Veriopt

February 8, 2022

Abstract

The Veriopt project aims to prove the optimization pass of the GraalVM compiler. The GraalVM compiler includes a sophisticated Intermediate Representation (IR) in the form of a sea-of-nodes based graph structure. We first define the IR graph structure in the Isabelle/HOL interactive theorem prover. We subsequently give the evaluation of the structure a semantics based on the current understanding of the purpose of each IR graph node. Optimization phases are then encoded including the static analysis passes required for an optimization. Each optimization phase is proved to be correct by proving that a bisimulation exists between the unoptimized and optimized graphs. The following document has been automatically generated from the Isabelle/HOL source to provide a very comprehensive definition of the semantics and optimizations introduced by the Veriopt project.

Contents

1 Runtime Values and Arithmetic

```
theory Values
  imports
    HOL−Library.Word
    HOL−Library.Signed-Division
    HOL−Library.Float
    HOL−Library.LaTeXsugar
begin
```

In order to properly implement the IR semantics we first introduce a type that represents runtime values. These runtime values represent the full range of primitive types currently allowed by our semantics, ranging from basic integer types to object references and arrays.

Note that Java supports 64, 32, 16, 8 signed ints, plus 1 bit (boolean) ints, but during calculations the smaller sizes are expanded to 32 bits, so here we model just 32 and 64 bit values.

An object reference is an option type where the *None* object reference points to the static fields. This is examined more closely in our definition of the heap.

```
type-synonym int64 = 64 word — long
type-synonym int32 = 32 word — int
type-synonym int16 = 16 word — short
type-synonym int8 = 8 word — char
type-synonym int1 = 1 word — boolean

type-synonym objref = nat option

datatype (discs-sels) Value =
  UndefVal |
  IntVal32 32 word |
  IntVal64 64 word |

  ObjRef objref |
  ObjStr string


fun wf-bool :: Value ⇒ bool where
  wf-bool (IntVal32 v) = (v = 0 ∨ v = 1) |
  wf-bool - = False

fun val-to-bool :: Value ⇒ bool where
  val-to-bool (IntVal32 val) = (if val = 0 then False else True) |
  val-to-bool (IntVal64 val) = (if val = 0 then False else True) |
  val-to-bool v = False

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

value sint(word-of-int (1) :: int1)

fun is-int-val :: Value ⇒ bool where
  is-int-val (IntVal32 v) = True |
  is-int-val (IntVal64 v) = True |
  is-int-val - = False
```

We need to introduce arithmetic operations which agree with the JVM.

Within the JVM, bytecode arithmetic operations are performed on 32 or 64 bit integers, unboxing where appropriate.

The following collection of intval functions correspond to the JVM arithmetic operations. We merge the 32 and 64 bit operations into a single function, even though the stamp of each IRNode tells us exactly what the bit widths will be. These merged functions know to make it easier to do the instantiation of Value as 'plus', etc. It might be worse for reasoning, because it could cause more case analysis, but this does not seem to be a problem in practice.

```
fun intval-add :: Value ⇒ Value ⇒ Value where
  intval-add (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1+v2)) |
  intval-add (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1+v2)) |
  intval-add - - = UndefVal

instantiation Value :: ab-semigroup-add
begin

definition plus-Value :: Value ⇒ Value ⇒ Value where
  plus-Value = intval-add

print-locale! ab-semigroup-add

instance proof
  fix a b c :: Value
  show a + b + c = a + (b + c)
    apply (simp add: plus-Value-def)
    apply (induction a; induction b; induction c; auto)
    done
  show a + b = b + a
    apply (simp add: plus-Value-def)
    apply (induction a; induction b; auto)
    done
qed
end


fun intval-sub :: Value ⇒ Value ⇒ Value where
  intval-sub (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1−v2)) |
```

intval-sub (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*v1*−*v2*)) |
intval-sub - - = *UndefVal*

instantiation *Value* :: *minus*
begin

definition *minus-Value* :: *Value* ⇒ *Value* ⇒ *Value* **where**
minus-Value = *intval-sub*

instance proof qed
end

fun *intval-mul* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-mul (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*v1*∗*v2*)) |
intval-mul (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*v1*∗*v2*)) |
intval-mul - - = *UndefVal*

instantiation *Value* :: *times*
begin

definition *times-Value* :: *Value* ⇒ *Value* ⇒ *Value* **where**
times-Value = *intval-mul*

instance proof qed
end

fun *intval-div* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-div (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*word-of-int*((*sint v1*) *sdiv* (*sint v2*)))) |
intval-div (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*word-of-int*((*sint v1*) *sdiv* (*sint v2*)))) |
intval-div - - = *UndefVal*

instantiation *Value* :: *divide*
begin

definition *divide-Value* :: *Value* ⇒ *Value* ⇒ *Value* **where**
divide-Value = *intval-div*

instance proof qed
end

fun *intval-mod* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-mod (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*word-of-int*((*sint v1*) *smod* (*sint v2*)))) |

intval-mod (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*word-of-int*((*sint v1*) *smod* (*sint v2*)))) |
intval-mod - - = *UndefVal*

instantiation *Value* :: *modulo*
begin

definition *modulo-Value* :: *Value* ⇒ *Value* ⇒ *Value* **where**
modulo-Value = *intval-mod*

instance proof qed
end

context
includes *bit-operations-syntax*
begin

fun *intval-and* :: *Value* ⇒ *Value* ⇒ *Value* (**infix** &&∗ *64*) **where**
intval-and (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*v1 AND v2*)) |
intval-and (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*v1 AND v2*)) |
intval-and - - = *UndefVal*

fun *intval-or* :: *Value* ⇒ *Value* ⇒ *Value* (**infix** ||∗ *59*) **where**
intval-or (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*v1 OR v2*)) |
intval-or (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*v1 OR v2*)) |
intval-or - - = *UndefVal*

fun *intval-xor* :: *Value* ⇒ *Value* ⇒ *Value* (**infix** ^∗ *59*) **where**
intval-xor (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*v1 XOR v2*)) |
intval-xor (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*v1 XOR v2*)) |
intval-xor - - = *UndefVal*

fun *intval-equals* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-equals (*IntVal32 v1*) (*IntVal32 v2*) = *bool-to-val* (*v1* = *v2*) |
intval-equals (*IntVal64 v1*) (*IntVal64 v2*) = *bool-to-val* (*v1* = *v2*) |
intval-equals - - = *UndefVal*

fun *intval-less-than* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-less-than (*IntVal32 v1*) (*IntVal32 v2*) = *bool-to-val* (*v1* <*s v2*) |
intval-less-than (*IntVal64 v1*) (*IntVal64 v2*) = *bool-to-val* (*v1* <*s v2*) |
intval-less-than - - = *UndefVal*

fun *intval-below* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-below (*IntVal32 v1*) (*IntVal32 v2*) = *bool-to-val* (*v1* < *v2*) |
intval-below (*IntVal64 v1*) (*IntVal64 v2*) = *bool-to-val* (*v1* < *v2*) |

```
  intval-below - - = UndefVal

fun intval-not :: Value ⇒ Value where
  intval-not (IntVal32 v) = (IntVal32 (NOT v)) |
  intval-not (IntVal64 v) = (IntVal64 (NOT v)) |
  intval-not - = UndefVal

fun intval-negate :: Value ⇒ Value where
  intval-negate (IntVal32 v) = IntVal32 (− v) |
  intval-negate (IntVal64 v) = IntVal64 (− v) |
  intval-negate - = UndefVal

fun intval-abs :: Value ⇒ Value where
  intval-abs (IntVal32 v) = (if (v) <s 0 then (IntVal32 (− v)) else (IntVal32 v)) |
  intval-abs (IntVal64 v) = (if (v) <s 0 then (IntVal64 (− v)) else (IntVal64 v)) |
  intval-abs - = UndefVal

fun intval-conditional :: Value ⇒ Value ⇒ Value ⇒ Value where
  intval-conditional cond tv fv = (if (val-to-bool cond) then tv else fv)

fun intval-logic-negation :: Value ⇒ Value where
  intval-logic-negation (IntVal32 v) = (if v = 0 then (IntVal32 1) else (IntVal32
0)) |
  intval-logic-negation (IntVal64 v) = (if v = 0 then (IntVal64 1) else (IntVal64
0)) |
  intval-logic-negation - = UndefVal



definition shiftl (infix << 75) where
  shiftl w n = (push-bit n) w

lemma shiftl-power[simp]: (x::('a::len) word) ∗ (2 ^ j) = x << j
  unfolding shiftl-def apply (induction j)
   apply simp unfolding funpow-Suc-right
  by (metis (no-types, opaque-lifting) push-bit-eq-mult)

lemma (x::('a::len) word) ∗ ((2 ^ j) + 1) = x << j + x
  by (simp add: distrib-left)

lemma (x::('a::len) word) ∗ ((2 ^ j) − 1) = x << j − x
  by (simp add: right-diff-distrib)

lemma (x::('a::len) word) ∗ ((2^j) + (2^k)) = x << j + x << k
  by (simp add: distrib-left)

lemma (x::('a::len) word) ∗ ((2^j) − (2^k)) = x << j − x << k
  by (simp add: right-diff-distrib)
```

definition *shiftr* (**infix** *>>> 75*) **where**
  shiftr w n = (drop-bit n) w

value (*255 :: 8 word*) *>>>* (*2 :: nat*)

definition *signed-shiftr :: 'a :: len word ⇒ nat ⇒ 'a :: len word* (**infix** *>> 75*) **where**
  signed-shiftr w n = word-of-int ((sint w) div (2 ^ n))

value (*128 :: 8 word*) *>> 2*

fun *intval-left-shift :: Value ⇒ Value ⇒ Value* **where**
  intval-left-shift (IntVal32 v1) (IntVal32 v2) = IntVal32 (v1 << unat (v2 AND 0x1f)) |
  intval-left-shift (IntVal64 v1) (IntVal64 v2) = IntVal64 (v1 << unat (v2 AND 0x3f)) |
  intval-left-shift - - = UndefVal

fun *intval-right-shift :: Value ⇒ Value ⇒ Value* **where**
  intval-right-shift (IntVal32 v1) (IntVal32 v2) = IntVal32 (v1 >> unat (v2 AND 0x1f)) |
  intval-right-shift (IntVal64 v1) (IntVal64 v2) = IntVal64 (v1 >> unat (v2 AND 0x3f)) |
  intval-right-shift - - = UndefVal

fun *intval-uright-shift :: Value ⇒ Value ⇒ Value* **where**
  intval-uright-shift (IntVal32 v1) (IntVal32 v2) = IntVal32 (v1 >>> unat (v2 AND 0x1f)) |
  intval-uright-shift (IntVal64 v1) (IntVal64 v2) = IntVal64 (v1 >>> unat (v2 AND 0x3f)) |
  intval-uright-shift - - = UndefVal

end

lemma *intval-add-sym*:
  shows *intval-add a b = intval-add b a*
  by (*induction a*; *induction b*; *auto*)

```
code-deps intval-add
code-thms intval-add
```

```
lemma intval-add (IntVal32 (2^31−1)) (IntVal32 (2^31−1)) = IntVal32 (−2)
  by eval
lemma intval-add (IntVal64 (2^31−1)) (IntVal64 (2^31−1)) = IntVal64 4294967294
  by eval

end
```

2 Nodes

2.1 Types of Nodes

```
theory IRNodes
  imports
    Values
begin
```

The GraalVM IR is represented using a graph data structure. Here we define the nodes that are contained within the graph. Each node represents a Node subclass in the GraalVM compiler, the node classes have annotated fields to indicate input and successor edges.

We represent these classes with each IRNode constructor explicitly labelling a reference to the node IDs that it stores as inputs and successors.

The inputs_of and successors_of functions partition those labelled references into input edges and successor edges of a node.

To identify each Node, we use a simple natural number index. Zero is always the start node in a graph. For human readability, within nodes we write INPUT (or special case thereof) instead of ID for input edges, and SUCC instead of ID for control-flow successor edges. Optional edges are handled as ”INPUT option” etc.

```
type-synonym ID = nat
type-synonym INPUT = ID
type-synonym INPUT-ASSOC = ID
type-synonym INPUT-STATE = ID
type-synonym INPUT-GUARD = ID
type-synonym INPUT-COND = ID
type-synonym INPUT-EXT = ID
type-synonym SUCC = ID
```

```
datatype (discs-sels) IRNode =
  AbsNode (ir-value: INPUT)
  | AddNode (ir-x: INPUT) (ir-y: INPUT)
```

| *AndNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)

| *BeginNode* (*ir-next*: *SUCC*)

| *BytecodeExceptionNode* (*ir-arguments*: *INPUT list*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *ConditionalNode* (*ir-condition*: *INPUT-COND*) (*ir-trueValue*: *INPUT*) (*ir-falseValue*: *INPUT*)

| *ConstantNode* (*ir-const*: *Value*)

| *DynamicNewArrayNode* (*ir-elementType*: *INPUT*) (*ir-length*: *INPUT*) (*ir-voidClass-opt*: *INPUT option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *EndNode*

| *ExceptionObjectNode* (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *FrameState* (*ir-monitorIds*: *INPUT-ASSOC list*) (*ir-outerFrameState-opt*: *INPUT-STATE option*) (*ir-values-opt*: *INPUT list option*) (*ir-virtualObjectMappings-opt*: *INPUT-STATE list option*)

| *IfNode* (*ir-condition*: *INPUT-COND*) (*ir-trueSuccessor*: *SUCC*) (*ir-falseSuccessor*: *SUCC*)

| *IntegerBelowNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)

| *IntegerEqualsNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)

| *IntegerLessThanNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)

| *InvokeNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *InvokeWithExceptionNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*) (*ir-exceptionEdge*: *SUCC*)

| *IsNullNode* (*ir-value*: *INPUT*)

| *KillingBeginNode* (*ir-next*: *SUCC*)

| *LeftShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)

| *LoadFieldNode* (*ir-nid*: *ID*) (*ir-field*: *string*) (*ir-object-opt*: *INPUT option*) (*ir-next*: *SUCC*)

| *LogicNegationNode* (*ir-value*: *INPUT-COND*)

| *LoopBeginNode* (*ir-ends*: *INPUT-ASSOC list*) (*ir-overflowGuard-opt*: *INPUT-GUARD option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *LoopEndNode* (*ir-loopBegin*: *INPUT-ASSOC*)

| *LoopExitNode* (*ir-loopBegin*: *INPUT-ASSOC*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *MergeNode* (*ir-ends*: *INPUT-ASSOC list*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *MethodCallTargetNode* (*ir-targetMethod*: *string*) (*ir-arguments*: *INPUT list*)

| *MulNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)

| *NarrowNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)

| *NegateNode* (*ir-value*: *INPUT*)

| *NewArrayNode* (*ir-length*: *INPUT*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *NewInstanceNode* (*ir-nid*: *ID*) (*ir-instanceClass*: *string*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *NotNode* (*ir-value*: *INPUT*)

| *OrNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)

| *ParameterNode* (*ir-index*: *nat*)
| *PiNode* (*ir-object*: *INPUT*) (*ir-guard-opt*: *INPUT-GUARD option*)
| *ReturnNode* (*ir-result-opt*: *INPUT option*) (*ir-memoryMap-opt*: *INPUT-EXT option*)
| *RightShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *ShortCircuitOrNode* (*ir-x*: *INPUT-COND*) (*ir-y*: *INPUT-COND*)
| *SignExtendNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)
| *SignedDivNode* (*ir-nid*: *ID*) (*ir-x*: *INPUT*) (*ir-y*: *INPUT*) (*ir-zeroCheck-opt*: *INPUT-GUARD option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *SignedRemNode* (*ir-nid*: *ID*) (*ir-x*: *INPUT*) (*ir-y*: *INPUT*) (*ir-zeroCheck-opt*: *INPUT-GUARD option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *StartNode* (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *StoreFieldNode* (*ir-nid*: *ID*) (*ir-field*: *string*) (*ir-value*: *INPUT*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-object-opt*: *INPUT option*) (*ir-next*: *SUCC*)
| *SubNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *UnsignedRightShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *UnwindNode* (*ir-exception*: *INPUT*)
| *ValuePhiNode* (*ir-nid*: *ID*) (*ir-values*: *INPUT list*) (*ir-merge*: *INPUT-ASSOC*)
| *ValueProxyNode* (*ir-value*: *INPUT*) (*ir-loopExit*: *INPUT-ASSOC*)
| *XorNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *ZeroExtendNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)
| *NoNode*

| *RefNode* (*ir-ref*:*ID*)

fun *opt-to-list* :: *'a option* ⇒ *'a list* **where**
opt-to-list None = [] |
opt-to-list (*Some v*) = [*v*]

fun *opt-list-to-list* :: *'a list option* ⇒ *'a list* **where**
opt-list-to-list None = [] |
opt-list-to-list (*Some x*) = *x*

The following functions, inputs_of and successors_of, are automatically generated from the GraalVM compiler. Their purpose is to partition the node edges into input or successor edges.

fun *inputs-of* :: *IRNode* ⇒ *ID list* **where**
inputs-of-AbsNode:
inputs-of (*AbsNode value*) = [*value*] |
inputs-of-AddNode:
inputs-of (*AddNode x y*) = [*x*, *y*] |
inputs-of-AndNode:
inputs-of (*AndNode x y*) = [*x*, *y*] |

inputs-of-BeginNode:

inputs-of (BeginNode next) = [] |

inputs-of-BytecodeExceptionNode:

inputs-of (BytecodeExceptionNode arguments stateAfter next) = arguments @ (opt-to-list stateAfter) |

inputs-of-ConditionalNode:

inputs-of (ConditionalNode condition trueValue falseValue) = [condition, trueValue, falseValue] |

inputs-of-ConstantNode:

inputs-of (ConstantNode const) = [] |

inputs-of-DynamicNewArrayNode:

inputs-of (DynamicNewArrayNode elementType length0 voidClass stateBefore next) = [elementType, length0] @ (opt-to-list voidClass) @ (opt-to-list stateBefore) |

inputs-of-EndNode:

inputs-of (EndNode) = [] |

inputs-of-ExceptionObjectNode:

inputs-of (ExceptionObjectNode stateAfter next) = (opt-to-list stateAfter) |

inputs-of-FrameState:

inputs-of (FrameState monitorIds outerFrameState values virtualObjectMappings) = monitorIds @ (opt-to-list outerFrameState) @ (opt-list-to-list values) @ (opt-list-to-list virtualObjectMappings) |

inputs-of-IfNode:

inputs-of (IfNode condition trueSuccessor falseSuccessor) = [condition] |

inputs-of-IntegerBelowNode:

inputs-of (IntegerBelowNode x y) = [x, y] |

inputs-of-IntegerEqualsNode:

inputs-of (IntegerEqualsNode x y) = [x, y] |

inputs-of-IntegerLessThanNode:

inputs-of (IntegerLessThanNode x y) = [x, y] |

inputs-of-InvokeNode:

inputs-of (InvokeNode nid0 callTarget classInit stateDuring stateAfter next) = callTarget # (opt-to-list classInit) @ (opt-to-list stateDuring) @ (opt-to-list stateAfter) |

inputs-of-InvokeWithExceptionNode:

inputs-of (InvokeWithExceptionNode nid0 callTarget classInit stateDuring stateAfter next exceptionEdge) = callTarget # (opt-to-list classInit) @ (opt-to-list stateDuring) @ (opt-to-list stateAfter) |

inputs-of-IsNullNode:

inputs-of (IsNullNode value) = [value] |

inputs-of-KillingBeginNode:

inputs-of (KillingBeginNode next) = [] |

inputs-of-LeftShiftNode:

inputs-of (LeftShiftNode x y) = [x, y] |

inputs-of-LoadFieldNode:

inputs-of (LoadFieldNode nid0 field object next) = (opt-to-list object) |

inputs-of-LogicNegationNode:

inputs-of (LogicNegationNode value) = [value] |

inputs-of-LoopBeginNode:

inputs-of (*LoopBeginNode ends overflowGuard stateAfter next*) = *ends* @ (*opt-to-list overflowGuard*) @ (*opt-to-list stateAfter*) |
inputs-of-LoopEndNode:
inputs-of (*LoopEndNode loopBegin*) = [*loopBegin*] |
inputs-of-LoopExitNode:
inputs-of (*LoopExitNode loopBegin stateAfter next*) = *loopBegin* # (*opt-to-list stateAfter*) |
inputs-of-MergeNode:
inputs-of (*MergeNode ends stateAfter next*) = *ends* @ (*opt-to-list stateAfter*) |
inputs-of-MethodCallTargetNode:
inputs-of (*MethodCallTargetNode targetMethod arguments*) = *arguments* |
inputs-of-MulNode:
inputs-of (*MulNode x y*) = [*x*, *y*] |
inputs-of-NarrowNode:
inputs-of (*NarrowNode inputBits resultBits value*) = [*value*] |
inputs-of-NegateNode:
inputs-of (*NegateNode value*) = [*value*] |
inputs-of-NewArrayNode:
inputs-of (*NewArrayNode length0 stateBefore next*) = *length0* # (*opt-to-list stateBefore*) |
inputs-of-NewInstanceNode:
inputs-of (*NewInstanceNode nid0 instanceClass stateBefore next*) = (*opt-to-list stateBefore*) |
inputs-of-NotNode:
inputs-of (*NotNode value*) = [*value*] |
inputs-of-OrNode:
inputs-of (*OrNode x y*) = [*x*, *y*] |
inputs-of-ParameterNode:
inputs-of (*ParameterNode index*) = [] |
inputs-of-PiNode:
inputs-of (*PiNode object guard*) = *object* # (*opt-to-list guard*) |
inputs-of-ReturnNode:
inputs-of (*ReturnNode result memoryMap*) = (*opt-to-list result*) @ (*opt-to-list memoryMap*) |
inputs-of-RightShiftNode:
inputs-of (*RightShiftNode x y*) = [*x*, *y*] |
inputs-of-ShortCircuitOrNode:
inputs-of (*ShortCircuitOrNode x y*) = [*x*, *y*] |
inputs-of-SignExtendNode:
inputs-of (*SignExtendNode inputBits resultBits value*) = [*value*] |
inputs-of-SignedDivNode:
inputs-of (*SignedDivNode nid0 x y zeroCheck stateBefore next*) = [*x*, *y*] @ (*opt-to-list zeroCheck*) @ (*opt-to-list stateBefore*) |
inputs-of-SignedRemNode:
inputs-of (*SignedRemNode nid0 x y zeroCheck stateBefore next*) = [*x*, *y*] @ (*opt-to-list zeroCheck*) @ (*opt-to-list stateBefore*) |
inputs-of-StartNode:
inputs-of (*StartNode stateAfter next*) = (*opt-to-list stateAfter*) |
inputs-of-StoreFieldNode:

inputs-of (*StoreFieldNode nid0 field value stateAfter object next*) = *value* # (*opt-to-list stateAfter*) @ (*opt-to-list object*) |
inputs-of-SubNode:
inputs-of (*SubNode x y*) = [*x*, *y*] |
inputs-of-UnsignedRightShiftNode:
inputs-of (*UnsignedRightShiftNode x y*) = [*x*, *y*] |
inputs-of-UnwindNode:
inputs-of (*UnwindNode exception*) = [*exception*] |
inputs-of-ValuePhiNode:
inputs-of (*ValuePhiNode nid0 values merge*) = *merge* # *values* |
inputs-of-ValueProxyNode:
inputs-of (*ValueProxyNode value loopExit*) = [*value*, *loopExit*] |
inputs-of-XorNode:
inputs-of (*XorNode x y*) = [*x*, *y*] |
inputs-of-ZeroExtendNode:
inputs-of (*ZeroExtendNode inputBits resultBits value*) = [*value*] |
inputs-of-NoNode: *inputs-of* (*NoNode*) = [] |

inputs-of-RefNode: *inputs-of* (*RefNode ref*) = [*ref*]

