Veriopt Theories

November 24, 2022

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

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

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

datatype IRBinaryOp =
    BinAdd
  | BinMul
  | BinSub
  | BinAnd
  | BinOr
  | BinXor
  | BinShortCircuitOr
  | BinLeftShift
  | BinRightShift
  | BinURightShift
  | BinIntegerEquals
  | BinIntegerLessThan
  | BinIntegerBelow

datatype (discs-sels) IRExpr =
    UnaryExpr (ir-uop: IRUnaryOp) (ir-value: IRExpr)
  | BinaryExpr (ir-op: IRBinaryOp) (ir-x: IRExpr) (ir-y: IRExpr)
  | ConditionalExpr (ir-condition: IRExpr) (ir-trueValue: IRExpr) (ir-falseValue:
IRExpr)

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


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

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

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

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

1.2 Functions for re-calculating stamps

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

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

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

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

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

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

  stamp-unary op - = IllegalStamp

fun stamp-binary :: IRBinaryOp ⇒ Stamp ⇒ Stamp ⇒ Stamp where
  stamp-binary op (IntegerStamp b1 lo1 hi1) (IntegerStamp b2 lo2 hi2) =
    (if op ∈ binary-shift-ops then unrestricted-stamp (IntegerStamp b1 lo1 hi1)
     else if b1 ≠ b2 then IllegalStamp else
      (if op ∈ binary-fixed-32-ops
       then unrestricted-stamp (IntegerStamp 32 lo1 hi1)
       else unrestricted-stamp (IntegerStamp b1 lo1 hi1))) |

  stamp-binary op - - = IllegalStamp

fun stamp-expr :: IRExpr ⇒ Stamp where
  stamp-expr (UnaryExpr op x) = stamp-unary op (stamp-expr x) |
  stamp-expr (BinaryExpr bop x y) = stamp-binary bop (stamp-expr x) (stamp-expr
y) |
  stamp-expr (ConstantExpr val) = constantAsStamp val |
  stamp-expr (LeafExpr i s) = s |
  stamp-expr (ParameterExpr i s) = s |
  stamp-expr (ConditionalExpr c t f) = meet (stamp-expr t) (stamp-expr f)
```

```
export-code stamp-unary stamp-binary stamp-expr
```

1.3 Data-flow Tree Evaluation

```
fun unary-eval :: IRUnaryOp ⇒ 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
```

```
fun bin-eval :: IRBinaryOp ⇒ Value ⇒ Value ⇒ Value where
  bin-eval BinAdd v1 v2 = intval-add v1 v2 |
  bin-eval BinMul v1 v2 = intval-mul v1 v2 |
  bin-eval BinSub v1 v2 = intval-sub v1 v2 |
  bin-eval BinAnd v1 v2 = intval-and v1 v2 |
  bin-eval BinOr  v1 v2 = intval-or v1 v2 |
  bin-eval BinXor v1 v2 = intval-xor v1 v2 |
  bin-eval BinShortCircuitOr v1 v2 = intval-short-circuit-or v1 v2 |
  bin-eval BinLeftShift v1 v2 = intval-left-shift v1 v2 |
  bin-eval BinRightShift v1 v2 = intval-right-shift v1 v2 |
  bin-eval BinURightShift v1 v2 = intval-uright-shift v1 v2 |
  bin-eval BinIntegerEquals v1 v2 = intval-equals v1 v2 |
  bin-eval BinIntegerLessThan v1 v2 = intval-less-than v1 v2 |
  bin-eval BinIntegerBelow v1 v2 = intval-below v1 v2
```

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

```
inductive not-undef-or-fail :: Value ⇒ 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;
    branch = (if val-to-bool cond then te else fe);
    [m,p] ⊢ branch ↦ result;
    result ≠ 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 ([-,-] ⊢ - ↦L
- 55)
  for m p where

  EvalNil:
  [m,p] ⊢ [] ↦L [] |
  EvalCons:
  ⟦[m,p] ⊢ x ↦ xval;
    [m,p] ⊢ yy ↦_L yyval⟧
    ⟹ [m,p] ⊢ (x#yy) ↦_L (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 (- ≐ - 55) where
  (e1 ≐ e2) = (∀ m p v. (([m,p] ⊢ e1 ↦ v) ⟷ ([m,p] ⊢ e2 ↦ 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]:
    (e₂ ≤ e₁) ⟷ (∀ m p v. (([m,p] ⊢ e₁ ↦ v) ⟶ ([m,p] ⊢ e₂ ↦ v)))

definition
  lt-expr-def [simp]:
    (e₁ < e₂) ⟷ (e₁ ≤ e₂ ∧ ¬ (e₁ ≐ e₂))
```

```
instance proof
  fix x y z :: IRExpr
  show x < y ⟷ x ≤ y ∧ ¬ (y ≤ x) by (simp add: equiv-exprs-def; auto)
  show x ≤ x by simp
  show x ≤ y ⟹ y ≤ z ⟹ x ≤ z by simp
qed

end

abbreviation (output) Refines :: IRExpr ⇒ IRExpr ⇒ bool (infix ⊒ 64)
  where e₁ ⊒ e₂ ≡ (e₂ ≤ e₁)
```

1.5 Stamp Masks

A stamp can contain additional range information in the form of masks. A stamp has an up mask and a down mask, corresponding to 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
  using up-spec
  using bit-and-iff bit-eq-iff bit-not-iff bit-unsigned-iff down-spec
  by (smt (verit, best) bit.double-compl)

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

```
lemma not-must-implies-either:
  [m, p] ⊢ 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
  using assms
 by (smt (z3) and.commute and.right-neutral and-zero-eq bit.compl-zero bit.conj-cancel-right
bit.conj-disj-distribs(1) ucast-id up-spec word-bw-assocs(1) word-not-dist(2))

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
  using assms
 by (smt (z3) and-zero-eq bit.conj-cancel-left bit.conj-disj-distribs(1) bit.conj-disj-distribs(2)
bit.de-Morgan-disj down-spec or-eq-not-not-and ucast-id up-spec word-ao-absorbs(2)
word-ao-absorbs(8) word-bw-lcs(1) word-not-dist(2))

end

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
  unfolding IRExpr-up-def IRExpr-down-def
  apply unfold-locales
  by (simp add: ucast-minus-one)+
end
```

2 Tree to Graph

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

2.1 Subgraph to Data-flow Tree

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

export-code find-node-and-stamp


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


inductive
  rep :: IRGraph ⇒ ID ⇒ IRExpr ⇒ bool (- ⊢ - ≃ - 55)
  for g where

  ConstantNode:
  ⟦kind g n = ConstantNode c⟧
    ⟹ g ⊢ n ≃ (ConstantExpr c) |

  ParameterNode:
  ⟦kind g n = ParameterNode i;
    stamp g n = s⟧
    ⟹ g  ⊢ n ≃ (ParameterExpr i s) |

  ConditionalNode:
  ⟦kind g n = ConditionalNode c t f;
    g ⊢ c ≃ ce;
    g ⊢ t ≃ te;
    g ⊢ f ≃ fe⟧
```

$\Longrightarrow g \vdash n \simeq (\mathit{ConditionalExpr}\ \mathit{ce}\ \mathit{te}\ \mathit{fe})$ |

$\mathit{AbsNode}$:
$\llbracket \mathit{kind}\ g\ n = \mathit{AbsNode}\ x;$
$g \vdash x \simeq \mathit{xe}\rrbracket$
$\Longrightarrow g \vdash n \simeq (\mathit{UnaryExpr}\ \mathit{UnaryAbs}\ \mathit{xe})$ |

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

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

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

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

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

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

$\mathit{AndNode}$:
$\llbracket \mathit{kind}\ g\ n = \mathit{AndNode}\ x\ y;$
$g \vdash x \simeq \mathit{xe};$
$g \vdash y \simeq \mathit{ye}\rrbracket$
$\Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr}\ \mathit{BinAnd}\ \mathit{xe}\ \mathit{ye})$ |

OrNode:

$$\begin{array}{l}
[\![\mathit{kind}\ g\ n = \mathit{OrNode}\ x\ y;\\
\quad g \vdash x \simeq \mathit{xe};\\
\quad g \vdash y \simeq \mathit{ye}]\!]\\
\quad \Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr}\ \mathit{BinOr}\ \mathit{xe}\ \mathit{ye})\ |
\end{array}$$

XorNode:

$$\begin{array}{l}
[\![\mathit{kind}\ g\ n = \mathit{XorNode}\ x\ y;\\
\quad g \vdash x \simeq \mathit{xe};\\
\quad g \vdash y \simeq \mathit{ye}]\!]\\
\quad \Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr}\ \mathit{BinXor}\ \mathit{xe}\ \mathit{ye})\ |
\end{array}$$

ShortCircuitOrNode:

$$\begin{array}{l}
[\![\mathit{kind}\ g\ n = \mathit{ShortCircuitOrNode}\ x\ y;\\
\quad g \vdash x \simeq \mathit{xe};\\
\quad g \vdash y \simeq \mathit{ye}]\!]\\
\quad \Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr}\ \mathit{BinShortCircuitOr}\ \mathit{xe}\ \mathit{ye})\ |
\end{array}$$

LeftShiftNode:

$$\begin{array}{l}
[\![\mathit{kind}\ g\ n = \mathit{LeftShiftNode}\ x\ y;\\
\quad g \vdash x \simeq \mathit{xe};\\
\quad g \vdash y \simeq \mathit{ye}]\!]\\
\Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr}\ \mathit{BinLeftShift}\ \mathit{xe}\ \mathit{ye})\ |
\end{array}$$

RightShiftNode:

$$\begin{array}{l}
[\![\mathit{kind}\ g\ n = \mathit{RightShiftNode}\ x\ y;\\
\quad g \vdash x \simeq \mathit{xe};\\
\quad g \vdash y \simeq \mathit{ye}]\!]\\
\Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr}\ \mathit{BinRightShift}\ \mathit{xe}\ \mathit{ye})\ |
\end{array}$$

UnsignedRightShiftNode:

$$\begin{array}{l}
[\![\mathit{kind}\ g\ n = \mathit{UnsignedRightShiftNode}\ x\ y;\\
\quad g \vdash x \simeq \mathit{xe};\\
\quad g \vdash y \simeq \mathit{ye}]\!]\\
\Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr}\ \mathit{BinURightShift}\ \mathit{xe}\ \mathit{ye})\ |
\end{array}$$

IntegerBelowNode:

$$\begin{array}{l}
[\![\mathit{kind}\ g\ n = \mathit{IntegerBelowNode}\ x\ y;\\
\quad g \vdash x \simeq \mathit{xe};\\
\quad g \vdash y \simeq \mathit{ye}]\!]\\
\quad \Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr}\ \mathit{BinIntegerBelow}\ \mathit{xe}\ \mathit{ye})\ |
\end{array}$$

IntegerEqualsNode:

$$\begin{array}{l}
[\![\mathit{kind}\ g\ n = \mathit{IntegerEqualsNode}\ x\ y;\\
\quad g \vdash x \simeq \mathit{xe};\\
\quad g \vdash y \simeq \mathit{ye}]\!]\\
\quad \Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr}\ \mathit{BinIntegerEquals}\ \mathit{xe}\ \mathit{ye})\ |
\end{array}$$

IntegerLessThanNode:

$$[\![\mathit{kind}\ g\ n = \mathit{IntegerLessThanNode}\ x\ y;$$

$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr}\ \mathit{BinIntegerLessThan}\ xe\ ye)\ |$

NarrowNode:
$\llbracket \mathit{kind}\ g\ n = \mathit{NarrowNode}\ \mathit{inputBits}\ \mathit{resultBits}\ x;$
$g \vdash x \simeq xe\rrbracket$
$\Longrightarrow g \vdash n \simeq (\mathit{UnaryExpr}\ (\mathit{UnaryNarrow}\ \mathit{inputBits}\ \mathit{resultBits})\ xe)\ |$

SignExtendNode:
$\llbracket \mathit{kind}\ g\ n = \mathit{SignExtendNode}\ \mathit{inputBits}\ \mathit{resultBits}\ x;$
$g \vdash x \simeq xe\rrbracket$
$\Longrightarrow g \vdash n \simeq (\mathit{UnaryExpr}\ (\mathit{UnarySignExtend}\ \mathit{inputBits}\ \mathit{resultBits})\ xe)\ |$

ZeroExtendNode:
$\llbracket \mathit{kind}\ g\ n = \mathit{ZeroExtendNode}\ \mathit{inputBits}\ \mathit{resultBits}\ x;$
$g \vdash x \simeq xe\rrbracket$
$\Longrightarrow g \vdash n \simeq (\mathit{UnaryExpr}\ (\mathit{UnaryZeroExtend}\ \mathit{inputBits}\ \mathit{resultBits})\ xe)\ |$

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

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

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

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

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

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

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

```
definition wf-term-graph :: MapState ⇒ Params ⇒ IRGraph ⇒ ID ⇒ bool where
  wf-term-graph m p g n = (∃ e. (g ⊢ n ≃ e) ∧ (∃ v. ([m, p] ⊢ e ↦ v)))
```

```
values {t. eg2-sq ⊢ 4 ≃ t}
```

