      then have r: UnaryExpr (UnaryNarrow inputBits resultBits) e1' ≥
                  UnaryExpr (UnaryNarrow inputBits resultBits) e2'
        by (meson a mono-unary)
      then show ?thesis
        by (metis n ev)
    next
      case False
      have g1 ⊢ xn ≃ xe1
        by (simp add: m)
      have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
      using NarrowNode False b encodes-contains l not-excluded-keep-type not-in-g
singleton-iff
        by (metis-node-eq-ternary NarrowNode)
      then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr (UnaryNarrow inputBits resultBits)
xe2) ∧
                                UnaryExpr (UnaryNarrow inputBits resultBits) xe1 ≥
                                  UnaryExpr (UnaryNarrow inputBits resultBits) xe2
        by (metis NarrowNode.prems l mono-unary rep.NarrowNode)
      then show ?thesis
        by meson
    qed
  next
    case (SignExtendNode n inputBits resultBits x xe1)
```

```
    have k: g1 ⊢ n ≃ UnaryExpr (UnarySignExtend inputBits resultBits) xe1
    using SignExtendNode by (simp add: SignExtendNode.hyps(2) rep.SignExtendNode)
    obtain xn where l: kind g1 n = SignExtendNode inputBits resultBits xn
      using SignExtendNode.hyps(1) by simp
    then have m: g1 ⊢ xn ≃ xe1
      using SignExtendNode.hyps(1,2) by simp
    then show ?case
    proof (cases xn = n')
      case True
      then have n: xe1 = e1'
        using m by (simp add: repDet c)
     then have ev: g2 ⊢ n ≃ UnaryExpr (UnarySignExtend inputBits resultBits)
e2'
       using l by (simp add: True d rep.SignExtendNode SignExtendNode.prems)
      then have r: UnaryExpr (UnarySignExtend inputBits resultBits) e1' ≥
                UnaryExpr (UnarySignExtend inputBits resultBits) e2'
        by (meson a mono-unary)
      then show ?thesis
        by (metis n ev)
    next
      case False
      have g1 ⊢ xn ≃ xe1
        by (simp add: m)
      have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
         using SignExtendNode False b encodes-contains l not-excluded-keep-type
not-in-g
            singleton-iff
        by (metis-node-eq-ternary SignExtendNode)
        then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr (UnarySignExtend inputBits
resultBits) xe2) ∧
                             UnaryExpr (UnarySignExtend inputBits resultBits)
xe1 ≥
                           UnaryExpr (UnarySignExtend inputBits resultBits) xe2
        by (metis SignExtendNode.prems l mono-unary rep.SignExtendNode)
      then show ?thesis
        by meson
    qed
  next
    case (ZeroExtendNode n inputBits resultBits x xe1)
    have k: g1 ⊢ n ≃ UnaryExpr (UnaryZeroExtend inputBits resultBits) xe1
    using ZeroExtendNode by (simp add: ZeroExtendNode.hyps(2) rep.ZeroExtendNode)
    obtain xn where l: kind g1 n = ZeroExtendNode inputBits resultBits xn
      using ZeroExtendNode.hyps(1) by simp
    then have m: g1 ⊢ xn ≃ xe1
      using ZeroExtendNode.hyps(1,2) by simp
    then show ?case
    proof (cases xn = n')
      case True
      then have n: xe1 = e1'
```

using *m* **by** (*simp add*: *repDet c*)
then have *ev*: *g2* ⊢ *n* ≃ *UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *e2′*
using *l* **by** (*simp add*: *ZeroExtendNode.prems True d rep.ZeroExtendNode*)
then have *r*: *UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *e1′* ≥
UnaryExpr (*UnaryZeroExtend inputBits resultBits*) *e2′*
by (*meson a mono-unary*)
then show *?thesis*
by (*metis n ev*)
next
case *False*
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *m*)
have ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *ZeroExtendNode b encodes-contains l not-excluded-keep-type not-in-g singleton-iff*
False
by (*metis-node-eq-ternary ZeroExtendNode*)
then have ∃ *xe2*. (*g2* ⊢ *n* ≃ *UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *xe2*) ∧
UnaryExpr (*UnaryZeroExtend inputBits resultBits*) *xe1* ≥
UnaryExpr (*UnaryZeroExtend inputBits resultBits*) *xe2*
by (*metis ZeroExtendNode.prems l mono-unary rep.ZeroExtendNode*)
then show *?thesis*
by *meson*
qed
next
case (*LeafNode n s*)
then show *?case*
by (*metis eq-refl rep.LeafNode*)
next
case (*PiNode n′ gu*)
then show *?case*
by (*metis encodes-contains not-excluded-keep-type not-in-g rep.PiNode repDet singleton-iff*
a b c d)
next
case (*RefNode n′*)
then show *?case*
by (*metis a b c d no-encoding not-excluded-keep-type rep.RefNode repDet singletonD*)
next
case (*IsNullNode n*)
then show *?case*
by (*metis insertE mono-unary no-encoding not-excluded-keep-type rep.IsNullNode repDet emptyE*
a b c d)
qed

qed
qed

lemma *graph-semantics-preservation-subscript*:
assumes *a*: ${e_1}' \geq {e_2}'$
assumes *b*: $(\{n\} \trianglelefteq \textit{as-set } g_1) \subseteq \textit{as-set } g_2$
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 *assms* **by** (*simp add*: *graph-semantics-preservation*)

lemma *tree-to-graph-rewriting*:
$e_1 \geq e_2$
$\land (g_1 \vdash n \simeq e_1) \land$ *maximal-sharing* g_1
$\land (\{n\} \trianglelefteq \textit{as-set } g_1) \subseteq \textit{as-set } g_2$
$\land (g_2 \vdash n \simeq e_2) \land$ *maximal-sharing* g_2
$\Longrightarrow$ *graph-refinement* g_1 g_2
by (*auto simp add*: *graph-semantics-preservation*)

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** *auto*