fun *successors-of* :: *IRNode* ⇒ *ID list* **where**
successors-of-AbsNode:
successors-of (*AbsNode value*) = [] |
successors-of-AddNode:
successors-of (*AddNode x y*) = [] |
successors-of-AndNode:
successors-of (*AndNode x y*) = [] |
successors-of-BeginNode:
successors-of (*BeginNode next*) = [*next*] |
successors-of-BytecodeExceptionNode:
successors-of (*BytecodeExceptionNode arguments stateAfter next*) = [*next*] |
successors-of-ConditionalNode:
successors-of (*ConditionalNode condition trueValue falseValue*) = [] |
successors-of-ConstantNode:
successors-of (*ConstantNode const*) = [] |
successors-of-DynamicNewArrayNode:
successors-of (*DynamicNewArrayNode elementType length0 voidClass stateBefore next*) = [*next*] |
successors-of-EndNode:
successors-of (*EndNode*) = [] |
successors-of-ExceptionObjectNode:
successors-of (*ExceptionObjectNode stateAfter next*) = [*next*] |
successors-of-FrameState:
successors-of (*FrameState monitorIds outerFrameState values virtualObjectMappings*) = [] |
successors-of-IfNode:
successors-of (*IfNode condition trueSuccessor falseSuccessor*) = [*trueSuccessor*,

falseSuccessor] |
successors-of-IntegerBelowNode:
successors-of (*IntegerBelowNode x y*) = [] |
successors-of-IntegerEqualsNode:
successors-of (*IntegerEqualsNode x y*) = [] |
successors-of-IntegerLessThanNode:
successors-of (*IntegerLessThanNode x y*) = [] |
successors-of-InvokeNode:
successors-of (*InvokeNode nid0 callTarget classInit stateDuring stateAfter next*) = [*next*] |
successors-of-InvokeWithExceptionNode:
successors-of (*InvokeWithExceptionNode nid0 callTarget classInit stateDuring stateAfter next exceptionEdge*) = [*next*, *exceptionEdge*] |
successors-of-IsNullNode:
successors-of (*IsNullNode value*) = [] |
successors-of-KillingBeginNode:
successors-of (*KillingBeginNode next*) = [*next*] |
successors-of-LeftShiftNode:
successors-of (*LeftShiftNode x y*) = [] |
successors-of-LoadFieldNode:
successors-of (*LoadFieldNode nid0 field object next*) = [*next*] |
successors-of-LogicNegationNode:
successors-of (*LogicNegationNode value*) = [] |
successors-of-LoopBeginNode:
successors-of (*LoopBeginNode ends overflowGuard stateAfter next*) = [*next*] |
successors-of-LoopEndNode:
successors-of (*LoopEndNode loopBegin*) = [] |
successors-of-LoopExitNode:
successors-of (*LoopExitNode loopBegin stateAfter next*) = [*next*] |
successors-of-MergeNode:
successors-of (*MergeNode ends stateAfter next*) = [*next*] |
successors-of-MethodCallTargetNode:
successors-of (*MethodCallTargetNode targetMethod arguments*) = [] |
successors-of-MulNode:
successors-of (*MulNode x y*) = [] |
successors-of-NarrowNode:
successors-of (*NarrowNode inputBits resultBits value*) = [] |
successors-of-NegateNode:
successors-of (*NegateNode value*) = [] |
successors-of-NewArrayNode:
successors-of (*NewArrayNode length0 stateBefore next*) = [*next*] |
successors-of-NewInstanceNode:
successors-of (*NewInstanceNode nid0 instanceClass stateBefore next*) = [*next*] |
successors-of-NotNode:
successors-of (*NotNode value*) = [] |
successors-of-OrNode:
successors-of (*OrNode x y*) = [] |
successors-of-ParameterNode:
successors-of (*ParameterNode index*) = [] |

successors-of-PiNode:
successors-of (*PiNode object guard*) = [] |
successors-of-ReturnNode:
successors-of (*ReturnNode result memoryMap*) = [] |
successors-of-RightShiftNode:
successors-of (*RightShiftNode x y*) = [] |
successors-of-ShortCircuitOrNode:
successors-of (*ShortCircuitOrNode x y*) = [] |
successors-of-SignExtendNode:
successors-of (*SignExtendNode inputBits resultBits value*) = [] |
successors-of-SignedDivNode:
successors-of (*SignedDivNode nid0 x y zeroCheck stateBefore next*) = [*next*] |
successors-of-SignedRemNode:
successors-of (*SignedRemNode nid0 x y zeroCheck stateBefore next*) = [*next*] |
successors-of-StartNode:
successors-of (*StartNode stateAfter next*) = [*next*] |
successors-of-StoreFieldNode:
successors-of (*StoreFieldNode nid0 field value stateAfter object next*) = [*next*] |
successors-of-SubNode:
successors-of (*SubNode x y*) = [] |
successors-of-UnsignedRightShiftNode:
successors-of (*UnsignedRightShiftNode x y*) = [] |
successors-of-UnwindNode:
successors-of (*UnwindNode exception*) = [] |
successors-of-ValuePhiNode:
successors-of (*ValuePhiNode nid0 values merge*) = [] |
successors-of-ValueProxyNode:
successors-of (*ValueProxyNode value loopExit*) = [] |
successors-of-XorNode:
successors-of (*XorNode x y*) = [] |
successors-of-ZeroExtendNode:
successors-of (*ZeroExtendNode inputBits resultBits value*) = [] |
successors-of-NoNode: *successors-of* (*NoNode*) = [] |

successors-of-RefNode: *successors-of* (*RefNode ref*) = [*ref*]

lemma *inputs-of* (*FrameState x* (*Some y*) (*Some z*) *None*) = *x* @ [*y*] @ *z*
unfolding *inputs-of-FrameState* **by** *simp*
lemma *successors-of* (*FrameState x* (*Some y*) (*Some z*) *None*) = []
unfolding *inputs-of-FrameState* **by** *simp*

lemma *inputs-of* (*IfNode c t f*) = [*c*]
unfolding *inputs-of-IfNode* **by** *simp*
lemma *successors-of* (*IfNode c t f*) = [*t*, *f*]
unfolding *successors-of-IfNode* **by** *simp*

```
lemma inputs-of (EndNode) = [] ∧ successors-of (EndNode) = []
  unfolding inputs-of-EndNode successors-of-EndNode by simp

end
```

2.2 Hierarchy of Nodes

```
theory IRNodeHierarchy
imports IRNodes
begin
```

It is helpful to introduce a node hierarchy into our formalization. Often the GraalVM compiler relies on explicit type checks to determine which operations to perform on a given node, we try to mimic the same functionality by using a suite of predicate functions over the IRNode class to determine inheritance.

As one would expect, the function is<ClassName>Type will be true if the node parameter is a subclass of the ClassName within the GraalVM compiler.

These functions have been automatically generated from the compiler.

```
fun is-EndNode :: IRNode ⇒ bool where
  is-EndNode EndNode = True |
  is-EndNode - = False


fun is-VirtualState :: IRNode ⇒ bool where
  is-VirtualState n = ((is-FrameState n))

fun is-BinaryArithmeticNode :: IRNode ⇒ bool where
  is-BinaryArithmeticNode n = ((is-AddNode n) ∨ (is-AndNode n) ∨ (is-MulNode
n) ∨ (is-OrNode n) ∨ (is-SubNode n) ∨ (is-XorNode n))

fun is-ShiftNode :: IRNode ⇒ bool where
 is-ShiftNode n = ((is-LeftShiftNode n) ∨ (is-RightShiftNode n) ∨ (is-UnsignedRightShiftNode
n))

fun is-BinaryNode :: IRNode ⇒ bool where
  is-BinaryNode n = ((is-BinaryArithmeticNode n) ∨ (is-ShiftNode n))

fun is-AbstractLocalNode :: IRNode ⇒ bool where
  is-AbstractLocalNode n = ((is-ParameterNode n))

fun is-IntegerConvertNode :: IRNode ⇒ bool where
   is-IntegerConvertNode n = ((is-NarrowNode n) ∨ (is-SignExtendNode n) ∨
(is-ZeroExtendNode n))

fun is-UnaryArithmeticNode :: IRNode ⇒ bool where
```

is-UnaryArithmeticNode n = ((*is-AbsNode n*) ∨ (*is-NegateNode n*) ∨ (*is-NotNode n*))

fun *is-UnaryNode* :: *IRNode* ⇒ *bool* **where**
is-UnaryNode n = ((*is-IntegerConvertNode n*) ∨ (*is-UnaryArithmeticNode n*))

fun *is-PhiNode* :: *IRNode* ⇒ *bool* **where**
is-PhiNode n = ((*is-ValuePhiNode n*))

fun *is-FloatingGuardedNode* :: *IRNode* ⇒ *bool* **where**
is-FloatingGuardedNode n = ((*is-PiNode n*))

fun *is-UnaryOpLogicNode* :: *IRNode* ⇒ *bool* **where**
is-UnaryOpLogicNode n = ((*is-IsNullNode n*))

fun *is-IntegerLowerThanNode* :: *IRNode* ⇒ *bool* **where**
is-IntegerLowerThanNode n = ((*is-IntegerBelowNode n*) ∨ (*is-IntegerLessThanNode n*))

fun *is-CompareNode* :: *IRNode* ⇒ *bool* **where**
is-CompareNode n = ((*is-IntegerEqualsNode n*) ∨ (*is-IntegerLowerThanNode n*))

fun *is-BinaryOpLogicNode* :: *IRNode* ⇒ *bool* **where**
is-BinaryOpLogicNode n = ((*is-CompareNode n*))

fun *is-LogicNode* :: *IRNode* ⇒ *bool* **where**
is-LogicNode n = ((*is-BinaryOpLogicNode n*) ∨ (*is-LogicNegationNode n*) ∨ (*is-ShortCircuitOrNode n*) ∨ (*is-UnaryOpLogicNode n*))

fun *is-ProxyNode* :: *IRNode* ⇒ *bool* **where**
is-ProxyNode n = ((*is-ValueProxyNode n*))

fun *is-FloatingNode* :: *IRNode* ⇒ *bool* **where**
is-FloatingNode n = ((*is-AbstractLocalNode n*) ∨ (*is-BinaryNode n*) ∨ (*is-ConditionalNode n*) ∨ (*is-ConstantNode n*) ∨ (*is-FloatingGuardedNode n*) ∨ (*is-LogicNode n*) ∨ (*is-PhiNode n*) ∨ (*is-ProxyNode n*) ∨ (*is-UnaryNode n*))

fun *is-AccessFieldNode* :: *IRNode* ⇒ *bool* **where**
is-AccessFieldNode n = ((*is-LoadFieldNode n*) ∨ (*is-StoreFieldNode n*))

fun *is-AbstractNewArrayNode* :: *IRNode* ⇒ *bool* **where**
is-AbstractNewArrayNode n = ((*is-DynamicNewArrayNode n*) ∨ (*is-NewArrayNode n*))

fun *is-AbstractNewObjectNode* :: *IRNode* ⇒ *bool* **where**
is-AbstractNewObjectNode n = ((*is-AbstractNewArrayNode n*) ∨ (*is-NewInstanceNode n*))

fun *is-IntegerDivRemNode* :: *IRNode* ⇒ *bool* **where**

is-IntegerDivRemNode n = ((*is-SignedDivNode n*) ∨ (*is-SignedRemNode n*))

fun *is-FixedBinaryNode* :: *IRNode* ⇒ *bool* **where**
is-FixedBinaryNode n = ((*is-IntegerDivRemNode n*))

fun *is-DeoptimizingFixedWithNextNode* :: *IRNode* ⇒ *bool* **where**
is-DeoptimizingFixedWithNextNode n = ((*is-AbstractNewObjectNode n*) ∨ (*is-FixedBinaryNode n*))

fun *is-AbstractMemoryCheckpoint* :: *IRNode* ⇒ *bool* **where**
is-AbstractMemoryCheckpoint n = ((*is-BytecodeExceptionNode n*) ∨ (*is-InvokeNode n*))

fun *is-AbstractStateSplit* :: *IRNode* ⇒ *bool* **where**
is-AbstractStateSplit n = ((*is-AbstractMemoryCheckpoint n*))

fun *is-AbstractMergeNode* :: *IRNode* ⇒ *bool* **where**
is-AbstractMergeNode n = ((*is-LoopBeginNode n*) ∨ (*is-MergeNode n*))

fun *is-BeginStateSplitNode* :: *IRNode* ⇒ *bool* **where**
is-BeginStateSplitNode n = ((*is-AbstractMergeNode n*) ∨ (*is-ExceptionObjectNode n*) ∨ (*is-LoopExitNode n*) ∨ (*is-StartNode n*))

fun *is-AbstractBeginNode* :: *IRNode* ⇒ *bool* **where**
is-AbstractBeginNode n = ((*is-BeginNode n*) ∨ (*is-BeginStateSplitNode n*) ∨ (*is-KillingBeginNode n*))

fun *is-FixedWithNextNode* :: *IRNode* ⇒ *bool* **where**
is-FixedWithNextNode n = ((*is-AbstractBeginNode n*) ∨ (*is-AbstractStateSplit n*) ∨ (*is-AccessFieldNode n*) ∨ (*is-DeoptimizingFixedWithNextNode n*))

fun *is-WithExceptionNode* :: *IRNode* ⇒ *bool* **where**
is-WithExceptionNode n = ((*is-InvokeWithExceptionNode n*))

fun *is-ControlSplitNode* :: *IRNode* ⇒ *bool* **where**
is-ControlSplitNode n = ((*is-IfNode n*) ∨ (*is-WithExceptionNode n*))

fun *is-ControlSinkNode* :: *IRNode* ⇒ *bool* **where**
is-ControlSinkNode n = ((*is-ReturnNode n*) ∨ (*is-UnwindNode n*))

fun *is-AbstractEndNode* :: *IRNode* ⇒ *bool* **where**
is-AbstractEndNode n = ((*is-EndNode n*) ∨ (*is-LoopEndNode n*))

fun *is-FixedNode* :: *IRNode* ⇒ *bool* **where**
is-FixedNode n = ((*is-AbstractEndNode n*) ∨ (*is-ControlSinkNode n*) ∨ (*is-ControlSplitNode n*) ∨ (*is-FixedWithNextNode n*))

fun *is-CallTargetNode* :: *IRNode* ⇒ *bool* **where**
is-CallTargetNode n = ((*is-MethodCallTargetNode n*))

fun *is-ValueNode* :: *IRNode* ⇒ *bool* **where**
is-ValueNode n = ((*is-CallTargetNode n*) ∨ (*is-FixedNode n*) ∨ (*is-FloatingNode n*))

fun *is-Node* :: *IRNode* ⇒ *bool* **where**
is-Node n = ((*is-ValueNode n*) ∨ (*is-VirtualState n*))

fun *is-MemoryKill* :: *IRNode* ⇒ *bool* **where**
is-MemoryKill n = ((*is-AbstractMemoryCheckpoint n*))

fun *is-NarrowableArithmeticNode* :: *IRNode* ⇒ *bool* **where**
is-NarrowableArithmeticNode n = ((*is-AbsNode n*) ∨ (*is-AddNode n*) ∨ (*is-AndNode n*) ∨ (*is-MulNode n*) ∨ (*is-NegateNode n*) ∨ (*is-NotNode n*) ∨ (*is-OrNode n*) ∨ (*is-ShiftNode n*) ∨ (*is-SubNode n*) ∨ (*is-XorNode n*))

fun *is-AnchoringNode* :: *IRNode* ⇒ *bool* **where**
is-AnchoringNode n = ((*is-AbstractBeginNode n*))

fun *is-DeoptBefore* :: *IRNode* ⇒ *bool* **where**
is-DeoptBefore n = ((*is-DeoptimizingFixedWithNextNode n*))

fun *is-IndirectCanonicalization* :: *IRNode* ⇒ *bool* **where**
is-IndirectCanonicalization n = ((*is-LogicNode n*))

fun *is-IterableNodeType* :: *IRNode* ⇒ *bool* **where**
is-IterableNodeType n = ((*is-AbstractBeginNode n*) ∨ (*is-AbstractMergeNode n*) ∨ (*is-FrameState n*) ∨ (*is-IfNode n*) ∨ (*is-IntegerDivRemNode n*) ∨ (*is-InvokeWithExceptionNode n*) ∨ (*is-LoopBeginNode n*) ∨ (*is-LoopExitNode n*) ∨ (*is-MethodCallTargetNode n*) ∨ (*is-ParameterNode n*) ∨ (*is-ReturnNode n*) ∨ (*is-ShortCircuitOrNode n*))

fun *is-Invoke* :: *IRNode* ⇒ *bool* **where**
is-Invoke n = ((*is-InvokeNode n*) ∨ (*is-InvokeWithExceptionNode n*))

fun *is-Proxy* :: *IRNode* ⇒ *bool* **where**
is-Proxy n = ((*is-ProxyNode n*))

fun *is-ValueProxy* :: *IRNode* ⇒ *bool* **where**
is-ValueProxy n = ((*is-PiNode n*) ∨ (*is-ValueProxyNode n*))

fun *is-ValueNodeInterface* :: *IRNode* ⇒ *bool* **where**
is-ValueNodeInterface n = ((*is-ValueNode n*))

fun *is-ArrayLengthProvider* :: *IRNode* ⇒ *bool* **where**
is-ArrayLengthProvider n = ((*is-AbstractNewArrayNode n*) ∨ (*is-ConstantNode n*))

fun *is-StampInverter* :: *IRNode* ⇒ *bool* **where**
is-StampInverter n = ((*is-IntegerConvertNode n*) ∨ (*is-NegateNode n*) ∨ (*is-NotNode*

n))

fun *is-GuardingNode* :: *IRNode* ⇒ *bool* **where**
is-GuardingNode n = ((*is-AbstractBeginNode n*))

fun *is-SingleMemoryKill* :: *IRNode* ⇒ *bool* **where**
is-SingleMemoryKill n = ((*is-BytecodeExceptionNode n*) ∨ (*is-ExceptionObjectNode n*) ∨ (*is-InvokeNode n*) ∨ (*is-InvokeWithExceptionNode n*) ∨ (*is-KillingBeginNode n*) ∨ (*is-StartNode n*))

fun *is-LIRLowerable* :: *IRNode* ⇒ *bool* **where**
is-LIRLowerable n = ((*is-AbstractBeginNode n*) ∨ (*is-AbstractEndNode n*) ∨ (*is-AbstractMergeNode n*) ∨ (*is-BinaryOpLogicNode n*) ∨ (*is-CallTargetNode n*) ∨ (*is-ConditionalNode n*) ∨ (*is-ConstantNode n*) ∨ (*is-IfNode n*) ∨ (*is-InvokeNode n*) ∨ (*is-InvokeWithExceptionNode n*) ∨ (*is-IsNullNode n*) ∨ (*is-LoopBeginNode n*) ∨ (*is-PiNode n*) ∨ (*is-ReturnNode n*) ∨ (*is-SignedDivNode n*) ∨ (*is-SignedRemNode n*) ∨ (*is-UnaryOpLogicNode n*) ∨ (*is-UnwindNode n*))

fun *is-GuardedNode* :: *IRNode* ⇒ *bool* **where**
is-GuardedNode n = ((*is-FloatingGuardedNode n*))

fun *is-ArithmeticLIRLowerable* :: *IRNode* ⇒ *bool* **where**
is-ArithmeticLIRLowerable n = ((*is-AbsNode n*) ∨ (*is-BinaryArithmeticNode n*) ∨ (*is-IntegerConvertNode n*) ∨ (*is-NotNode n*) ∨ (*is-ShiftNode n*) ∨ (*is-UnaryArithmeticNode n*))

fun *is-SwitchFoldable* :: *IRNode* ⇒ *bool* **where**
is-SwitchFoldable n = ((*is-IfNode n*))

fun *is-VirtualizableAllocation* :: *IRNode* ⇒ *bool* **where**
is-VirtualizableAllocation n = ((*is-NewArrayNode n*) ∨ (*is-NewInstanceNode n*))

fun *is-Unary* :: *IRNode* ⇒ *bool* **where**
is-Unary n = ((*is-LoadFieldNode n*) ∨ (*is-LogicNegationNode n*) ∨ (*is-UnaryNode n*) ∨ (*is-UnaryOpLogicNode n*))

fun *is-FixedNodeInterface* :: *IRNode* ⇒ *bool* **where**
is-FixedNodeInterface n = ((*is-FixedNode n*))

fun *is-BinaryCommutative* :: *IRNode* ⇒ *bool* **where**
is-BinaryCommutative n = ((*is-AddNode n*) ∨ (*is-AndNode n*) ∨ (*is-IntegerEqualsNode n*) ∨ (*is-MulNode n*) ∨ (*is-OrNode n*) ∨ (*is-XorNode n*))

fun *is-Canonicalizable* :: *IRNode* ⇒ *bool* **where**
is-Canonicalizable n = ((*is-BytecodeExceptionNode n*) ∨ (*is-ConditionalNode n*) ∨ (*is-DynamicNewArrayNode n*) ∨ (*is-PhiNode n*) ∨ (*is-PiNode n*) ∨ (*is-ProxyNode n*) ∨ (*is-StoreFieldNode n*) ∨ (*is-ValueProxyNode n*))

fun *is-UncheckedInterfaceProvider* :: *IRNode* ⇒ *bool* **where**

is-UncheckedInterfaceProvider n = ((*is-InvokeNode n*) ∨ (*is-InvokeWithExceptionNode n*) ∨ (*is-LoadFieldNode n*) ∨ (*is-ParameterNode n*))

fun *is-Binary* :: *IRNode* ⇒ *bool* **where**
is-Binary n = ((*is-BinaryArithmeticNode n*) ∨ (*is-BinaryNode n*) ∨ (*is-BinaryOpLogicNode n*) ∨ (*is-CompareNode n*) ∨ (*is-FixedBinaryNode n*) ∨ (*is-ShortCircuitOrNode n*))

fun *is-ArithmeticOperation* :: *IRNode* ⇒ *bool* **where**
is-ArithmeticOperation n = ((*is-BinaryArithmeticNode n*) ∨ (*is-IntegerConvertNode n*) ∨ (*is-ShiftNode n*) ∨ (*is-UnaryArithmeticNode n*))

fun *is-ValueNumberable* :: *IRNode* ⇒ *bool* **where**
is-ValueNumberable n = ((*is-FloatingNode n*) ∨ (*is-ProxyNode n*))

fun *is-Lowerable* :: *IRNode* ⇒ *bool* **where**
is-Lowerable n = ((*is-AbstractNewObjectNode n*) ∨ (*is-AccessFieldNode n*) ∨ (*is-BytecodeExceptionNode n*) ∨ (*is-ExceptionObjectNode n*) ∨ (*is-IntegerDivRemNode n*) ∨ (*is-UnwindNode n*))

fun *is-Virtualizable* :: *IRNode* ⇒ *bool* **where**
is-Virtualizable n = ((*is-IsNullNode n*) ∨ (*is-LoadFieldNode n*) ∨ (*is-PiNode n*) ∨ (*is-StoreFieldNode n*) ∨ (*is-ValueProxyNode n*))

fun *is-Simplifiable* :: *IRNode* ⇒ *bool* **where**
is-Simplifiable n = ((*is-AbstractMergeNode n*) ∨ (*is-BeginNode n*) ∨ (*is-IfNode n*) ∨ (*is-LoopExitNode n*) ∨ (*is-MethodCallTargetNode n*) ∨ (*is-NewArrayNode n*))

fun *is-StateSplit* :: *IRNode* ⇒ *bool* **where**
is-StateSplit n = ((*is-AbstractStateSplit n*) ∨ (*is-BeginStateSplitNode n*) ∨ (*is-StoreFieldNode n*))

fun *is-ConvertNode* :: *IRNode* ⇒ *bool* **where**
is-ConvertNode n = ((*is-IntegerConvertNode n*))

fun *is-sequential-node* :: *IRNode* ⇒ *bool* **where**
is-sequential-node (*StartNode - -*) = *True* |
is-sequential-node (*BeginNode -*) = *True* |
is-sequential-node (*KillingBeginNode -*) = *True* |
is-sequential-node (*LoopBeginNode - - - -*) = *True* |
is-sequential-node (*LoopExitNode - - -*) = *True* |
is-sequential-node (*MergeNode - - -*) = *True* |
is-sequential-node (*RefNode -*) = *True* |
is-sequential-node - = *False*