2.2 Data-flow Tree to Subgraph

```
fun unary-node :: IRUnaryOp ⇒ ID ⇒ IRNode where
  unary-node UnaryAbs v = AbsNode v |
  unary-node UnaryNot v = NotNode v |
  unary-node UnaryNeg v = NegateNode v |
  unary-node UnaryLogicNegation v = LogicNegationNode v |
  unary-node (UnaryNarrow ib rb) v = NarrowNode ib rb v |
  unary-node (UnarySignExtend ib rb) v = SignExtendNode ib rb v |
  unary-node (UnaryZeroExtend ib rb) v = ZeroExtendNode ib rb v
```

```
fun bin-node :: IRBinaryOp ⇒ ID ⇒ ID ⇒ IRNode where
  bin-node BinAdd x y = AddNode x y |
  bin-node BinMul x y = MulNode x y |
  bin-node BinSub x y = SubNode x y |
  bin-node BinAnd x y = AndNode x y |
  bin-node BinOr  x y = OrNode x y |
  bin-node BinXor x y = XorNode x y |
  bin-node BinShortCircuitOr x y = ShortCircuitOrNode x y |
  bin-node BinLeftShift x y = LeftShiftNode x y |
  bin-node BinRightShift x y = RightShiftNode x y |
  bin-node BinURightShift x y = UnsignedRightShiftNode x y |
  bin-node BinIntegerEquals x y = IntegerEqualsNode x y |
  bin-node BinIntegerLessThan x y = IntegerLessThanNode x y |
  bin-node BinIntegerBelow x y = IntegerBelowNode x y
```

```
inductive fresh-id :: IRGraph ⇒ 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
  unrep :: IRGraph ⇒ IRExpr ⇒ (IRGraph × ID) ⇒ bool (- ⊕ - ⇝ - 55)
   where

  ConstantNodeSame:
  ⟦find-node-and-stamp g (ConstantNode c, constantAsStamp c) = Some n⟧
    ⟹ g ⊕ (ConstantExpr c) ⇝ (g, n) |

  ConstantNodeNew:
  ⟦find-node-and-stamp g (ConstantNode c, constantAsStamp c) = None;
    n = get-fresh-id g;
    g' = add-node n (ConstantNode c, constantAsStamp c) g ⟧
    ⟹ g ⊕ (ConstantExpr c) ⇝ (g', n) |

  ParameterNodeSame:
  ⟦find-node-and-stamp g (ParameterNode i, s) = Some n⟧
    ⟹ g ⊕ (ParameterExpr i s) ⇝ (g, n) |

  ParameterNodeNew:
  ⟦find-node-and-stamp g (ParameterNode i, s) = None;
    n = get-fresh-id g;
    g' = add-node n (ParameterNode i, s) g⟧
    ⟹ g ⊕ (ParameterExpr i s) ⇝ (g', n) |

  ConditionalNodeSame:
  ⟦g ⊕ ce ⇝ (g2, c);
    g2 ⊕ te ⇝ (g3, t);
    g3 ⊕ fe ⇝ (g4, f);
    s' = meet (stamp g4 t) (stamp g4 f);
    find-node-and-stamp g4 (ConditionalNode c t f, s') = Some n⟧
    ⟹ g ⊕ (ConditionalExpr ce te fe) ⇝ (g4, n) |

  ConditionalNodeNew:
  ⟦g ⊕ ce ⇝ (g2, c);
    g2 ⊕ te ⇝ (g3, t);
    g3 ⊕ fe ⇝ (g4, f);
    s' = meet (stamp g4 t) (stamp g4 f);
    find-node-and-stamp g4 (ConditionalNode c t f, s') = None;
    n = get-fresh-id g4;
    g' = add-node n (ConditionalNode c t f, s') g4⟧
    ⟹ g ⊕ (ConditionalExpr ce te fe) ⇝ (g', n) |

  UnaryNodeSame:
  ⟦g ⊕ xe ⇝ (g2, x);
    s' = stamp-unary op (stamp g2 x);
    find-node-and-stamp g2 (unary-node op x, s') = Some n⟧
```

$\Longrightarrow g \oplus (\mathit{UnaryExpr}\ \mathit{op}\ \mathit{xe}) \leadsto (g2,\ n)$ |

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

BinaryNodeSame:
$[\![g \oplus \mathit{xe} \leadsto (g2,\ x);$
$g2 \oplus \mathit{ye} \leadsto (g3,\ y);$
$s' = \mathit{stamp\text{-}binary}\ \mathit{op}\ (\mathit{stamp}\ g3\ x)\ (\mathit{stamp}\ g3\ y);$
$\mathit{find\text{-}node\text{-}and\text{-}stamp}\ g3\ (\mathit{bin\text{-}node}\ \mathit{op}\ x\ y,\ s') = \mathit{Some}\ n]\!]$
$\Longrightarrow g \oplus (\mathit{BinaryExpr}\ \mathit{op}\ \mathit{xe}\ \mathit{ye}) \leadsto (g3,\ n)$ |

BinaryNodeNew:
$[\![g \oplus \mathit{xe} \leadsto (g2,\ x);$
$g2 \oplus \mathit{ye} \leadsto (g3,\ y);$
$s' = \mathit{stamp\text{-}binary}\ \mathit{op}\ (\mathit{stamp}\ g3\ x)\ (\mathit{stamp}\ g3\ y);$
$\mathit{find\text{-}node\text{-}and\text{-}stamp}\ g3\ (\mathit{bin\text{-}node}\ \mathit{op}\ x\ y,\ s') = \mathit{None};$
$n = \mathit{get\text{-}fresh\text{-}id}\ g3;$
$g' = \mathit{add\text{-}node}\ n\ (\mathit{bin\text{-}node}\ \mathit{op}\ x\ y,\ s')\ g3]\!]$
$\Longrightarrow g \oplus (\mathit{BinaryExpr}\ \mathit{op}\ \mathit{xe}\ \mathit{ye}) \leadsto (g',\ n)$ |

AllLeafNodes:
$[\![\mathit{stamp}\ g\ n = s;$
$\mathit{is\text{-}preevaluated}\ (\mathit{kind}\ g\ n)]\!]$
$\Longrightarrow g \oplus (\mathit{LeafExpr}\ n\ s) \leadsto (g,\ n)$

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

unrepRules

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

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

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

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

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

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

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

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

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

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

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

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

2.3 Lift Data-flow Tree Semantics

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

2.4 Graph Refinement

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

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

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

2.5 Maximal Sharing

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

end

2.6 Formedness Properties

theory *Form*
imports
Semantics.TreeToGraph
begin

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

definition *wf-closed* **where**
$\textit{wf-closed}\ g =$
$(\forall\ n \in \textit{ids}\ g\ .$

```
      inputs g n ⊆ ids g ∧
      succ g n ⊆ ids g ∧
      kind g n ≠ NoNode)

definition wf-phis where
  wf-phis g =
    (∀ n ∈ ids g.
      is-PhiNode (kind g n) ⟶
      length (ir-values (kind g n))
       = length (ir-ends
          (kind g (ir-merge (kind g n)))))

definition wf-ends where
  wf-ends g =
    (∀ n ∈ ids g .
      is-AbstractEndNode (kind g n) ⟶
      card (usages g n) > 0)

fun wf-graph :: IRGraph ⇒ bool where
  wf-graph g = (wf-start g ∧ wf-closed g ∧ wf-phis g ∧ 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 ⇒ bool where
  wf-stamps g = (∀ n ∈ ids g .
    (∀ v m p e . (g ⊢ n ≃ e) ∧ ([m, p] ⊢ e ↦ v) ⟶ valid-value v (stamp-expr e)))

fun wf-stamp :: IRGraph ⇒ (ID ⇒ Stamp) ⇒ bool where
  wf-stamp g s = (∀ n ∈ ids g .
    (∀ v m p e . (g ⊢ n ≃ e) ∧ ([m, p] ⊢ e ↦ v) ⟶ valid-value v (s n)))

lemma wf-empty: wf-graph start-end-graph
  unfolding start-end-graph-def wf-folds by simp

lemma wf-eg2-sq: wf-graph eg2-sq
  unfolding eg2-sq-def wf-folds by simp

fun wf-logic-node-inputs :: IRGraph ⇒ ID ⇒ 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 auto

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
  using changeonly.simps by blast
```

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 eval-usages.simps eval-uses.intros(1)
  by (simp add: ids.rep-eq)

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

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

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



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

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

lemma inp-in-g-wf:
  assumes wf-graph g
  assumes n ∈ inputs g nid
  shows n ∈ ids g
  using assms unfolding wf-folds
  using inp-in-g by blast


lemma kind-unchanged:
  assumes nid ∈ 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
    using unchanged.simps by blast
qed

lemma stamp-unchanged:
  assumes nid ∈ ids g1
  assumes unchanged (eval-usages g1 nid) g1 g2
  shows stamp g1 nid = stamp g2 nid
  by (meson assms(1) assms(2) eval-usages-self unchanged.elims(2))


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

lemma eval-usages:
  assumes us = eval-usages g nid
  assumes nid′ ∈ ids g
  shows eval-uses g nid nid′ ⟷ nid′ ∈ us (is ?P ⟷ ?Q)
  using assms eval-usages.simps
  by (simp add: ids.rep-eq)
```

```
lemma inputs-are-uses:
  assumes nid′ ∈ inputs g nid
  shows eval-uses g nid nid′
  by (metis assms use-inp)

lemma inputs-are-usages:
  assumes nid′ ∈ inputs g nid
  assumes nid′ ∈ ids g
  shows nid′ ∈ eval-usages g nid
  using assms(1) assms(2) eval-usages inputs-are-uses by blast

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(1) assms(2) 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 eval-usages
  using assms(1) assms(2) assms(3) 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 auto

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

lemma eval-in-ids:
  assumes [g, m, p] ⊢ nid ↦ v
  shows nid ∈ ids g
  using assms using encodeeval-def encode-in-ids
  by auto