lemma *subset-kind*[*simp*]: *as-set g1* ⊆ *as-set g2* ⟹ *g1* ⊢ *n* ≃ *e* ⟹ *kind g1 n* = *kind g2 n*
using *eval-contains-id 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 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: ReverseBytesNode)
                    apply (solve-subset-eval as-set: assms(1) eval: BitCountNode)
                     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: DivNode)
                  apply (solve-subset-eval as-set: assms(1) eval: ModNode)
                 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: IntegerTestNode)
    apply (solve-subset-eval as-set: assms(1) eval: IntegerNormalizeCompareNode)
      apply (solve-subset-eval as-set: assms(1) eval: IntegerMulHighNode)
      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)
      apply (solve-subset-eval as-set: assms(1) eval: PiNode)
  apply (solve-subset-eval as-set: assms(1) eval: RefNode)
  by (solve-subset-eval as-set: assms(1) eval: IsNullNode)

lemma subset-refines:
  assumes as-set g1 ⊆ as-set g2
  shows graph-refinement g1 g2
proof –
  have ids g1 ⊆ ids g2
    using assms 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*
by (*meson equal-refines subset-implies-evals assms 1 2*)
qed
qed

lemma *graph-construction*:
$e_1 \geq e_2$
$\wedge$ *as-set* $g_1 \subseteq$ *as-set* g_2
$\wedge$ ($g_2 \vdash n \simeq e_2$)
$\Longrightarrow$ ($g_2 \vdash n \trianglelefteq e_1$) $\wedge$ *graph-refinement* g_1 g_2
by (*meson encodeeval.simps graph-represents-expression-def le-expr-def subset-refines*)

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*
by (*metis* (*mono-tags*, *lifting*) *find-Some-iff find-node-and-stamp.simps assms*)

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

lemma *find-new-kind*:
assumes g' = *add-node nid* (*node*, *s*) *g*
assumes *node* $\neq$ *NoNode*
shows *kind* g' *nid* = *node*
by (*simp add*: *add-node-lookup assms*)

lemma *find-new-stamp*:
assumes g' = *add-node nid* (*node*, *s*) *g*
assumes *node* $\neq$ *NoNode*
shows *stamp* g' *nid* = *s*
by (*simp add*: *assms add-node-lookup*)

lemma *sorted-bottom*:
assumes *finite xs*
assumes $x \in xs$
shows $x \leq$ *last*(*sorted-list-of-set*(*xs*::*nat set*))
proof −
obtain *largest* **where** *largest*: *largest* = *last* (*sorted-list-of-set*(*xs*))
by *simp*
obtain *sortedList* **where** *sortedList*: *sortedList* = *sorted-list-of-set*(*xs*)
by *simp*
have *step*: $\forall$ *i*. *0* < *i* $\wedge$ *i* < (*length* (*sortedList*)) $\longrightarrow$ *sortedList*!(*i*−*1*) $\leq$ *sortedList*!(*i*)

unfolding *sortedList* **apply** *auto*
by (*metis diff-le-self sorted-list-of-set.length-sorted-key-list-of-set sorted-nth-mono sorted-list-of-set*(*2*))
have *finalElement*: *last* (*sorted-list-of-set*(*xs*)) = *sorted-list-of-set*(*xs*)!(*length* (*sorted-list-of-set*(*xs*)) − *1*)
using *assms last-conv-nth sorted-list-of-set.sorted-key-list-of-set-eq-Nil-iff* **by** *blast*
have *contains0*: (*x* ∈ *xs*) = (*x* ∈ *set* (*sorted-list-of-set*(*xs*)))
using *assms*(*1*) **by** *auto*
have *lastLargest*: ((*x* ∈ *xs*) ⟶ (*largest* ≥ *x*))
using *step* **unfolding** *largest finalElement* **apply** *auto*
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-pred assms*(*1*) *card-Diff1-less in-set-conv-nth sorted-list-of-set.length-sorted-key-list-of-set card-Diff-singleton-if less-Suc-eq-le sorted-list-of-set.sorted-sorted-key-list-of-set length-pos-if-in-set sorted-nth-mono contains0*)
then show *?thesis*
by (*simp add*: *assms largest*)
qed

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

lemma *fresh-ids*:
assumes *n* = *get-fresh-id g*
shows *n* ∉ *ids g*
proof −
have *finite* (*ids g*)
by (*simp add*: *Rep-IRGraph*)
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.ReverseBytesNode*)
apply (*metis no-encoding rep.BitCountNode*)
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.DivNode*)
apply (*metis no-encoding rep.ModNode*)
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.IntegerTestNode*)
apply (*metis no-encoding rep.IntegerNormalizeCompareNode*)
apply (*metis no-encoding rep.IntegerMulHighNode*)
apply (*metis no-encoding rep.NarrowNode*)
apply (*metis no-encoding rep.SignExtendNode*)
apply (*metis no-encoding rep.ZeroExtendNode*)
apply (*metis no-encoding rep.LeafNode*)
apply (*metis no-encoding rep.PiNode*)
apply (*metis no-encoding rep.RefNode*)
by (*metis no-encoding rep.IsNullNode*)
done

lemma *fresh-node-subset*:
assumes $n \notin ids\ g$
assumes $g' = add\text{-}node\ n\ (k, s)\ g$
shows $as\text{-}set\ g \subseteq as\text{-}set\ g'$
by (*smt* (*z3*) *Collect-mono-iff Diff-idemp Diff-insert-absorb add-changed as-set-def unchanged.simps*
disjoint-change assms)

lemma *unique-subset*:
assumes *unique g node* (g', n)
shows $as\text{-}set\ g \subseteq as\text{-}set\ g'$
using *assms fresh-ids fresh-node-subset*
by (*metis Pair-inject old.prod.exhaust subsetI unique.cases*)

lemma *unrep-subset*:
assumes $(g \oplus e \rightsquigarrow (g', n))$
shows $as\text{-}set\ g \subseteq as\text{-}set\ g'$
using *assms*
proof (*induction g e* (g', n) *arbitrary*: $g'\ n$)
case (*UnrepConstantNode g c n* g')
then show *?case* **using** *unique-subset* **by** *simp*
next

case (*UnrepParameterNode g i s n*)
then show *?case* **using** *unique-subset* **by** *simp*
next
case (*UnrepConditionalNode g ce g2 c te g3 t fe g4 f s′ n*)