The following convenience function is useful in determining if two IRNodes are of the same type irregardless of their edges. It will return true if both the node parameters are the same node class.

fun *is-same-ir-node-type* :: *IRNode* ⇒ *IRNode* ⇒ *bool* **where**

```
is-same-ir-node-type n1 n2 = (
  ((is-AbsNode n1) ∧ (is-AbsNode n2)) ∨
  ((is-AddNode n1) ∧ (is-AddNode n2)) ∨
  ((is-AndNode n1) ∧ (is-AndNode n2)) ∨
  ((is-BeginNode n1) ∧ (is-BeginNode n2)) ∨
  ((is-BytecodeExceptionNode n1) ∧ (is-BytecodeExceptionNode n2)) ∨
  ((is-ConditionalNode n1) ∧ (is-ConditionalNode n2)) ∨
  ((is-ConstantNode n1) ∧ (is-ConstantNode n2)) ∨
  ((is-DynamicNewArrayNode n1) ∧ (is-DynamicNewArrayNode n2)) ∨
  ((is-EndNode n1) ∧ (is-EndNode n2)) ∨
  ((is-ExceptionObjectNode n1) ∧ (is-ExceptionObjectNode n2)) ∨
  ((is-FrameState n1) ∧ (is-FrameState n2)) ∨
  ((is-IfNode n1) ∧ (is-IfNode n2)) ∨
  ((is-IntegerBelowNode n1) ∧ (is-IntegerBelowNode n2)) ∨
  ((is-IntegerEqualsNode n1) ∧ (is-IntegerEqualsNode n2)) ∨
  ((is-IntegerLessThanNode n1) ∧ (is-IntegerLessThanNode n2)) ∨
  ((is-InvokeNode n1) ∧ (is-InvokeNode n2)) ∨
  ((is-InvokeWithExceptionNode n1) ∧ (is-InvokeWithExceptionNode n2)) ∨
  ((is-IsNullNode n1) ∧ (is-IsNullNode n2)) ∨
  ((is-KillingBeginNode n1) ∧ (is-KillingBeginNode n2)) ∨
  ((is-LoadFieldNode n1) ∧ (is-LoadFieldNode n2)) ∨
  ((is-LogicNegationNode n1) ∧ (is-LogicNegationNode n2)) ∨
  ((is-LoopBeginNode n1) ∧ (is-LoopBeginNode n2)) ∨
  ((is-LoopEndNode n1) ∧ (is-LoopEndNode n2)) ∨
  ((is-LoopExitNode n1) ∧ (is-LoopExitNode n2)) ∨
  ((is-MergeNode n1) ∧ (is-MergeNode n2)) ∨
  ((is-MethodCallTargetNode n1) ∧ (is-MethodCallTargetNode n2)) ∨
  ((is-MulNode n1) ∧ (is-MulNode n2)) ∨
  ((is-NegateNode n1) ∧ (is-NegateNode n2)) ∨
  ((is-NewArrayNode n1) ∧ (is-NewArrayNode n2)) ∨
  ((is-NewInstanceNode n1) ∧ (is-NewInstanceNode n2)) ∨
  ((is-NotNode n1) ∧ (is-NotNode n2)) ∨
  ((is-OrNode n1) ∧ (is-OrNode n2)) ∨
  ((is-ParameterNode n1) ∧ (is-ParameterNode n2)) ∨
  ((is-PiNode n1) ∧ (is-PiNode n2)) ∨
  ((is-ReturnNode n1) ∧ (is-ReturnNode n2)) ∨
  ((is-ShortCircuitOrNode n1) ∧ (is-ShortCircuitOrNode n2)) ∨
  ((is-SignedDivNode n1) ∧ (is-SignedDivNode n2)) ∨
  ((is-StartNode n1) ∧ (is-StartNode n2)) ∨
  ((is-StoreFieldNode n1) ∧ (is-StoreFieldNode n2)) ∨
  ((is-SubNode n1) ∧ (is-SubNode n2)) ∨
  ((is-UnwindNode n1) ∧ (is-UnwindNode n2)) ∨
  ((is-ValuePhiNode n1) ∧ (is-ValuePhiNode n2)) ∨
  ((is-ValueProxyNode n1) ∧ (is-ValueProxyNode n2)) ∨
  ((is-XorNode n1) ∧ (is-XorNode n2)))
```

end

3 Stamp Typing

```
theory Stamp
  imports Values
begin
```

The GraalVM compiler uses the Stamp class to store range and type information for a given node in the IR graph. We model the Stamp class as a datatype, Stamp, and provide a number of functions on the datatype which correspond to the class methods within the compiler.

Stamp information is used in a variety of ways in optimizations, and so, we additionally provide a number of lemmas which help to prove future optimizations.

```
datatype Stamp =
  VoidStamp
  | IntegerStamp (stp-bits: nat) (stpi-lower: int) (stpi-upper: int)

  | KlassPointerStamp (stp-nonNull: bool) (stp-alwaysNull: bool)
  | MethodCountersPointerStamp (stp-nonNull: bool) (stp-alwaysNull: bool)
  | MethodPointersStamp (stp-nonNull: bool) (stp-alwaysNull: bool)
 | ObjectStamp (stp-type: string) (stp-exactType: bool) (stp-nonNull: bool) (stp-alwaysNull:
bool)
  | RawPointerStamp (stp-nonNull: bool) (stp-alwaysNull: bool)
  | IllegalStamp

fun bit-bounds :: nat ⇒ (int × int) where
  bit-bounds bits = (((2 ^ bits) div 2) * −1, ((2 ^ bits) div 2) − 1)
```

— A stamp which includes the full range of the type

```
fun unrestricted-stamp :: Stamp ⇒ Stamp where
  unrestricted-stamp VoidStamp = VoidStamp |
   unrestricted-stamp (IntegerStamp bits lower upper) = (IntegerStamp bits (fst
(bit-bounds bits)) (snd (bit-bounds bits))) |

 unrestricted-stamp (KlassPointerStamp nonNull alwaysNull) = (KlassPointerStamp
False False) |
 unrestricted-stamp (MethodCountersPointerStamp nonNull alwaysNull) = (MethodCountersPointerStamp
False False) |
 unrestricted-stamp (MethodPointersStamp nonNull alwaysNull) = (MethodPointersStamp
False False) |
 unrestricted-stamp (ObjectStamp type exactType nonNull alwaysNull) = (ObjectStamp
'''' False False False) |
  unrestricted-stamp - = IllegalStamp

fun is-stamp-unrestricted :: Stamp ⇒ bool where
```

```
  is-stamp-unrestricted s = (s = unrestricted-stamp s)

— A stamp which provides type information but has an empty range of values
fun empty-stamp :: Stamp ⇒ Stamp where
  empty-stamp VoidStamp = VoidStamp |
  empty-stamp (IntegerStamp bits lower upper) = (IntegerStamp bits (snd (bit-bounds
bits)) (fst (bit-bounds bits))) |

  empty-stamp (KlassPointerStamp nonNull alwaysNull) = (KlassPointerStamp
nonNull alwaysNull) |
  empty-stamp (MethodCountersPointerStamp nonNull alwaysNull) = (MethodCountersPointerStamp
nonNull alwaysNull) |
  empty-stamp (MethodPointersStamp nonNull alwaysNull) = (MethodPointersStamp
nonNull alwaysNull) |
  empty-stamp (ObjectStamp type exactType nonNull alwaysNull) = (ObjectStamp
'''' True True False) |
  empty-stamp stamp = IllegalStamp

fun is-stamp-empty :: Stamp ⇒ bool where
  is-stamp-empty (IntegerStamp b lower upper) = (upper < lower) |

  is-stamp-empty x = False

— Calculate the meet stamp of two stamps
fun meet :: Stamp ⇒ Stamp ⇒ Stamp where
  meet VoidStamp VoidStamp = VoidStamp |
  meet (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1 ≠ b2 then IllegalStamp else
    (IntegerStamp b1 (min l1 l2) (max u1 u2))
  ) |

  meet (KlassPointerStamp nn1 an1) (KlassPointerStamp nn2 an2) = (
    KlassPointerStamp (nn1 ∧ nn2) (an1 ∧ an2)
  ) |
  meet (MethodCountersPointerStamp nn1 an1) (MethodCountersPointerStamp
nn2 an2) = (
    MethodCountersPointerStamp (nn1 ∧ nn2) (an1 ∧ an2)
  ) |
  meet (MethodPointersStamp nn1 an1) (MethodPointersStamp nn2 an2) = (
    MethodPointersStamp (nn1 ∧ nn2) (an1 ∧ an2)
  ) |
  meet s1 s2 = IllegalStamp

— Calculate the join stamp of two stamps
fun join :: Stamp ⇒ Stamp ⇒ Stamp where
  join VoidStamp VoidStamp = VoidStamp |
  join (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1 ≠ b2 then IllegalStamp else
    (IntegerStamp b1 (max l1 l2) (min u1 u2))
  ) |

  join (KlassPointerStamp nn1 an1) (KlassPointerStamp nn2 an2) = (
    if ((nn1 ∨ nn2) ∧ (an1 ∨ an2))
    then (empty-stamp (KlassPointerStamp nn1 an1))
    else (KlassPointerStamp (nn1 ∨ nn2) (an1 ∨ an2))
  ) |
  join (MethodCountersPointerStamp nn1 an1) (MethodCountersPointerStamp nn2
an2) = (
    if ((nn1 ∨ nn2) ∧ (an1 ∨ an2))
    then (empty-stamp (MethodCountersPointerStamp nn1 an1))
    else (MethodCountersPointerStamp (nn1 ∨ nn2) (an1 ∨ an2))
  ) |
  join (MethodPointersStamp nn1 an1) (MethodPointersStamp nn2 an2) = (
    if ((nn1 ∨ nn2) ∧ (an1 ∨ an2))
    then (empty-stamp (MethodPointersStamp nn1 an1))
    else (MethodPointersStamp (nn1 ∨ nn2) (an1 ∨ an2))
  ) |
  join s1 s2 = IllegalStamp
```

— In certain circumstances a stamp provides enough information to evaluate a value as a stamp, the asConstant function converts the stamp to a value where one can be inferred.

```
fun asConstant :: Stamp ⇒ Value where
  asConstant (IntegerStamp b l h) = (if l = h then IntVal64 (word-of-int l) else
UndefVal) |
  asConstant - = UndefVal
```

— Determine if two stamps never have value overlaps i.e. their join is empty

```
fun alwaysDistinct :: Stamp ⇒ Stamp ⇒ bool where
  alwaysDistinct stamp1 stamp2 = is-stamp-empty (join stamp1 stamp2)
```

— Determine if two stamps must always be the same value i.e. two equal constants

```
fun neverDistinct :: Stamp ⇒ Stamp ⇒ bool where
  neverDistinct stamp1 stamp2 = (asConstant stamp1 = asConstant stamp2 ∧
asConstant stamp1 ≠ UndefVal)

fun constantAsStamp :: Value ⇒ Stamp where
  constantAsStamp (IntVal32 v) = (IntegerStamp (nat 32) (sint v) (sint v)) |
  constantAsStamp (IntVal64 v) = (IntegerStamp (nat 64) (sint v) (sint v)) |

  constantAsStamp - = IllegalStamp
```

— Define when a runtime value is valid for a stamp

```
fun valid-value :: Value ⇒ Stamp ⇒ bool where
  valid-value (IntVal32 v) (IntegerStamp b l h) = (b=32 ∧ (sint v ≥ l) ∧ (sint v ≤
h)) |
  valid-value (IntVal64 v) (IntegerStamp b l h) = (b=64 ∧ (sint v ≥ l) ∧ (sint v ≤
h)) |

  valid-value (ObjRef ref) (ObjectStamp klass exact nonNull alwaysNull) = False |
  valid-value stamp val = False
```

```
fun compatible :: Stamp ⇒ Stamp ⇒ bool where
  compatible (IntegerStamp b1 - -) (IntegerStamp b2 - -) = (b1 = b2) |
  compatible (VoidStamp) (VoidStamp) = True |
  compatible - - = False

fun stamp-under :: Stamp ⇒ Stamp ⇒ bool where
  stamp-under x y = ((stpi-upper x) < (stpi-lower y))
```

— The most common type of stamp within the compiler (apart from the VoidStamp) is a 32 bit integer stamp with an unrestricted range. We use *default-stamp* as it is a frequently used stamp.

```
definition default-stamp :: Stamp where
  default-stamp = (unrestricted-stamp (IntegerStamp 32 0 0))

end
```

4 Graph Representation

```
theory IRGraph
  imports
    IRNodeHierarchy
    Stamp
    HOL−Library.FSet
    HOL.Relation
begin
```

This theory defines the main Graal data structure - an entire IR Graph.

IRGraph is defined as a partial map with a finite domain. The finite domain is required to be able to generate code and produce an interpreter.

```
typedef IRGraph = {g :: ID ⇀ (IRNode × Stamp) . finite (dom g)}
proof −
  have finite(dom(Map.empty)) ∧ ran Map.empty = {} by auto
  then show ?thesis
    by fastforce
qed

setup-lifting type-definition-IRGraph

lift-definition ids :: IRGraph ⇒ ID set
  is λg. {nid ∈ dom g . ∄ s. g nid = (Some (NoNode, s))} .
```

```
fun with-default :: 'c ⇒ ('b ⇒ 'c) ⇒ (('a ⇀ 'b) ⇒ 'a ⇒ 'c) where
  with-default def conv = (λm k.
    (case m k of None ⇒ def | Some v ⇒ conv v))

lift-definition kind :: IRGraph ⇒ (ID ⇒ IRNode)
  is with-default NoNode fst .

lift-definition stamp :: IRGraph ⇒ ID ⇒ Stamp
  is with-default IllegalStamp snd .

lift-definition add-node :: ID ⇒ (IRNode × Stamp) ⇒ IRGraph ⇒ IRGraph
  is λnid k g. if fst k = NoNode then g else g(nid ↦ k) by simp

lift-definition remove-node :: ID ⇒ IRGraph ⇒ IRGraph
  is λnid g. g(nid := None) by simp

lift-definition replace-node :: ID ⇒ (IRNode × Stamp) ⇒ IRGraph ⇒ IRGraph
  is λnid k g. if fst k = NoNode then g else g(nid ↦ k) by simp

lift-definition as-list :: IRGraph ⇒ (ID × IRNode × Stamp) list
  is λg. map (λk. (k, the (g k))) (sorted-list-of-set (dom g)) .

fun no-node :: (ID × (IRNode × Stamp)) list ⇒ (ID × (IRNode × Stamp)) list
where
  no-node g = filter (λn. fst (snd n) ≠ NoNode) g

lift-definition irgraph :: (ID × (IRNode × Stamp)) list ⇒ IRGraph
  is map-of ∘ no-node
  by (simp add: finite-dom-map-of)

definition as-set :: IRGraph ⇒ (ID × (IRNode × Stamp)) set where
  as-set g = {(n, kind g n, stamp g n) | n . n ∈ ids g}

definition true-ids :: IRGraph ⇒ ID set where
  true-ids g = ids g − {n ∈ ids g. ∃ n' . kind g n = RefNode n'}

definition domain-subtraction :: 'a set ⇒ ('a × 'b) set ⇒ ('a × 'b) set
  (infix ⊴ 30) where
  domain-subtraction s r = {(x, y) . (x, y) ∈ r ∧ x ∉ s}

notation (latex)
  domain-subtraction (- ⩤ -)


code-datatype irgraph

fun filter-none where
  filter-none g = {nid ∈ dom g . ∄ s. g nid = (Some (NoNode, s))}
```

```
lemma no-node-clears:
  res = no-node xs ⟶ (∀ x ∈ set res. fst (snd x) ≠ NoNode)
  by simp

lemma dom-eq:
  assumes ∀ x ∈ set xs. fst (snd x) ≠ NoNode
  shows filter-none (map-of xs) = dom (map-of xs)
  unfolding filter-none.simps using assms map-of-SomeD
  by fastforce

lemma fil-eq:
  filter-none (map-of (no-node xs)) = set (map fst (no-node xs))
  using no-node-clears
  by (metis dom-eq dom-map-of-conv-image-fst list.set-map)

lemma irgraph[code]: ids (irgraph m) = set (map fst (no-node m))
  unfolding irgraph-def ids-def using fil-eq
   by (smt Rep-IRGraph comp-apply eq-onp-same-args filter-none.simps ids.abs-eq
ids-def irgraph.abs-eq irgraph.rep-eq irgraph-def mem-Collect-eq)

lemma [code]: Rep-IRGraph (irgraph m) = map-of (no-node m)
  using Abs-IRGraph-inverse
  by (simp add: irgraph.rep-eq)


— Get the inputs set of a given node ID
fun inputs :: IRGraph ⇒ ID ⇒ ID set where
  inputs g nid = set (inputs-of (kind g nid))
— Get the successor set of a given node ID
fun succ :: IRGraph ⇒ ID ⇒ ID set where
  succ g nid = set (successors-of (kind g nid))
— Gives a relation between node IDs - between a node and its input nodes
fun input-edges :: IRGraph ⇒ ID rel where
  input-edges g = (⋃ i ∈ ids g. {(i,j)|j. j ∈ (inputs g i)})
— Find all the nodes in the graph that have nid as an input - the usages of nid
fun usages :: IRGraph ⇒ ID ⇒ ID set where
  usages g nid = {j. j ∈ ids g ∧ (j,nid) ∈ input-edges g}
fun successor-edges :: IRGraph ⇒ ID rel where
  successor-edges g = (⋃ i ∈ ids g. {(i,j)|j . j ∈ (succ  g i)})
fun predecessors :: IRGraph ⇒ ID ⇒ ID set where
  predecessors g nid = {j. j ∈ ids g ∧ (j,nid) ∈ successor-edges g}
fun nodes-of :: IRGraph ⇒ (IRNode ⇒ bool) ⇒ ID set where
  nodes-of g sel = {nid ∈ ids g . sel (kind g nid)}
fun edge :: (IRNode ⇒ 'a) ⇒ ID ⇒ IRGraph ⇒ 'a where
  edge sel nid g = sel (kind g nid)

fun filtered-inputs :: IRGraph ⇒ ID ⇒ (IRNode ⇒ bool) ⇒ ID list where
  filtered-inputs g nid f = filter (f ∘ (kind g)) (inputs-of (kind g nid))
```

```
fun filtered-successors :: IRGraph ⇒ ID ⇒ (IRNode ⇒ bool) ⇒ ID list where
  filtered-successors g nid f = filter (f ∘ (kind g)) (successors-of (kind g nid))
fun filtered-usages :: IRGraph ⇒ ID ⇒ (IRNode ⇒ bool) ⇒ ID set where
  filtered-usages g nid f = {n ∈ (usages g nid). f (kind g n)}

fun is-empty :: IRGraph ⇒ bool where
  is-empty g = (ids g = {})

fun any-usage :: IRGraph ⇒ ID ⇒ ID where
  any-usage g nid = hd (sorted-list-of-set (usages g nid))

lemma ids-some[simp]: x ∈ ids g ⟷ kind g x ≠ NoNode
proof −
  have that: x ∈ ids g ⟶ kind g x ≠ NoNode
    using ids.rep-eq kind.rep-eq by force
  have kind g x ≠ NoNode ⟶ x ∈ ids g
    unfolding with-default.simps kind-def ids-def
    by (cases Rep-IRGraph g x = None; auto)
  from this that show ?thesis by auto
qed

lemma not-in-g:
  assumes nid ∉ ids g
  shows kind g nid = NoNode
  using assms ids-some by blast

lemma valid-creation[simp]:
  finite (dom g) ⟷ Rep-IRGraph (Abs-IRGraph g) = g
  using Abs-IRGraph-inverse by (metis Rep-IRGraph mem-Collect-eq)

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

lemma [simp]: finite (ids (irgraph g))
  by (simp add: finite-dom-map-of)

lemma [simp]: finite (dom g) ⟶ ids (Abs-IRGraph g) = {nid ∈ dom g . ∄ s. g
nid = Some (NoNode, s)}
  using ids.rep-eq by simp

lemma [simp]: finite (dom g) ⟶ kind (Abs-IRGraph g) = (λx . (case g x of None
⇒ NoNode | Some n ⇒ fst n))
  by (simp add: kind.rep-eq)

lemma [simp]: finite (dom g) ⟶ stamp (Abs-IRGraph g) = (λx . (case g x of
None ⇒ IllegalStamp | Some n ⇒ snd n))
  using stamp.abs-eq stamp.rep-eq by auto

lemma [simp]: ids (irgraph g) = set (map fst (no-node g))
```

```
  using irgraph by auto

lemma [simp]: kind (irgraph g) = (λnid. (case (map-of (no-node g)) nid of None
⇒ NoNode | Some n ⇒ fst n))
  using irgraph.rep-eq kind.transfer kind.rep-eq by auto

lemma [simp]: stamp (irgraph g) = (λnid. (case (map-of (no-node g)) nid of None
⇒ IllegalStamp | Some n ⇒ snd n))
  using irgraph.rep-eq stamp.transfer stamp.rep-eq by auto

lemma map-of-upd: (map-of g)(k ↦ v) = (map-of ((k, v) # g))
  by simp


lemma [code]: replace-node nid k (irgraph g) = (irgraph ( ((nid, k) # g)))
proof (cases fst k = NoNode)
  case True
  then show ?thesis
    by (metis (mono-tags, lifting) Rep-IRGraph-inject filter.simps(2) irgraph.abs-eq
no-node.simps replace-node.rep-eq snd-conv)
next
  case False
  then show ?thesis unfolding irgraph-def replace-node-def no-node.simps
    by (smt (verit, best) Rep-IRGraph comp-apply eq-onp-same-args filter.simps(2)
id-def irgraph.rep-eq map-fun-apply map-of-upd mem-Collect-eq no-node.elims
replace-node.abs-eq replace-node-def snd-eqD)
qed

lemma [code]: add-node nid k (irgraph g) = (irgraph (((nid, k) # g)))
  by (smt (z3) Rep-IRGraph-inject add-node.rep-eq filter.simps(2) irgraph.rep-eq
map-of-upd no-node.simps snd-conv)

lemma add-node-lookup:
  gup = add-node nid (k, s) g ⟶
    (if k ≠ NoNode then kind gup nid = k ∧ stamp gup nid = s else kind gup nid
= kind g nid)
proof (cases k = NoNode)
  case True
  then show ?thesis
    by (simp add: add-node.rep-eq kind.rep-eq)
next
  case False
  then show ?thesis
    by (simp add: kind.rep-eq add-node.rep-eq stamp.rep-eq)
qed

lemma remove-node-lookup:
  gup = remove-node nid g ⟶ kind gup nid = NoNode ∧ stamp gup nid =
IllegalStamp
```

by (*simp add*: *kind.rep-eq remove-node.rep-eq stamp.rep-eq*)

lemma *replace-node-lookup*[*simp*]:
gup = *replace-node nid* (*k*, *s*) *g* ∧ *k* ≠ *NoNode* ⟶ *kind gup nid* = *k* ∧ *stamp gup nid* = *s*
by (*simp add*: *replace-node.rep-eq kind.rep-eq stamp.rep-eq*)

lemma *replace-node-unchanged*:
gup = *replace-node nid* (*k*, *s*) *g* ⟶ (∀ *n* ∈ (*ids g* − {*nid*}) . *n* ∈ *ids g* ∧ *n* ∈ *ids gup* ∧ *kind g n* = *kind gup n*)
by (*simp add*: *kind.rep-eq replace-node.rep-eq*)

4.0.1 Example Graphs

Example 1: empty graph (just a start and end node)

definition *start-end-graph*:: *IRGraph* **where**
start-end-graph = *irgraph* [(*0*, *StartNode None 1*, *VoidStamp*), (*1*, *ReturnNode None None*, *VoidStamp*)]

Example 2: public static int sq(int x) return x * x;

[1 P(0)] / [0 Start] [4 *] | / V / [5 Return]

definition *eg2-sq* :: *IRGraph* **where**
eg2-sq = *irgraph* [
  (*0*, *StartNode None 5*, *VoidStamp*),
  (*1*, *ParameterNode 0*, *default-stamp*),
  (*4*, *MulNode 1 1*, *default-stamp*),
  (*5*, *ReturnNode* (*Some 4*) *None*, *default-stamp*)
 ]

value *input-edges eg2-sq*
value *usages eg2-sq 1*

end

4.1 Control-flow Graph Traversal

theory
Traversal
imports
IRGraph
begin

type-synonym *Seen* = *ID set*

nextEdge helps determine which node to traverse next by returning the first successor edge that isn't in the set of already visited nodes. If there is not an appropriate successor, None is returned instead.

```
fun nextEdge :: Seen ⇒ ID ⇒ IRGraph ⇒ ID option where
  nextEdge seen nid g =
    (let nids = (filter (λnid'. nid' ∉ seen) (successors-of (kind g nid))) in
     (if length nids > 0 then Some (hd nids) else None))
```

pred determines which node, if any, acts as the predecessor of another.