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

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
    using dom nc kind-unchanged
    by metis
  then show ?case using rep.ConstantNode
    by presburger
next
  case (ParameterNode n i s)
  then have kind g2 n = ParameterNode i
    by (metis kind-unchanged)
  then show ?case
   by (metis ParameterNode.hyps(2) ParameterNode.prems(1) ParameterNode.prems(3)
rep.ParameterNode stamp-unchanged)
next
  case (ConditionalNode n c t f ce te fe)
  then have kind g2 n = ConditionalNode c t f
    by (metis kind-unchanged)
  have c ∈ eval-usages g1 n ∧ t ∈ eval-usages g1 n ∧ f ∈ eval-usages g1 n
    using inputs-of-ConditionalNode
     by (metis ConditionalNode.hyps(1) ConditionalNode.hyps(2) ConditionalNode.hyps(3) ConditionalNode.hyps(4) encode-in-ids inputs.simps inputs-are-usages
list.set-intros(1) set-subset-Cons subset-code(1))
  then show ?case using transitive-kind-same
   by (metis ConditionalNode.hyps(1) ConditionalNode.prems(1) IRNodes.inputs-of-ConditionalNode
‹kind g2 n = ConditionalNode c t f› child-unchanged inputs.simps list.set-intros(1)
local.ConditionalNode(5) local.ConditionalNode(6) local.ConditionalNode(7) local.ConditionalNode(9)
rep.ConditionalNode set-subset-Cons subset-code(1) unchanged.elims(2))
next
  case (AbsNode n x xe)
  then have kind g2 n = AbsNode x
    using kind-unchanged
    by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-AbsNode
     by (metis AbsNode.hyps(1) AbsNode.hyps(2) encode-in-ids inputs.simps inputs-are-usages list.set-intros(1))
  then show ?case
    by (metis AbsNode.IH AbsNode.hyps(1) AbsNode.prems(1) AbsNode.prems(3)
IRNodes.inputs-of-AbsNode ‹kind g2 n = AbsNode x› child-member-in child-unchanged
local.wf member-rec(1) rep.AbsNode unchanged.simps)
next
  case (NotNode n x xe)
  then have kind g2 n = NotNode x
    using kind-unchanged
    by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-NotNode
     by (metis NotNode.hyps(1) NotNode.hyps(2) encode-in-ids inputs.simps
inputs-are-usages list.set-intros(1))
  then show ?case
    by (metis NotNode.IH NotNode.hyps(1) NotNode.prems(1) NotNode.prems(3)
IRNodes.inputs-of-NotNode ‹kind g2 n = NotNode x› child-member-in child-unchanged
local.wf member-rec(1) rep.NotNode unchanged.simps)
next
  case (NegateNode n x xe)
  then have kind g2 n = NegateNode x
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-NegateNode
    by (metis NegateNode.hyps(1) NegateNode.hyps(2) encode-in-ids inputs.simps
inputs-are-usages list.set-intros(1))
  then show ?case
     by (metis IRNodes.inputs-of-NegateNode NegateNode.IH NegateNode.hyps(1)
NegateNode.prems(1) NegateNode.prems(3) ‹kind g2 n = NegateNode x› child-member-in
child-unchanged local.wf member-rec(1) rep.NegateNode unchanged.elims(1))
next
  case (LogicNegationNode n x xe)
  then have kind g2 n = LogicNegationNode x
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-LogicNegationNode inputs-of-are-usages
   by (metis LogicNegationNode.hyps(1) LogicNegationNode.hyps(2) encode-in-ids
member-rec(1))
  then show ?case
     by (metis IRNodes.inputs-of-LogicNegationNode LogicNegationNode.IH
LogicNegationNode.hyps(1) LogicNegationNode.hyps(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
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-LogicNegationNode inputs-of-are-usages
   by (metis AddNode.hyps(1) AddNode.hyps(2) AddNode.hyps(3) IRNodes.inputs-of-AddNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case
    by (metis AddNode.IH(1) AddNode.IH(2) AddNode.hyps(1) AddNode.hyps(2)
AddNode.hyps(3) AddNode.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
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-LogicNegationNode inputs-of-are-usages
   by (metis MulNode.hyps(1) MulNode.hyps(2) MulNode.hyps(3) IRNodes.inputs-of-MulNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using MulNode inputs-of-MulNode
   by (metis ‹kind g2 n = MulNode x y› child-unchanged inputs.simps list.set-intros(1)
rep.MulNode set-subset-Cons subset-iff unchanged.elims(2))
next
  case (SubNode n x y xe ye)
  then have kind g2 n = SubNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-LogicNegationNode inputs-of-are-usages
   by (metis SubNode.hyps(1) SubNode.hyps(2) SubNode.hyps(3) IRNodes.inputs-of-SubNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using SubNode inputs-of-SubNode
   by (metis ‹kind g2 n = SubNode x y› child-member child-unchanged encode-in-ids
ids-some member-rec(1) rep.SubNode)
next
  case (AndNode n x y xe ye)
  then have kind g2 n = AndNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-LogicNegationNode inputs-of-are-usages
   by (metis AndNode.hyps(1) AndNode.hyps(2) AndNode.hyps(3) IRNodes.inputs-of-AndNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using AndNode inputs-of-AndNode
   by (metis ‹kind g2 n = AndNode x y› child-unchanged inputs.simps list.set-intros(1)
rep.AndNode set-subset-Cons subset-iff unchanged.elims(2))
next
  case (OrNode n x y xe ye)
  then have kind g2 n = OrNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-OrNode inputs-of-are-usages
   by (metis OrNode.hyps(1) OrNode.hyps(2) OrNode.hyps(3) IRNodes.inputs-of-OrNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using OrNode inputs-of-OrNode
   by (metis ‹kind g2 n = OrNode x y› child-member child-unchanged encode-in-ids
ids-some member-rec(1) rep.OrNode)
next
  case (XorNode n x y xe ye)
  then have kind g2 n = XorNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-XorNode inputs-of-are-usages
   by (metis XorNode.hyps(1) XorNode.hyps(2) XorNode.hyps(3) IRNodes.inputs-of-XorNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using XorNode inputs-of-XorNode
      by (metis ‹kind g2 n = XorNode x y› child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.XorNode)
next
  case (ShortCircuitOrNode n x y xe ye)
  then have kind g2 n = ShortCircuitOrNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-XorNode inputs-of-are-usages
    by (metis ShortCircuitOrNode.hyps(1) ShortCircuitOrNode.hyps(2)
ShortCircuitOrNode.hyps(3) IRNodes.inputs-of-ShortCircuitOrNode encode-in-ids in-mono
inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using ShortCircuitOrNode inputs-of-ShortCircuitOrNode
    by (metis ‹kind g2 n = ShortCircuitOrNode x y› child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.ShortCircuitOrNode)
next
case (LeftShiftNode n x y xe ye)
  then have kind g2 n = LeftShiftNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-XorNode inputs-of-are-usages
     by (metis LeftShiftNode.hyps(1) LeftShiftNode.hyps(2) LeftShiftNode.hyps(3)
IRNodes.inputs-of-LeftShiftNode encode-in-ids in-mono inputs.simps inputs-are-usages
list.set-intros(1) set-subset-Cons)
  then show ?case using LeftShiftNode inputs-of-LeftShiftNode
     by (metis ‹kind g2 n = LeftShiftNode x y› child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.LeftShiftNode)
next
case (RightShiftNode n x y xe ye)
  then have kind g2 n = RightShiftNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-RightShiftNode inputs-of-are-usages
   by (metis RightShiftNode.hyps(1) RightShiftNode.hyps(2) RightShiftNode.hyps(3)
IRNodes.inputs-of-RightShiftNode encode-in-ids in-mono inputs.simps inputs-are-usages
list.set-intros(1) set-subset-Cons)
  then show ?case using RightShiftNode inputs-of-RightShiftNode
      by (metis ‹kind g2 n = RightShiftNode x y› child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.RightShiftNode)
next
```

```
case (UnsignedRightShiftNode n x y xe ye)
  then have kind g2 n = UnsignedRightShiftNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-UnsignedRightShiftNode inputs-of-are-usages
    by (metis UnsignedRightShiftNode.hyps(1) UnsignedRightShiftNode.hyps(2) UnsignedRightShiftNode.hyps(3) IRNodes.inputs-of-UnsignedRightShiftNode encode-in-ids
in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using UnsignedRightShiftNode inputs-of-UnsignedRightShiftNode
    by (metis ‹kind g2 n = UnsignedRightShiftNode x y› child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.UnsignedRightShiftNode)
next
  case (IntegerBelowNode n x y xe ye)
  then have kind g2 n = IntegerBelowNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-IntegerBelowNode inputs-of-are-usages
    by (metis IntegerBelowNode.hyps(1) IntegerBelowNode.hyps(2) IntegerBelowNode.hyps(3) IRNodes.inputs-of-IntegerBelowNode encode-in-ids in-mono inputs.simps
inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using IntegerBelowNode inputs-of-IntegerBelowNode
    by (metis ‹kind g2 n = IntegerBelowNode x y› child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.IntegerBelowNode)
next
  case (IntegerEqualsNode n x y xe ye)
  then have kind g2 n = IntegerEqualsNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-IntegerEqualsNode inputs-of-are-usages
    by (metis IntegerEqualsNode.hyps(1) IntegerEqualsNode.hyps(2) IntegerEqualsNode.hyps(3) IRNodes.inputs-of-IntegerEqualsNode encode-in-ids in-mono inputs.simps
inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using IntegerEqualsNode inputs-of-IntegerEqualsNode
    by (metis ‹kind g2 n = IntegerEqualsNode x y› child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.IntegerEqualsNode)
next
  case (IntegerLessThanNode n x y xe ye)
  then have kind g2 n = IntegerLessThanNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-IntegerLessThanNode inputs-of-are-usages
    by (metis IntegerLessThanNode.hyps(1) IntegerLessThanNode.hyps(2) IntegerLessThanNode.hyps(3) IRNodes.inputs-of-IntegerLessThanNode encode-in-ids in-mono
inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using IntegerLessThanNode inputs-of-IntegerLessThanNode
    by (metis ‹kind g2 n = IntegerLessThanNode x y› child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.IntegerLessThanNode)
next
  case (NarrowNode n ib rb x xe)
  then have kind g2 n = NarrowNode ib rb x
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-NarrowNode inputs-of-are-usages
   by (metis NarrowNode.hyps(1) NarrowNode.hyps(2) IRNodes.inputs-of-NarrowNode
encode-in-ids inputs.simps inputs-are-usages list.set-intros(1))
  then show ?case using NarrowNode inputs-of-NarrowNode
     by (metis ‹kind g2 n = NarrowNode ib rb x› child-unchanged inputs.elims
list.set-intros(1) rep.NarrowNode unchanged.simps)
next
  case (SignExtendNode n ib rb x xe)
  then have kind g2 n = SignExtendNode ib rb x
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-SignExtendNode inputs-of-are-usages
    by (metis SignExtendNode.hyps(1) SignExtendNode.hyps(2) encode-in-ids
inputs.simps inputs-are-usages list.set-intros(1))
  then show ?case using SignExtendNode inputs-of-SignExtendNode
   by (metis ‹kind g2 n = SignExtendNode ib rb x› child-member-in child-unchanged
in-set-member list.set-intros(1) rep.SignExtendNode unchanged.elims(2))
next
  case (ZeroExtendNode n ib rb x xe)
  then have kind g2 n = ZeroExtendNode ib rb x
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-ZeroExtendNode inputs-of-are-usages
   by (metis ZeroExtendNode.hyps(1) ZeroExtendNode.hyps(2) IRNodes.inputs-of-ZeroExtendNode
encode-in-ids inputs.simps inputs-are-usages list.set-intros(1))
  then show ?case using ZeroExtendNode inputs-of-ZeroExtendNode
   by (metis ‹kind g2 n = ZeroExtendNode ib rb x› child-member-in child-unchanged
member-rec(1) rep.ZeroExtendNode unchanged.simps)
next
  case (LeafNode n s)
  then show ?case
    by (metis kind-unchanged rep.LeafNode stamp-unchanged)
next
  case (RefNode n n')
  then have kind g2 n = RefNode n'
    using kind-unchanged by metis
  then have n' ∈ eval-usages g1 n
     by (metis IRNodes.inputs-of-RefNode RefNode.hyps(1) RefNode.hyps(2)
encode-in-ids inputs.elims inputs-are-usages list.set-intros(1))
  then show ?case
   by (metis IRNodes.inputs-of-RefNode RefNode.IH RefNode.hyps(1) RefNode.hyps(2)
RefNode.prems(1) ‹kind g2 n = RefNode n'› child-unchanged encode-in-ids
inputs.elims list.set-intros(1) local.wf rep.RefNode)
qed
qed
```

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


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

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

lemma add-node-unchanged:
  assumes new ∉ 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
    using eval-usages.simps by blast
  then have changeonly {new} g gup
    using assms add-changed by blast
  then show ?thesis using assms add-node-def disjoint-change
    using Diff-insert-absorb by auto
qed

lemma eval-uses-imp:
  ((nid' ∈ ids g ∧ nid = nid')
    ∨ nid' ∈ inputs g nid
    ∨ (∃ nid'' . eval-uses g nid nid'' ∧ eval-uses g nid'' nid'))
    ⟷ eval-uses g nid nid'
  using use0 use-inp use-trans
  by (meson eval-uses.simps)

lemma wf-use-ids:
  assumes wf-graph g
  assumes nid ∈ ids g
  assumes eval-uses g nid nid'
  shows nid' ∈ ids g
  using assms(3)
proof (induction rule: eval-uses.induct)
  case use0
  then show ?case by simp
next
  case use-inp
  then show ?case
    using assms(1) inp-in-g-wf by blast
next
  case use-trans
  then show ?case by blast
qed

lemma no-external-use:
  assumes wf-graph g
  assumes nid′ ∉ ids g
  assumes nid ∈ ids g
  shows ¬(eval-uses g nid nid′)
proof −
  have 0: nid ≠ nid′
    using assms by blast
  have inp: nid′ ∉ inputs g nid
    using assms
    using inp-in-g-wf by blast
  have rec-0: ∄n . n ∈ ids g ∧ n = nid′
    using assms by blast
  have rec-inp: ∄n . n ∈ ids g ∧ n ∈ inputs g nid′
    using assms(2) inp-in-g by blast
  have rec: ∄nid′′ . eval-uses g nid nid′′ ∧ eval-uses g nid′′ nid′
    using wf-use-ids assms(1) assms(2) assms(3) by blast
  from inp 0 rec show ?thesis
    using eval-uses-imp by blast
qed

end
```

3 Control-flow Semantics

```
theory IRStepObj
  imports
    TreeToGraph
begin
```

3.1 Object Heap

The heap model we introduce maps field references to object instances to runtime values. We use the H[f][p] heap representation. See \cite{heap−reps−2011}. We also introduce the DynamicHeap type which allocates new object references sequentially storing the next free object reference as 'Free'.

heapdef

```
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 :: ('a, 'b) DynamicHeap ⇒ ('a, 'b) DynamicHeap × Value
where
  h-new-inst (h, n) = ((h,n+1), (ObjRef (Some n)))

type-synonym FieldRefHeap = (string, objref) DynamicHeap
```

```
definition new-heap :: ('a, 'b) DynamicHeap where
  new-heap =  ((λf. λp. 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)

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

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

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

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

Intraprocedural semantics are given as a small-step semantics.

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

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

SequentialNode:
$[\![\mathit{is\text{-}sequential\text{-}node}\ (\mathit{kind}\ g\ \mathit{nid});$
$\mathit{nid}' = (\mathit{successors\text{-}of}\ (\mathit{kind}\ g\ \mathit{nid}))!\mathit{0}]\!]$
$\Longrightarrow g,\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow (\mathit{nid}',\ m,\ h)\ |$

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

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

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

$m' = \mathit{set\text{-}phis}\ \mathit{phis}\ \mathit{vs}\ m]\!]$
$\Longrightarrow g,\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow (\mathit{merge},\ m',\ h)\ |$

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

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

SignedDivNode:
$[\![\mathit{kind}\ g\ \mathit{nid} = (\mathit{SignedDivNode}\ \mathit{nid}\ x\ y\ \mathit{zero}\ \mathit{sb}\ \mathit{nxt});$
$g \vdash x \simeq \mathit{xe};$
$g \vdash y \simeq \mathit{ye};$

$[m,\ p] \vdash xe \mapsto v1$;
$[m,\ p] \vdash ye \mapsto v2$;
$v = (intval\text{-}div\ v1\ v2)$;
$m' =\ m(nid := v)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \to (nxt,\ m',\ h)$ |

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

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

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

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

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* = $Signature \rightharpoonup IRGraph$

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

```
FieldRefHeap ⇒ (IRGraph × ID × MapState × Params) list × FieldRefHeap ⇒
bool
  (- ⊢ - ⟶ - 55)
  for P where

  Lift:
  ⟦g, p ⊢ (nid, m, h) → (nid′, m′, h′)⟧
    ⟹ P ⊢ ((g,nid,m,p)#stk, h) ⟶ ((g,nid′,m′,p)#stk, h′) |