then show *?case* **using** *unique-subset* **by** *blast*
next
case (*UnrepUnaryNode g xe g2 x s′ op n*)
then show *?case* **using** *unique-subset* **by** *blast*
next
case (*UnrepBinaryNode g xe g2 x ye g3 y s′ op n*)
then show *?case* **using** *unique-subset* **by** *blast*
next
case (*AllLeafNodes g n s*)
then show *?case*
by *auto*
qed

lemma *fresh-node-preserves-other-nodes*:
assumes $n' = \textit{get-fresh-id } g$
assumes $g' = \textit{add-node } n' \ (k, s) \ g$
shows $\forall \ n \in \textit{ids } g \ . \ (g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$
using *assms* **apply** *auto*
by (*metis fresh-node-subset subset-implies-evals fresh-ids assms*)

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

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

lemma *unrep-unchanged*:
assumes $g \oplus e \leadsto (g', n)$
shows $\forall \ n \in \textit{ids } g \ . \ \forall \ e. \ (g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$
by (*meson subset-implies-evals unrep-subset assms*)

lemma *unique-kind*:
assumes *unique g* (*node*, *s*) (*g′*, *nid*)
assumes $\textit{node} \neq \textit{NoNode}$
shows $\textit{kind } g' \ \textit{nid} = \textit{node} \land \textit{stamp } g' \ \textit{nid} = s$
using *assms find-exists-kind add-node-lookup*
by (*smt* (*verit*, *del-insts*) *Pair-inject find-exists-stamp unique.cases*)

lemma *unique-eval*:
assumes *unique g* (*n*, *s*) (*g′*, *nid*)
shows $g \vdash \textit{nid}' \simeq e \Longrightarrow g' \vdash \textit{nid}' \simeq e$

using *assms subset-implies-evals unique-subset* **by** *blast*

lemma *unrep-eval*:
assumes *unrep g e* (g', nid)
shows $g \vdash nid' \simeq e' \Longrightarrow g' \vdash nid' \simeq e'$
using *assms subset-implies-evals no-encoding unrep-unchanged* **by** *blast*

lemma *unary-node-nonode*:
unary-node op x $\neq$ *NoNode*
by (*cases op*; *auto*)

lemma *bin-node-nonode*:
bin-node op x y $\neq$ *NoNode*
by (*cases op*; *auto*)

theorem *term-graph-reconstruction*:
$g \oplus e \rightsquigarrow (g', n) \Longrightarrow (g' \vdash n \simeq e) \wedge$ *as-set g* $\subseteq$ *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 (*UnrepConstantNode g c* g_1 *n*)
then show *?case*
using *ConstantNode unique-kind* **by** *blast*
next
case (*UnrepParameterNode g i s* g_1 *n*)
then show *?case*
using *ParameterNode unique-kind*
by (*metis IRNode.distinct*(*3695*))
next
case (*UnrepConditionalNode g ce* g_1 *c te* g_2 *t fe* g_3 *f* s' g_4 *n*)
then show *?case*
using *unique-kind unique-eval unrep-eval*
by (*meson ConditionalNode IRNode.distinct*(*965*))
next
case (*UnrepUnaryNode g xe* g_1 *x* s' *op* g_2 *n*)
then have *k*: *kind* g_2 *n* = *unary-node op x*
using *unique-kind unary-node-nonode* **by** *simp*
then have $g_2 \vdash x \simeq xe$
using *UnrepUnaryNode unique-eval* **by** *blast*
then show *?case*
using *k* **apply** (*cases op*)
using *unary-node.simps*(*1*,*2*,*3*,*4*,*5*,*6*,*7*,*8*,*9*,*10*)
AbsNode NegateNode NotNode LogicNegationNode NarrowNode SignExtendNode ZeroExtendNode
IsNullNode ReverseBytesNode BitCountNode
by *presburger+*
next
case (*UnrepBinaryNode g xe* g_1 *x ye* g_2 *y* s' *op* g_3 *n*)

then have *k*: *kind* g_3 *n* = *bin-node op x y*
using *unique-kind bin-node-nonode* **by** *simp*
have *x*: $g_3 \vdash x \simeq xe$
using *UnrepBinaryNode unique-eval unrep-eval* **by** *blast*
have *y*: $g_3 \vdash y \simeq ye$
using *UnrepBinaryNode unique-eval unrep-eval* **by** *blast*
then show *?case*
using *x k* **apply** (*cases op*)
using *bin-node.simps*(*1*,*2*,*3*,*4*,*5*,*6*,*7*,*8*,*9*,*10*,*11*,*12*,*13*,*14*,*15*,*16*,*17*,*18*)
AddNode MulNode DivNode ModNode SubNode AndNode OrNode ShortCircuitOrNode LeftShiftNode RightShiftNode
UnsignedRightShiftNode IntegerEqualsNode IntegerLessThanNode IntegerBelowNode XorNode
IntegerTestNode IntegerNormalizeCompareNode IntegerMulHighNode
by *metis+*
next
case (*AllLeafNodes g n s*)
then show *?case*
by (*simp add*: *rep.LeafNode*)
qed
done

lemma *ref-refinement*:
assumes $g \vdash n \simeq e_1$
assumes *kind g n′* = *RefNode n*
shows $g \vdash n' \trianglelefteq e_1$
by (*meson equal-refines graph-represents-expression-def RefNode assms*)

lemma *unrep-refines*:
assumes $g \oplus e \rightsquigarrow (g', n)$
shows *graph-refinement g g′*
using *assms* **by** (*simp add*: *unrep-subset subset-refines*)

lemma *add-new-node-refines*:
assumes $n \notin$ *ids g*
assumes *g′* = *add-node n* (*k*, *s*) *g*
shows *graph-refinement g g′*
using *assms* **by** (*simp add*: *fresh-node-subset subset-refines*)

lemma *add-node-as-set*:
assumes *g′* = *add-node n* (*k*, *s*) *g*
shows ($\{n\} \trianglelefteq$ *as-set g*) $\subseteq$ *as-set g′*
unfolding *assms*
by (*smt* (*verit*, *ccfv-SIG*) *case-prodE changeonly.simps mem-Collect-eq prod.sel*(*1*) *subsetI assms*
add-changed as-set-def domain-subtraction-def)

theorem *refined-insert*:
assumes $e_1 \geq e_2$

assumes $g_1 \oplus e_2 \leadsto (g_2, n')$
shows $(g_2 \vdash n' \trianglelefteq e_1) \wedge$ *graph-refinement* g_1 g_2
using *assms graph-construction term-graph-reconstruction* **by** *blast*

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

lemma *unwrap-sorted*: *set* (*sorted-list-of-set* (*ids g*)) = *ids g*
using *ids-finite* **by** *simp*