Merge nodes represent a special case where-in the predecessor exists as an input edge of the merge node, to simplify the traversal we treat only the first input end node as the predecessor, ignoring that multiple nodes may act as a successor.

For all other nodes, the predecessor is the first element of the predecessors set. Note that in a well-formed graph there should only be one element in the predecessor set.

```
fun pred :: IRGraph ⇒ ID ⇒ ID option where
  pred g nid = (case kind g nid of
    (MergeNode ends - -) ⇒ Some (hd ends) |
    - ⇒
      (if IRGraph.predecessors g nid = {}
        then None else
        Some (hd (sorted-list-of-set (IRGraph.predecessors g nid)))
      )
  )
```

Here we try to implement a generic fork of the control-flow traversal algorithm that was initially implemented for the ConditionalElimination phase

```
type-synonym 'a TraversalState = (ID × Seen × 'a)

inductive Step
  :: ('a TraversalState ⇒ 'a) ⇒ IRGraph ⇒ 'a TraversalState ⇒ 'a TraversalState
option ⇒ bool
  for sa g where
```

— Hit a BeginNode with an IfNode predecessor which represents the start of a basic block for the IfNode. 1. nid' will be the successor of the begin node. 2. Find the first and only predecessor. 3. Extract condition from the preceding IfNode. 4. Negate condition if the begin node is second branch (we've taken the else branch of the condition) 5. Add the condition or the negated condition to stack 6. Perform any stamp updates based on the condition using the registerNewCondition function and place them on the top of the stack of stamp information

```
  ⟦kind g nid = BeginNode nid';

    nid ∉ seen;
    seen' = {nid} ∪ seen;

    Some ifcond = pred g nid;
    kind g ifcond = IfNode cond t f;
```

$\quad analysis' = sa\ (nid,\ seen,\ analysis)\rrbracket$
$\Longrightarrow Step\ sa\ g\ (nid,\ seen,\ analysis)\ (Some\ (nid',\ seen',\ analysis'))\ |$

— Hit an EndNode 1. nid' will be the usage of EndNode 2. pop the conditions and stamp stack
$\llbracket kind\ g\ nid = EndNode;$

$\quad nid \notin seen;$
$\quad seen' = \{nid\} \cup seen;$

$\quad nid' = any\text{-}usage\ g\ nid;$

$\quad analysis' = sa\ (nid,\ seen,\ analysis)\rrbracket$
$\Longrightarrow Step\ sa\ g\ (nid,\ seen,\ analysis)\ (Some\ (nid',\ seen',\ analysis'))\ |$

— We can find a successor edge that is not in seen, go there
$\llbracket \neg(is\text{-}EndNode\ (kind\ g\ nid));$
$\quad \neg(is\text{-}BeginNode\ (kind\ g\ nid));$

$\quad nid \notin seen;$
$\quad seen' = \{nid\} \cup seen;$

$\quad Some\ nid' = nextEdge\ seen'\ nid\ g;$

$\quad analysis' = sa\ (nid,\ seen,\ analysis)\rrbracket$
$\Longrightarrow Step\ sa\ g\ (nid,\ seen,\ analysis)\ (Some\ (nid',\ seen',\ analysis'))\ |$

— We can cannot find a successor edge that is not in seen, give back None
$\llbracket \neg(is\text{-}EndNode\ (kind\ g\ nid));$
$\quad \neg(is\text{-}BeginNode\ (kind\ g\ nid));$

$\quad nid \notin seen;$
$\quad seen' = \{nid\} \cup seen;$

$\quad None = nextEdge\ seen'\ nid\ g\rrbracket$
$\quad \Longrightarrow Step\ sa\ g\ (nid,\ seen,\ analysis)\ None\ |$

— We've already seen this node, give back None
$\llbracket nid \in seen\rrbracket \Longrightarrow Step\ sa\ g\ (nid,\ seen,\ analysis)\ None$

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

end

4.2 Structural Graph Comparison

theory
Comparison

```
imports
  IRGraph
begin
```

We introduce a form of structural graph comparison that is able to assert structural equivalence of graphs which differ in zero or more reference node chains for any given nodes.

```
fun find-ref-nodes :: IRGraph ⇒ (ID ⇀ ID) where
find-ref-nodes g = map-of
 (map (λn. (n, ir-ref (kind g n))) (filter (λid. is-RefNode (kind g id)) (sorted-list-of-set
(ids g))))

fun replace-ref-nodes :: IRGraph ⇒ (ID ⇀ ID) ⇒ ID list ⇒ ID list where
replace-ref-nodes g m xs = map (λid. (case (m id) of Some other ⇒ other | None
⇒ id)) xs

fun find-next :: ID list ⇒ ID set ⇒ ID option where
  find-next to-see seen = (let l = (filter (λnid. nid ∉ seen) to-see)
    in (case l of [] ⇒ None | xs ⇒ Some (hd xs)))

inductive reachables :: IRGraph ⇒ ID list ⇒ ID set ⇒ ID set ⇒ bool where
reachables g [] {} {} |
⟦None = find-next to-see seen⟧ ⟹ reachables g to-see seen seen |
⟦Some n = find-next to-see seen;
  node = kind g n;
  new = (inputs-of node) @ (successors-of node);
   reachables g (to-see @ new) ({n} ∪ seen) seen′ ⟧ ⟹ reachables g to-see seen
seen′

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

reachables .

inductive nodeEq :: (ID ⇀ ID) ⇒ IRGraph ⇒ ID ⇒ IRGraph ⇒ ID ⇒ bool
where
⟦ kind g1 n1 = RefNode ref; nodeEq m g1 ref g2 n2 ⟧ ⟹ nodeEq m g1 n1 g2 n2 |
⟦ x = kind g1 n1;
  y = kind g2 n2;
  is-same-ir-node-type x y;
  replace-ref-nodes g1 m (successors-of x) = successors-of y;
  replace-ref-nodes g1 m (inputs-of x) = inputs-of y ⟧
  ⟹ nodeEq m g1 n1 g2 n2

code-pred [show-modes] nodeEq .

fun diffNodesGraph :: IRGraph ⇒ IRGraph ⇒ ID set where
diffNodesGraph g1 g2 = (let refNodes = find-ref-nodes g1 in
    { n . n ∈ Predicate.the (reachables-i-i-i-o g1 [0] {}) ∧ (case refNodes n of Some
- ⇒ False | - ⇒ True) ∧ ¬(nodeEq refNodes g1 n g2 n)})
```

fun *diffNodesInfo* :: *IRGraph* ⇒ *IRGraph* ⇒ (*ID* × *IRNode* × *IRNode*) *set* **where**
diffNodesInfo g1 g2 = {(*nid*, *kind g1 nid*, *kind g2 nid*) | *nid* . *nid* ∈ *diffNodesGraph g1 g2*}

fun *eqGraph* :: *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
eqGraph isabelle-graph graal-graph = ((*diffNodesGraph isabelle-graph graal-graph*) = {})

end

5 Data-flow Semantics

theory *IRTreeEval*
  imports
    Graph.Values
    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*)

5.1 Data-flow Tree Representation

datatype *IRUnaryOp* =

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

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

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

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


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

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

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

typedef GroundExpr = { e :: IRExpr . is-ground e }
```

using *is-ground.simps(6)* **by** *blast*

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

stamp-unary op - = IllegalStamp

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

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

stamp-binary op - - = IllegalStamp

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

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

5.2 Data-flow Tree Evaluation

fun *unary-eval* :: *IRUnaryOp ⇒ Value ⇒ Value* **where**
unary-eval UnaryAbs v = intval-abs v |
unary-eval UnaryNeg v = intval-negate v |
unary-eval UnaryNot v = intval-not v |
unary-eval UnaryLogicNegation v = intval-logic-negation v |
unary-eval op v1 = UndefVal

fun *bin-eval* :: *IRBinaryOp ⇒ Value ⇒ Value ⇒ Value* **where**
bin-eval BinAdd v1 v2 = intval-add v1 v2 |
bin-eval BinMul v1 v2 = intval-mul v1 v2 |
bin-eval BinSub v1 v2 = intval-sub v1 v2 |
bin-eval BinAnd v1 v2 = intval-and v1 v2 |
bin-eval BinOr v1 v2 = intval-or v1 v2 |
bin-eval BinXor v1 v2 = intval-xor v1 v2 |
bin-eval BinLeftShift v1 v2 = intval-left-shift v1 v2 |

```
  bin-eval BinRightShift v1 v2 = intval-right-shift v1 v2 |
  bin-eval BinURightShift v1 v2 = intval-uright-shift v1 v2 |
  bin-eval BinIntegerEquals v1 v2 = intval-equals v1 v2 |
  bin-eval BinIntegerLessThan v1 v2 = intval-less-than v1 v2 |
  bin-eval BinIntegerBelow v1 v2 = intval-below v1 v2


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

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

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

  ConstantExpr:
  ⟦valid-value c (constantAsStamp 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 ↦ v;
    v ≠ UndefVal⟧
    ⟹ [m,p] ⊢ (ConditionalExpr ce te fe) ↦ v |

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

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

  LeafExpr:
  ⟦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 [IntVal32 5] sq-param0 v}


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

5.3 Data-flow Tree Refinement

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

```
definition equiv-exprs :: IRExpr ⇒ IRExpr ⇒ bool (- ≐ - 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** $\sqsubseteq$ *65*)

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

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

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

end

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

end

5.4 Data-flow Tree Theorems

theory *IRTreeEvalThms*
imports
IRTreeEval
begin

5.4.1 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
```

5.4.2 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-int32[elim]:
 shows valid-value val (IntegerStamp 32 l h) ⟹
    (∃ v. val = IntVal32 v)
 apply (rule val-to-bool.cases[of val])
 using Value.distinct by simp+
```

```
lemma valid-int64[elim]:
 shows valid-value val (IntegerStamp 64 l h) ⟹
    (∃ v. val = IntVal64 v)
 apply (rule val-to-bool.cases[of val])
 using Value.distinct by simp+
```

```
lemmas valid-value-elims =
 valid-VoidStamp
 valid-ObjStamp
 valid-int32
 valid-int64
```

```
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 by auto
```

```
lemma leafint32:
  assumes ev: [m,p] ⊢ LeafExpr i (IntegerStamp 32 lo hi) ↦ val
  shows ∃ v. val = (IntVal32 v)

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


lemma leafint64:
  assumes ev: [m,p] ⊢ LeafExpr i (IntegerStamp 64 lo hi) ↦ val
  shows ∃ v. val = (IntVal64 v)

proof –
  have valid-value val (IntegerStamp 64 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 valid32 [simp]:
  assumes valid-value val (IntegerStamp 32 lo hi)
  shows ∃ v. (val = (IntVal32 v) ∧ lo ≤ sint v ∧ sint v ≤ hi)
  using assms valid-int32 by force

lemma valid64 [simp]:
  assumes valid-value val (IntegerStamp 64 lo hi)
  shows ∃ v. (val = (IntVal64 v) ∧ lo ≤ sint v ∧ sint v ≤ hi)
  using assms valid-int64 by force

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

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

5.4.3 Example Data-flow Optimisations

```
lemma a0a-helper [simp]:
  assumes a: valid-value v (IntegerStamp 32 lo hi)
  shows intval-add v (IntVal32 0) = v
proof –
  obtain v32 :: int32 where v = (IntVal32 v32) using a valid32 by blast
  then show ?thesis by simp
qed
```

```
lemma a0a: (BinaryExpr BinAdd (LeafExpr 1 default-stamp) (ConstantExpr (IntVal32
0)))
            ≥ (LeafExpr 1 default-stamp)
  by (auto simp add: evaltree.LeafExpr)
```

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

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

5.4.4 Monotonicity of Expression Refinement

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

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

```
lemma mono-unary:
  assumes e ≥ e′
  shows (UnaryExpr op e) ≥ (UnaryExpr op e′)
```

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

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

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

lemma *stamp32*:
$\exists\, v$. $xv = \mathit{IntVal32}\ v \longleftrightarrow$ *valid-value xv* (*IntegerStamp 32 lo hi*)
using *valid-int32*
by (*metis* (*full-types*) *Value.inject*(*1*) *zero-neq-one*)

lemma *stamp64*:
$\exists\, v$. $xv = \mathit{IntVal64}\ v \longleftrightarrow$ *valid-value xv* (*IntegerStamp 64 lo hi*)
using *valid-int64*
by (*metis* (*full-types*) *Value.inject*(*2*) *zero-neq-one*)

lemma *stamprange*:
valid-value v s $\longrightarrow$ ($\exists\, b\ lo\ hi.$ ($s = \mathit{IntegerStamp}\ b\ lo\ hi$) $\wedge$ ($b = 32 \vee b = 64$))
using *valid-value.elims stamp32 stamp64*
by (*smt* (*verit*, *del-insts*))

lemma *compatible-trans*:
compatible x y $\wedge$ *compatible y z* $\Longrightarrow$ *compatible x z*
by (*smt* (*verit*, *best*) *compatible.elims*(*2*) *compatible.simps*(*1*))

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

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

```
  then have ce′: [m,p] ⊢ 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


end
```

6 Tree to Graph

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

6.1 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⟧
```

$\Longrightarrow g \vdash n \simeq (\mathit{ConstantExpr}\ c)$ |

$\mathit{ParameterNode}$:
$[\![\mathit{kind}\ g\ n = \mathit{ParameterNode}\ i;$
$\mathit{stamp}\ g\ n = s]\!]$
$\Longrightarrow g\ \vdash n \simeq (\mathit{ParameterExpr}\ i\ s)$ |

$\mathit{ConditionalNode}$:
$[\![\mathit{kind}\ g\ n = \mathit{ConditionalNode}\ c\ t\ f;$
$g \vdash c \simeq ce;$
$g \vdash t \simeq te;$
$g \vdash f \simeq fe]\!]$
$\Longrightarrow g \vdash n \simeq (\mathit{ConditionalExpr}\ ce\ te\ fe)$ |

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

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

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

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

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

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

$\mathit{SubNode}$:

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

AndNode:

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

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:

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

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:

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

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

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

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

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

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

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

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

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

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

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

RepCons:

```
⟦g ⊢ x ≃ xe;
  g ⊢ xs ≃_L xse⟧
  ⟹ g ⊢ x#xs ≃_L xe#xse
```

```
code-pred (modes: i ⇒ i ⇒ o ⇒ 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}
```

6.2 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 BinLeftShift x y = LeftShiftNode x y |
  bin-node BinRightShift x y = RightShiftNode x y |
  bin-node BinURightShift x y = UnsignedRightShiftNode x y |
  bin-node BinIntegerEquals x y = IntegerEqualsNode x y |
  bin-node BinIntegerLessThan x y = IntegerLessThanNode x y |
  bin-node BinIntegerBelow x y = IntegerBelowNode x y
```

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

```
inductive fresh-id :: IRGraph ⇒ ID ⇒ bool where
```

```
  n ∉ ids g ⟹ fresh-id g n

code-pred fresh-id .


fun get-fresh-id :: IRGraph ⇒ 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⟧
    ⟹ g ◁ (UnaryExpr op xe) ↝ (g2, n) |

  UnaryNodeNew:
  ⟦g ◁ xe ↝ (g2, x);
    s′ = stamp-unary op (stamp g2 x);
    find-node-and-stamp g2 (unary-node op x, s′) = None;
    n = get-fresh-id g2;
    g′ = add-node n (unary-node op x, s′) g2⟧
    ⟹ g ◁ (UnaryExpr op xe) ↝ (g′, n) |

  BinaryNodeSame:
  ⟦g ◁ xe ↝ (g2, x);
    g2 ◁ ye ↝ (g3, y);
    s′ = stamp-binary op (stamp g3 x) (stamp g3 y);
    find-node-and-stamp g3 (bin-node op x y, s′) = Some n⟧
    ⟹ g ◁ (BinaryExpr op xe ye) ↝ (g3, n) |

  BinaryNodeNew:
  ⟦g ◁ xe ↝ (g2, x);
    g2 ◁ ye ↝ (g3, y);
    s′ = stamp-binary op (stamp g3 x) (stamp g3 y);
    find-node-and-stamp g3 (bin-node op x y, s′) = None;
    n = get-fresh-id g3;
    g′ = add-node n (bin-node op x y, s′) g3⟧
    ⟹ g ◁ (BinaryExpr op xe ye) ↝ (g′, n) |

  AllLeafNodes:
  ⟦stamp g n = s;
    is-preevaluated (kind g n)⟧
    ⟹ g ◁ (LeafExpr n s) ↝ (g, n)


code-pred (modes: i ⇒ i ⇒ o ⇒ bool as unrepE)
  unrep .
```

unrepRules

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

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

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

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

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

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

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

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

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

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

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

values {(*n*, *g*) . (*eg2-sq* ◁ *sq-param0* ⇝ (*g*, *n*))}

6.3 Lift Data-flow Tree Semantics

definition *encodeeval* :: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *ID* ⇒ *Value* ⇒ *bool*
([-,-,-] ⊢ - ↦ - *50*)
where
encodeeval g m p n v = (∃ *e*. (*g* ⊢ *n* ≃ *e*) ∧ ([*m*,*p*] ⊢ *e* ↦ *v*))

6.4 Graph Refinement

definition *graph-represents-expression* :: *IRGraph* ⇒ *ID* ⇒ *IRExpr* ⇒ *bool*
(- ⊢ - ⊴ - *50*)
where
(*g* ⊢ *n* ⊴ *e*) = (∃ *e′* . (*g* ⊢ *n* ≃ *e′*) ∧ (*e′* ≤ *e*))

definition *graph-refinement* :: *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
graph-refinement g_1 g_2 =
((*ids* g_1 ⊆ *ids* g_2) ∧
(∀ *n* . *n* ∈ *ids* g_1 ⟶ (∀ *e*. (g_1 ⊢ *n* ≃ *e*) ⟶ (g_2 ⊢ *n* ⊴ *e*))))

lemma *graph-refinement*:
graph-refinement g1 g2 ⟹ (∀ *n m p v*. *n* ∈ *ids g1* ⟶ ([*g1*, *m*, *p*] ⊢ *n* ↦ *v*) ⟶ ([*g2*, *m*, *p*] ⊢ *n* ↦ *v*))
by (*meson encodeeval-def graph-refinement-def graph-represents-expression-def le-expr-def*)

6.5 Maximal Sharing

definition *maximal-sharing*:
maximal-sharing g = (∀ n_1 n_2 . n_1 ∈ *true-ids g* ∧ n_2 ∈ *true-ids g* ⟶
(∀ *e*. (*g* ⊢ n_1 ≃ *e*) ∧ (*g* ⊢ n_2 ≃ *e*) ⟶ n_1 = n_2))

end

6.6 Formedness Properties

theory *Form*
imports
Semantics.TreeToGraph
begin

definition *wf-start* **where**
wf-start g = (*0* ∈ *ids g* ∧
is-StartNode (*kind g 0*))

definition *wf-closed* **where**
wf-closed g =
(∀ *n* ∈ *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
```

6.7 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
    Semantics.IRTreeEval
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 (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
```

6.8 Tree to Graph Theorems

```
theory TreeToGraphThms
imports
  TreeToGraph
  IRTreeEvalThms
  IRGraphFrames
  HOL−Eisbach.Eisbach
  HOL−Eisbach.Eisbach-Tools
begin
```

6.8.1 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 ⊢ n ≃ e ⟹
   kind g n = SubNode x y ⟹
   (∃ xe ye. e = BinaryExpr BinSub xe ye)
  by (induction rule: rep.induct; auto)

lemma rep-mul [rep]:
  g ⊢ n ≃ e ⟹
   kind g n = MulNode x y ⟹
```

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

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

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

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

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

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

lemma rep-integer-less-than [rep]:
  g ⊢ 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 ≃ e₁) ⟹ (g ⊢ n ≃ e₂) ⟹ e₁ = e₂
proof (induction arbitrary: e₂ rule: rep.induct)
  case (ConstantNode n c)
  then show ?case using rep-constant by auto
next
  case (ParameterNode n i s)
  then show ?case
    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 (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
```

6.8.2 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›)
```

6.8.3 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 (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: 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)
```

6.8.4 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.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
  subgoal premises e 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 LeftShiftNode bin-node.simps(7) apply presburger
      using RightShiftNode bin-node.simps(8) apply presburger
      using UnsignedRightShiftNode bin-node.simps(9) apply presburger
```

```
      using IntegerEqualsNode bin-node.simps(10) apply presburger
      using IntegerLessThanNode bin-node.simps(11) apply presburger
      using IntegerBelowNode bin-node.simps(12) 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 LeftShiftNode bin-node.simps(7) apply presburger
      using RightShiftNode bin-node.simps(8) apply presburger
      using UnsignedRightShiftNode bin-node.simps(9) apply presburger
      using IntegerEqualsNode bin-node.simps(10) apply presburger
      using IntegerLessThanNode bin-node.simps(11) apply presburger
      using IntegerBelowNode bin-node.simps(12) 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 ◁ e ⇝ (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*)

end

7 Control-flow Semantics

theory *IRStepObj*
  imports
    TreeToGraph
begin

7.1 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)
```

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

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

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

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

    i = find-index nid (inputs-of (kind g merge));
    phis = (phi-list g merge);
    inps = (phi-inputs g i phis);
    g ⊢ inps ≃_L inpsE;
    [m, p] ⊢ inpsE ↦_L vs;

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


  NewInstanceNode:
    ⟦kind g nid = (NewInstanceNode nid f obj nid′);
      (h′, ref) = h-new-inst h;
      m′ = m(nid := ref)⟧
    ⟹ g, p ⊢ (nid, m, h) → (nid′, m′, h′) |

  LoadFieldNode:
    ⟦kind g nid = (LoadFieldNode nid f (Some obj) nid′);
      g ⊢ obj ≃ objE;
      [m, p] ⊢ objE ↦ ObjRef ref;
      h-load-field f ref h = v;
      m′ = m(nid := v)⟧
    ⟹ g, p ⊢ (nid, m, h) → (nid′, m′, h) |

  SignedDivNode:
    ⟦kind g nid = (SignedDivNode nid x y zero sb nxt);
      g ⊢ x ≃ xe;
      g ⊢ y ≃ ye;
      [m, p] ⊢ xe ↦ v1;
      [m, p] ⊢ ye ↦ v2;
      v = (intval-div v1 v2);
      m′ =  m(nid := v)⟧
    ⟹ g, p ⊢ (nid, m, h) → (nxt, m′, h) |

  SignedRemNode:
    ⟦kind g nid = (SignedRemNode nid x y zero sb nxt);
      g ⊢ x ≃ xe;
      g ⊢ y ≃ ye;
      [m, p] ⊢ xe ↦ v1;
      [m, p] ⊢ ye ↦ v2;
      v = (intval-mod v1 v2);
      m′ =  m(nid := v)⟧
    ⟹ g, p ⊢ (nid, m, h) → (nxt, m′, h) |

  StaticLoadFieldNode:
    ⟦kind g nid = (LoadFieldNode nid f None nid′);
      h-load-field f None h = v;
      m′ =  m(nid := v)⟧
    ⟹ g, p ⊢ (nid, m, h) → (nid′, m′, h) |

  StoreFieldNode:
    ⟦kind g nid = (StoreFieldNode nid f newval - (Some obj) nid′);
      g ⊢ newval ≃ newvalE;
      g ⊢ obj ≃ objE;
      [m, p] ⊢ newvalE ↦ val;
      [m, p] ⊢ objE ↦ ObjRef ref;
      h′ = h-store-field f ref val h;
      m′ =  m(nid := val)⟧
    ⟹ g, p ⊢ (nid, m, h) → (nid′, m′, h′) |

  StaticStoreFieldNode:
    ⟦kind g nid = (StoreFieldNode nid f newval - None nid′);
      g ⊢ newval ≃ newvalE;
      [m, p] ⊢ newvalE ↦ val;
      h′ = h-store-field f None val h;
      m′ =  m(nid := val)⟧
    ⟹ g, p ⊢ (nid, m, h) → (nid′, m′, h′)
```

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

7.3 Interprocedural Semantics

type-synonym *Signature* = *string*
type-synonym *Program* = *Signature* ⇀ *IRGraph*

inductive *step-top* :: *Program* ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *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 .
```

7.4 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 .
```

7.4.1 Heap Testing

```
definition p3:: Params where
  p3 = [IntVal32 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) →*4* res}

end
```

7.5 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.