  InvokeNodeStep:
  ⟦is-Invoke (kind g nid);

    callTarget = ir-callTarget (kind g nid);
    kind g callTarget = (MethodCallTargetNode targetMethod arguments);
    Some targetGraph = P targetMethod;
    m′ = new-map-state;
    g ⊢ arguments ≃_L argsE;
    [m, p] ⊢ argsE ↦_L p′⟧
    ⟹ P ⊢ ((g,nid,m,p)#stk, h) ⟶ ((targetGraph,0,m′,p′)#(g,nid,m,p)#stk, h)
|

  ReturnNode:
  ⟦kind g nid = (ReturnNode (Some expr) -);
    g ⊢ expr ≃ e;
    [m, p] ⊢ e ↦ v;

    cm′ = cm(cnid := v);
    cnid′ = (successors-of (kind cg cnid))!0⟧
    ⟹ P ⊢ ((g,nid,m,p)#(cg,cnid,cm,cp)#stk, h) ⟶ ((cg,cnid′,cm′,cp)#stk, h)
|

  ReturnNodeVoid:
  ⟦kind g nid = (ReturnNode None -);
    cm′ = cm(cnid := (ObjRef (Some (2048))));

    cnid′ = (successors-of (kind cg cnid))!0⟧
    ⟹ P ⊢ ((g,nid,m,p)#(cg,cnid,cm,cp)#stk, h) ⟶ ((cg,cnid′,cm′,cp)#stk, h)
|

  UnwindNode:
  ⟦kind g nid = (UnwindNode exception);

    g ⊢ exception ≃ exceptionE;
    [m, p] ⊢ exceptionE ↦ e;

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

    cm′ = cm(cnid := e)⟧
  ⟹ P ⊢ ((g,nid,m,p)#(cg,cnid,cm,cp)#stk, h) ⟶ ((cg,exEdge,cm′,cp)#stk, h)
```

```
code-pred (modes: i ⇒ i ⇒ o ⇒ bool) step-top .
```

3.4 Big-step Execution

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

fun has-return :: MapState ⇒ bool where
  has-return m = (m 0 ≠ UndefVal)

inductive exec :: Program
      ⇒ (IRGraph × ID × MapState × Params) list × FieldRefHeap
      ⇒ Trace
      ⇒ (IRGraph × ID × MapState × Params) list × FieldRefHeap
      ⇒ Trace
      ⇒ bool
  (- ⊢ - | - ⟶* - | -)
  for P
  where
  ⟦P ⊢ (((g,nid,m,p)#xs),h) ⟶ (((g′,nid′,m′,p′)#ys),h′);
    ¬(has-return m′);

    l′ = (l @ [(g,nid,m,p)]);

    exec P (((g′,nid′,m′,p′)#ys),h′) l′ next-state l″⟧
    ⟹ exec P (((g,nid,m,p)#xs),h) l next-state l″

  |
  ⟦P ⊢ (((g,nid,m,p)#xs),h) ⟶ (((g′,nid′,m′,p′)#ys),h′);
    has-return m′;

    l′ = (l @ [(g,nid,m,p)])⟧
    ⟹ exec P (((g,nid,m,p)#xs),h) l (((g′,nid′,m′,p′)#ys),h′) l′
code-pred (modes: i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ bool as Exec) exec .


inductive exec-debug :: Program
     ⇒ (IRGraph × ID × MapState × Params) list × FieldRefHeap
     ⇒ nat
     ⇒ (IRGraph × ID × MapState × Params) list × FieldRefHeap
     ⇒ bool
  (-⊢-⟶*-* -)
  where
  ⟦n > 0;
    p ⊢ s ⟶ s′;
    exec-debug p s′ (n − 1) s″⟧
    ⟹ exec-debug p s n s″ |

  ⟦n = 0⟧
  ⟹ exec-debug p s n s
code-pred (modes: i ⇒ i ⇒ i ⇒ o ⇒ bool) exec-debug .
```

3.4.1 Heap Testing

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


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

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

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


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

definition eg4-sq :: IRGraph where
  eg4-sq = irgraph [
    (0, StartNode None 4, VoidStamp),
    (1, ParameterNode 0, default-stamp),
    (3, MulNode 1 1, default-stamp),
   (4, NewInstanceNode 4 ''obj-class'' None 5, ObjectStamp ''obj-class'' True True
True),
    (5, StoreFieldNode 5 field-sq 3 None (Some 4) 6, VoidStamp),
    (6, ReturnNode (Some 3) None, default-stamp)
   ]


values {h-load-field field-sq (Some 0) (prod.snd res) | res.
        (λx. Some 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*:
  $[m,p] \vdash e \mapsto_L v1 \Longrightarrow$
   $[m,p] \vdash e \mapsto_L v2 \Longrightarrow$
  $v1 = v2$
  apply (*induction arbitrary*: *v2 rule*: *evaltrees.induct*)
   apply (*elim EvalTreeE*; *auto*)
  using *evalDet* **by** *force*

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* $\neq$ *ObjRef v*
  by (*cases op*; *cases x*; *auto*)

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

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

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

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

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


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

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

```
lemma validStampIntConst:
  assumes v = IntVal b ival
  assumes 0 < b ∧ b ≤ 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 int-signed-value-bounds
    by presburger
  have s: constantAsStamp v = IntegerStamp b (int-signed-value b ival) (int-signed-value
b ival)
    using assms(1) constantAsStamp.simps(1) by blast
  then show ?thesis
    unfolding s valid-stamp.simps
    using assms(2) assms bnds by linarith
qed
```

```
lemma validDefIntConst:
  assumes v: v = IntVal b ival
  assumes 0 < b ∧ 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 int-signed-value-bounds
    by presburger
  have s: constantAsStamp v = IntegerStamp b (int-signed-value b ival) (int-signed-value
b ival)
    using assms(1) constantAsStamp.simps(1) by blast
  then show ?thesis
    unfolding s unfolding v unfolding valid-value.simps
    using assms validStampIntConst
    by simp
qed
```

3.5.3 Evaluation Results are Valid

A valid value cannot be $UndefVal$.

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


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

lemma valid-ObjStamp[elim]:
  shows valid-value val (ObjectStamp klass exact nonNull alwaysNull) ⟹
      (∃ v. val = ObjRef v)
  using valid-value.simps by (metis val-to-bool.cases)

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] ⊢ e ↦ v) ⟹ v ≠ UndefVal
  apply (induction rule: evaltree.induct)
  using valid-not-undef wf-value-def by auto


lemma leafint:
  assumes ev: [m,p] ⊢ LeafExpr i (IntegerStamp b lo hi) ↦ val
  shows ∃ b v. val = (IntVal b v)

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


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

lemma valid-value-signed-int-range [simp]:
  assumes valid-value val (IntegerStamp b lo hi)
  assumes lo < 0
  shows ∃ v. (val = IntVal b v ∧
             lo ≤ int-signed-value b v ∧
             int-signed-value b v ≤ hi)
  using assms valid-int
  by (metis valid-value.simps(1))
```

3.5.4 Example Data-flow Optimisations

3.5.5 Monotonicity of Expression Refinement

We prove that each subexpression position is monotonic. That is, optimizing a subexpression anywhere deep inside a top-level expression also optimizes that top-level expression.

Note that we might also be able to do this via reusing Isabelle's *mono* operator (HOL.Orderings theory), proving instantiations like $mono(UnaryExprop)$, but it is not obvious how to do this for both arguments of the binary expressions.

```
lemma mono-unary:
  assumes e ≥ e′
  shows (UnaryExpr op e) ≥ (UnaryExpr op e′)
  using UnaryExpr assms by auto

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

lemma never-void:
  assumes [m, p] ⊢ x ↦ xv
  assumes valid-value xv (stamp-expr xe)
  shows stamp-expr xe ≠ VoidStamp
  using valid-value.simps
  using assms(2) by force
```

```
lemma compatible-trans:
  compatible x y ∧ compatible y z ⟹ compatible x z
  by (cases x; cases y; cases z; simp del: valid-stamp.simps)

lemma compatible-refl:
  compatible x y ⟹ compatible y x
  using compatible.elims(2) by fastforce
```

```
lemma mono-conditional:
  assumes ce ≥ ce′
  assumes te ≥ te′
  assumes fe ≥ fe′
  shows (ConditionalExpr ce te fe) ≥ (ConditionalExpr ce′ te′ fe′)
proof (simp only: le-expr-def; (rule allI)+; rule impI)
  fix m p v
  assume a: [m,p] ⊢ ConditionalExpr ce te fe ↦ v
  then obtain cond where ce: [m,p] ⊢ ce ↦ cond by auto
  then have ce′: [m,p] ⊢ ce′ ↦ cond using assms by auto

  define branch  where b:  branch  = (if val-to-bool cond then te else fe)
  define branch′ where b′: branch′ = (if val-to-bool cond then te′ else fe′)
  then have beval: [m,p] ⊢ branch ↦ v using a b ce evalDet by blast

  from beval have [m,p] ⊢ branch′ ↦ v using assms b b′ by auto
  then show [m,p] ⊢ ConditionalExpr ce′ te′ fe′ ↦ v
    using ConditionalExpr ce′ b′
    using a by blast
qed
```

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 $bin_e val$ / $unary_e val$ level, simply by saying $unfoldingunfold_e valtree$.

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

```
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 ir-resultBits op)
proof (cases op ∈ normal-unary)
  case True
  then show ?thesis
   by (metis def empty-iff insert-iff intval-abs.elims intval-bits.simps intval-logic-negation.elims
intval-negate.elims intval-not.elims unary-eval.simps(1) unary-eval.simps(2) unary-eval.simps(3)
unary-eval.simps(4))
next
  case False
  consider ib ob where op = UnaryNarrow ib ob |
          ib ob where op = UnaryZeroExtend ib ob |
          ib ob where op = UnarySignExtend ib ob
    by (metis False IRUnaryOp.exhaust insert-iff)
  then show ?thesis
  proof (cases)
    case 1
    then show ?thesis
     by (metis False IRUnaryOp.sel(4) def intval-narrow.elims unary-eval.simps(5))
  next
    case 2
    then show ?thesis
    by (metis False IRUnaryOp.sel(6) def intval-zero-extend.elims unary-eval.simps(7))
  next
    case 3
    then show ?thesis
    by (metis False IRUnaryOp.sel(5) def intval-sign-extend.elims unary-eval.simps(6))
  qed
qed

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

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

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

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

```
    by (smt (z3) EvalTreeE(6) Value.simps(11) valid-value.elims(1) valid-value.simps(1))

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


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

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


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

```
lemma bin-eval-inputs-are-ints:
  assumes bin-eval op x y = 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 apply (cases op; cases x; cases y; simp)
    using that by blast+
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
    using unfold-binary by auto
  then have b = (if op ∈ normal-unary then intval-bits xv else ir-resultBits op)
    using unary-eval-new-int
    by (metis Value.disc(1) Value.discI(1) Value.sel(1) new-int.simps)
  then show ?case
    by (metis xv UnaryExpr.IH unary-normal-bitsize unary-not-normal-bitsize)
next
```

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


lemma unfold-binary-width:
  assumes op ∉ binary-fixed-32-ops ∧ op ∉ binary-shift-ops
  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 force+ apply auto[1]
    using assms apply (cases op; auto)
          apply (smt (verit) intval-add.elims Value.inject(1))
    using intval-mul.elims Value.inject(1)
         apply (smt (verit) new-int.simps new-int-bin.simps)
    using intval-sub.elims Value.inject(1)
        apply (smt (verit) new-int.simps new-int-bin.simps)
    using intval-and.elims Value.inject(1)
       apply (smt (verit) new-int.simps new-int-bin.simps take-bit-and)
    using intval-or.elims Value.inject(1)
      apply (smt (verit) new-int.simps new-int-bin.simps take-bit-or)
    using intval-xor.elims Value.inject(1)
     apply (smt (verit) new-int.simps new-int-bin.simps take-bit-xor)
  by blast

next
  assume R: ?R
  then obtain x y where [m,p] ⊢ 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 ⊢ *n* ≃ *e* ⟹
kind g n = *ConstantNode c* ⟹
e = *ConstantExpr c*
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-parameter* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
kind g n = *ParameterNode i* ⟹
(∃ *s*. *e* = *ParameterExpr i s*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-conditional* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
kind g n = *ConditionalNode c t f* ⟹
(∃ *ce te fe*. *e* = *ConditionalExpr ce te fe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-abs* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
kind g n = *AbsNode x* ⟹
(∃ *xe*. *e* = *UnaryExpr UnaryAbs xe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-not* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
kind g n = *NotNode x* ⟹
(∃ *xe*. *e* = *UnaryExpr UnaryNot xe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-negate* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
kind g n = *NegateNode x* ⟹
(∃ *xe*. *e* = *UnaryExpr UnaryNeg xe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-logicnegation* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
kind g n = *LogicNegationNode x* ⟹
(∃ *xe*. *e* = *UnaryExpr UnaryLogicNegation xe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-add* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
kind g n = *AddNode x y* ⟹
(∃ *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-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 ⊢ *n* ≃ *e* ⟹
kind g n = *UnsignedRightShiftNode x y* ⟹
(∃ *xe ye*. *e* = *BinaryExpr BinURightShift xe ye*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-integer-below* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
kind g n = *IntegerBelowNode x y* ⟹
(∃ *xe ye*. *e* = *BinaryExpr BinIntegerBelow xe ye*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-integer-equals* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
kind g n = *IntegerEqualsNode x y* ⟹
(∃ *xe ye*. *e* = *BinaryExpr BinIntegerEquals xe ye*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-integer-less-than* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
kind g n = *IntegerLessThanNode x y* ⟹
(∃ *xe ye*. *e* = *BinaryExpr BinIntegerLessThan xe ye*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-narrow* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
kind g n = *NarrowNode ib rb x* ⟹
(∃ *x*. *e* = *UnaryExpr* (*UnaryNarrow ib rb*) *x*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-sign-extend* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
kind g n = *SignExtendNode ib rb x* ⟹
(∃ *x*. *e* = *UnaryExpr* (*UnarySignExtend ib rb*) *x*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-zero-extend* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
kind g n = *ZeroExtendNode ib rb x* ⟹
(∃ *x*. *e* = *UnaryExpr* (*UnaryZeroExtend ib rb*) *x*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-load-field* [*rep*]:
g ⊢ *n* ≃ *e* ⟹
is-preevaluated (*kind g n*) ⟹
(∃ *s*. *e* = *LeafExpr n s*)
by (*induction rule*: *rep.induct*; *auto*)