lemma *find-none*:
assumes *find-node-and-stamp g* (*k*, *s*) = *None*
shows $\forall$ $n \in$ *ids g*. *kind g n* $\neq$ *k* $\vee$ *stamp g n* $\neq$ *s*
proof −
have ($\nexists n$. $n \in$ *ids g* $\wedge$ (*kind g n* = *k* $\wedge$ *stamp g n* = *s*))
by (*metis* (*mono-tags*) *unwrap-sorted find-None-iff find-node-and-stamp.simps assms*)
then show *?thesis*
by *auto*
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 ¬(*is-preevaluated* (*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 ReverseBytesNode*)
apply (*metis BitCountNode*)
apply (*metis NotNode*)
apply (*metis NegateNode*)
apply (*metis LogicNegationNode*)
apply (*metis AddNode*)
apply (*metis MulNode*)
apply (*metis DivNode*)
apply (*metis ModNode*)
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 IntegerTestNode*)
apply (*metis IntegerNormalizeCompareNode*)
apply (*metis IntegerMulHighNode*)
apply (*metis NarrowNode*)
apply (*metis SignExtendNode*)
apply (*metis ZeroExtendNode*)
defer
apply (*metis PiNode*)
apply (*metis RefNode*)
apply (*metis IsNullNode*)
by *blast*
done
done

lemma *new-node-not-present*:
assumes *find-node-and-stamp g* (*node*, *s*) = *None*
assumes n = *get-fresh-id g*
assumes g' = *add-node n* (*node*, *s*) *g*
shows $\forall\ n' \in$ *true-ids g*. ($\forall$ *e*. (($g \vdash n \simeq e$) $\wedge$ ($g \vdash n' \simeq e$)) $\longrightarrow n = n'$)
using *assms encode-in-ids fresh-ids* **by** *blast*

lemma *true-ids-def*:
true-ids g = {$n \in$ *ids g*. $\neg$(*is-RefNode* (*kind g n*)) $\wedge$ ((*kind g n*) $\neq$ *NoNode*)}
using *true-ids-def* **by** (*auto simp add*: *is-RefNode-def*)

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

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

lemma *ids-add-update-v1*:
assumes $g' =$ *add-node nid* (k, s) g
assumes $k \neq$ *NoNode*
shows *dom* (*Rep-IRGraph* g') $=$ *dom* (*Rep-IRGraph* g) $\cup$ {*nid*}
by (*simp add*: *add-node.rep-eq assms*)

lemma *ids-add-update-v2*:
assumes $g' =$ *add-node nid* (k, s) g
assumes $k \neq$ *NoNode*
shows *nid* $\in$ *ids* g'
by (*simp add*: *find-new-kind assms*)

lemma *add-node-ids-subset*:
assumes $n \in$ *ids* g
assumes $g' =$ *add-node* n *node* g
shows *ids* $g' =$ *ids* $g \cup \{n\}$
using *assms replace-node.rep-eq* **by** (*auto simp add*: *replace-node-def ids.rep-eq add-node-def*)

lemma *convert-maximal*:
assumes $\forall n\ n'.\ n \in$ *true-ids* $g \wedge n' \in$ *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* **by** (*auto simp add*: *maximal-sharing*)

lemma *add-node-set-eq*:
assumes $k \neq$ *NoNode*
assumes $n \notin$ *ids* g
shows *as-set* (*add-node* n (k, s) g) $=$ *as-set* $g \cup \{(n, (k, s))\}$
using *assms* **unfolding** *as-set-def* **by** (*transfer*; *auto*)

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

lemma *true-ids*:
true-ids $g =$ *ids* $g - \{n \in$ *ids* $g.$ *is-RefNode* (*kind* g n)$\}$

unfolding *true-ids-def* **by** *fastforce*

lemma *as-set-ids*:
assumes *as-set* g = *as-set* g'
shows *ids* g = *ids* g'
by (*metis antisym equalityD1 graph-refinement-def subset-refines assms*)

lemma *ids-add-update*:
assumes $k \neq$ *NoNode*
assumes $n \notin$ *ids* g
assumes g' = *add-node* n (k, s) g
shows *ids* $g' =$ *ids* $g \cup \{n\}$
by (*smt* (*z3*) *Diff-idemp Diff-insert-absorb Un-commute add-node.rep-eq insert-is-Un insert-Collect*
add-node-def ids.rep-eq ids-add-update-v1 insertE assms replace-node-unchanged Collect-cong
map-upd-Some-unfold mem-Collect-eq replace-node-def ids-add-update-v2)

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

lemma *new-def*:
assumes (*new* $\trianglelefteq$ *as-set* g') = *as-set* g
shows $n \in$ *ids* $g \longrightarrow n \notin$ *new*
using *assms* **apply** *auto* **unfolding** *as-set-def*
by (*smt* (*z3*) *as-set-def case-prodD domain-subtraction-def mem-Collect-eq assms ids-some*)

lemma *add-preserves-rep*:
assumes *unchanged*: (*new* $\trianglelefteq$ *as-set* g') = *as-set* g
assumes *closed*: *wf-closed* g
assumes *existed*: $n \in$ *ids* g
assumes $g' \vdash n \simeq e$
shows $g \vdash n \simeq e$
proof (*cases* $n \in$ *new*)
case *True*
have $n \notin$ *ids* g
using *unchanged True as-set-def* **unfolding** *domain-subtraction-def* **by** *blast*
then show *?thesis*
using *existed* **by** *simp*

next
  case *False*
  have *kind-eq*: ∀ *n′* . *n′* ∉ *new* ⟶ *kind g n′* = *kind g′ n′*
    — can be more general than *stamp_eq* because NoNode default is equal
    apply (*rule allI*; *rule impI*)
    by (*smt* (*z3*) *case-prodE domain-subtraction-def ids-some mem-Collect-eq subsetI unchanged*
        not-excluded-keep-type)
  from *False* **have** *stamp-eq*: ∀ *n′* ∈ *ids g′* . *n′* ∉ *new* ⟶ *stamp g n′* = *stamp g′ n′*
    by (*metis equalityE not-excluded-keep-type unchanged*)
  show *?thesis*
    using *assms*(*4*) *kind-eq stamp-eq False*
  proof (*induction n e rule*: *rep.induct*)
    case (*ConstantNode n c*)
    then show *?case*
      by (*simp add*: *rep.ConstantNode*)
  next
    case (*ParameterNode n i s*)
    then show *?case*
      by (*metis no-encoding rep.ParameterNode*)
  next
    case (*ConditionalNode n c t f ce te fe*)
    have *kind*: *kind g n* = *ConditionalNode c t f*
      by (*simp add*: *kind-eq ConditionalNode.prems*(*3*) *ConditionalNode.hyps*(*1*))