7.5.1 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 IRN-
```

```
ode.distinct IRNode.inject(11) Pair-inject step.simps
    by (smt (z3) IRNode.distinct IRNode.inject(12) Pair-inject step.simps)
next
  case (EndNodes nid merge 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) IRNode.distinct(2311) IRNode.distinct(2313) IRNode.inject(31) Pair-inject)
next
  case (LoadFieldNode nid f obj nxt m ref h v m′)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using LoadFieldNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject Value.inject(3)
option.distinct(1) option.inject)
next
  case (StaticLoadFieldNode nid f nxt h v m′ m)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StaticLoadFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StaticLoadFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StaticLoadFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StaticLoadFieldNode step.cases
   by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject option.distinct(1))
next
  case (StoreFieldNode nid f newval uu obj nxt m val ref h′ h m′)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StoreFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StoreFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StoreFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StoreFieldNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject Value.inject(3)
option.distinct(1) option.inject)
next
  case (StaticStoreFieldNode nid f newval uv nxt m val h′ h m′)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StaticStoreFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StaticStoreFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StaticStoreFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StoreFieldNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject StaticStoreFieldNode.hyps(1) StaticStoreFieldNode.hyps(2) StaticStoreFieldNode.hyps(3)
StaticStoreFieldNode.hyps(4) StaticStoreFieldNode.hyps(5) option.distinct(1))
next
  case (SignedDivNode nid x y zero sb nxt m v1 v2 v m′ h)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: SignedDivNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: SignedDivNode.hyps(1))
  from notseq notend
  show ?case using SignedDivNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1091) IRNode.distinct(1739) IRNode.distinct(2311)
IRNode.distinct(2601) IRNode.distinct(2605) IRNode.inject(40) Pair-inject)
next
  case (SignedRemNode nid x y zero sb nxt m v1 v2 v m′ h)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: SignedRemNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: SignedRemNode.hyps(1))
  from notseq notend
  show ?case using SignedRemNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1093) IRNode.distinct(1741) IRNode.distinct(2313)
IRNode.distinct(2601) IRNode.distinct(2627) IRNode.inject(41) Pair-inject)
qed

lemma stepRefNode:
  ⟦kind g nid = RefNode nid'⟧ ⟹ g, p ⊢ (nid,m,h) → (nid',m,h)
  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
```

8 Proof Infrastructure

8.1 Bisimulation

```
theory Bisimulation
imports
```

Stuttering
begin

inductive *weak-bisimilar* :: $\mathit{ID} \Rightarrow \mathit{IRGraph} \Rightarrow \mathit{IRGraph} \Rightarrow \mathit{bool}$
(- . - ∼ -) **for** *nid* **where**
$[\![\forall P'.\ (g\ m\ p\ h \vdash \mathit{nid} \leadsto P') \longrightarrow (\exists\, Q'\ .\ (g'\ m\ p\ h \vdash \mathit{nid} \leadsto Q') \wedge P' = Q');$
$\forall\, Q'.\ (g'\ m\ p\ h \vdash \mathit{nid} \leadsto Q') \longrightarrow (\exists\, P'\ .\ (g\ m\ p\ h \vdash \mathit{nid} \leadsto P') \wedge P' = Q')]\!]$
$\Longrightarrow \mathit{nid}\ .\ g \sim g'$

A strong bisimilution between no-op transitions

inductive *strong-noop-bisimilar* :: $\mathit{ID} \Rightarrow \mathit{IRGraph} \Rightarrow \mathit{IRGraph} \Rightarrow \mathit{bool}$
(- | - ∼ -) **for** *nid* **where**
$[\![\forall P'.\ (g,\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow P') \longrightarrow (\exists\, Q'\ .\ (g',\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow Q') \wedge P' = Q');$
$\forall\, Q'.\ (g',\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow Q') \longrightarrow (\exists\, P'\ .\ (g,\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow P') \wedge P' = Q')]\!]$
$\Longrightarrow \mathit{nid}\ |\ g \sim g'$

lemma *lockstep-strong-bisimilulation*:
assumes $g' = \mathit{replace\text{-}node}\ \mathit{nid}\ \mathit{node}\ g$
assumes $g,\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow (\mathit{nid}',\ m,\ h)$
assumes $g',\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow (\mathit{nid}',\ m,\ h)$
shows $\mathit{nid}\ |\ g \sim g'$
using *assms*(*2*) *assms*(*3*) *stepDet strong-noop-bisimilar.simps* **by** *metis*

lemma *no-step-bisimulation*:
assumes $\forall\, m\ p\ h\ \mathit{nid}'\ m'\ h'.\ \neg(g,\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow (\mathit{nid}',\ m',\ h'))$
assumes $\forall\, m\ p\ h\ \mathit{nid}'\ m'\ h'.\ \neg(g',\ p \vdash (\mathit{nid},\ m,\ h) \rightarrow (\mathit{nid}',\ m',\ h'))$
shows $\mathit{nid}\ |\ g \sim g'$
using *assms*
by (*simp add*: *assms*(*1*) *assms*(*2*) *strong-noop-bisimilar.intros*)

end

8.2 Graph Rewriting

theory
Rewrites
imports
Semantics.IRGraphFrames
Stuttering
begin

fun *replace-usages* :: $\mathit{ID} \Rightarrow \mathit{ID} \Rightarrow \mathit{IRGraph} \Rightarrow \mathit{IRGraph}$ **where**
$\mathit{replace\text{-}usages}\ \mathit{nid}\ \mathit{nid}'\ g = \mathit{replace\text{-}node}\ \mathit{nid}\ (\mathit{RefNode}\ \mathit{nid}',\ \mathit{stamp}\ g\ \mathit{nid}')\ g$

lemma *replace-usages-effect*:

assumes *g′ = replace-usages nid nid′ g*
shows *kind g′ nid = RefNode nid′*
using *assms replace-node-lookup replace-usages.simps*
by (*metis IRNode.distinct*(*2755*))

lemma *replace-usages-changeonly*:
assumes *nid ∈ ids g*
assumes *g′ = replace-usages nid nid′ g*
shows *changeonly* {*nid*} *g g′*
using *assms* **unfolding** *replace-usages.simps*
by (*metis add-changed add-node-def replace-node-def*)

lemma *replace-usages-unchanged*:
assumes *nid ∈ ids g*
assumes *g′ = replace-usages nid nid′ g*
shows *unchanged* (*ids g* − {*nid*}) *g g′*
using *assms* **unfolding** *replace-usages.simps*
using *assms*(*2*) *disjoint-change replace-usages-changeonly* **by** *presburger*

fun *nextNid* :: *IRGraph ⇒ ID* **where**
nextNid g = (*Max* (*ids g*)) + *1*

lemma *max-plus-one*:
fixes *c* :: *ID set*
shows ⟦*finite c*; *c* ≠ {}⟧ ⟹ (*Max c*) + *1* ∉ *c*
by (*meson Max-gr-iff less-add-one less-irrefl*)

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

lemma *nextNidNotIn*:
ids g ≠ {} ⟶ *nextNid g* ∉ *ids g*
unfolding *nextNid.simps*
using *ids-finite max-plus-one* **by** *blast*

fun *constantCondition* :: *bool ⇒ ID ⇒ IRNode ⇒ IRGraph ⇒ IRGraph* **where**
constantCondition val nid (*IfNode cond t f*) *g* =
replace-node nid (*IfNode* (*nextNid g*) *t f*, *stamp g nid*)
(*add-node* (*nextNid g*) ((*ConstantNode* (*bool-to-val val*)), *constantAsStamp* (*bool-to-val val*)) *g*) |
constantCondition cond nid - g = *g*

lemma *constantConditionTrue*:
assumes *kind g ifcond* = *IfNode cond t f*
assumes *g′* = *constantCondition True ifcond* (*kind g ifcond*) *g*
shows *g′*, *p* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)

proof –

have *ifn*: ⋀ *c t f*. *IfNode c t f* ≠ *NoNode*

by *simp*

then have *if′*: *kind g′ ifcond* = *IfNode* (*nextNid g*) *t f*

using *assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *replace-node-lookup*

by *presburger*

have *truedef*: *bool-to-val True* = (*IntVal32 1*)

by *auto*

from *ifn* **have** *ifcond* ≠ (*nextNid g*)

by (*metis assms*(*1*) *emptyE ids-some nextNidNotIn*)

moreover have ⋀ *c*. *ConstantNode c* ≠ *NoNode* **by** *simp*

ultimately have *kind g′* (*nextNid g*) = *ConstantNode* (*bool-to-val True*)

using *add-changed add-node-def assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *not-in-g other-node-unchanged replace-node-def replace-node-lookup singletonD*

by (*smt* (*z3*) *DiffI add-node-lookup replace-node-unchanged*)

then have *c′*: *kind g′* (*nextNid g*) = *ConstantNode* (*IntVal32 1*)

using *truedef* **by** *simp*

have *valid-value* (*IntVal32 1*) (*constantAsStamp* (*IntVal32 1*))

unfolding *constantAsStamp.simps valid-value.simps*

using *nat-numeral* **by** *blast*

then have [*g′*, *m*, *p*] ⊢ *nextNid g* ↦ *IntVal32 1*

using *ConstantExpr ConstantNode Value.distinct*(*1*) ‹*kind g′* (*nextNid g*) = *ConstantNode* (*bool-to-val True*)› *encodeeval-def truedef*

by *metis*

from *if′ c′* **show** *?thesis* **using** *IfNode*

by (*metis* (*no-types*, *opaque-lifting*) *val-to-bool.simps*(*1*) ‹[*g′*,*m*,*p*] ⊢ *nextNid g* ↦ *IntVal32 1*› *encodeeval-def zero-neq-one*)

qed

lemma *constantConditionFalse*:

assumes *kind g ifcond* = *IfNode cond t f*

assumes *g′* = *constantCondition False ifcond* (*kind g ifcond*) *g*

shows *g′*, *p* ⊢ (*ifcond*, *m*, *h*) → (*f*, *m*, *h*)

proof –

have *ifn*: ⋀ *c t f*. *IfNode c t f* ≠ *NoNode*

by *simp*

then have *if′*: *kind g′ ifcond* = *IfNode* (*nextNid g*) *t f*

by (*metis assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *replace-node-lookup*)

have *falsedef*: *bool-to-val False* = (*IntVal32 0*)

by *auto*

from *ifn* **have** *ifcond* ≠ (*nextNid g*)

by (*metis assms*(*1*) *equals0D ids-some nextNidNotIn*)

moreover have ⋀ *c*. *ConstantNode c* ≠ *NoNode* **by** *simp*

ultimately have *kind g′* (*nextNid g*) = *ConstantNode* (*bool-to-val False*)

by (*smt* (*z3*) *add-changed add-node-def assms*(*1*) *assms*(*2*) *constantCondition.simps*(*1*) *not-in-g other-node-unchanged replace-node-def replace-node-lookup singletonD*)

then have *c′*: *kind g′* (*nextNid g*) = *ConstantNode* (*IntVal32 0*)

using *falsedef* **by** *simp*

```
  have valid-value (IntVal32 0) (constantAsStamp (IntVal32 0))
    unfolding constantAsStamp.simps valid-value.simps
    using nat-numeral by blast
  then have [g′, m, p] ⊢ nextNid g ↦ IntVal32 0
    by (metis ConstantExpr ConstantNode ‹kind g′ (nextNid g) = ConstantNode (bool-to-val False)› encodeeval-def falsedef)
  from if′ c′ show ?thesis using IfNode
    by (metis (no-types, opaque-lifting) val-to-bool.simps(1) ‹[g′,m,p] ⊢ nextNid g ↦ IntVal32 0› encodeeval-def)
qed

lemma diff-forall:
  assumes ∀ n∈ids g − {nid}. cond n
  shows ∀ n. n ∈ ids g ∧ n ∉ {nid} ⟶ cond n
  by (meson Diff-iff assms)

lemma replace-node-changeonly:
  assumes g′ = replace-node nid node g
  shows changeonly {nid} g g′
  using assms replace-node-unchanged
  unfolding changeonly.simps using diff-forall
  by (metis add-changed add-node-def changeonly.simps replace-node-def)

lemma add-node-changeonly:
  assumes g′ = add-node nid node g
  shows changeonly {nid} g g′
   by (metis Rep-IRGraph-inverse add-node.rep-eq assms replace-node.rep-eq replace-node-changeonly)

lemma constantConditionNoEffect:
  assumes ¬(is-IfNode (kind g nid))
  shows g = constantCondition b nid (kind g nid) g
  using assms apply (cases kind g nid)
  using constantCondition.simps
  apply presburger+
  apply (metis is-IfNode-def)
  using constantCondition.simps
  by presburger+

lemma constantConditionIfNode:
  assumes kind g nid = IfNode cond t f
  shows constantCondition val nid (kind g nid) g =
    replace-node nid (IfNode (nextNid g) t f, stamp g nid)
      (add-node (nextNid g) ((ConstantNode (bool-to-val val)), constantAsStamp (bool-to-val val)) g)
  using constantCondition.simps
  by (simp add: assms)

lemma constantCondition-changeonly:
  assumes nid ∈ ids g
  assumes g' = constantCondition b nid (kind g nid) g
  shows changeonly {nid} g g'
proof (cases is-IfNode (kind g nid))
  case True
  have nextNid g ∉ ids g
    using nextNidNotIn by (metis emptyE)
  then show ?thesis using assms
   using replace-node-changeonly add-node-changeonly unfolding changeonly.simps
    using True constantCondition.simps(1) is-IfNode-def
    by (metis (no-types, lifting) insert-iff)
next
  case False
  have g = g'
    using constantConditionNoEffect
    using False assms(2) by blast
  then show ?thesis by simp
qed


lemma constantConditionNoIf:
  assumes ∀ cond t f. kind g ifcond ≠ IfNode cond t f
  assumes g' = constantCondition val ifcond (kind g ifcond) g
  shows ∃ nid' .(g m p h ⊢ ifcond ⇝ nid') ⟷ (g' m p h ⊢ ifcond ⇝ nid')
proof –
  have g' = g
    using assms(2) assms(1)
    using constantConditionNoEffect
    by (metis IRNode.collapse(11))
  then show ?thesis by simp
qed

lemma constantConditionValid:
  assumes kind g ifcond = IfNode cond t f
  assumes [g, m, p] ⊢ cond ↦ v
  assumes const = val-to-bool v
  assumes g' = constantCondition const ifcond (kind g ifcond) g
  shows ∃ nid' .(g m p h ⊢ ifcond ⇝ nid') ⟷ (g' m p h ⊢ ifcond ⇝ nid')
proof (cases const)
  case True
  have ifstep: g, p ⊢ (ifcond, m, h) → (t, m, h)
    by (meson IfNode True assms(1) assms(2) assms(3) encodeeval-def)
  have ifstep': g', p ⊢ (ifcond, m, h) → (t, m, h)
    using constantConditionTrue
    using True assms(1) assms(4) by presburger
  from ifstep ifstep' show ?thesis
    using StutterStep by blast
next
  case False
  have ifstep: g, p ⊢ (ifcond, m, h) → (f, m, h)
    by (meson IfNode False assms(1) assms(2) assms(3) encodeeval-def)
  have ifstep': g', p ⊢ (ifcond, m, h) → (f, m, h)
    using constantConditionFalse
    using False assms(1) assms(4) by presburger
  from ifstep ifstep' show ?thesis
    using StutterStep by blast
qed

end
```

8.3 Stuttering

```
theory Stuttering
  imports
    Semantics.IRStepThms
begin

inductive stutter:: IRGraph ⇒ MapState ⇒ Params ⇒ FieldRefHeap ⇒ ID ⇒
ID ⇒ bool (- - - - ⊢ - ⇝ - 55)
  for g m p h where

  StutterStep:
  ⟦g, p ⊢ (nid,m,h) → (nid',m,h)⟧
   ⟹ g m p h ⊢ nid ⇝ nid' |

  Transitive:
  ⟦g, p ⊢ (nid,m,h) → (nid'',m,h);
    g m p h ⊢ nid'' ⇝ nid'⟧
   ⟹ g m p h ⊢ nid ⇝ nid'

lemma stuttering-successor:
  assumes (g, p ⊢ (nid, m, h) → (nid', m, h))
  shows {P'. (g m p h ⊢ nid ⇝ P')} = {nid'} ∪ {nid''. (g m p h ⊢ nid' ⇝ nid'')}
proof −
  have nextin: nid' ∈ {P'. (g m p h ⊢ nid ⇝ P')}
    using assms StutterStep by blast
  have nextsubset: {nid''. (g m p h ⊢ nid' ⇝ nid'')} ⊆ {P'. (g m p h ⊢ nid ⇝ P')}
    by (metis Collect-mono assms stutter.Transitive)
  have ∀ n ∈ {P'. (g m p h ⊢ nid ⇝ P')} . n = nid' ∨ n ∈ {nid''. (g m p h ⊢ nid'
⇝ nid'')}
    using stepDet
    by (metis (no-types, lifting) Pair-inject assms mem-Collect-eq stutter.simps)
  then show ?thesis
    using insert-absorb mk-disjoint-insert nextin nextsubset by auto
qed

end
```

8.4 Evaluation Stamp Theorems

theory StampEvalThms
imports Semantics.IRTreeEvalThms
begin

8.4.1 Evaluated Value Satisfies Stamps

lemma size32: size v = 32 **for** v :: 32 word
using size-word.rep-eq
using One-nat-def add.right-neutral add-Suc-right len-of-numeral-defs(2) len-of-numeral-defs(3) mult.right-neutral mult-Suc-right numeral-2-eq-2 numeral-Bit0
by (smt (verit, del-insts) mult.commute)

lemma size64: size v = 64 **for** v :: 64 word
using size-word.rep-eq
using One-nat-def add.right-neutral add-Suc-right len-of-numeral-defs(2) len-of-numeral-defs(3) mult.right-neutral mult-Suc-right numeral-2-eq-2 numeral-Bit0
by (smt (verit, del-insts) mult.commute)

lemma signed-int-bottom32: −(((2::int) ^ 31)) ≤ sint (v::int32)
using sint-range-size size32
by (smt (verit, ccfv-SIG) One-nat-def Suc-pred add-Suc add-Suc-right eval-nat-numeral(3) nat.inject numeral-2-eq-2 numeral-Bit0 numeral-Bit1 zero-less-numeral)

lemma signed-int-top32: (2 ^ 31) − 1 ≥ sint (v::int32)
using sint-range-size size32
by (smt (verit, ccfv-SIG) One-nat-def Suc-pred add-Suc add-Suc-right eval-nat-numeral(3) nat.inject numeral-2-eq-2 numeral-Bit0 numeral-Bit1 zero-less-numeral)

lemma lower-bounds-equiv32: −(((2::int) ^ 31)) = (2::int) ^ 32 div 2 * − 1
by fastforce

lemma upper-bounds-equiv32: (2::int) ^ 31 = (2::int) ^ 32 div 2
by simp

lemma bit-bounds-min32: ((fst (bit-bounds 32))) ≤ (sint (v::int32))
unfolding bit-bounds.simps fst-def **using** signed-int-bottom32 lower-bounds-equiv32
by auto

lemma bit-bounds-max32: ((snd (bit-bounds 32))) ≥ (sint (v::int32))
unfolding bit-bounds.simps fst-def **using** signed-int-top32 upper-bounds-equiv32
by auto

lemma signed-int-bottom64: −(((2::int) ^ 63)) ≤ sint (v::int64)
using sint-range-size size64
by (smt (verit, ccfv-SIG) One-nat-def Suc-pred add-Suc add-Suc-right eval-nat-numeral(3) nat.inject numeral-2-eq-2 numeral-Bit0 numeral-Bit1 zero-less-numeral)

lemma signed-int-top64: (2 ^ 63) − 1 ≥ sint (v::int64)