```
lemma rep-ref [rep]:
  g ⊢ n ≃ e ⟹
   kind g n = RefNode n′ ⟹
   g ⊢ n′ ≃ e
  by (induction rule: rep.induct; auto)
```

```
method solve-det uses node =
  (match node in kind - - = node - for node ⇒
    ‹match rep in r: - ⟹ - = node - ⟹ - ⇒
      ‹match IRNode.inject in i: (node - = node -) = - ⇒
        ‹match RepE in e: - ⟹ (⋀x. - = node x ⟹ -) ⟹ - ⇒
          ‹match IRNode.distinct in d: node - ≠ RefNode - ⇒
            ‹metis i e r d›››› |
   match node in kind - - = node - - for node ⇒
    ‹match rep in r: - ⟹ - = node - - ⟹ - ⇒
      ‹match IRNode.inject in i: (node - - = node - -) = - ⇒
        ‹match RepE in e: - ⟹ (⋀x y. - = node x y ⟹ -) ⟹ - ⇒
          ‹match IRNode.distinct in d: node - - ≠ RefNode - ⇒
            ‹metis i e r d›››› |
   match node in kind - - = node - - - for node ⇒
    ‹match rep in r: - ⟹ - = node - - - ⟹ - ⇒
      ‹match IRNode.inject in i: (node - - - = node - - -) = - ⇒
        ‹match RepE in e: - ⟹ (⋀x y z. - = node x y z ⟹ -) ⟹ - ⇒
          ‹match IRNode.distinct in d: node - - - ≠ RefNode - ⇒
            ‹metis i e r d›››› |
  match node in kind - - = node - - - for node ⇒
    ‹match rep in r: - ⟹ - = node - - - ⟹ - ⇒
      ‹match IRNode.inject in i: (node - - - = node - - -) = - ⇒
        ‹match RepE in e: - ⟹ (⋀x. - = node - - x ⟹ -) ⟹ - ⇒
          ‹match IRNode.distinct in d: node - - - ≠ RefNode - ⇒
            ‹metis i e r d›››››)
```

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

```
lemma repDet:
  shows (g ⊢ n ≃ e_1) ⟹ (g ⊢ n ≃ e_2) ⟹ e_1 = e_2
proof (induction arbitrary: e_2 rule: rep.induct)
  case (ConstantNode n c)
  then show ?case using rep-constant by auto
next
  case (ParameterNode n i s)
  then show ?case
    by (metis IRNode.disc(2685) ParameterNodeE is-RefNode-def rep-parameter)
next
  case (ConditionalNode n c t f ce te fe)
  then show ?case
    using IRNode.distinct(593)
    using IRNode.inject(6) ConditionalNodeE rep-conditional
    by metis
next
  case (AbsNode n x xe)
  then show ?case
    by (solve-det node: AbsNode)
next
  case (NotNode n x xe)
  then show ?case
    by (solve-det node: NotNode)
next
  case (NegateNode n x xe)
  then show ?case
    by (solve-det node: NegateNode)
next
  case (LogicNegationNode n x xe)
  then show ?case
    by (solve-det node: LogicNegationNode)
next
  case (AddNode n x y xe ye)
  then show ?case
    by (solve-det node: AddNode)
next
  case (MulNode n x y xe ye)
  then show ?case
    by (solve-det node: MulNode)
next
  case (SubNode n x y xe ye)
  then show ?case
    by (solve-det node: SubNode)
next
  case (AndNode n x y xe ye)
  then show ?case
    by (solve-det node: AndNode)
next
  case (OrNode n x y xe ye)
  then show ?case
    by (solve-det node: OrNode)
next
  case (XorNode n x y xe ye)
  then show ?case
    by (solve-det node: XorNode)
next
  case (ShortCircuitOrNode n x y xe ye)
  then show ?case
    by (solve-det node: ShortCircuitOrNode)
next
  case (LeftShiftNode n x y xe ye)
  then show ?case
    by (solve-det node: LeftShiftNode)
```

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

lemma repAllDet:
  g ⊢ xs ≃L e1 ⟹
   g ⊢ xs ≃L e2 ⟹
   e1 = e2
proof (induction arbitrary: e2 rule: replist.induct)
  case RepNil
  then show ?case
```

```
    using replist.cases by auto
next
  case (RepCons x xe xs xse)
  then show ?case
    by (metis list.distinct(1) list.sel(1) list.sel(3) repDet replist.cases)
qed

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

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

3.8.2 Monotonicity of Graph Refinement

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

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

lemma mono-not:
  assumes kind g1 n = NotNode x ∧ kind g2 n = NotNode x
  assumes (g1 ⊢ x ≃ xe1) ∧ (g2 ⊢ x ≃ xe2)
  assumes xe1 ≥ xe2
  assumes (g1 ⊢ n ≃ e1) ∧ (g2 ⊢ n ≃ e2)
  shows e1 ≥ e2
  by (metis NotNode assms(1) assms(2) assms(3) assms(4) mono-unary repDet)

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

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

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

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

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

lemma mono-conditional-graph:
  assumes kind g1 n = ConditionalNode c t f ∧ kind g2 n = ConditionalNode c t f
  assumes (g1 ⊢ c ≃ ce1) ∧ (g2 ⊢ c ≃ ce2)
  assumes (g1 ⊢ t ≃ te1) ∧ (g2 ⊢ t ≃ te2)
  assumes (g1 ⊢ f ≃ fe1) ∧ (g2 ⊢ f ≃ fe2)
  assumes ce1 ≥ ce2 ∧ te1 ≥ te2 ∧ fe1 ≥ fe2
  assumes (g1 ⊢ n ≃ e1) ∧ (g2 ⊢ n ≃ e2)
  shows e1 ≥ e2
  using ConditionalNodeE IRNode.inject(6) assms(1) assms(2) assms(3) assms(4) assms(5) assms(6) mono-conditional repDet rep-conditional
  by (smt (verit, best) ConditionalNode)

lemma mono-add:
  assumes kind g1 n = AddNode x y ∧ kind g2 n = AddNode x y
  assumes (g1 ⊢ x ≃ xe1) ∧ (g2 ⊢ x ≃ xe2)
  assumes (g1 ⊢ y ≃ ye1) ∧ (g2 ⊢ y ≃ ye2)
  assumes xe1 ≥ xe2 ∧ ye1 ≥ ye2
  assumes (g1 ⊢ n ≃ e1) ∧ (g2 ⊢ n ≃ e2)
  shows e1 ≥ e2
  using mono-binary assms AddNodeE IRNode.inject(2) repDet rep-add
  by (metis IRNode.distinct(205))

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


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

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›)+
  apply force
  by fastforce

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

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

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

3.8.3 Lift Data-flow Tree Refinement to Graph Refinement

```
theorem graph-semantics-preservation:
  assumes a: e1' ≥ e2'
  assumes b: ({n'} ⊴ as-set g1) ⊆ as-set g2
  assumes c: g1 ⊢ n' ≃ e1'
  assumes d: g2 ⊢ n' ≃ e2'
  shows graph-refinement g1 g2
  unfolding graph-refinement-def apply rule
  apply (metis b d ids-some no-encoding not-excluded-keep-type singleton-iff subsetI)
  apply (rule allI) apply (rule impI) apply (rule allI) apply (rule impI)
  unfolding graph-represents-expression-def
proof −
  fix n e1
  assume e: n ∈ ids g1
  assume f: (g1 ⊢ n ≃ e1)

  show ∃ e2. (g2 ⊢ n ≃ e2) ∧ e1 ≥ e2
  proof (cases n = n')
    case True
    have g: e1 = e1' using c f True repDet by simp
    have h: (g2 ⊢ n ≃ e2') ∧ e1' ≥ e2'
      using True a d by blast
    then show ?thesis
      using g by blast
  next
    case False
    have n ∉ {n'}
      using False by simp
    then have i: kind g1 n = kind g2 n ∧ stamp g1 n = stamp g2 n
      using not-excluded-keep-type
      using b e by presburger
    show ?thesis using f i
    proof (induction e1)
      case (ConstantNode n c)
      then show ?case
        by (metis eq-refl rep.ConstantNode)
    next
      case (ParameterNode n i s)
      then show ?case
        by (metis eq-refl rep.ParameterNode)
    next
      case (ConditionalNode n c t f ce1 te1 fe1)
      have k: g1 ⊢ n ≃ ConditionalExpr ce1 te1 fe1 using f ConditionalNode
        by (simp add: ConditionalNode.hyps(2) rep.ConditionalNode)
      obtain cn tn fn where l: kind g1 n = ConditionalNode cn tn fn
        using ConditionalNode.hyps(1) by blast
      then have mc: g1 ⊢ cn ≃ ce1
        using ConditionalNode.hyps(1) ConditionalNode.hyps(2) by fastforce
      from l have mt: g1 ⊢ tn ≃ te1
        using ConditionalNode.hyps(1) ConditionalNode.hyps(3) by fastforce
      from l have mf: g1 ⊢ fn ≃ fe1
        using ConditionalNode.hyps(1) ConditionalNode.hyps(4) by fastforce
      then show ?case
      proof −
        have g1 ⊢ cn ≃ ce1 using mc by simp
        have g1 ⊢ tn ≃ te1 using mt by simp
        have g1 ⊢ fn ≃ fe1 using mf by simp
        have cer: ∃ ce2. (g2 ⊢ cn ≃ ce2) ∧ ce1 ≥ ce2
          using ConditionalNode
          using a b c d l no-encoding not-excluded-keep-type repDet singletonD
          by (metis-node-eq-ternary ConditionalNode)
        have ter: ∃ te2. (g2 ⊢ tn ≃ te2) ∧ te1 ≥ te2
         using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
          by (metis-node-eq-ternary ConditionalNode)
        have ∃ fe2. (g2 ⊢ fn ≃ fe2) ∧ fe1 ≥ fe2
         using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
          by (metis-node-eq-ternary ConditionalNode)
          then have ∃ ce2 te2 fe2. (g2 ⊢ n ≃ ConditionalExpr ce2 te2 fe2) ∧
ConditionalExpr ce1 te1 fe1 ≥ ConditionalExpr ce2 te2 fe2
          using ConditionalNode.prems l rep.ConditionalNode cer ter
          by (smt (verit) mono-conditional)
        then show ?thesis
          by meson
      qed
    next
      case (AbsNode n x xe1)
      have k: g1 ⊢ n ≃ UnaryExpr UnaryAbs xe1 using f AbsNode
        by (simp add: AbsNode.hyps(2) rep.AbsNode)
      obtain xn where l: kind g1 n = AbsNode xn
        using AbsNode.hyps(1) by blast
      then have m: g1 ⊢ xn ≃ xe1
        using AbsNode.hyps(1) AbsNode.hyps(2) by fastforce
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1' using c m repDet by simp
        then have ev: g2 ⊢ n ≃ UnaryExpr UnaryAbs e2' using AbsNode.hyps(1)
l m n
```