    then have *isin*: *n* ∈ *ids g*
      by *simp*
    have *inputs*: {*c*, *t*, *f*} = *inputs g n*
      by (*simp add*: *kind*)
    have *c* ∈ *ids g* ∧ *t* ∈ *ids g* ∧ *f* ∈ *ids g*
      using *closed wf-closed-def isin inputs* **by** *blast*
    then have *c* ∉ *new* ∧ *t* ∉ *new* ∧ *f* ∉ *new*
      using *unchanged* **by** (*simp add*: *new-def*)
    then show *?case*
      by (*simp add*: *rep.ConditionalNode ConditionalNode*)
  next
    case (*AbsNode n x xe*)
    then have *kind*: *kind g n* = *AbsNode x*
      by *simp*
    then have *isin*: *n* ∈ *ids g*
      by *simp*
    have *inputs*: {*x*} = *inputs g n*
      by (*simp add*: *kind*)
    have *x* ∈ *ids g*
      using *closed wf-closed-def isin inputs* **by** *blast*
    then have *x* ∉ *new*
      using *unchanged* **by** (*simp add*: *new-def*)
    then show *?case*
      by (*simp add*: *AbsNode rep.AbsNode*)

next
case (*ReverseBytesNode n x xe*)
then have *kind*: *kind g n* = *ReverseBytesNode x*
by *simp*
then have *isin*: *n* ∈ *ids g*
by *simp*
have *inputs*: {*x*} = *inputs g n*
by (*simp add*: *kind*)
have *x* ∈ *ids g*
using *closed wf-closed-def isin inputs* **by** *blast*
then have *x* ∉ *new*
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
using *ReverseBytesNode.IH kind kind-eq rep.ReverseBytesNode stamp-eq* **by** *blast*
next
case (*BitCountNode n x xe*)
then have *kind*: *kind g n* = *BitCountNode x*
by *simp*
then have *isin*: *n* ∈ *ids g*
by *simp*
have *inputs*: {*x*} = *inputs g n*
by (*simp add*: *kind*)
have *x* ∈ *ids g*
using *closed wf-closed-def isin inputs* **by** *blast*
then have *x* ∉ *new*
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
using *BitCountNode.IH kind kind-eq rep.BitCountNode stamp-eq* **by** *blast*
next
case (*NotNode n x xe*)
then have *kind*: *kind g n* = *NotNode x*
by *simp*
then have *isin*: *n* ∈ *ids g*
by *simp*
have *inputs*: {*x*} = *inputs g n*
by (*simp add*: *kind*)
have *x* ∈ *ids g*
using *closed wf-closed-def isin inputs* **by** *blast*
then have *x* ∉ *new*
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *NotNode rep.NotNode*)
next
case (*NegateNode n x xe*)
then have *kind*: *kind g n* = *NegateNode x*
by *simp*
then have *isin*: *n* ∈ *ids g*
by *simp*

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

by (*simp add*: *MulNode rep.MulNode*)
next
case (*DivNode n x y xe ye*)
then have *kind*: *kind g n* = *SignedFloatingIntegerDivNode x y*
by *simp*
then have *isin*: $n \in$ *ids g*
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
by (*simp add*: *kind*)
have $x \in$ *ids g* $\wedge$ $y \in$ *ids g*
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin$ *new* $\wedge$ $y \notin$ *new*
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *DivNode rep.DivNode*)
next
case (*ModNode n x y xe ye*)
then have *kind*: *kind g n* = *SignedFloatingIntegerRemNode x y*
by *simp*
then have *isin*: $n \in$ *ids g*
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
by (*simp add*: *kind*)
have $x \in$ *ids g* $\wedge$ $y \in$ *ids g*
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin$ *new* $\wedge$ $y \notin$ *new*
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *ModNode rep.ModNode*)
next
case (*SubNode n x y xe ye*)
then have *kind*: *kind g n* = *SubNode x y*
by *simp*
then have *isin*: $n \in$ *ids g*
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
by (*simp add*: *kind*)
have $x \in$ *ids g* $\wedge$ $y \in$ *ids g*
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin$ *new* $\wedge$ $y \notin$ *new*
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *SubNode rep.SubNode*)
next
case (*AndNode n x y xe ye*)
then have *kind*: *kind g n* = *AndNode x y*
by *simp*
then have *isin*: $n \in$ *ids g*
by *simp*

have *inputs*: $\{x, y\}$ = *inputs g n*
  by (*simp add*: *kind*)
have $x \in$ *ids g* $\wedge$ $y \in$ *ids g*
  using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin$ *new* $\wedge$ $y \notin$ *new*
  using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
  by (*simp add*: *AndNode rep.AndNode*)
next
  case (*OrNode n x y xe ye*)
  then have *kind*: *kind g n* = *OrNode x y*
    by *simp*
  then have *isin*: $n \in$ *ids g*
    by *simp*
  have *inputs*: $\{x, y\}$ = *inputs g n*
    by (*simp add*: *kind*)
  have $x \in$ *ids g* $\wedge$ $y \in$ *ids g*
    using *closed wf-closed-def isin inputs* **by** *blast*
  then have $x \notin$ *new* $\wedge$ $y \notin$ *new*
    using *unchanged* **by** (*simp add*: *new-def*)
  then show *?case*
    by (*simp add*: *OrNode rep.OrNode*)
next
  case (*XorNode n x y xe ye*)
  then have *kind*: *kind g n* = *XorNode x y*
    by *simp*
  then have *isin*: $n \in$ *ids g*
    by *simp*
  have *inputs*: $\{x, y\}$ = *inputs g n*
    by (*simp add*: *kind*)
  have $x \in$ *ids g* $\wedge$ $y \in$ *ids g*
    using *closed wf-closed-def isin inputs* **by** *blast*
  then have $x \notin$ *new* $\wedge$ $y \notin$ *new*
    using *unchanged* **by** (*simp add*: *new-def*)
  then show *?case*
    by (*simp add*: *XorNode rep.XorNode*)
next
  case (*ShortCircuitOrNode n x y xe ye*)
  then have *kind*: *kind g n* = *ShortCircuitOrNode x y*
    by *simp*
  then have *isin*: $n \in$ *ids g*
    by *simp*
  have *inputs*: $\{x, y\}$ = *inputs g n*
    by (*simp add*: *kind*)