```
  using sint-range-size size64
 by (smt (verit, ccfv-SIG) One-nat-def Suc-pred add-Suc add-Suc-right eval-nat-numeral(3)
nat.inject numeral-2-eq-2 numeral-Bit0 numeral-Bit1 zero-less-numeral)

lemma lower-bounds-equiv64: −(((2::int) ^ 63)) = (2::int) ^ 64 div 2 ∗ − 1
  by fastforce

lemma upper-bounds-equiv64: (2::int) ^ 63 = (2::int) ^ 64 div 2
  by simp

lemma bit-bounds-min64: ((fst (bit-bounds 64))) ≤ (sint (v::int64))
 unfolding bit-bounds.simps fst-def using signed-int-bottom64 lower-bounds-equiv64
  by auto

lemma bit-bounds-max64: ((snd (bit-bounds 64))) ≥ (sint (v::int64))
  unfolding bit-bounds.simps fst-def using signed-int-top64 upper-bounds-equiv64
  by auto

lemma unrestricted-32bit-always-valid:
  valid-value (IntVal32 v) (unrestricted-stamp (IntegerStamp 32 lo hi))
  using valid-value.simps(1) bit-bounds-min32 bit-bounds-max32
  using unrestricted-stamp.simps(2) by presburger

lemma unrestricted-64bit-always-valid:
  valid-value (IntVal64 v) (unrestricted-stamp (IntegerStamp 64 lo hi))
  using valid-value.simps(2) bit-bounds-min64 bit-bounds-max64
  using unrestricted-stamp.simps(2) by presburger

lemma unary-undef: val = UndefVal ⟹ unary-eval op val = UndefVal
  by (cases op; auto)

lemma unary-obj: val = ObjRef x ⟹ unary-eval op val = UndefVal
  by (cases op; auto)

lemma unary-eval-implies-valid-value:
  assumes [m,p] ⊢ expr ↦ val
  assumes result = unary-eval op val
  assumes result ≠ UndefVal
  assumes valid-value val (stamp-expr expr)
  shows valid-value result (stamp-expr (UnaryExpr op expr))
proof −
  have is-IntVal: ∃ x y. result = IntVal32 x ∨ result = IntVal64 y
    using assms(2,3) apply (cases op; auto; cases val; auto)
    apply metis
    by metis
  then have is-IntegerStamp (stamp-expr expr)
    using assms(2,3,4) apply (cases (stamp-expr expr); auto)
    using valid-VoidStamp unary-undef apply simp
    using valid-VoidStamp unary-undef apply simp
```

using *valid-ObjStamp unary-obj* **apply** *fastforce*

using *valid-ObjStamp unary-obj* **by** *fastforce*

then obtain *b lo hi* **where** *stamp-expr-def*: *stamp-expr expr* = (*IntegerStamp b lo hi*)

using *is-IntegerStamp-def* **by** *auto*

then have *stamp-expr* (*UnaryExpr op expr*) = *unrestricted-stamp* (*IntegerStamp b lo hi*)

using *stamp-expr.simps*(*1*) *stamp-unary.simps*(*1*) **by** *presburger*

from *stamp-expr-def* **have** *bit32*: *b* = *32* ⟹ ∃ *x*. *result* = *IntVal32 x*

using *assms*(*2*,*3*,*4*) **by** (*cases op*; *auto*; *cases val*; *auto*)

from *stamp-expr-def* **have** *bit64*: *b* = *64* ⟹ ∃ *x*. *result* = *IntVal64 x*

using *assms*(*2*,*3*,*4*) **by** (*cases op*; *auto*; *cases val*; *auto*)

show *?thesis* **using** *valid-value.simps*(*1*,*2*)

unrestricted-32bit-always-valid unrestricted-64bit-always-valid stamp-expr-def

bit32 bit64

by (*metis* ‹*stamp-expr* (*UnaryExpr op expr*) = *unrestricted-stamp* (*IntegerStamp b lo hi*)› *assms*(*4*) *valid32or64-both*)

qed

lemma *binary-undef*: *v1* = *UndefVal* ∨ *v2* = *UndefVal* ⟹ *bin-eval op v1 v2* = *UndefVal*

by (*cases op*; *auto*)

lemma *binary-obj*: *v1* = *ObjRef x* ∨ *v2* = *ObjRef y* ⟹ *bin-eval op v1 v2* = *UndefVal*

by (*cases op*; *auto*)

lemma *binary-eval-bits-equal*:

assumes *result* = *bin-eval op val1 val2*

assumes *result* ≠ *UndefVal*

assumes *valid-value val1* (*IntegerStamp b1 lo1 hi1*)

assumes *valid-value val2* (*IntegerStamp b2 lo2 hi2*)

shows *b1* = *b2*

using *assms*

by (*cases op*; *cases val1*; *cases val2*; *auto*)

lemma *binary-eval-values*:

assumes ∃ *x y*. *result* = *IntVal32 x* ∨ *result* = *IntVal64 y*

assumes *result* = *bin-eval op val1 val2*

shows ∃ *x32 x64 y32 y64*. *val1* = *IntVal32 x32* ∧ *val2* = *IntVal32 y32* ∨ *val1* = *IntVal64 x64* ∧ *val2* = *IntVal64 y64*

using *assms* **apply** (*cases result*)

apply *simp* **apply** (*cases op*; *cases val1*; *cases val2*; *auto*)

apply (*cases op*; *cases val1*; *cases val2*; *auto*) **by** *auto+*

lemma *binary-eval-implies-valid-value*:

assumes [*m*,*p*] ⊢ *expr1* ↦ *val1*

assumes [*m*,*p*] ⊢ *expr2* ↦ *val2*

```
  assumes result = bin-eval op val1 val2
  assumes result ≠ UndefVal
  assumes valid-value val1 (stamp-expr expr1)
  assumes valid-value val2 (stamp-expr expr2)
  shows valid-value result (stamp-expr (BinaryExpr op expr1 expr2))
proof −
  have is-IntVal: ∃ x y. result = IntVal32 x ∨ result = IntVal64 y
    using assms(1,2,3,4) apply (cases op; auto; cases val1; auto; cases val2; auto)
    by (meson Values.bool-to-val.elims)+
  then have expr1-intstamp: is-IntegerStamp (stamp-expr expr1)
   using assms(1,3,4,5) apply (cases (stamp-expr expr1); auto simp: valid-VoidStamp
binary-undef)
    using valid-ObjStamp binary-obj apply (metis assms(4))
    using valid-ObjStamp binary-obj by (metis assms(4))
  from is-IntVal have expr2-intstamp: is-IntegerStamp (stamp-expr expr2)
   using assms(2,3,4,6) apply (cases (stamp-expr expr2); auto simp: valid-VoidStamp
binary-undef)
    using valid-ObjStamp binary-obj apply (metis assms(4))
    using valid-ObjStamp binary-obj by (metis assms(4))
 from expr1-intstamp obtain b1 lo1 hi1 where stamp-expr1-def: stamp-expr expr1
= (IntegerStamp b1 lo1 hi1)
    using is-IntegerStamp-def by auto
   from expr2-intstamp obtain b2 lo2 hi2 where stamp-expr2-def: stamp-expr
expr2 = (IntegerStamp b2 lo2 hi2)
    using is-IntegerStamp-def by auto

  have ∃ x32 x64 y32 y64 . (val1 = IntVal32 x32 ∧ val2 = IntVal32 y32) ∨ (val1
= IntVal64 x64 ∧ val2 = IntVal64 y64)
    using is-IntVal assms(3) binary-eval-values
    by presburger

  have b1 = b2
    using assms(3,4,5,6) stamp-expr1-def stamp-expr2-def
    using binary-eval-bits-equal
    by auto
  then have stamp-def: stamp-expr (BinaryExpr op expr1 expr2) =
     (case op ∈ fixed-32 of True ⇒ unrestricted-stamp (IntegerStamp 32 lo1 hi1)|
False ⇒ unrestricted-stamp (IntegerStamp b1 lo1 hi1))
    using stamp-expr.simps(2) stamp-binary.simps(1)
    using stamp-expr1-def stamp-expr2-def by presburger
  from stamp-expr1-def have bit32: b1 = 32 ⟹ ∃ x. result = IntVal32 x
    using assms apply (cases op; cases val1; cases val2; auto)
    by (meson Values.bool-to-val.elims)+
  from stamp-expr1-def have bit64: b1 = 64 ∧ op ∉ fixed-32 ⟹ ∃ x y. result =
IntVal64 x
    using assms apply (cases op; cases val1; cases val2; simp)
    using fixed-32-def by auto+
  from stamp-expr1-def have fixed: op ∈ fixed-32 ⟹ ∃ x y. result = IntVal32 x
    using assms unfolding fixed-32-def apply (cases op; auto)
    apply (cases val1; cases val2; auto)
    using bit32 apply fastforce
      apply (meson Values.bool-to-val.elims)
     apply (cases val1; cases val2; auto)
    using bit32 apply fastforce
     apply (meson Values.bool-to-val.elims)
      apply (cases val1; cases val2; auto)
    using bit32 apply fastforce
    by (meson Values.bool-to-val.elims)

  show ?thesis apply (cases op ∈ fixed-32) defer using valid-value.simps(1,2)
    unrestricted-32bit-always-valid unrestricted-64bit-always-valid stamp-expr1-def
    bit32 bit64 stamp-def apply auto
    using ‹∃ x32 x64 y32 y64. val1 = IntVal32 x32 ∧ val2 = IntVal32 y32 ∨ val1
= IntVal64 x64 ∧ val2 = IntVal64 y64› assms(5) apply auto[1]
    using fixed by force
qed

lemma stamp-meet-is-valid:
  assumes valid-value val stamp1 ∨ valid-value val stamp2
  assumes meet stamp1 stamp2 ≠ IllegalStamp
  shows valid-value val (meet stamp1 stamp2)
  using assms proof (cases stamp1)
  case VoidStamp
  then show ?thesis
      by (metis Stamp.exhaust assms(1) assms(2) meet.simps(1) meet.simps(37)
meet.simps(44) meet.simps(51) meet.simps(58) meet.simps(65) meet.simps(66) meet.simps(67))
next
  case (IntegerStamp b lo hi)
  obtain b2 lo2 hi2 where stamp2-def: stamp2 = IntegerStamp b2 lo2 hi2
   by (metis IntegerStamp assms(2) meet.simps(45) meet.simps(52) meet.simps(59)
meet.simps(6) meet.simps(65) meet.simps(66) meet.simps(67) unrestricted-stamp.cases)
  then have b = b2 using meet.simps(2) assms(2)
    by (metis IntegerStamp)
  then have meet-def: meet stamp1 stamp2 = (IntegerStamp b (min lo lo2) (max
hi hi2))
    by (simp add: IntegerStamp stamp2-def)
  then show ?thesis proof (cases b = 32)
    case True
    then obtain x where val-def: val = IntVal32 x
      using IntegerStamp assms(1) valid32
      using ‹b = b2› stamp2-def by blast
    have min: sint x ≥ min lo lo2
      using val-def
      using IntegerStamp assms(1)
      using stamp2-def by force
    have max: sint x ≤ max hi hi2
      using val-def
      using IntegerStamp assms(1)
      using stamp2-def by force
    from min max show ?thesis
      by (simp add: True meet-def val-def)
  next
    case False
    then have bit64: b = 64
      using assms(1) IntegerStamp valid-value.simps
      valid32or64-both
      by (metis ‹b = b2› stamp2-def)
    then obtain x where val-def: val = IntVal64 x
      using IntegerStamp assms(1) valid64
      using ‹b = b2› stamp2-def by blast
    have min: sint x ≥ min lo lo2
      using val-def
      using IntegerStamp assms(1)
      using stamp2-def by force
    have max: sint x ≤ max hi hi2
      using val-def
      using IntegerStamp assms(1)
      using stamp2-def by force
    from min max show ?thesis
      by (simp add: bit64 meet-def val-def)
  qed
next
  case (KlassPointerStamp x31 x32)
  then show ?thesis using assms valid-value.elims(2)
    by (metis meet.simps(14) valid-value.simps(21))
next
  case (MethodCountersPointerStamp x41 x42)
  then show ?thesis using assms valid-value.elims(2)
    by (metis meet.simps(39) valid-value.simps(22))
next
  case (MethodPointersStamp x51 x52)
then show ?thesis using assms valid-value.elims(2)
  by (metis meet.simps(40) valid-value.simps(23))
next
  case (ObjectStamp x61 x62 x63 x64)
  then show ?thesis using assms
    using meet.simps(34) by blast
next
  case (RawPointerStamp x71 x72)
  then show ?thesis using assms
    using meet.simps(35) by blast
next
  case IllegalStamp
  then show ?thesis using assms
    using meet.simps(36) by blast
qed
```

```
lemma conditional-eval-implies-valid-value:
  assumes [m,p] ⊢ cond ↦ condv
  assumes expr = (if val-to-bool condv then expr1 else expr2)
  assumes [m,p] ⊢ expr ↦ val
  assumes val ≠ UndefVal
  assumes valid-value condv (stamp-expr cond)
  assumes valid-value val (stamp-expr expr)
  assumes compatible (stamp-expr expr1) (stamp-expr expr2)
  shows valid-value val (stamp-expr (ConditionalExpr cond expr1 expr2))
proof –
  have meet (stamp-expr expr1) (stamp-expr expr2) ≠ IllegalStamp
    using assms
   by (metis Stamp.distinct(13) Stamp.distinct(25) compatible.elims(2) meet.simps(1)
meet.simps(2))
  then show ?thesis using stamp-meet-is-valid using stamp-expr.simps(6)
    using assms(2) assms(6) by presburger
qed

experiment begin
lemma stamp-implies-valid-value:
  assumes [m,p] ⊢ expr ↦ val
  shows valid-value val (stamp-expr expr)
  using assms proof (induction expr val)
case (UnaryExpr expr val result op)
  then show ?case using unary-eval-implies-valid-value by simp
next
  case (BinaryExpr expr1 val1 expr2 val2 result op)
  then show ?case using binary-eval-implies-valid-value by simp
next
  case (ConditionalExpr cond condv expr expr1 expr2 val)
  then show ?case using conditional-eval-implies-valid-value sorry
next
  case (ParameterExpr x1 x2)
  then show ?case by auto
next
  case (LeafExpr x1 x2)
  then show ?case by auto
next
  case (ConstantExpr x)
  then show ?case by auto
qed

lemma value-range:
  assumes [m, p] ⊢ e ↦ v
  shows v ∈ {val . valid-value val (stamp-expr e)}
  using assms sorry
end
```

```
lemma upper-bound-32:
  assumes val = IntVal32 v
  assumes ∃ l h. s = (IntegerStamp 32 l h)
  shows valid-value val s ⟹ sint v ≤ (stpi-upper s)
  using assms by force

lemma upper-bound-64:
  assumes val = IntVal64 v
  assumes ∃ l h. s = (IntegerStamp 64 l h)
  shows valid-value val s ⟹ sint v ≤ (stpi-upper s)
  using assms by force

lemma lower-bound-32:
  assumes val = IntVal32 v
  assumes ∃ l h. s = (IntegerStamp 32 l h)
  shows valid-value val s ⟹ sint v ≥ (stpi-lower s)
  using assms by force

lemma lower-bound-64:
  assumes val = IntVal64 v
  assumes ∃ l h. s = (IntegerStamp 64 l h)
  shows valid-value val s ⟹ sint v ≥ (stpi-lower s)
  using assms
  by force


lemma stamp-under-semantics:
  assumes stamp-under (stamp-expr x) (stamp-expr y)
  assumes [m, p] ⊢ (BinaryExpr BinIntegerLessThan x y) ↦ v
  assumes xvalid: (∀ m p v. ([m, p] ⊢ x ↦ v) ⟶ valid-value v (stamp-expr x))
  assumes yvalid: (∀ m p v. ([m, p] ⊢ y ↦ v) ⟶ valid-value v (stamp-expr y))
  shows val-to-bool v
proof −
  obtain xval where xval-def: [m, p] ⊢ x ↦ xval
    using assms(2) by blast
  obtain yval where yval-def: [m, p] ⊢ y ↦ yval
    using assms(2) by blast
  have is-IntVal32 xval ∨ is-IntVal64 xval
      by (metis BinaryExprE Value.exhaust-disc assms(2) bin-eval.simps(11) binary-obj binary-undef evalDet intval-less-than.simps(9) is-ObjRef-def is-ObjStr-def xval-def)
  have is-IntVal32 yval ∨ is-IntVal64 yval
      by (metis BinaryExprE Value.exhaust-disc assms(2) bin-eval.simps(11) binary-obj binary-undef evalDet intval-less-than.simps(16) is-ObjRef-def is-ObjStr-def yval-def)
  have is-IntVal32 xval = is-IntVal32 yval
    by (metis BinaryExprE Value.collapse(2) ‹is-IntVal32 xval ∨ is-IntVal64 xval› ‹is-IntVal32 yval ∨ is-IntVal64 yval› assms(2) bin-eval.simps(11) evalDet intval-less-than.simps(12) intval-less-than.simps(5) is-IntVal32-def xval-def yval-def)
  have is-IntVal64 xval = is-IntVal64 yval
   using ‹is-IntVal32 xval = is-IntVal32 yval› ‹is-IntVal32 xval ∨ is-IntVal64 xval›
‹is-IntVal32 yval ∨ is-IntVal64 yval› by blast
  have (intval-less-than xval yval) ≠ UndefVal
    using assms(2)
    by (metis BinaryExprE bin-eval.simps(11) evalDet xval-def yval-def)
  have is-IntVal32 xval ⟹ ((∃ lo hi. stamp-expr x = IntegerStamp 32 lo hi) ∧
(∃ lo hi. stamp-expr y = IntegerStamp 32 lo hi))
    using assms(2) binary-eval-bits-equal valid-value.elims(2) xval-def
   by (metis Value.distinct(9) Value.distinct-disc(9) ‹is-IntVal32 yval ∨ is-IntVal64
yval› ‹is-IntVal64 xval = is-IntVal64 yval› is-IntVal32-def xvalid yval-def yvalid)
  have is-IntVal64 xval ⟹ ((∃ lo hi. stamp-expr x = IntegerStamp 64 lo hi) ∧
(∃ lo hi. stamp-expr y = IntegerStamp 64 lo hi))
     by (metis (full-types) ‹is-IntVal64 xval = is-IntVal64 yval› is-IntVal64-def
stamprange valid-value.simps(2) xval-def xvalid yval-def yvalid)
  have xvalid: valid-value xval (stamp-expr x)
    using xvalid xval-def by auto
  have yvalid: valid-value yval (stamp-expr y)
    using yvalid yval-def by auto
  { assume c: is-IntVal32 xval
    obtain xxval where x32: xval = IntVal32 xxval
      using c is-IntVal32-def by blast
    obtain yyval where y32: yval = IntVal32 yyval
      using ‹is-IntVal32 xval = is-IntVal32 yval› c is-IntVal32-def by auto
    have xs: ∃ lo hi. stamp-expr x = IntegerStamp 32 lo hi
     by (simp add: ‹is-IntVal32 xval ⟹ (∃lo hi. stamp-expr x = IntegerStamp 32
lo hi) ∧ (∃lo hi. stamp-expr y = IntegerStamp 32 lo hi)› c)
    have ys: ∃ lo hi. stamp-expr y = IntegerStamp 32 lo hi
      using ‹is-IntVal32 xval ⟹ (∃lo hi. stamp-expr x = IntegerStamp 32 lo hi)
∧ (∃lo hi. stamp-expr y = IntegerStamp 32 lo hi)› c by blast
    have sint xxval ≤ stpi-upper (stamp-expr x)
      using upper-bound-32 x32 xs xvalid by presburger
    have stpi-lower (stamp-expr y) ≤ sint yyval
      using lower-bound-32 y32 ys yvalid by presburger
    have stpi-upper (stamp-expr x) < stpi-lower (stamp-expr y)
      using assms(1) unfolding stamp-under.simps
      by auto
    then have xxval <s yyval
      using assms(1) unfolding stamp-under.simps
      using ‹sint xxval ⊑ stpi-upper (stamp-expr x)› ‹stpi-lower (stamp-expr y) ⊑
sint yyval› word-sless-alt by fastforce
    then have (intval-less-than xval yval) = IntVal32 1
      by (simp add: x32 y32)
  }
  note case32 = this
  { assume c: is-IntVal64 xval
    obtain xxval where x64: xval = IntVal64 xxval
      using c is-IntVal64-def by blast
    obtain yyval where y64: yval = IntVal64 yyval
```

```
      using ‹is-IntVal64 xval = is-IntVal64 yval› c is-IntVal64-def by auto
    have xs: ∃ lo hi. stamp-expr x = IntegerStamp 64 lo hi
      by (simp add: ‹is-IntVal64 xval ⟹ (∃lo hi. stamp-expr x = IntegerStamp 64 lo hi) ∧ (∃lo hi. stamp-expr y = IntegerStamp 64 lo hi)› c)
    have ys: ∃ lo hi. stamp-expr y = IntegerStamp 64 lo hi
      using ‹is-IntVal64 xval ⟹ (∃lo hi. stamp-expr x = IntegerStamp 64 lo hi) ∧ (∃lo hi. stamp-expr y = IntegerStamp 64 lo hi)› c by blast
    have sint xxval ≤ stpi-upper (stamp-expr x)
      using upper-bound-64 x64 xs xvalid by presburger
    have stpi-lower (stamp-expr y) ≤ sint yyval
      using lower-bound-64 y64 ys yvalid by presburger
    have stpi-upper (stamp-expr x) < stpi-lower (stamp-expr y)
      using assms(1) unfolding stamp-under.simps
      by auto
    then have xxval <s yyval
      using assms(1) unfolding stamp-under.simps
      using ‹sint xxval ⊑ stpi-upper (stamp-expr x)› ‹stpi-lower (stamp-expr y) ⊑ sint yyval› word-sless-alt by fastforce
    then have (intval-less-than xval yval) = IntVal32 1
      by (simp add: x64 y64)
  }
  note case64 = this
  have (intval-less-than xval yval) = IntVal32 1
    using ‹is-IntVal32 xval ∨ is-IntVal64 xval› case32 case64 by fastforce
  then show ?thesis
   by (metis BinaryExprE assms(2) bin-eval.simps(11) evalDet val-to-bool.simps(1) xval-def yval-def zero-neq-one)
qed

lemma stamp-under-semantics-inversed:
  assumes stamp-under (stamp-expr y) (stamp-expr x)
  assumes [m, p] ⊢ (BinaryExpr BinIntegerLessThan x y) ↦ v
  assumes xvalid: (∀ m p v. ([m, p] ⊢ x ↦ v) ⟶ valid-value v (stamp-expr x))
  assumes yvalid: (∀ m p v. ([m, p] ⊢ y ↦ v) ⟶ valid-value v (stamp-expr y))
  shows ¬(val-to-bool v)
proof –
  obtain xval where xval-def: [m, p] ⊢ x ↦ xval
    using assms(2) by blast
  obtain yval where yval-def: [m, p] ⊢ y ↦ yval
    using assms(2) by blast
  have is-IntVal32 xval ∨ is-IntVal64 xval
    by (metis is-IntVal32-def is-IntVal64-def xvalid valid-value.elims(2) xval-def)
  have is-IntVal32 yval ∨ is-IntVal64 yval
    by (metis is-IntVal32-def is-IntVal64-def yvalid valid-value.elims(2) yval-def)
  have is-IntVal32 xval = is-IntVal32 yval
    by (metis BinaryExprE Value.collapse(2) ‹is-IntVal32 xval ∨ is-IntVal64 xval› ‹is-IntVal32 yval ∨ is-IntVal64 yval› assms(2) bin-eval.simps(11) evalDet intval-less-than.simps(12) intval-less-than.simps(5) is-IntVal32-def xval-def yval-def)
  have is-IntVal64 xval = is-IntVal64 yval
```