```
          using AbsNode.prems True d rep.AbsNode by simp
        then have r: UnaryExpr UnaryAbs e1' ≥ UnaryExpr UnaryAbs e2'
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using AbsNode
        using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
          by (metis-node-eq-unary AbsNode)
          then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr UnaryAbs xe2) ∧ UnaryExpr
UnaryAbs xe1 ≥ UnaryExpr UnaryAbs xe2
          by (metis AbsNode.prems l mono-unary rep.AbsNode)
        then show ?thesis
          by meson
      qed
    next
      case (NotNode n x xe1)
      have k: g1 ⊢ n ≃ UnaryExpr UnaryNot xe1 using f NotNode
        by (simp add: NotNode.hyps(2) rep.NotNode)
      obtain xn where l: kind g1 n = NotNode xn
        using NotNode.hyps(1) by blast
      then have m: g1 ⊢ xn ≃ xe1
        using NotNode.hyps(1) NotNode.hyps(2) by fastforce
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1' using c m repDet by simp
        then have ev: g2 ⊢ n ≃ UnaryExpr UnaryNot e2' using NotNode.hyps(1)
l m n
          using NotNode.prems True d rep.NotNode by simp
        then have r: UnaryExpr UnaryNot e1' ≥ UnaryExpr UnaryNot e2'
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using NotNode
          using False i b l not-excluded-keep-type singletonD no-encoding
          by (metis-node-eq-unary NotNode)
          then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr UnaryNot xe2) ∧ UnaryExpr
UnaryNot xe1 ≥ UnaryExpr UnaryNot xe2
          by (metis NotNode.prems l mono-unary rep.NotNode)
        then show ?thesis
          by meson
      qed
    next
      case (NegateNode n x xe1)
      have k: g1 ⊢ n ≃ UnaryExpr UnaryNeg xe1 using f NegateNode
        by (simp add: NegateNode.hyps(2) rep.NegateNode)
      obtain xn where l: kind g1 n = NegateNode xn
        using NegateNode.hyps(1) by blast
      then have m: g1 ⊢ xn ≃ xe1
        using NegateNode.hyps(1) NegateNode.hyps(2) by fastforce
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1' using c m repDet by simp
      then have ev: g2 ⊢ n ≃ UnaryExpr UnaryNeg e2' using NegateNode.hyps(1) l m n
          using NegateNode.prems True d rep.NegateNode by simp
        then have r: UnaryExpr UnaryNeg e1' ≥ UnaryExpr UnaryNeg e2'
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using NegateNode
          using False i 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 f LogicNegationNode
        by (simp add: LogicNegationNode.hyps(2) rep.LogicNegationNode)
      obtain xn where l: kind g1 n = LogicNegationNode xn
        using LogicNegationNode.hyps(1) by blast
      then have m: g1 ⊢ xn ≃ xe1
        using LogicNegationNode.hyps(1) LogicNegationNode.hyps(2) by fastforce
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1' using c m repDet by simp
        then have ev: g2 ⊢ n ≃ UnaryExpr UnaryLogicNegation e2' using LogicNegationNode.hyps(1) l m n
          using LogicNegationNode.prems True d rep.LogicNegationNode by simp
        then have r: UnaryExpr UnaryLogicNegation e1' ≥ UnaryExpr UnaryLogicNegation e2'
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using LogicNegationNode
          using False i 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 f AddNode
        by (simp add: AddNode.hyps(2) rep.AddNode)
      obtain xn yn where l: kind g1 n = AddNode xn yn
        using AddNode.hyps(1) by blast
      then have mx: g1 ⊢ xn ≃ xe1
        using AddNode.hyps(1) AddNode.hyps(2) by fastforce
      from l have my: g1 ⊢ yn ≃ ye1
        using AddNode.hyps(1) AddNode.hyps(3) by fastforce
      then show ?case
      proof -
        have g1 ⊢ xn ≃ xe1 using mx by simp
        have g1 ⊢ yn ≃ ye1 using my by simp
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using AddNode
          using 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
          using 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 f MulNode
```

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

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

```
        then show ?thesis
          by meson
      qed
    next
      case (NarrowNode n inputBits resultBits x xe1)
      have k: g1 ⊢ n ≃ UnaryExpr (UnaryNarrow inputBits resultBits) xe1 using f NarrowNode
        by (simp add: NarrowNode.hyps(2) rep.NarrowNode)
      obtain xn where l: kind g1 n = NarrowNode inputBits resultBits xn
        using NarrowNode.hyps(1) by blast
      then have m: g1 ⊢ xn ≃ xe1
        using NarrowNode.hyps(1) NarrowNode.hyps(2)
        by auto
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1' using c m repDet by simp
        then have ev: g2 ⊢ n ≃ UnaryExpr (UnaryNarrow inputBits resultBits) e2' using NarrowNode.hyps(1) l m n
          using NarrowNode.prems True d rep.NarrowNode by simp
        then have r: UnaryExpr (UnaryNarrow inputBits resultBits) e1' ≥ UnaryExpr (UnaryNarrow inputBits resultBits) e2'
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using NarrowNode
          using 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 f SignExtendNode
        by (simp add: SignExtendNode.hyps(2) rep.SignExtendNode)
      obtain xn where l: kind g1 n = SignExtendNode inputBits resultBits xn
        using SignExtendNode.hyps(1) by blast
      then have m: g1 ⊢ xn ≃ xe1
        using SignExtendNode.hyps(1) SignExtendNode.hyps(2)
        by auto
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1' using c m repDet by simp
      then have ev: g2 ⊢ n ≃ UnaryExpr (UnarySignExtend inputBits resultBits) e2' using SignExtendNode.hyps(1) l m n
          using SignExtendNode.prems True d rep.SignExtendNode by simp
          then have r: UnaryExpr (UnarySignExtend inputBits resultBits) e1' ≥ UnaryExpr (UnarySignExtend inputBits resultBits) e2'
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using SignExtendNode
        using 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 f ZeroExtendNode
        by (simp add: ZeroExtendNode.hyps(2) rep.ZeroExtendNode)
      obtain xn where l: kind g1 n = ZeroExtendNode inputBits resultBits xn
        using ZeroExtendNode.hyps(1) by blast
      then have m: g1 ⊢ xn ≃ xe1
        using ZeroExtendNode.hyps(1) ZeroExtendNode.hyps(2)
        by auto
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1' using c m repDet by simp
      then have ev: g2 ⊢ n ≃ UnaryExpr (UnaryZeroExtend inputBits resultBits) e2' using ZeroExtendNode.hyps(1) l m n
          using ZeroExtendNode.prems True d rep.ZeroExtendNode by simp
          then have r: UnaryExpr (UnaryZeroExtend inputBits resultBits) e1' ≥ UnaryExpr (UnaryZeroExtend inputBits resultBits) e2'
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
```

```
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using ZeroExtendNode
        using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
          by (metis-node-eq-ternary ZeroExtendNode)
       then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr (UnaryZeroExtend inputBits 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 (RefNode n')
      then show ?case
         by (metis a b c d no-encoding not-excluded-keep-type rep.RefNode repDet singletonD)
    qed
  qed
qed

lemma graph-semantics-preservation-subscript:
  assumes a: e_1' ≥ e_2'
  assumes b: ({n} ⊴ as-set g_1) ⊆ as-set g_2
  assumes c: g_1 ⊢ n ≃ e_1'
  assumes d: g_2 ⊢ n ≃ e_2'
  shows graph-refinement g_1 g_2
  using graph-semantics-preservation assms by simp


lemma tree-to-graph-rewriting:
  e_1 ≥ e_2
  ∧ (g_1 ⊢ n ≃ e_1) ∧ maximal-sharing g_1
  ∧ ({n} ⊴ as-set g_1) ⊆ as-set g_2
  ∧ (g_2 ⊢ n ≃ e_2) ∧ maximal-sharing g_2
  ⟹ graph-refinement g_1 g_2
  using graph-semantics-preservation
  by auto

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


lemma eval-contains-id[simp]: g1 ⊢ n ≃ e ⟹ n ∈ ids g1
  using no-encoding by blast

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

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

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



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

```
      apply (solve-subset-eval as-set: assms(1) eval: IntegerLessThanNode)
     apply (solve-subset-eval as-set: assms(1) eval: NarrowNode)
    apply (solve-subset-eval as-set: assms(1) eval: SignExtendNode)
   apply (solve-subset-eval as-set: assms(1) eval: ZeroExtendNode)
  apply (solve-subset-eval as-set: assms(1) eval: LeafNode)
 by (solve-subset-eval as-set: assms(1) eval: RefNode)

lemma subset-refines:
  assumes as-set g1 ⊆ as-set g2
  shows graph-refinement g1 g2
proof −
  have ids g1 ⊆ ids g2 using assms unfolding as-set-def
    by blast
  then show ?thesis unfolding graph-refinement-def apply rule
    apply (rule allI) apply (rule impI) apply (rule allI) apply (rule impI)
    unfolding graph-represents-expression-def
    proof −
      fix n e1
      assume 1:n ∈ ids g1
      assume 2:g1 ⊢ n ≃ e1

      show ∃ e2. (g2 ⊢ n ≃ e2) ∧ e1 ≥ e2
        using assms 1 2 using subset-implies-evals
        by (meson equal-refines)
    qed
  qed

lemma graph-construction:
  e₁ ≥ e₂
  ∧ as-set g₁ ⊆ as-set g₂
  ∧ (g₂ ⊢ n ≃ e₂)
  ⟹ (g₂ ⊢ n ⊴ e₁) ∧ graph-refinement g₁ g₂
  using subset-refines
  by (meson encodeeval-def graph-represents-expression-def le-expr-def)
```

3.8.4 Term Graph Reconstruction

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

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

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

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

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

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

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

lemma graph-unchanged-rep-unchanged:
  assumes ∀ n ∈ ids g. kind g n = kind g′ n
  assumes ∀ n ∈ ids g. stamp g n = stamp g′ n
  shows (g ⊢ n ≃ e) ⟶ (g′ ⊢ n ≃ e)
  apply (rule impI) subgoal premises e using e assms
    apply (induction n e)
                         apply (metis no-encoding rep.ConstantNode)
                        apply (metis no-encoding rep.ParameterNode)
                       apply (metis no-encoding rep.ConditionalNode)
                      apply (metis no-encoding rep.AbsNode)
                     apply (metis no-encoding rep.NotNode)
                    apply (metis no-encoding rep.NegateNode)
                    apply (metis no-encoding rep.LogicNegationNode)
                   apply (metis no-encoding rep.AddNode)
                  apply (metis no-encoding rep.MulNode)
                 apply (metis no-encoding rep.SubNode)
                apply (metis no-encoding rep.AndNode)
               apply (metis no-encoding rep.OrNode)
                apply (metis no-encoding rep.XorNode)
               apply (metis no-encoding rep.ShortCircuitOrNode)
              apply (metis no-encoding rep.LeftShiftNode)
             apply (metis no-encoding rep.RightShiftNode)
            apply (metis no-encoding rep.UnsignedRightShiftNode)
           apply (metis no-encoding rep.IntegerBelowNode)
          apply (metis no-encoding rep.IntegerEqualsNode)
         apply (metis no-encoding rep.IntegerLessThanNode)
        apply (metis no-encoding rep.NarrowNode)
       apply (metis no-encoding rep.SignExtendNode)
      apply (metis no-encoding rep.ZeroExtendNode)
     apply (metis no-encoding rep.LeafNode)
    by (metis no-encoding rep.RefNode)
  done

lemma fresh-node-subset:
  assumes n ∉ ids g
  assumes g′ = add-node n (k, s) g
  shows as-set g ⊆ as-set g′
  using assms
  by (smt (verit, del-insts) Collect-mono-iff Diff-idemp Diff-insert-absorb add-changed
as-set-def disjoint-change unchanged.simps)