  have $x \in$ *ids g* $\wedge$ $y \in$ *ids g*
    using *closed wf-closed-def isin inputs* **by** *blast*
  then have $x \notin$ *new* $\wedge$ $y \notin$ *new*
    using *unchanged* **by** (*simp add*: *new-def*)
  then show *?case*

by (*simp add*: *ShortCircuitOrNode rep.ShortCircuitOrNode*)
next
case (*LeftShiftNode n x y xe ye*)
then have *kind*: *kind g n* = *LeftShiftNode x y*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\} = inputs\ g\ n$
by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *LeftShiftNode rep.LeftShiftNode*)
next
case (*RightShiftNode n x y xe ye*)
then have *kind*: *kind g n* = *RightShiftNode x y*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\} = inputs\ g\ n$
by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *RightShiftNode rep.RightShiftNode*)
next
case (*UnsignedRightShiftNode n x y xe ye*)
then have *kind*: *kind g n* = *UnsignedRightShiftNode x y*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\} = inputs\ g\ n$
by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *UnsignedRightShiftNode rep.UnsignedRightShiftNode*)
next
case (*IntegerBelowNode n x y xe ye*)
then have *kind*: *kind g n* = *IntegerBelowNode x y*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*

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

by (*simp add*: *IntegerTestNode rep.IntegerTestNode*)
next
case (*IntegerNormalizeCompareNode n x y xe ye*)
then have *kind*: *kind g n* = *IntegerNormalizeCompareNode x y*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
using *IntegerNormalizeCompareNode.IH*(*1*,*2*) *kind kind-eq rep.IntegerNormalizeCompareNode stamp-eq* **by** *blast*
next
case (*IntegerMulHighNode n x y xe ye*)
then have *kind*: *kind g n* = *IntegerMulHighNode x y*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
using *IntegerMulHighNode.IH*(*1*,*2*) *kind kind-eq rep.IntegerMulHighNode stamp-eq* **by** *blast*
next
case (*NarrowNode n inputBits resultBits x xe*)
then have *kind*: *kind g n* = *NarrowNode inputBits resultBits x*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x\}$ = *inputs g n*
by (*simp add*: *kind*)
have $x \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *NarrowNode rep.NarrowNode*)
next
case (*SignExtendNode n inputBits resultBits x xe*)
then have *kind*: *kind g n* = *SignExtendNode inputBits resultBits x*
by *simp*

```
    then have isin: n ∈ ids g
      by simp
    have inputs: {x} = inputs g n
      by (simp add: kind)
    have x ∈ ids g
      using closed wf-closed-def isin inputs by blast
    then have x ∉ new
      using unchanged by (simp add: new-def)
    then show ?case
      by (simp add: SignExtendNode rep.SignExtendNode)
  next
    case (ZeroExtendNode n inputBits resultBits x xe)
    then have kind: kind g n = ZeroExtendNode inputBits resultBits x
      by simp
    then have isin: n ∈ ids g
      by simp
    have inputs: {x} = inputs g n
      by (simp add: kind)
    have x ∈ ids g
      using closed wf-closed-def isin inputs by blast
    then have x ∉ new
      using unchanged by (simp add: new-def)
    then show ?case
      by (simp add: ZeroExtendNode rep.ZeroExtendNode)
  next
    case (LeafNode n s)
    then show ?case
      by (metis no-encoding rep.LeafNode)
  next
    case (PiNode n n' gu e)
    then have kind: kind g n = PiNode n' gu
      by simp
    then have isin: n ∈ ids g
      by simp
    have inputs: set (n' # (opt-to-list gu)) = inputs g n
      by (simp add: kind)
    have n' ∈ ids g
      by (metis in-mono list.set-intros(1) inputs isin wf-closed-def closed)
    then show ?case
       using PiNode.IH kind kind-eq new-def rep.PiNode stamp-eq unchanged by
blast
  next
    case (RefNode n n' e)
    then have kind: kind g n = RefNode n'
      by simp
    then have isin: n ∈ ids g
      by simp
    have inputs: {n'} = inputs g n
      by (simp add: kind)
```

have $n' \in$ *ids g*
using *closed wf-closed-def isin inputs* **by** *blast*
then have $n' \notin$ *new*
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *RefNode rep.RefNode*)
next
case (*IsNullNode n v*)
then have *kind*: *kind g n* = *IsNullNode v*
by *simp*
then have *isin*: $n \in$ *ids g*
by *simp*
have *inputs*: $\{v\}$ = *inputs g n*
by (*simp add*: *kind*)
have $v \in$ *ids g*
using *closed wf-closed-def isin inputs* **by** *blast*
then have $v \notin$ *new*
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *rep.IsNullNode stamp-eq kind-eq kind IsNullNode.IH*)
qed
qed

lemma *not-in-no-rep*:
$n \notin$ *ids g* $\Longrightarrow \forall e. \neg(g \vdash n \simeq e)$
using *eval-contains-id* **by** *auto*

lemma *unary-inputs*:
assumes *kind g n* = *unary-node op x*
shows *inputs g n* = $\{x\}$
by (*cases op*; *auto simp add*: *assms*)

lemma *unary-succ*:
assumes *kind g n* = *unary-node op x*
shows *succ g n* = $\{\}$
by (*cases op*; *auto simp add*: *assms*)

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

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

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

lemma *unrep-preserves-contains*:
  assumes $n \in ids\ g$
  assumes $g \oplus e \leadsto (g', n')$
  shows $n \in ids\ g'$
  by (*meson subsetD unrep-ids-subset assms*)

lemma *unique-preserves-closure*:
  assumes *wf-closed g*
  assumes *unique g* (*node*, *s*) (g', *n*)