```
    using ‹is-IntVal32 xval = is-IntVal32 yval› ‹is-IntVal32 xval ∨ is-IntVal64 xval›
‹is-IntVal32 yval ∨ is-IntVal64 yval› by blast
  have (intval-less-than xval yval) ≠ UndefVal
    using assms(2)
    by (metis BinaryExprE bin-eval.simps(11) evalDet xval-def yval-def)
  have is-IntVal32 xval ⟹ ((∃ lo hi. stamp-expr x = IntegerStamp 32 lo hi) ∧
(∃ lo hi. stamp-expr y = IntegerStamp 32 lo hi))
     by (smt (verit) BinaryExprE Value.discI(2) Value.distinct-disc(9) assms(2)
binary-eval-bits-equal xvalid yvalid valid-value.elims(2) xval-def)
  have is-IntVal64 xval ⟹ ((∃ lo hi. stamp-expr x = IntegerStamp 64 lo hi) ∧
(∃ lo hi. stamp-expr y = IntegerStamp 64 lo hi))
     by (smt (verit, best) BinaryExprE ‹intval-less-than xval yval ≠ UndefVal›
assms(2) binary-eval-bits-equal intval-less-than.simps(5) is-IntVal64-def xvalid yvalid
valid-value.elims(2) yval-def)
  have xvalid: valid-value xval (stamp-expr x)
    using xvalid xval-def by auto
  have yvalid: valid-value yval (stamp-expr y)
    using yvalid yval-def by auto
  { assume c: is-IntVal32 xval
    obtain xxval where x32: xval = IntVal32 xxval
      using c is-IntVal32-def by blast
    obtain yyval where y32: yval = IntVal32 yyval
      using ‹is-IntVal32 xval = is-IntVal32 yval› c is-IntVal32-def by auto
    have xs: ∃ lo hi. stamp-expr x = IntegerStamp 32 lo hi
     by (simp add: ‹is-IntVal32 xval ⟹ (∃ lo hi. stamp-expr x = IntegerStamp 32
lo hi) ∧ (∃ lo hi. stamp-expr y = IntegerStamp 32 lo hi)› c)
    have ys: ∃ lo hi. stamp-expr y = IntegerStamp 32 lo hi
      using ‹is-IntVal32 xval ⟹ (∃ lo hi. stamp-expr x = IntegerStamp 32 lo hi)
∧ (∃ lo hi. stamp-expr y = IntegerStamp 32 lo hi)› c by blast
    have sint yyval ≤ stpi-upper (stamp-expr y)
      using y32 ys yvalid by force
    have stpi-lower (stamp-expr x) ≤ sint xxval
      using x32 xs xvalid by force
    have stpi-upper (stamp-expr y) < stpi-lower (stamp-expr x)
      using assms(1) unfolding stamp-under.simps
      by auto
    then have yyval <s xxval
      using assms(1) unfolding stamp-under.simps
      using ‹sint yyval ⊑ stpi-upper (stamp-expr y)› ‹stpi-lower (stamp-expr x) ⊑
sint xxval› word-sless-alt by fastforce
    then have (intval-less-than xval yval) = IntVal32 0
      using signed.less-not-sym x32 y32 by fastforce
  }
  note case32 = this
  { assume c: is-IntVal64 xval
    obtain xxval where x64: xval = IntVal64 xxval
      using c is-IntVal64-def by blast
    obtain yyval where y64: yval = IntVal64 yyval
      using ‹is-IntVal64 xval = is-IntVal64 yval› c is-IntVal64-def by auto
    have xs: ∃ lo hi. stamp-expr x = IntegerStamp 64 lo hi
      by (simp add: ‹is-IntVal64 xval ⟹ (∃lo hi. stamp-expr x = IntegerStamp 64 lo hi) ∧ (∃lo hi. stamp-expr y = IntegerStamp 64 lo hi)› c)
    have ys: ∃ lo hi. stamp-expr y = IntegerStamp 64 lo hi
      using ‹is-IntVal64 xval ⟹ (∃lo hi. stamp-expr x = IntegerStamp 64 lo hi) ∧ (∃lo hi. stamp-expr y = IntegerStamp 64 lo hi)› c by blast
    have sint yyval ≤ stpi-upper (stamp-expr y)
      using y64 ys yvalid by force
    have stpi-lower (stamp-expr x) ≤ sint xxval
      using x64 xs xvalid by force
    have stpi-upper (stamp-expr y) < stpi-lower (stamp-expr x)
      using assms(1) unfolding stamp-under.simps
      by auto
    then have yyval <s xxval
      using assms(1) unfolding stamp-under.simps
      using ‹sint yyval ⊑ stpi-upper (stamp-expr y)› ‹stpi-lower (stamp-expr x) ⊑ sint xxval› word-sless-alt by fastforce
    then have (intval-less-than xval yval) = IntVal32 0
      using signed.less-imp-triv x64 y64 by fastforce
  }
  note case64 = this
  have (intval-less-than xval yval) = IntVal32 0
    using ‹is-IntVal32 xval ∨ is-IntVal64 xval› case32 case64 by fastforce
  then show ?thesis
    by (metis BinaryExprE assms(2) bin-eval.simps(11) evalDet val-to-bool.simps(1) xval-def yval-def)
qed

end
```

9 Optization DSLs

```
theory Markup
  imports Semantics.IRTreeEval Snippets.Snipping
begin

datatype 'a Rewrite =
  Transform 'a 'a (- ↦ - 10) |
  Conditional 'a 'a bool (- ↦ - when - 70) |
  Sequential 'a Rewrite 'a Rewrite |
  Transitive 'a Rewrite

datatype 'a ExtraNotation =
  ConditionalNotation 'a 'a 'a (- ? - : -) |
  EqualsNotation 'a 'a (- eq -) |
  ConstantNotation 'a (const - 120) |
  TrueNotation (true) |
  FalseNotation (false)
```

ML-file ‹*markup.ML*›

```
ML ‹
structure IRExprTranslator : DSL-TRANSLATION =
struct
fun markup DSL-Tokens.Add = @{term BinaryExpr} $ @{term BinAdd}
  | markup DSL-Tokens.Sub = @{term BinaryExpr} $ @{term BinSub}
  | markup DSL-Tokens.Mul = @{term BinaryExpr} $ @{term BinMul}
  | markup DSL-Tokens.And = @{term BinaryExpr} $ @{term BinAnd}
  | markup DSL-Tokens.Abs = @{term UnaryExpr} $ @{term UnaryAbs}
  | markup DSL-Tokens.Less = @{term BinaryExpr} $ @{term BinIntegerLessThan}
  | markup DSL-Tokens.Equals = @{term BinaryExpr} $ @{term BinIntegerEquals}
  | markup DSL-Tokens.Not = @{term UnaryExpr} $ @{term UnaryLogicNegation}
  | markup DSL-Tokens.Negate = @{term UnaryExpr} $ @{term UnaryNeg}
  | markup DSL-Tokens.LeftShift = @{term BinaryExpr} $ @{term BinLeftShift}
  | markup DSL-Tokens.RightShift = @{term BinaryExpr} $ @{term BinRightShift}
  | markup DSL-Tokens.UnsignedRightShift = @{term BinaryExpr} $ @{term BinURightShift}
  | markup DSL-Tokens.Conditional = @{term ConditionalExpr}
  | markup DSL-Tokens.Constant = @{term ConstantExpr}
  | markup DSL-Tokens.TrueConstant = @{term ConstantExpr (IntVal32 1)}
  | markup DSL-Tokens.FalseConstant = @{term ConstantExpr (IntVal32 0)}
end

structure IntValTranslator : DSL-TRANSLATION =
struct
fun markup DSL-Tokens.Add = @{term intval-add}
  | markup DSL-Tokens.Sub = @{term intval-sub}
  | markup DSL-Tokens.Mul = @{term intval-mul}
  | markup DSL-Tokens.And = @{term intval-and}
  | markup DSL-Tokens.Abs = @{term intval-abs}
  | markup DSL-Tokens.Less = @{term intval-less-than}
  | markup DSL-Tokens.Equals = @{term intval-equals}
  | markup DSL-Tokens.Not = @{term intval-logic-negation}
  | markup DSL-Tokens.Negate = @{term intval-negate}
  | markup DSL-Tokens.LeftShift = @{term intval-left-shift}
  | markup DSL-Tokens.RightShift = @{term intval-right-shift}
  | markup DSL-Tokens.UnsignedRightShift = @{term intval-uright-shift}
  | markup DSL-Tokens.Conditional = @{term intval-conditional}
  | markup DSL-Tokens.Constant = @{term IntVal32}
  | markup DSL-Tokens.TrueConstant = @{term IntVal32 1}
  | markup DSL-Tokens.FalseConstant = @{term IntVal32 0}
end

structure IRExprMarkup = DSL-Markup(IRExprTranslator);
structure IntValMarkup = DSL-Markup(IntValTranslator);
›
```

ir expression translation

syntax *-expandExpr* :: *term* ⇒ *term* (*exp*[-])
parse-translation ‹ [(@{*syntax-const* *-expandExpr*} , *IRExprMarkup.markup-expr*)] ›

value expression translation

syntax *-expandIntVal* :: *term* ⇒ *term* (*val*[-])
parse-translation ‹ [(@{*syntax-const* *-expandIntVal*} , *IntValMarkup.markup-expr*)] ›

ir expression example

value *exp*[(e_1 < e_2) *?* e_1 : e_2]

ConditionalExpr (*BinaryExpr BinIntegerLessThan* e_1 e_2) e_1 e_2

value expression example

value *val*[(e_1 < e_2) *?* e_1 : e_2]

intval-conditional (*intval-less-than* e_1 e_2) e_1 e_2

value *exp*[((e_1 − e_2) + (*const* (*IntVal32 0*)) + e_2) ↦ e_1 *when True*]
value *val*[((e_1 − e_2) + (*const 0*) + e_2) ↦ e_1 *when True*]

end
theory *Phase*
  imports *Main*
begin

ML-file *map.ML*
ML-file *phase.ML*

end

9.1 Canonicalization DSL

theory *Canonicalization*
  imports
    Markup
    Phase
  keywords
    phase :: *thy-decl* **and**
    terminating :: *quasi-command* **and**
    print-phases :: *diag* **and**
    optimization :: *thy-goal-defn*

begin

```
ML ‹
datatype 'a Rewrite =
  Transform of 'a ∗ 'a |
  Conditional of 'a ∗ 'a ∗ term |
  Sequential of 'a Rewrite ∗ 'a Rewrite |
  Transitive of 'a Rewrite

type rewrite = {name: string, rewrite: term Rewrite}

structure RewriteRule : Rule =
struct
type T = rewrite;

fun pretty-rewrite ctxt (Transform (from, to)) =
      Pretty.block [
        Syntax.pretty-term ctxt from,
        Pretty.str ↦ ,
        Syntax.pretty-term ctxt to
      ]
  | pretty-rewrite ctxt (Conditional (from, to, cond)) =
      Pretty.block [
        Syntax.pretty-term ctxt from,
        Pretty.str ↦ ,
        Syntax.pretty-term ctxt to,
        Pretty.str  when ,
        Syntax.pretty-term ctxt cond
      ]
  | pretty-rewrite - - = Pretty.str not implemented

fun pretty ctxt t =
  Pretty.block [
    Pretty.str ((#name t) ^ : ),
    pretty-rewrite ctxt (#rewrite t)
  ]
end

structure RewritePhase = DSL-Phase(RewriteRule);

val - =
  Outer-Syntax.command command-keyword‹phase› enter an optimization phase
   (Parse.binding −−| Parse.$$$ terminating −− Parse.const −−| Parse.begin
     >> (Toplevel.begin-main-target true o RewritePhase.setup));

fun print-phases ctxt =
  let
    val thy = Proof-Context.theory-of ctxt;
    fun print phase = RewritePhase.pretty phase ctxt
  in
    map print (RewritePhase.phases thy)
  end

fun print-optimizations thy =
  print-phases thy |> Pretty.writeln-chunks

val - =
  Outer-Syntax.command command-keyword ‹print-phases›
    print debug information for optimizations
    (Scan.succeed
      (Toplevel.keep (print-optimizations o Toplevel.context-of)));
›
```

ML-file *rewrites.ML*

```
fun rewrite-preservation :: IRExpr Rewrite ⇒ bool where
  rewrite-preservation (Transform x y) = (y ≤ x) |
  rewrite-preservation (Conditional x y cond) = (cond ⟶ (y ≤ x)) |
 rewrite-preservation (Sequential x y) = (rewrite-preservation x ∧ rewrite-preservation
y) |
  rewrite-preservation (Transitive x) = rewrite-preservation x

fun rewrite-termination :: IRExpr Rewrite ⇒ (IRExpr ⇒ nat) ⇒ bool where
  rewrite-termination (Transform x y) trm = (trm x > trm y) |
  rewrite-termination (Conditional x y cond) trm = (cond ⟶ (trm x > trm y)) |
 rewrite-termination (Sequential x y) trm = (rewrite-termination x trm ∧ rewrite-termination
y trm) |
  rewrite-termination (Transitive x) trm = rewrite-termination x trm

fun intval :: Value Rewrite ⇒ bool where
  intval (Transform x y) = (x ≠ UndefVal ∧ y ≠ UndefVal ⟶ x = y) |
  intval (Conditional x y cond) = (cond ⟶ (x = y)) |
  intval (Sequential x y) = (intval x ∧ intval y) |
  intval (Transitive x) = intval x

ML ‹
structure System : RewriteSystem =
struct
val preservation = @{const rewrite-preservation};
val termination = @{const rewrite-termination};
val intval = @{const intval};
end

structure DSL = DSL-Rewrites(System);

val - =
  Outer-Syntax.local-theory-to-proof command-keyword ‹optimization›
    define an optimization and open proof obligation
  (Parse-Spec.thm-name : −− Parse.term
    >> DSL.rewrite-cmd);
›

end
```

10 Canonicalization Phase

```
theory Common
  imports
    OptimizationDSL.Canonicalization
    HOL−Eisbach.Eisbach
begin

fun size :: IRExpr ⇒ nat where
  size (UnaryExpr op e) = (size e) + 1 |
  size (BinaryExpr BinAdd x y) = (size x) + ((size y) ∗ 2) |
  size (BinaryExpr op x y) = (size x) + (size y) |
  size (ConditionalExpr cond t f) = (size cond) + (size t) + (size f) + 2 |
  size (ConstantExpr c) = 1 |
  size (ParameterExpr ind s) = 2 |
  size (LeafExpr nid s) = 2 |
  size (ConstantVar c) = 2 |
  size (VariableExpr x s) = 2

method unfold-optimization =
  (unfold rewrite-preservation.simps, unfold rewrite-termination.simps,
    unfold intval.simps,
    rule conjE, simp, simp del: le-expr-def)
  | (unfold rewrite-preservation.simps, unfold rewrite-termination.simps,
    rule conjE, simp, simp del: le-expr-def)

end
```

10.1 Conditional Expression

```
theory ConditionalPhase
  imports
    Common
    Proofs.StampEvalThms
begin

phase Conditional
  terminating size
begin

lemma negates: is-IntVal32 e ∨ is-IntVal64 e ⟹ val-to-bool (val[e]) ≡ ¬(val-to-bool
(val[¬e]))
  by (smt (verit, best) Value.disc(1) Value.disc(10) Value.disc(4) Value.disc(5)
Value.disc(6) Value.disc(9) intval-logic-negation.elims val-to-bool.simps(1) val-to-bool.simps(2)
zero-neq-one)

optimization negate-condition: ((¬e) ? x : y) ↦ (e ? y : x)
    apply unfold-optimization apply simp using negates
   using ConditionalExprE UnaryExprE intval-logic-negation.elims unary-eval.simps(4)
val-to-bool.simps(1) val-to-bool.simps(2) zero-neq-one
    apply (smt (verit) ConditionalExpr)
    unfolding size.simps by simp

optimization const-true: (true ? x : y) ↦ x
   apply unfold-optimization
   apply force
  unfolding size.simps by simp

optimization const-false: (false ? x : y) ↦ y
   apply unfold-optimization
   apply force
  unfolding size.simps by simp

optimization equal-branches: (e ? x : x) ↦ x
   apply unfold-optimization
   apply force
  unfolding size.simps by auto


definition wff-stamps :: bool where
  wff-stamps = (∀ m p expr val . ([m,p] ⊢ expr ↦ val) ⟶ valid-value val (stamp-expr
expr))

definition wf-stamp :: IRExpr ⇒ bool where
  wf-stamp e = (∀ m p v. ([m, p] ⊢ e ↦ v) ⟶ valid-value v (stamp-expr e))

optimization condition-bounds-x: ((x < y) ? x : y) ↦ x
    when (stamp-under (stamp-expr x) (stamp-expr y) ∧ wf-stamp x ∧ wf-stamp y)
   apply unfold-optimization
  using stamp-under-semantics
  using wf-stamp-def
  apply (smt (verit, best) ConditionalExprE le-expr-def stamp-under.simps)
  unfolding size.simps by simp

optimization condition-bounds-y: ((x < y) ? x : y) ↦ y
    when (stamp-under (stamp-expr y) (stamp-expr x) ∧ wf-stamp x ∧ wf-stamp y)
   apply unfold-optimization
  using stamp-under-semantics-inversed
  using wf-stamp-def
  apply (smt (verit, best) ConditionalExprE le-expr-def stamp-under.simps)
  unfolding size.simps by simp
```

```
optimization b[intval]: ((x eq y) ? x : y) ↦ y
   apply unfold-optimization
     apply (smt (z3) bool-to-val.simps(2) intval-equals.elims val-to-bool.simps(1)
val-to-bool.simps(3))
    unfolding intval.simps
   apply (smt (z3) BinaryExprE ConditionalExprE Value.inject(1) Value.inject(2)
bin-eval.simps(10) bool-to-val.simps(2) evalDet intval-equals.simps(1) intval-equals.simps(10)
intval-equals.simps(12) intval-equals.simps(15) intval-equals.simps(16) intval-equals.simps(2)
intval-equals.simps(5) intval-equals.simps(8) intval-equals.simps(9) le-expr-def val-to-bool.cases
val-to-bool.elims(2))
  unfolding size.simps by auto

end

end
```

11 Conditional Elimination Phase

```
theory ConditionalElimination
  imports
    Semantics.IRGraphFrames
    Proofs.Stuttering
    Proofs.Rewrites
    Proofs.Bisimulation
begin
```

11.1 Individual Elimination Rules

We introduce a TriState as in the Graal compiler to represent when static analysis can tell us information about the value of a boolean expression. Unknown = No information can be inferred KnownTrue/KnownFalse = We can infer the expression will always be true or false.

```
datatype TriState = Unknown | KnownTrue | KnownFalse
```

The implies relation corresponds to the LogicNode.implies method from the compiler which attempts to infer when one logic nodes value can be inferred from a known logic node.

```
inductive implies :: IRGraph ⇒ IRNode ⇒ IRNode ⇒ TriState ⇒ bool
  (- ⊢ - & - ↪ -) for g where
  eq-imp-less:
  g ⊢ (IntegerEqualsNode x y) & (IntegerLessThanNode x y) ↪ KnownFalse |
  eq-imp-less-rev:
  g ⊢ (IntegerEqualsNode x y) & (IntegerLessThanNode y x) ↪ KnownFalse |
  less-imp-rev-less:
  g ⊢ (IntegerLessThanNode x y) & (IntegerLessThanNode y x) ↪ KnownFalse |
  less-imp-not-eq:
```

$g \vdash (\mathit{IntegerLessThanNode}\ x\ y)\ \&\ (\mathit{IntegerEqualsNode}\ x\ y) \hookrightarrow \mathit{KnownFalse}\ |$
less-imp-not-eq-rev:
$g \vdash (\mathit{IntegerLessThanNode}\ x\ y)\ \&\ (\mathit{IntegerEqualsNode}\ y\ x) \hookrightarrow \mathit{KnownFalse}\ |$

x-imp-x:
$g \vdash x\ \&\ x \hookrightarrow \mathit{KnownTrue}\ |$

negate-false:
$\llbracket g \vdash x\ \&\ (\mathit{kind}\ g\ y) \hookrightarrow \mathit{KnownTrue} \rrbracket \Longrightarrow g \vdash x\ \&\ (\mathit{LogicNegationNode}\ y) \hookrightarrow \mathit{KnownFalse}\ |$
negate-true:
$\llbracket g \vdash x\ \&\ (\mathit{kind}\ g\ y) \hookrightarrow \mathit{KnownFalse} \rrbracket \Longrightarrow g \vdash x\ \&\ (\mathit{LogicNegationNode}\ y) \hookrightarrow \mathit{KnownTrue}$

Total relation over partial implies relation

inductive *condition-implies* :: $\mathit{IRGraph} \Rightarrow \mathit{IRNode} \Rightarrow \mathit{IRNode} \Rightarrow \mathit{TriState} \Rightarrow \mathit{bool}$
(- $\vdash$ - & - $\rightharpoonup$ -) **for** *g* **where**
$\llbracket \neg(g \vdash a\ \&\ b \hookrightarrow \mathit{imp}) \rrbracket \Longrightarrow (g \vdash a\ \&\ b \rightharpoonup \mathit{Unknown})\ |$
$\llbracket (g \vdash a\ \&\ b \hookrightarrow \mathit{imp}) \rrbracket \Longrightarrow (g \vdash a\ \&\ b \rightharpoonup \mathit{imp})$

inductive *implies-tree* :: $\mathit{IRExpr} \Rightarrow \mathit{IRExpr} \Rightarrow \mathit{bool} \Rightarrow \mathit{bool}$
(- & - $\hookrightarrow$ -) **where**
eq-imp-less:
$(\mathit{BinaryExpr}\ \mathit{BinIntegerEquals}\ x\ y)\ \&\ (\mathit{BinaryExpr}\ \mathit{BinIntegerLessThan}\ x\ y) \hookrightarrow \mathit{False}\ |$
eq-imp-less-rev:
$(\mathit{BinaryExpr}\ \mathit{BinIntegerEquals}\ x\ y)\ \&\ (\mathit{BinaryExpr}\ \mathit{BinIntegerLessThan}\ y\ x) \hookrightarrow \mathit{False}\ |$
less-imp-rev-less:
$(\mathit{BinaryExpr}\ \mathit{BinIntegerLessThan}\ x\ y)\ \&\ (\mathit{BinaryExpr}\ \mathit{BinIntegerLessThan}\ y\ x) \hookrightarrow \mathit{False}\ |$
less-imp-not-eq:
$(\mathit{BinaryExpr}\ \mathit{BinIntegerLessThan}\ x\ y)\ \&\ (\mathit{BinaryExpr}\ \mathit{BinIntegerEquals}\ x\ y) \hookrightarrow \mathit{False}\ |$
less-imp-not-eq-rev:
$(\mathit{BinaryExpr}\ \mathit{BinIntegerLessThan}\ x\ y)\ \&\ (\mathit{BinaryExpr}\ \mathit{BinIntegerEquals}\ y\ x) \hookrightarrow \mathit{False}\ |$

x-imp-x:
$x\ \&\ x \hookrightarrow \mathit{True}\ |$

negate-false:
$\llbracket x\ \&\ y \hookrightarrow \mathit{True} \rrbracket \Longrightarrow x\ \&\ (\mathit{UnaryExpr}\ \mathit{UnaryLogicNegation}\ y) \hookrightarrow \mathit{False}\ |$
negate-true:
$\llbracket x\ \&\ y \hookrightarrow \mathit{False} \rrbracket \Longrightarrow x\ \&\ (\mathit{UnaryExpr}\ \mathit{UnaryLogicNegation}\ y) \hookrightarrow \mathit{True}$

Proofs that the implies relation is correct with respect to the existing evaluation semantics.