lemma unrep-subset:
  assumes (g ⊕ e ⇝ (g′, n))
  shows as-set g ⊆ as-set g′
  using assms proof (induction g e (g′, n) arbitrary: g′ n)
  case (ConstantNodeSame g c n)
  then show ?case by blast
next
  case (ConstantNodeNew g c n g′)
  then show ?case using fresh-ids fresh-node-subset
    by presburger
next
  case (ParameterNodeSame g i s n)
  then show ?case by blast
next
  case (ParameterNodeNew g i s n g′)
  then show ?case using fresh-ids fresh-node-subset
    by presburger
next
  case (ConditionalNodeSame g ce g2 c te g3 t fe g4 f s′ n)
  then show ?case by blast
```

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

lemma fresh-node-preserves-other-nodes:
  assumes n' = get-fresh-id g
  assumes g' = add-node n' (k, s) g
  shows ∀ n ∈ ids g . (g ⊢ n ≃ e) ⟶ (g' ⊢ n ≃ e)
  using assms
 by (smt (verit, ccfv-SIG) Diff-idemp Diff-insert-absorb add-changed disjoint-change fresh-ids graph-unchanged-rep-unchanged unchanged.elims(2))

lemma found-node-preserves-other-nodes:
  assumes find-node-and-stamp g (k, s) = Some n
  shows ∀ n ∈ ids g. (g ⊢ n ≃ e) ⟷ (g ⊢ n ≃ e)
  using assms
  by blast

lemma unrep-ids-subset[simp]:
  assumes g ⊕ e ⇝ (g', n)
  shows ids g ⊆ ids g'
  using assms unrep-subset
  by (meson graph-refinement-def subset-refines)

lemma unrep-unchanged:
  assumes g ⊕ e ⇝ (g', n)
  shows ∀ n ∈ ids g . ∀ e. (g ⊢ n ≃ e) ⟶ (g' ⊢ n ≃ e)
  using assms unrep-subset fresh-node-preserves-other-nodes
  by (meson subset-implies-evals)
```

```
theorem term-graph-reconstruction:
  g ⊕ e ⇝ (g′, n) ⟹ (g′ ⊢ n ≃ e) ∧ as-set g ⊆ as-set g′
  subgoal premises e apply (rule conjI) defer
    using e unrep-subset apply blast using e
  proof (induction g e (g′, n) arbitrary: g′ n)
    case (ConstantNodeSame g′ c n)
    then have kind g′ n = ConstantNode c
      using find-exists-kind local.ConstantNodeSame by blast
    then show ?case using ConstantNode by blast
  next
    case (ConstantNodeNew g c)
    then show ?case
      using ConstantNode IRNode.distinct(683) add-node-lookup by presburger
  next
    case (ParameterNodeSame i s)
    then show ?case
      by (metis ParameterNode find-exists-kind find-exists-stamp)
  next
    case (ParameterNodeNew g i s)
    then show ?case
      by (metis IRNode.distinct(2447) ParameterNode add-node-lookup)
  next
    case (ConditionalNodeSame g ce g2 c te g3 t fe g4 f s′ n)
    then have k: kind g4 n = ConditionalNode c t f
      using find-exists-kind by blast
    have c: g4 ⊢ c ≃ ce using local.ConditionalNodeSame unrep-unchanged
      using no-encoding by blast
    have t: g4 ⊢ t ≃ te using local.ConditionalNodeSame unrep-unchanged
      using no-encoding by blast
    have f: g4 ⊢ f ≃ fe using local.ConditionalNodeSame unrep-unchanged
      using no-encoding by blast
    then show ?case using c t f
      using ConditionalNode k by blast
  next
    case (ConditionalNodeNew g ce g2 c te g3 t fe g4 f s′ n g′)
    moreover have ConditionalNode c t f ≠ NoNode
      using unary-node.elims by blast
    ultimately have k: kind g′ n = ConditionalNode c t f
      using find-new-kind local.ConditionalNodeNew
      by presburger
    then have c: g′ ⊢ c ≃ ce using local.ConditionalNodeNew unrep-unchanged
      using no-encoding
      by (metis ConditionalNodeNew.hyps(9) fresh-node-preserves-other-nodes)
    then have t: g′ ⊢ t ≃ te using local.ConditionalNodeNew unrep-unchanged
      using no-encoding fresh-node-preserves-other-nodes
      by metis
    then have f: g′ ⊢ f ≃ fe using local.ConditionalNodeNew unrep-unchanged
      using no-encoding fresh-node-preserves-other-nodes
      by metis
    then show ?case using c t f
      using ConditionalNode k by blast
  next
    case (UnaryNodeSame g xe g′ x s′ op n)
    then have k: kind g′ n = unary-node op x
      using find-exists-kind local.UnaryNodeSame by blast
    then have g′ ⊢ x ≃ xe using local.UnaryNodeSame by blast
    then show ?case using k
      apply (cases op)
      using AbsNode unary-node.simps(1) apply presburger
      using NegateNode unary-node.simps(3) apply presburger
      using NotNode unary-node.simps(2) apply presburger
      using LogicNegationNode unary-node.simps(4) apply presburger
      using NarrowNode unary-node.simps(5) apply presburger
      using SignExtendNode unary-node.simps(6) apply presburger
      using ZeroExtendNode unary-node.simps(7) by presburger
  next
    case (UnaryNodeNew g xe g2 x s′ op n g′)
    moreover have unary-node op x ≠ NoNode
      using unary-node.elims by blast
    ultimately have k: kind g′ n = unary-node op x
      using find-new-kind local.UnaryNodeNew
      by presburger
    have x ∈ ids g2 using local.UnaryNodeNew
      using eval-contains-id by blast
    then have x ≠ n using local.UnaryNodeNew(5) fresh-ids by blast
    have g′ ⊢ x ≃ xe using local.UnaryNodeNew fresh-node-preserves-other-nodes
      using ‹x ∈ ids g2› by blast
    then show ?case using k
      apply (cases op)
      using AbsNode unary-node.simps(1) apply presburger
      using NegateNode unary-node.simps(3) apply presburger
      using NotNode unary-node.simps(2) apply presburger
      using LogicNegationNode unary-node.simps(4) apply presburger
      using NarrowNode unary-node.simps(5) apply presburger
      using SignExtendNode unary-node.simps(6) apply presburger
      using ZeroExtendNode unary-node.simps(7) by presburger
  next
    case (BinaryNodeSame g xe g2 x ye g3 y s′ op n)
    then have k: kind g3 n = bin-node op x y
      using find-exists-kind by blast
    have x: g3 ⊢ x ≃ xe using local.BinaryNodeSame unrep-unchanged
      using no-encoding by blast
    have y: g3 ⊢ y ≃ ye using local.BinaryNodeSame unrep-unchanged
      using no-encoding by blast
    then show ?case using x y k apply (cases op)
      using AddNode bin-node.simps(1) apply presburger
      using MulNode bin-node.simps(2) apply presburger
      using SubNode bin-node.simps(3) apply presburger
```

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

lemma ref-refinement:
  assumes g ⊢ n ≃ e1
  assumes kind g n' = RefNode n
  shows g ⊢ n' ⊴ e1
```

using *assms RefNode*
by (*meson equal-refines graph-represents-expression-def*)

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

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

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

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

lemma *ids-finite*: *finite* (*ids* g)
using *Rep-IRGraph ids.rep-eq* **by** *simp*

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

lemma *find-none*:
assumes *find-node-and-stamp* $g\ (k, s) =$ *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* $g\ n = s))$
using *assms* **unfolding** *find-node-and-stamp.simps* **using** *find-None-iff unwrap-sorted*
by (*metis* (*mono-tags*, *lifting*))
then show *?thesis*
by *blast*
qed

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

3.8.5 Data-flow Tree to Subgraph Preserves Maximal Sharing

```
lemma same-kind-stamp-encodes-equal:
  assumes kind g n = kind g n′
  assumes stamp g n = stamp g n′
  assumes ¬(is-preevaluated (kind g n))
  shows ∀ e. (g ⊢ n ≃ e) ⟶ (g ⊢ n′ ≃ e)
  apply (rule allI)
  subgoal for e
    apply (rule impI)
    subgoal premises eval using eval assms
      apply (induction e)
    using ConstantNode apply presburger
    using ParameterNode apply presburger
                       apply (metis ConditionalNode)
                       apply (metis AbsNode)
                      apply (metis NotNode)
                     apply (metis NegateNode)
                    apply (metis LogicNegationNode)
                   apply (metis AddNode)
                  apply (metis MulNode)
                 apply (metis SubNode)
                apply (metis AndNode)
               apply (metis OrNode)
               apply (metis XorNode)
               apply (metis ShortCircuitOrNode)
             apply (metis LeftShiftNode)
            apply (metis RightShiftNode)
           apply (metis UnsignedRightShiftNode)
          apply (metis IntegerBelowNode)
         apply (metis IntegerEqualsNode)
        apply (metis IntegerLessThanNode)
```

```
       apply (metis NarrowNode)
      apply (metis SignExtendNode)
     apply (metis ZeroExtendNode)
    defer
     apply (metis RefNode)
    by blast
    done
  done

lemma new-node-not-present:
  assumes find-node-and-stamp g (node, s) = None
  assumes n = get-fresh-id g
  assumes g′ = add-node n (node, s) g
  shows ∀ n′ ∈ true-ids g. (∀ e. ((g ⊢ n ≃ e) ∧ (g ⊢ n′ ≃ e)) ⟶ n = n′)
  using assms
  using encode-in-ids fresh-ids by blast

lemma true-ids-def:
  true-ids g = {n ∈ ids g. ¬(is-RefNode (kind g n)) ∧ ((kind g n) ≠ NoNode)}
  unfolding true-ids-def ids-def
  using ids-def is-RefNode-def by fastforce

lemma add-node-some-node-def:
  assumes k ≠ NoNode
  assumes g′ = add-node nid (k, s) g
  shows g′ = Abs-IRGraph ((Rep-IRGraph g)(nid ↦ (k, s)))
  using assms
  by (metis Rep-IRGraph-inverse add-node.rep-eq fst-conv)

lemma ids-add-update-v1:
  assumes g′ = add-node nid (k, s) g
  assumes k ≠ NoNode
  shows dom (Rep-IRGraph g′) = dom (Rep-IRGraph g) ∪ {nid}
  using assms ids.rep-eq add-node-some-node-def
  by (simp add: add-node.rep-eq)

lemma ids-add-update-v2:
  assumes g′ = add-node nid (k, s) g
  assumes k ≠ NoNode
  shows nid ∈ ids g′
  using assms
  using find-new-kind ids-some by presburger

lemma add-node-ids-subset:
  assumes n ∈ ids g
  assumes g′ = add-node n node g
  shows ids g′ = ids g ∪ {n}
  using assms unfolding add-node-def
  apply (cases fst node = NoNode)
```

using *ids.rep-eq replace-node.rep-eq replace-node-def* **apply** *auto*[*1*]
unfolding *ids-def*
by (*smt* (*verit*, *best*) *Collect-cong Un-insert-right dom-fun-upd fst-conv fun-upd-apply ids.rep-eq ids-def insert-absorb mem-Collect-eq option.inject option.simps*(*3*) *replace-node.rep-eq replace-node-def sup-bot.right-neutral*)

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

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

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

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

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

lemma *ids-add-update*:
assumes $k \neq \textit{NoNode}$
assumes $n \notin \textit{ids}\ g$
assumes $g' = \textit{add-node}\ n\ (k,\ s)\ g$
shows $\textit{ids}\ g' = \textit{ids}\ g \cup \{n\}$
using *assms* **apply** (*subst assms*(*3*)) **using** *add-node-set-eq as-set-ids*
by (*smt* (*verit*, *del-insts*) *Collect-cong Diff-idemp Diff-insert-absorb Un-commute add-node.rep-eq add-node-def ids.rep-eq ids-add-update-v1 ids-add-update-v2 insertE*

insert-Collect insert-is-Un map-upd-Some-unfold mem-Collect-eq replace-node-def replace-node-unchanged)

lemma *true-ids-add-update*:
assumes $k \neq$ *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* $\cup \{n\}$
using *assms* **using** *true-ids ids-add-update*
by (*smt* (*z3*) *Collect-cong Diff-iff Diff-insert-absorb Un-commute add-node-def find-new-kind insert-Diff-if insert-is-Un mem-Collect-eq replace-node-def replace-node-unchanged*)

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

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* **unfolding** *as-set-def domain-subtraction-def*
by *blast*
then show *?thesis* **using** *existed* **by** *simp*
next
case *False*
then have *kind-eq*: $\forall\ n'$. $n' \notin$ *new* $\longrightarrow$ *kind g* $n' =$ *kind* g' n'
— can be more general than *stamp_eq* because NoNode default is equal
using *unchanged not-excluded-keep-type*
by (*smt* (*z3*) *case-prodE domain-subtraction-def ids-some mem-Collect-eq subsetI*)
from *False* **have** *stamp-eq*: $\forall\ n' \in$ *ids* g' . $n' \notin$ *new* $\longrightarrow$ *stamp g* $n' =$ *stamp* g' n'
using *unchanged not-excluded-keep-type*
by (*metis equalityE*)
show *?thesis* **using** *assms*(*4*) *kind-eq stamp-eq False*
proof (*induction n e rule*: *rep.induct*)
case (*ConstantNode n c*)
then show *?case*
using *rep.ConstantNode kind-eq* **by** *presburger*

```
  next
    case (ParameterNode n i s)
    then show ?case
      using rep.ParameterNode
      by (metis no-encoding)
  next
    case (ConditionalNode n c t f ce te fe)
    have kind: kind g n = ConditionalNode c t f
       using ConditionalNode.hyps(1) ConditionalNode.prems(3) kind-eq by presburger
    then have isin: n ∈ ids g
      by simp
    have inputs: {c, t, f} = inputs g n
     using kind unfolding inputs.simps using inputs-of-ConditionalNode by simp
    have c ∈ ids g ∧ t ∈ ids g ∧ f ∈ ids g
      using closed unfolding wf-closed-def
      using isin inputs by blast
    then have c ∉ new ∧ t ∉ new ∧ f ∉ new
      using new-def unchanged by blast
    then show ?case using ConditionalNode apply simp
      using rep.ConditionalNode by presburger
  next
    case (AbsNode n x xe)
    then have kind: kind g n = AbsNode x
      by simp
    then have isin: n ∈ ids g
      by simp
    have inputs: {x} = inputs g n
      using kind unfolding inputs.simps by simp
    have x ∈ ids g
      using closed unfolding wf-closed-def
      using isin inputs by blast
    then have x ∉ new
      using new-def unchanged by blast
    then show ?case
      using AbsNode
      using rep.AbsNode by presburger
  next
    case (NotNode n x xe)
    then have kind: kind g n = NotNode x
      by simp
    then have isin: n ∈ ids g
      by simp
    have inputs: {x} = inputs g n
      using kind unfolding inputs.simps by simp
    have x ∈ ids g
      using closed unfolding wf-closed-def
      using isin inputs by blast
    then have x ∉ new
    using new-def unchanged by blast
  then show ?case using NotNode
    using rep.NotNode by presburger
next
  case (NegateNode n x xe)
  then have kind: kind g n = NegateNode x
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new
    using new-def unchanged by blast
  then show ?case using NegateNode
    using rep.NegateNode by presburger
next
  case (LogicNegationNode n x xe)
  then have kind: kind g n = LogicNegationNode x
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new
    using new-def unchanged by blast
  then show ?case using LogicNegationNode
    using rep.LogicNegationNode by presburger
next
  case (AddNode n x y xe ye)
  then have kind: kind g n = AddNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using AddNode
    using rep.AddNode by presburger
next
```

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

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