  assumes *set* (*inputs-of node*) $\subseteq$ *ids g* $\wedge$
    set (*successors-of node*) $\subseteq$ *ids g* $\wedge$
    node $\neq$ *NoNode*
  shows *wf-closed* g'
  using *assms*
  by (*smt* (*verit*, *del-insts*) *Pair-inject UnE add-changed fresh-ids graph-refinement-def ids-add-update inputs.simps other-node-unchanged singletonD subset-refines subset-trans succ.simps unique.cases unique-kind unique-subset wf-closed-def*)

lemma *unrep-preserves-closure*:
  assumes *wf-closed g*
  assumes $g \oplus e \leadsto (g', n)$
  shows *wf-closed* g'
  using *assms*(*2*,*1*) *wf-closed-def*
  proof (*induction g e* (g', *n*) *arbitrary*: g' *n*)
  next
    case (*UnrepConstantNode g c* g' *n*)
    then show *?case* **using** *unique-preserves-closure*
    by (*metis IRNode.distinct*(*1077*) *IRNodes.inputs-of-ConstantNode IRNodes.successors-of-ConstantNode empty-subsetI list.set*(*1*))
  next
    case (*UnrepParameterNode g i s n*)
    then show *?case* **using** *unique-preserves-closure*
      by (*metis IRNode.distinct*(*3695*) *IRNodes.inputs-of-ParameterNode IRNodes.successors-of-ParameterNode empty-subsetI list.set*(*1*))
  next
    case (*UnrepConditionalNode g ce* g_1 *c te* g_2 *t fe* g_3 *f* s' g_4 *n*)
    then have *c*: *wf-closed* g_3
      by *fastforce*
    have *k*: *kind* g_4 *n* = *ConditionalNode c t f*
     using *UnrepConditionalNode IRNode.distinct*(*965*) *unique-kind* **by** *presburger*
    have $\{c, t, f\} \subseteq ids\ g_4$ **using** *unrep-contains*
     by (*metis UnrepConditionalNode.hyps*(*1*) *UnrepConditionalNode.hyps*(*3*) *UnrepConditionalNode.hyps*(*5*) *UnrepConditionalNode.hyps*(*8*) *empty-subsetI graph-refinement-def*

insert-subsetI subset-iff subset-refines unique-subset unrep-ids-subset)
also have *inputs* g_4 *n* = {*c*, *t*, *f*} ∧ *succ* g_4 *n* = {}
using *k* **by** *simp*
moreover have *inputs* g_4 *n* ⊆ *ids* g_4 ∧ *succ* g_4 *n* ⊆ *ids* g_4 ∧ *kind* g_4 *n* ≠ *NoNode*
using *k*
by (*metis IRNode.distinct*(*965*) *calculation empty-subsetI*)
ultimately show *?case* **using** *c unique-preserves-closure UnrepConditionalNode*
by (*metis empty-subsetI inputs.simps insert-subsetI k succ.simps unrep-contains unrep-preserves-contains*)
next
case (*UnrepUnaryNode g xe* g_1 *x s′ op* g_2 *n*)
then have *c*: *wf-closed* g_1
by *fastforce*
have *k*: *kind* g_2 *n* = *unary-node op x*
using *UnrepUnaryNode unique-kind unary-node-nonode* **by** *blast*
have {*x*} ⊆ *ids* g_2 **using** *unrep-contains*
by (*metis UnrepUnaryNode.hyps*(*1*) *UnrepUnaryNode.hyps*(*4*) *encodes-contains ids-some singletonD subsetI term-graph-reconstruction unique-eval*)
also have *inputs* g_2 *n* = {*x*} ∧ *succ* g_2 *n* = {}
using *k*
by (*meson unary-inputs unary-succ*)
moreover have *inputs* g_2 *n* ⊆ *ids* g_2 ∧ *succ* g_2 *n* ⊆ *ids* g_2 ∧ *kind* g_2 *n* ≠ *NoNode*
using *k*
by (*metis calculation*(*1*) *calculation*(*2*) *empty-subsetI unary-node-nonode*)
ultimately show *?case* **using** *c unique-preserves-closure UnrepUnaryNode*
by (*metis empty-subsetI inputs.simps insert-subsetI k succ.simps unrep-contains*)
next
case (*UnrepBinaryNode g xe* g_1 *x ye* g_2 *y s′ op* g_3 *n*)
then have *c*: *wf-closed* g_2
by *fastforce*
have *k*: *kind* g_3 *n* = *bin-node op x y*
using *UnrepBinaryNode unique-kind bin-node-nonode* **by** *blast*
have {*x*, *y*} ⊆ *ids* g_3 **using** *unrep-contains*
by (*metis UnrepBinaryNode.hyps*(*1*) *UnrepBinaryNode.hyps*(*3*) *UnrepBinaryNode.hyps*(*6*) *empty-subsetI graph-refinement-def insert-absorb insert-subset subset-refines unique-subset unrep-refines*)
also have *inputs* g_3 *n* = {*x*, *y*} ∧ *succ* g_3 *n* = {}
using *k*
by (*meson binary-inputs binary-succ*)
moreover have *inputs* g_3 *n* ⊆ *ids* g_3 ∧ *succ* g_3 *n* ⊆ *ids* g_3 ∧ *kind* g_3 *n* ≠ *NoNode*
using *k*
by (*metis calculation*(*1*) *calculation*(*2*) *empty-subsetI bin-node-nonode*)
ultimately show *?case* **using** *c unique-preserves-closure UnrepBinaryNode*
by (*metis empty-subsetI inputs.simps insert-subsetI k succ.simps unrep-contains unrep-preserves-contains*)

next
  case (*AllLeafNodes g n s*)
  then show *?case*
    by *simp*
qed

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

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'*:
  $(g, p \vdash state \rightarrow next) \Longrightarrow$
  $(g, p \vdash state \rightarrow next') \Longrightarrow next = next'$
proof (*induction arbitrary*: *next'* *rule*: *step.induct*)
  case (*SequentialNode nid nid' m h*)
  have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
   by (*metis SequentialNode.hyps*(*1*) *is-AbstractEndNode.simps is-EndNode.elims*(*2*) *is-LoopEndNode-def is-sequential-node.simps*(*18*) *is-sequential-node.simps*(*36*))
 from *SequentialNode* **show** *?case* **apply** (*elim StepE*) **using** *is-sequential-node.simps*
   apply *blast*
  apply *force* **apply** *force* **apply** *force*

using *notend*
apply (*metis* (*no-types*, *lifting*) *Pair-inject is-AbstractEndNode.simps*)
by *force+*
next
case (*FixedGuardNode nid cond before next m val nid′ h*)
then show *?case* **apply** (*elim StepE*)
by *force+*
next
case (*BytecodeExceptionNode nid args st nid′ exceptionType h′ ref h m′ m*)
then show *?case* **apply** (*elim StepE*)
by *force+*
next
case (*IfNode nid cond tb fb m val nid′ h*)