```
experiment begin
lemma logic-negate-type:
  assumes [m, p] ⊢ UnaryExpr UnaryLogicNegation x ↦ v
  assumes v ≠ UndefVal
  shows ∃ v2. [m, p] ⊢ x ↦ IntVal32 v2
proof –
  obtain ve where ve: [m, p] ⊢ x ↦ ve
    using assms(1) by blast
  then have [m, p] ⊢ UnaryExpr UnaryLogicNegation x ↦ unary-eval UnaryLogicNegation ve
    by (metis UnaryExprE assms(1) evalDet)
  then show ?thesis using assms unary-eval.elims evalDet ve IRUnaryOp.distinct
    sorry
qed


lemma logic-negation-relation-tree:
  assumes [m, p] ⊢ y ↦ val
  assumes [m, p] ⊢ UnaryExpr UnaryLogicNegation y ↦ invval
  assumes intval ≠ UndefVal
  shows val-to-bool val ⟷ ¬(val-to-bool invval)
proof –
  obtain v where invval = unary-eval UnaryLogicNegation v
    using assms(2) by blast
  then have [m, p] ⊢ y ↦ v using UnaryExprE assms(1,2) sorry
  then show ?thesis sorry
  qed

lemma logic-negation-relation:
  assumes [g, m, p] ⊢ y ↦ val
  assumes kind g neg = LogicNegationNode y
  assumes [g, m, p] ⊢ neg ↦ invval
  assumes intval ≠ UndefVal
  shows val-to-bool val ⟷ ¬(val-to-bool invval)
proof –
  obtain yencode where g ⊢ y ≃ yencode
    using assms(1) encodeeval-def by auto
  then have g ⊢ neg ≃ UnaryExpr UnaryLogicNegation yencode
    using rep.intros(7) assms(2) by simp
  then have [m, p] ⊢ UnaryExpr UnaryLogicNegation yencode ↦ invval
    using assms(3) encodeeval-def
    by (metis repDet)
  obtain v1 where [g, m, p] ⊢ y ↦ IntVal32 v1
    using assms(1,2,3,4) using logic-negate-type sorry
  have invval = bool-to-val (¬(val-to-bool val))
    using assms(1,2,3) evalDet unary-eval.simps(4)
   by (smt (verit, ccfv-SIG) UnaryExprE ‹[m,p] ⊢ UnaryExpr UnaryLogicNegation yencode ↦ invval› ‹g ⊢ y ≃ yencode› bool-to-val.simps(1) bool-to-val.simps(2) en-
codeeval-def intval-logic-negation.simps(1) logic-negate-type repDet val-to-bool.simps(1))
  have val-to-bool invval ⟷ ¬(val-to-bool val)
    using ‹invval = bool-to-val (¬ val-to-bool val)› by force
  then show ?thesis
    by simp
qed
end

lemma implies-valid:
  assumes x & y ↪ imp
  assumes [m, p] ⊢ x ↦ v1
  assumes [m, p] ⊢ y ↦ v2
  assumes v1 ≠ UndefVal ∧ v2 ≠ UndefVal
  shows (imp ⟶ (val-to-bool v1 ⟶ val-to-bool v2)) ∧
        (¬imp ⟶ (val-to-bool v1 ⟶ ¬(val-to-bool v2)))
    (is (?TP ⟶ ?TC) ∧ (?FP ⟶ ?FC))
  apply (intro conjI; rule impI)
proof −
  assume KnownTrue: ?TP
  show ?TC
 using assms(1) KnownTrue assms(2−) proof (induct x y imp rule: implies-tree.induct)
    case (eq-imp-less x y)
    then show ?case by simp
  next
    case (eq-imp-less-rev x y)
    then show ?case by simp
  next
    case (less-imp-rev-less x y)
    then show ?case by simp
  next
    case (less-imp-not-eq x y)
    then show ?case by simp
  next
    case (less-imp-not-eq-rev x y)
    then show ?case by simp
  next
    case (x-imp-x)
    then show ?case
      by (metis evalDet)
  next
    case (negate-false x1)
    then show ?case using evalDet
      using assms(2,3) by blast
  next
    case (negate-true y)
    then show ?case
      sorry
  qed
next
```

```
  assume KnownFalse: ?FP
 show ?FC using assms KnownFalse proof (induct x y imp rule: implies-tree.induct)
    case (eq-imp-less x y)
    obtain xval where xval: [m, p] ⊢ x ↦ xval
      using eq-imp-less(1) eq-imp-less.prems(3)
      by blast
    then obtain yval where yval: [m, p] ⊢ y ↦ yval
      using eq-imp-less.prems(3)
      using eq-imp-less.prems(2) by blast
     have eqeval: [m, p] ⊢ (BinaryExpr BinIntegerEquals x y) ↦ intval-equals xval
yval
      using xval yval evaltree.BinaryExpr
      by (metis BinaryExprE bin-eval.simps(10) eq-imp-less.prems(1) evalDet)
   have lesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan x y) ↦ intval-less-than
xval yval
      using xval yval evaltree.BinaryExpr
      by (metis BinaryExprE bin-eval.simps(11) eq-imp-less.prems(2) evalDet)
   have val-to-bool (intval-equals xval yval) ⟶ ¬(val-to-bool (intval-less-than xval
yval))
      using assms(4) apply (cases xval; cases yval; auto)
         apply (metis (full-types) val-to-bool.simps(1) Values.bool-to-val.simps(2)
signed.less-irrefl)
    by (metis (mono-tags) val-to-bool.simps(1) Values.bool-to-val.elims signed.order.strict-implies-not-eq)
    then show ?case
      using eqeval lesseval
      by (metis eq-imp-less.prems(1) eq-imp-less.prems(2) evalDet)
  next
    case (eq-imp-less-rev x y)
    obtain xval where xval: [m, p] ⊢ x ↦ xval
      using eq-imp-less-rev.prems(3)
      using eq-imp-less-rev.prems(2) by blast
    obtain yval where yval: [m, p] ⊢ y ↦ yval
      using eq-imp-less-rev.prems(3)
      using eq-imp-less-rev.prems(2) by blast
     have eqeval: [m, p] ⊢ (BinaryExpr BinIntegerEquals x y) ↦ intval-equals xval
yval
      using xval yval evaltree.BinaryExpr
      by (metis BinaryExprE bin-eval.simps(10) eq-imp-less-rev.prems(1) evalDet)
   have lesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan y x) ↦ intval-less-than
yval xval
      using xval yval evaltree.BinaryExpr
      by (metis BinaryExprE bin-eval.simps(11) eq-imp-less-rev.prems(2) evalDet)
   have val-to-bool (intval-equals xval yval) ⟶ ¬(val-to-bool (intval-less-than yval
xval))
      using assms(4) apply (cases xval; cases yval; auto)
         apply (metis (full-types) val-to-bool.simps(1) Values.bool-to-val.simps(2)
signed.less-irrefl)
    by (metis (full-types) val-to-bool.simps(1) Values.bool-to-val.elims signed.order.strict-implies-not-eq)
    then show ?case
```

```
      using eqeval lesseval
      by (metis eq-imp-less-rev.prems(1) eq-imp-less-rev.prems(2) evalDet)
  next
    case (less-imp-rev-less x y)
    obtain xval where xval: [m, p] ⊢ x ↦ xval
      using less-imp-rev-less.prems(3)
      using less-imp-rev-less.prems(2) by blast
    obtain yval where yval: [m, p] ⊢ y ↦ yval
      using less-imp-rev-less.prems(3)
      using less-imp-rev-less.prems(2) by blast
   have lesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan x y) ↦ intval-less-than
xval yval
      using xval yval evaltree.BinaryExpr
     by (metis BinaryExprE bin-eval.simps(11) evalDet less-imp-rev-less.prems(1))
      have revlesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan y x) ↦ int-
val-less-than yval xval
      using xval yval evaltree.BinaryExpr
     by (metis BinaryExprE bin-eval.simps(11) evalDet less-imp-rev-less.prems(2))
    have val-to-bool (intval-less-than xval yval) ⟶ ¬(val-to-bool (intval-less-than
yval xval))
      using assms(4) apply (cases xval; cases yval; auto)
    apply (metis val-to-bool.simps(1) Values.bool-to-val.elims signed.not-less-iff-gr-or-eq)
      by (metis val-to-bool.simps(1) Values.bool-to-val.elims signed.less-asym')
    then show ?case
     by (metis evalDet less-imp-rev-less.prems(1) less-imp-rev-less.prems(2) lesseval
revlesseval)
  next
    case (less-imp-not-eq x y)
    obtain xval where xval: [m, p] ⊢ x ↦ xval
      using less-imp-not-eq.prems(3)
      using less-imp-not-eq.prems(1) by blast
    obtain yval where yval: [m, p] ⊢ y ↦ yval
      using less-imp-not-eq.prems(3)
      using less-imp-not-eq.prems(1) by blast
    have eqeval: [m, p] ⊢ (BinaryExpr BinIntegerEquals x y) ↦ intval-equals xval
yval
      using xval yval evaltree.BinaryExpr
      by (metis BinaryExprE bin-eval.simps(10) evalDet less-imp-not-eq.prems(2))
   have lesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan x y) ↦ intval-less-than
xval yval
      using xval yval evaltree.BinaryExpr
      by (metis BinaryExprE bin-eval.simps(11) evalDet less-imp-not-eq.prems(1))
   have val-to-bool (intval-less-than xval yval) ⟶ ¬(val-to-bool (intval-equals xval
yval))
      using assms(4) apply (cases xval; cases yval; auto)
     apply (metis (full-types) bool-to-val.simps(2) signed.less-imp-not-eq val-to-bool.simps(1))
    by (metis (full-types) bool-to-val.simps(2) signed.less-imp-not-eq2 val-to-bool.simps(1))
    then show ?case
       by (metis eqeval evalDet less-imp-not-eq.prems(1) less-imp-not-eq.prems(2)
lesseval)
  next
    case (less-imp-not-eq-rev x y)
    obtain xval where xval: [m, p] ⊢ x ↦ xval
      using less-imp-not-eq-rev.prems(3)
      using less-imp-not-eq-rev.prems(1) by blast
    obtain yval where yval: [m, p] ⊢ y ↦ yval
      using less-imp-not-eq-rev.prems(3)
      using less-imp-not-eq-rev.prems(1) by blast
    have eqeval: [m, p] ⊢ (BinaryExpr BinIntegerEquals y x) ↦ intval-equals yval
xval
      using xval yval evaltree.BinaryExpr
    by (metis BinaryExprE bin-eval.simps(10) evalDet less-imp-not-eq-rev.prems(2))
   have lesseval: [m, p] ⊢ (BinaryExpr BinIntegerLessThan x y) ↦ intval-less-than
xval yval
      using xval yval evaltree.BinaryExpr
    by (metis BinaryExprE bin-eval.simps(11) evalDet less-imp-not-eq-rev.prems(1))
   have val-to-bool (intval-less-than xval yval) ⟶ ¬(val-to-bool (intval-equals yval
xval))
      using assms(4) apply (cases xval; cases yval; auto)
    apply (metis (full-types) bool-to-val.simps(2) signed.less-imp-not-eq2 val-to-bool.simps(1))
    by (metis (full-types, opaque-lifting) val-to-bool.simps(1) Values.bool-to-val.elims
signed.dual-order.strict-implies-not-eq)
    then show ?case
    by (metis eqeval evalDet less-imp-not-eq-rev.prems(1) less-imp-not-eq-rev.prems(2)
lesseval)
  next
    case (x-imp-x x1)
    then show ?case by simp
  next
    case (negate-false x y)
    then show ?case sorry
  next
    case (negate-true x1)
    then show ?case by simp
  qed
qed

lemma implies-true-valid:
  assumes x & y ↪ imp
  assumes imp
  assumes [m, p] ⊢ x ↦ v1
  assumes [m, p] ⊢ y ↦ v2
  assumes v1 ≠ UndefVal ∧ v2 ≠ UndefVal
  shows val-to-bool v1 ⟶ val-to-bool v2
  using assms implies-valid
  by blast

lemma implies-false-valid:
  assumes x & y ↪ imp
  assumes ¬imp
  assumes [m, p] ⊢ x ↦ v1
  assumes [m, p] ⊢ y ↦ v2
  assumes v1 ≠ UndefVal ∧ v2 ≠ UndefVal
  shows val-to-bool v1 ⟶ ¬(val-to-bool v2)
  using assms implies-valid by blast
```

The following relation corresponds to the UnaryOpLogicNode.tryFold and BinaryOpLogicNode.tryFold methods and their associated concrete implementations.

The relation determines if a logic operation can be shown true or false through the stamp typing information.

```
inductive tryFold :: IRNode ⇒ (ID ⇒ Stamp) ⇒ bool ⇒ bool
  where
  ⟦alwaysDistinct (stamps x) (stamps y)⟧
    ⟹ tryFold (IntegerEqualsNode x y) stamps False |
  ⟦neverDistinct (stamps x) (stamps y)⟧
    ⟹ tryFold (IntegerEqualsNode x y) stamps True |
  ⟦is-IntegerStamp (stamps x);
    is-IntegerStamp (stamps y);
    stpi-upper (stamps x) < stpi-lower (stamps y)⟧
    ⟹ tryFold (IntegerLessThanNode x y) stamps True |
  ⟦is-IntegerStamp (stamps x);
    is-IntegerStamp (stamps y);
    stpi-lower (stamps x) ≥ stpi-upper (stamps y)⟧
    ⟹ tryFold (IntegerLessThanNode x y) stamps False
```

Proofs that show that when the stamp lookup function is well-formed, the tryFold relation correctly predicts the output value with respect to our evaluation semantics.

```
lemma
  assumes kind g nid = IntegerEqualsNode x y
  assumes [g, m, p] ⊢ nid ↦ v
  assumes v ≠ UndefVal
  assumes ([g, m, p] ⊢ x ↦ xval) ∧ ([g, m, p] ⊢ y ↦ yval)
  shows val-to-bool (intval-equals xval yval) ⟷ v = IntVal32 1
proof –
  have v = intval-equals xval yval
    using assms(1, 2, 3, 4) BinaryExprE IntegerEqualsNode bin-eval.simps(7)
    by (smt (verit) bin-eval.simps(10) encodeeval-def evalDet repDet)
  then show ?thesis using intval-equals.simps val-to-bool.simps sorry
qed

lemma tryFoldIntegerEqualsAlwaysDistinct:
  assumes wf-stamp g stamps
  assumes kind g nid = (IntegerEqualsNode x y)
  assumes [g, m, p] ⊢ nid ↦ v
  assumes alwaysDistinct (stamps x) (stamps y)
  shows v = IntVal32 0
proof –
  have ∀ val. ¬(valid-value val (join (stamps x) (stamps y)))
    using assms(1,4) unfolding alwaysDistinct.simps
   by (metis is-stamp-empty.elims(2) le-less-trans not-less valid32or64 valid-value.simps(1)
valid-value.simps(2))
  have ¬(∃ val . ([g, m, p] ⊢ x ↦ val) ∧ ([g, m, p] ⊢ y ↦ val))
      using assms(1,4) unfolding alwaysDistinct.simps wf-stamp.simps encodee-
val-def sorry
  then show ?thesis sorry
qed

lemma tryFoldIntegerEqualsNeverDistinct:
  assumes wf-stamp g stamps
  assumes kind g nid = (IntegerEqualsNode x y)
  assumes [g, m, p] ⊢ nid ↦ v
  assumes neverDistinct (stamps x) (stamps y)
  shows v = IntVal32 1
  using assms IntegerEqualsNodeE sorry

lemma tryFoldIntegerLessThanTrue:
  assumes wf-stamp g stamps
  assumes kind g nid = (IntegerLessThanNode x y)
  assumes [g, m, p] ⊢ nid ↦ v
  assumes stpi-upper (stamps x) < stpi-lower (stamps y)
  shows v = IntVal32 1
proof –
  have stamp-type: is-IntegerStamp (stamps x)
    using assms
    sorry
  obtain xval where xval: [g, m, p] ⊢ x ↦ xval
    using assms(2,3) sorry
  obtain yval where yval: [g, m, p] ⊢ y ↦ yval
    using assms(2,3) sorry
  have is-IntegerStamp (stamps x) ∧ is-IntegerStamp (stamps y)
    using assms(4)
    sorry
  then have val-to-bool (intval-less-than xval yval)
    sorry
  then show ?thesis
    sorry
qed

lemma tryFoldIntegerLessThanFalse:
  assumes wf-stamp g stamps
  assumes kind g nid = (IntegerLessThanNode x y)
  assumes [g, m, p] ⊢ nid ↦ v
  assumes stpi-lower (stamps x) ≥ stpi-upper (stamps y)
  shows v = IntVal32 0
  proof –
  have stamp-type: is-IntegerStamp (stamps x)
    using assms
    sorry
  obtain xval where xval: [g, m, p] ⊢ x ↦ xval
    using assms(2,3) sorry
  obtain yval where yval: [g, m, p] ⊢ y ↦ yval
    using assms(2,3) sorry
  have is-IntegerStamp (stamps x) ∧ is-IntegerStamp (stamps y)
    using assms(4)
    sorry
  then have ¬(val-to-bool (intval-less-than xval yval))
    sorry
  then show ?thesis
    sorry
qed

theorem tryFoldProofTrue:
  assumes wf-stamp g stamps
  assumes tryFold (kind g nid) stamps True
  assumes [g, m, p] ⊢ nid ↦ v
  shows val-to-bool v
  using assms(2) proof (induction kind g nid stamps True rule: tryFold.induct)
case (1 stamps x y)
  then show ?case using tryFoldIntegerEqualsAlwaysDistinct assms sorry
next
  case (2 stamps x y)
  then show ?case using tryFoldIntegerEqualsAlwaysDistinct assms sorry
next
  case (3 stamps x y)
  then show ?case using tryFoldIntegerLessThanTrue assms sorry
next
case (4 stamps x y)
  then show ?case using tryFoldIntegerLessThanFalse assms sorry
qed

theorem tryFoldProofFalse:
  assumes wf-stamp g stamps
  assumes tryFold (kind g nid) stamps False
  assumes [g, m, p] ⊢ nid ↦ v
  shows ¬(val-to-bool v)
using assms(2) proof (induction kind g nid stamps False rule: tryFold.induct)
case (1 stamps x y)
  then show ?case using tryFoldIntegerEqualsAlwaysDistinct assms sorry
next
case (2 stamps x y)
  then show ?case using tryFoldIntegerEqualsNeverDistinct assms sorry
next
```

```
  case (3 stamps x y)
  then show ?case using tryFoldIntegerLessThanTrue assms sorry
next
  case (4 stamps x y)
  then show ?case using tryFoldIntegerLessThanFalse assms sorry

qed
```

```
inductive-cases StepE:
  g, p ⊢ (nid,m,h) → (nid',m',h)
```

Perform conditional elimination rewrites on the graph for a particular node.

In order to determine conditional eliminations appropriately the rule needs two data structures produced by static analysis. The first parameter is the set of IRNodes that we know result in a true value when evaluated. The second parameter is a mapping from node identifiers to the flow-sensitive stamp.

The relation transforms the third parameter to the fifth parameter for a node identifier which represents the fourth parameter.

```
inductive ConditionalEliminationStep ::
  IRExpr set ⇒ (ID ⇒ Stamp) ⇒ IRGraph ⇒ ID ⇒ IRGraph ⇒ bool where
  impliesTrue:
  ⟦kind g ifcond = (IfNode cid t f);
    g ⊢ cid ≃ cond;
    ∃ ce ∈ conds . (ce & cond ↪ True);
    g' = constantCondition True ifcond (kind g ifcond) g
    ⟧ ⟹ ConditionalEliminationStep conds stamps g ifcond g' |

  impliesFalse:
  ⟦kind g ifcond = (IfNode cid t f);
    g ⊢ cid ≃ cond;
    ∃ ce ∈ conds . (ce & cond ↪ False);
    g' = constantCondition False ifcond (kind g ifcond) g
    ⟧ ⟹ ConditionalEliminationStep conds stamps g ifcond g' |

  tryFoldTrue:
  ⟦kind g ifcond = (IfNode cid t f);
    cond = kind g cid;
    tryFold (kind g cid) stamps True;
    g' = constantCondition True ifcond (kind g ifcond) g
    ⟧ ⟹ ConditionalEliminationStep conds stamps g ifcond g' |

  tryFoldFalse:
  ⟦kind g ifcond = (IfNode cid t f);
    cond = kind g cid;
    tryFold (kind g cid) stamps False;
  g' = constantCondition False ifcond (kind g ifcond) g
  ⟧ ⟹ ConditionalEliminationStep conds stamps g ifcond g'
```

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

thm *ConditionalEliminationStep.equation*

11.2 Control-flow Graph Traversal

```
type-synonym Seen = ID set
type-synonym Condition = IRNode
type-synonym Conditions = Condition list
type-synonym StampFlow = (ID ⇒ Stamp) list
```

nextEdge helps determine which node to traverse next by returning the first successor edge that isn't in the set of already visited nodes. If there is not an appropriate successor, None is returned instead.

```
fun nextEdge :: Seen ⇒ ID ⇒ IRGraph ⇒ ID option where
  nextEdge seen nid g =
    (let nids = (filter (λnid'. nid' ∉ seen) (successors-of (kind g nid))) in
     (if length nids > 0 then Some (hd nids) else None))
```

pred determines which node, if any, acts as the predecessor of another.

Merge nodes represent a special case where-in the predecessor exists as an input edge of the merge node, to simplify the traversal we treat only the first input end node as the predecessor, ignoring that multiple nodes may act as a successor.

For all other nodes, the predecessor is the first element of the predecessors set. Note that in a well-formed graph there should only be one element in the predecessor set.

```
fun pred :: IRGraph ⇒ ID ⇒ ID option where
  pred g nid = (case kind g nid of
    (MergeNode ends - -) ⇒ Some (hd ends) |
    - ⇒
      (if IRGraph.predecessors g nid = {}
        then None else
        Some (hd (sorted-list-of-set (IRGraph.predecessors g nid)))
      )
  )
```

When the basic block of an if statement is entered, we know that the condition of the preceding if statement must be true. As in the GraalVM compiler, we introduce the registerNewCondition funciton which roughly corresponds to the ConditionalEliminationPhase.registerNewCondition. This method updates the flow-sensitive stamp information based on the condition which we know must be true.

```
fun clip-upper :: Stamp ⇒ int ⇒ Stamp where
  clip-upper (IntegerStamp b l h) c = (IntegerStamp b l c) |
  clip-upper s c = s
fun clip-lower :: Stamp ⇒ int ⇒ Stamp where
  clip-lower (IntegerStamp b l h) c = (IntegerStamp b c h) |
  clip-lower s c = s

fun registerNewCondition :: IRGraph ⇒ Condition ⇒ (ID ⇒ Stamp) ⇒ (ID ⇒
Stamp) where

  registerNewCondition g (IntegerEqualsNode x y) stamps =
    (stamps(x := join (stamps x) (stamps y)))(y := join (stamps x) (stamps y)) |

  registerNewCondition g (IntegerLessThanNode x y) stamps =
    (stamps
      (x := clip-upper (stamps x) (stpi-lower (stamps y))))
      (y := clip-lower (stamps y) (stpi-upper (stamps x))) |
  registerNewCondition g - stamps = stamps

fun hdOr :: 'a list ⇒ 'a ⇒ 'a where
  hdOr (x # xs) de = x |
  hdOr [] de = de
```

The Step relation is a small-step traversal of the graph which handles transitions between individual nodes of the graph.

It relates a pairs of tuple of the current node, the set of seen nodes, the always true stack of IfNode conditions, and the flow-sensitive stamp information.

```
inductive Step
  :: IRGraph ⇒ (ID × Seen × Conditions × StampFlow) ⇒ (ID × Seen ×
Conditions × StampFlow) option ⇒ bool
  for g where
  — Hit a BeginNode with an IfNode predecessor which represents the start of a
basic block for the IfNode. 1. nid' will be the successor of the begin node. 2. Find
the first and only predecessor. 3. Extract condition from the preceding IfNode. 4.
Negate condition if the begin node is second branch (we've taken the else branch of
the condition) 5. Add the condition or the negated condition to stack 6. Perform
any stamp updates based on the condition using the registerNewCondition function
and place them on the top of the stack of stamp information
  ⟦kind g nid = BeginNode nid';

    nid ∉ seen;
    seen' = {nid} ∪ seen;

    Some ifcond = pred g nid;
    kind g ifcond = IfNode cond t f;

    i = find-index nid (successors-of (kind g ifcond));
    c = (if i = 0 then kind g cond else LogicNegationNode cond);
    conds' = c # conds;
```

$flow' = registerNewCondition\ g\ c\ (hdOr\ flow\ (stamp\ g))]\!]$
$\Longrightarrow Step\ g\ (nid,\ seen,\ conds,\ flow)\ (Some\ (nid',\ seen',\ conds',\ flow'\ \#\ flow))\ |$

— Hit an EndNode 1. nid' will be the usage of EndNode 2. pop the conditions and stamp stack

$[\![kind\ g\ nid = EndNode;$

$nid \notin seen;$
$seen' = \{nid\} \cup seen;$

$nid' = any\text{-}usage\ g\ nid;$

$conds' = tl\ conds;$
$flow' = tl\ flow]\!]$
$\Longrightarrow Step\ g\ (nid,\ seen,\ conds,\ flow)\ (Some\ (nid',\ seen',\ conds',\ flow'))\ |$

— We can find a successor edge that is not in seen, go there

$[\![\neg(is\text{-}EndNode\ (kind\ g\ nid));$
$\neg(is\text{-}BeginNode\ (kind\ g\ nid));$

$nid \notin seen;$
$seen' = \{nid\} \cup seen;$

$Some\ nid' = nextEdge\ seen'\ nid\ g]\!]$
$\Longrightarrow Step\ g\ (nid,\ seen,\ conds,\ flow)\ (Some\ (nid',\ seen',\ conds,\ flow))\ |$

— We can cannot find a successor edge that is not in seen, give back None

$[\![\neg(is\text{-}EndNode\ (kind\ g\ nid));$
$\neg(is\text{-}BeginNode\ (kind\ g\ nid));$

$nid \notin seen;$
$seen' = \{nid\} \cup seen;$

$None = nextEdge\ seen'\ nid\ g]\!]$
$\Longrightarrow Step\ g\ (nid,\ seen,\ conds,\ flow)\ None\ |$

— We've already seen this node, give back None

$[\![nid \in seen]\!] \Longrightarrow Step\ g\ (nid,\ seen,\ conds,\ flow)\ None$

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

The ConditionalEliminationPhase relation is responsible for combining the individual traversal steps from the Step relation and the optimizations from the ConditionalEliminationStep relation to perform a transformation of the whole graph.

end