```
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using LeftShiftNode
    using rep.LeftShiftNode by presburger
next
  case (RightShiftNode n x y xe ye)
  then have kind: kind g n = RightShiftNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using RightShiftNode
    using rep.RightShiftNode by presburger
next
  case (UnsignedRightShiftNode n x y xe ye)
  then have kind: kind g n = UnsignedRightShiftNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using UnsignedRightShiftNode
    using rep.UnsignedRightShiftNode by presburger
next
  case (IntegerBelowNode n x y xe ye)
  then have kind: kind g n = IntegerBelowNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using IntegerBelowNode
    using rep.IntegerBelowNode by presburger
next
  case (IntegerEqualsNode n x y xe ye)
  then have kind: kind g n = IntegerEqualsNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using IntegerEqualsNode
    using rep.IntegerEqualsNode by presburger
next
  case (IntegerLessThanNode n x y xe ye)
  then have kind: kind g n = IntegerLessThanNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using IntegerLessThanNode
    using rep.IntegerLessThanNode by presburger
next
  case (NarrowNode n inputBits resultBits x xe)
  then have kind: kind g n = NarrowNode inputBits resultBits x
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new
    using new-def unchanged by blast
  then show ?case using NarrowNode
    using rep.NarrowNode by presburger
next
  case (SignExtendNode n inputBits resultBits x xe)
  then have kind: kind g n = SignExtendNode inputBits resultBits x
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new
    using new-def unchanged by blast
  then show ?case using SignExtendNode
    using rep.SignExtendNode by presburger
next
  case (ZeroExtendNode n inputBits resultBits x xe)
  then have kind: kind g n = ZeroExtendNode inputBits resultBits x
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new
    using new-def unchanged by blast
  then show ?case using ZeroExtendNode
    using rep.ZeroExtendNode by presburger
next
  case (LeafNode n s)
  then show ?case
    by (metis no-encoding rep.LeafNode)
next
  case (RefNode n n′ e)
  then have kind: kind g n = RefNode n′
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {n′} = inputs g n
    using kind unfolding inputs.simps by simp
  have n′ ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have n′ ∉ new
    using new-def unchanged by blast
  then show ?case
    using RefNode
    using rep.RefNode by presburger
qed
```

```
qed

lemma not-in-no-rep:
  n ∉ ids g ⟹ ∀ e. ¬(g ⊢ n ≃ e)
  using eval-contains-id by blast



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

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

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

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

lemma unrep-contains:
  assumes g ⊕ e ⇝ (g′, n)
  shows n ∈ ids g′
  using assms
  using not-in-no-rep term-graph-reconstruction by blast

lemma unrep-preserves-contains:
  assumes n ∈ ids g
  assumes g ⊕ e ⇝ (g′, n′)
  shows n ∈ ids g′
  using assms
  by (meson subsetD unrep-ids-subset)

lemma unrep-preserves-closure:
  assumes wf-closed g
  assumes g ⊕ e ⇝ (g′, n)
  shows wf-closed g′
  using assms(2,1) unfolding wf-closed-def
  proof (induction g e (g′, n) arbitrary: g′ n)
    case (ConstantNodeSame g c n)
    then show ?case
```

```
      by blast
  next
    case (ConstantNodeNew g c n g')
    then have dom: ids g' = ids g ∪ {n}
      by (meson IRNode.distinct(683) add-node-ids-subset ids-add-update)
    have k: kind g' n = ConstantNode c
      using ConstantNodeNew add-node-lookup by simp
    then have inp: {} = inputs g' n
      unfolding inputs.simps by simp
    from k have suc: {} = succ g' n
      unfolding succ.simps by simp
    have inputs g' n ⊆ ids g' ∧ succ g' n ⊆ ids g' ∧ kind g' n ≠ NoNode
      using inp suc k by simp
    then show ?case
    by (smt (verit) ConstantNodeNew.hyps(3) ConstantNodeNew.prems Un-insert-right
add-changed changeonly.elims(2) dom inputs.simps insert-iff singleton-iff subset-insertI
subset-trans succ.simps sup-bot-right)
  next
    case (ParameterNodeSame g i s n)
    then show ?case by blast
  next
    case (ParameterNodeNew g i s n g')
    then have dom: ids g' = ids g ∪ {n}
      using IRNode.distinct(2447) fresh-ids ids-add-update by presburger
    have k: kind g' n = ParameterNode i
      using ParameterNodeNew add-node-lookup by simp
    then have inp: {} = inputs g' n
      unfolding inputs.simps by simp
    from k have suc: {} = succ g' n
      unfolding succ.simps by simp
    have inputs g' n ⊆ ids g' ∧ succ g' n ⊆ ids g' ∧ kind g' n ≠ NoNode
      using k inp suc by simp
    then show ?case
    by (smt (verit) ParameterNodeNew.hyps(3) ParameterNodeNew.prems Un-insert-right
add-node-as-set dom inputs.elims insertE not-excluded-keep-type order-trans single-
tonD subset-insertI succ.elims sup-bot-right)
  next
    case (ConditionalNodeSame g ce g2 c te g3 t fe g4 f s' n)
    then show ?case by blast
  next
    case (ConditionalNodeNew g ce g2 c te g3 t fe g4 f s' n g')
    then have dom: ids g' = ids g4 ∪ {n}
      by (meson IRNode.distinct(591) add-node-ids-subset ids-add-update)
    have k: kind g' n = ConditionalNode c t f
      using ConditionalNodeNew add-node-lookup by simp
    then have inp: {c, t, f} = inputs g' n
      unfolding inputs.simps by simp
    from k have suc: {} = succ g' n
      unfolding succ.simps by simp
    have inputs g′ n ⊆ ids g′ ∧ succ g′ n ⊆ ids g′ ∧ kind g′ n ≠ NoNode
      using k inp suc unrep-contains unrep-preserves-contains
      using ConditionalNodeNew(1,3,5,10)
       by (smt (verit) IRNode.simps(643) Un-insert-right bot.extremum dom insert-absorb insert-subset subset-insertI sup-bot-right)
    then show ?case using dom
       by (smt (z3) ConditionalNodeNew.hyps(10) ConditionalNodeNew.hyps(2) ConditionalNodeNew.hyps(4) ConditionalNodeNew.hyps(6) ConditionalNodeNew.prems Diff-eq-empty-iff Diff-iff Un-insert-right Un-upper1 add-node-def inputs.simps insertE replace-node-def replace-node-unchanged subset-trans succ.simps sup-bot-right)
  next
    case (UnaryNodeSame g xe g2 x s′ op n)
    then show ?case by blast
  next
    case (UnaryNodeNew g xe g2 x s′ op n g′)
    then have dom: ids g′ = ids g2 ∪ {n}
      by (metis add-node-ids-subset add-node-lookup ids-add-update ids-some unrep.UnaryNodeNew unrep-contains)
    have k: kind g′ n = unary-node op x
      using UnaryNodeNew add-node-lookup
      by (metis fresh-ids ids-some)
    then have inp: {x} = inputs g′ n
      using unary-inputs by simp
    from k have suc: {} = succ g′ n
      using unary-succ by simp
    have inputs g′ n ⊆ ids g′ ∧ succ g′ n ⊆ ids g′ ∧ kind g′ n ≠ NoNode
      using k inp suc unrep-contains unrep-preserves-contains
      using UnaryNodeNew(1,6)
    by (metis Un-upper1 dom empty-subsetI ids-some insert-not-empty insert-subsetI not-in-g-inputs subset-iff)
    then show ?case
    by (smt (verit) Un-insert-right UnaryNodeNew.hyps(2) UnaryNodeNew.hyps(6) UnaryNodeNew.prems add-changed changeonly.elims(2) dom inputs.simps insert-iff singleton-iff subset-insertI subset-trans succ.simps sup-bot-right)
  next
    case (BinaryNodeSame g xe g2 x ye g3 y s′ op n)
    then show ?case by blast
  next
    case (BinaryNodeNew g xe g2 x ye g3 y s′ op n g′)
    then have dom: ids g′ = ids g3 ∪ {n}
      by (metis binary-inputs fresh-ids ids-add-update ids-some insert-not-empty not-in-g-inputs)
    have k: kind g′ n = bin-node op x y
      using BinaryNodeNew add-node-lookup
      by (metis fresh-ids ids-some)
    then have inp: {x, y} = inputs g′ n
      using binary-inputs by simp
    from k have suc: {} = succ g′ n
      using binary-succ by simp
    have inputs g′ n ⊆ ids g′ ∧ succ g′ n ⊆ ids g′ ∧ kind g′ n ≠ NoNode
      using k inp suc unrep-contains unrep-preserves-contains
      using BinaryNodeNew(1,3,6)
     by (metis Un-upper1 dom empty-subsetI ids-some insert-not-empty insert-subsetI
not-in-g-inputs subset-iff)
    then show ?case using dom BinaryNodeNew
      by (smt (verit, del-insts) Diff-eq-empty-iff Diff-iff Un-insert-right Un-upper1
add-node-def inputs.simps insertE replace-node-def replace-node-unchanged subset-trans
succ.simps sup-bot-right)
  next
    case (AllLeafNodes g n s)
    then show ?case
      by blast
  qed

inductive-cases ConstUnrepE: g ⊕ (ConstantExpr x) ⇝ (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 ⊢ (nid,m,h) → next) ⟹
   (∀ next′. ((g, p ⊢ (nid,m,h) → next′) ⟶ next = next′))
proof (induction rule: step.induct)
```

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

```
is-LoopEndNode-def is-RefNode-def
    by metis
  have notnew: ¬(is-NewInstanceNode (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps
   using IRNode.distinct-disc(1442) is-EndNode.simps(29) is-NewInstanceNode-def
    by (metis IRNode.distinct-disc(1901) is-EndNode.simps(32))
  have notload: ¬(is-LoadFieldNode (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps
    using is-LoadFieldNode-def
    by (metis IRNode.distinct-disc(1706) is-EndNode.simps(21))
  have notstore: ¬(is-StoreFieldNode (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps is-StoreFieldNode-def
    by (metis IRNode.distinct-disc(1926) is-EndNode.simps(44))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
   using EndNodes.hyps(1) is-AbstractEndNode.simps is-SignedDivNode-def is-SignedRemNode-def
   using IRNode.distinct-disc(1498) IRNode.distinct-disc(1500) is-IntegerDivRemNode.simps
is-EndNode.simps(36) is-EndNode.simps(37)
    by auto
  from notseq notif notref notnew notload notstore notdivrem
  show ?case using EndNodes repAllDet evalAllDet
   by (smt (z3) is-IfNode-def is-LoadFieldNode-def is-NewInstanceNode-def is-RefNode-def
is-StoreFieldNode-def is-SignedDivNode-def is-SignedRemNode-def Pair-inject is-IntegerDivRemNode.elims(3)
step.cases)
next
  case (NewInstanceNode nid f obj nxt h′ ref h m′ m)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notif: ¬(is-IfNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notref: ¬(is-RefNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notload: ¬(is-LoadFieldNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notstore: ¬(is-StoreFieldNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notdivrem:  ¬(is-IntegerDivRemNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  from notseq notend notif notref notload notstore notdivrem
  show ?case using NewInstanceNode step.cases
     by (smt (z3) IRNode.disc(1028) IRNode.disc(2270) IRNode.discI(11) IRN-
ode.distinct(2311) IRNode.distinct(2313) IRNode.inject(31) Pair-inject)
next
  case (LoadFieldNode nid f obj nxt m ref h v m′)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  have notdivrem:  ¬(is-IntegerDivRemNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using LoadFieldNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject Value.inject(2)
option.distinct(1) option.inject)
next
  case (StaticLoadFieldNode nid f nxt h v m′ m)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StaticLoadFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StaticLoadFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StaticLoadFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StaticLoadFieldNode step.cases
   by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject option.distinct(1))
next
  case (StoreFieldNode nid f newval uu obj nxt m val ref h′ h m′)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StoreFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StoreFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StoreFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StoreFieldNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject Value.inject(2)
option.distinct(1) option.inject)
next
  case (StaticStoreFieldNode nid f newval uv nxt m val h′ h m′)
  then have notseq: ¬(is-sequential-node (kind g nid))
```

```
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StaticStoreFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StaticStoreFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StaticStoreFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StoreFieldNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject StaticStoreFieldNode.hyps(1)
StaticStoreFieldNode.hyps(2) StaticStoreFieldNode.hyps(3)
StaticStoreFieldNode.hyps(4) StaticStoreFieldNode.hyps(5) option.distinct(1))
next
  case (SignedDivNode nid x y zero sb nxt m v1 v2 v m' h)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: SignedDivNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: SignedDivNode.hyps(1))
  from notseq notend
  show ?case using SignedDivNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1091) IRNode.distinct(1739) IRNode.distinct(2311)
IRNode.distinct(2601) IRNode.distinct(2605) IRNode.inject(40) Pair-inject)
next
  case (SignedRemNode nid x y zero sb nxt m v1 v2 v m' h)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: SignedRemNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: SignedRemNode.hyps(1))
  from notseq notend
  show ?case using SignedRemNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1093) IRNode.distinct(1741) IRNode.distinct(2313)
IRNode.distinct(2601) IRNode.distinct(2627) IRNode.inject(41) Pair-inject)
qed

lemma stepRefNode:
  ⟦kind g nid = RefNode nid'⟧ ⟹ g, p ⊢ (nid,m,h) → (nid',m,h)
  using SequentialNode
 by (metis IRNodes.successors-of-RefNode is-sequential-node.simps(7) nth-Cons-0)

lemma IfNodeStepCases:
  assumes kind g nid = IfNode cond tb fb
  assumes g ⊢ cond ≃ condE
  assumes [m, p] ⊢ condE ↦ v
  assumes g, p ⊢ (nid, m, h) → (nid', m, h)
  shows nid′ ∈ {tb, fb}
  using step.IfNode repDet stepDet assms
  by (metis insert-iff old.prod.inject)

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

lemma IfNodeCond:
  assumes kind g nid = IfNode cond tb fb
  assumes g, p ⊢ (nid, m, h) → (nid′, m, h)
  shows ∃ condE v. ((g ⊢ cond ≃ condE) ∧ ([m, p] ⊢ condE ↦ v))
  using assms(2,1) by (induct (nid,m,h) (nid′,m,h) rule: step.induct; auto)

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

end
```