then show *?case* **apply** (*elim StepE*)
apply *force+*
— IfNode rule uses expression evaluation
using *graphDet* **apply** *fastforce*
by *force+*
next
case (*EndNodes nid merge i phis inps m vs m′ h*)
have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
using *EndNodes*
by (*metis is-AbstractEndNode.simps is-EndNode.elims*(*2*) *is-LoopEndNode-def is-sequential-node.simps*(*18*) *is-sequential-node.simps*(*36*))
from *EndNodes* **show** *?case* **apply** (*elim StepE*)
using *notseq* **apply** *force*
apply *force* **apply** *force* **apply** *force*
using *indexof-det*
unfolding *is-AbstractEndNode.simps*
is-AbstractMergeNode.simps any-usage.simps usages.simps inputs.simps ids-def
apply (*smt* (*verit*, *del-insts*) *Collect-cong encodeEvalAllDet ids-def ids-some old.prod.inject*)
by *force+*
next
case (*NewArrayNode nid len st nid′ m length′ arrayType h′ ref h refNo h″ m′*)
then show *?case* **apply** (*elim StepE*) **apply** *force+*
— NewArrayNode rule uses expression evaluation
using *graphDet* **apply** *fastforce*
by *force+*
next
case (*ArrayLengthNode nid x nid′ m ref h arrayVal length′ m′*)
then show *?case* **apply** (*elim StepE*) **apply** *force+*
— ArrayLengthNode rule uses expression evaluation
using *graphDet* **apply** *fastforce*
by *force+*
next
case (*LoadIndexedNode nid index guard array nid′ m indexVal ref h arrayVal loaded m′*)
then show *?case* **apply** (*elim StepE*) **apply** *force+*

— LoadIndexedNode rule uses expression evaluation
using *graphDet*
apply (*metis IRNode.inject*(*28*) *Pair-inject Value.inject*(*2*))
by *force+*
next
case (*StoreIndexedNode nid check val st index guard array nid′ m indexVal ref value h arrayVal updated h′ m′*)
then show *?case* **apply** (*elim StepE*) **apply** *force+*
— StoreIndexedNode rule uses expression evaluation
using *graphDet*
apply (*metis IRNode.inject*(*55*) *Pair-inject Value.inject*(*2*))
by *force+*
next
case (*NewInstanceNode nid cname obj nid′ h′ ref h m′ m*)
then show *?case* **apply** (*elim StepE*) **by** *force+*
next
case (*LoadFieldNode nid f obj nid′ m ref h v m′*)
then show *?case* **apply** (*elim StepE*) **apply** *force+*
— LoadFieldNode rule uses expression evaluation
using *graphDet* **apply** *fastforce*
by *force+*
next
case (*SignedDivNode nid x y zero sb nxt m v1 v2 v m′ h*)
then show *?case* **apply** (*elim StepE*) **apply** *force+*
— SignedDivNode rule uses expression evaluation
using *graphDet*
apply (*metis IRNode.inject*(*49*) *Pair-inject*)
by *force+*
next
case (*SignedRemNode nid x y zero sb nxt m v1 v2 v m′ h*)
then show *?case* **apply** (*elim StepE*) **apply** *force+*
— SignedRemNode rule uses expression evaluation
using *graphDet*
apply (*metis IRNode.inject*(*52*) *Pair-inject*)
by *force+*
next
case (*StaticLoadFieldNode nid f nid′ h v m′ m*)
then show *?case* **apply** (*elim StepE*) **by** *force+*
next
case (*StoreFieldNode nid f newval uu obj nid′ m val ref h′ h m′*)
then show *?case* **apply** (*elim StepE*) **apply** *force+*
— StoreFieldNode rule uses expression evaluation
using *graphDet*
apply (*metis IRNode.inject*(*54*) *Pair-inject Value.inject*(*2*) *option.inject*)
by *force+*
next
case (*StaticStoreFieldNode nid f newval uv nid′ m val h′ h m′*)
then show *?case* **apply** (*elim StepE*) **apply** *force+*
— StaticStoreFieldNode rule uses expression evaluation

using *graphDet* **by** *fastforce*
qed

theorem *stepDet*:
$(g,\ p \vdash (nid,m,h) \to next) \Longrightarrow$
$(\forall\ next'.\ ((g,\ p \vdash (nid,m,h) \to next') \longrightarrow next = next'))$
using *stepDet′* **by** *simp*

lemma *stepRefNode*:
$[\![kind\ g\ nid = RefNode\ nid']\!] \Longrightarrow g,\ p \vdash (nid,m,h) \to (nid',m,h)$
by (*metis IRNodes.successors-of-RefNode is-sequential-node.simps*(*7*) *nth-Cons-0 SequentialNode*)

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

lemma *IfNodeSeq*:
shows *kind g nid = IfNode cond tb fb* ⟶ ¬(*is-sequential-node* (*kind g nid*))
using *is-sequential-node.simps*(*18*,*19*) **by** *simp*

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

lemma *step-in-ids*:
assumes $g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m',\ h')$
shows $nid \in ids\ g$
using *assms* **apply** (*induct* (*nid*, *m*, *h*) (*nid′*, *m′*, *h′*) *rule*: *step.induct*) **apply** *fastforce*
prefer *4* **prefer** *14* **defer defer**
using *IRNode.distinct*(*1607*) *ids-some* **apply** *presburger*
using *IRNode.distinct*(*851*) *ids-some* **apply** *presburger*

using *IRNode.distinct*(*1805*) *ids-some* **apply** *presburger*
apply (*metis IRNode.distinct*(*3507*) *not-in-g*)
apply (*metis IRNode.distinct*(*497*) *not-in-g*)
apply (*metis IRNode.distinct*(*2897*) *not-in-g*)

apply (*metis IRNode.distinct*(*4085*) *not-in-g*)
using *IRNode.distinct*(*3557*) *ids-some* **apply** *presburger*
apply (*metis IRNode.distinct*(*2825*) *not-in-g*)

apply (*metis IRNode.distinct*(*3947*) *not-in-g*)
apply (*metis IRNode.distinct*(*4025*) *not-in-g*)
using *IRNode.distinct*(*2825*) *ids-some* **apply** *presburger*
apply (*metis IRNode.distinct*(*4067*) *not-in-g*)
apply (*metis IRNode.distinct*(*4067*) *not-in-g*)
using *IRNode.disc*(*1952*) *is-EndNode.simps*(*62*) *is-AbstractEndNode.simps not-in-g*
by (*metis IRNode.disc*(*2014*) *is-EndNode.simps*(*64*))

end