

Veriopt

January 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	3
2	Nodes	9
2.1	Types of Nodes	9
2.2	Hierarchy of Nodes	17
3	Stamp Typing	24
4	Graph Representation	27
4.0.1	Example Graphs	32
5	Data-flow Semantics	33
5.1	Data-flow Tree Representation	33
5.2	Data-flow Tree Evaluation	35
5.3	Data-flow Tree Refinement	37
6	Data-flow Expression-Tree Theorems	38
6.1	Extraction and Evaluation of Expression Trees is Deterministic.	38
6.2	Example Data-flow Optimisations	46
6.3	Monotonicity of Expression Optimization	46
7	Tree to Graph	47
8	Control-flow Semantics	74
8.1	Heap	74
8.2	Intraprocedural Semantics	74
8.3	Interprocedural Semantics	77
8.4	Big-step Execution	78
8.4.1	Heap Testing	79
9	Properties of Control-flow Semantics	80
10	Proof Infrastructure	85
10.1	Bisimulation	85
10.2	Formedness Properties	86
10.3	Dynamic Frames	87
10.4	Graph Rewriting	99
10.5	Stuttering	103
11	Canonicalization Phase	104
12	Canonicalization Phase	115

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 new type of runtime values. Our evaluation semantics are defined in terms of these 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 eventually arrays.

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.

Java supports 64, 32, 16, 8 signed ints, plus 1 bit (boolean) ints. Our Value type models this by keeping the value as an infinite precision signed int, but also carrying along the number of bits allowed.

So each $(\text{IntVal } b \ v)$ should satisfy the invariants:

$$b \in \{1::'a, 8::'a, 16::'a, 32::'a, 64::'a\}$$

$$1 < b \implies v \equiv \text{scast } (\text{signed-take-bit } b \ v)$$

```

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 Value =
  UndefVal |
  IntVal32 int32 |
  IntVal64 int64 |

  ObjRef objref |
  ObjStr string

```

We define integer values to be well-formed when their bit size is valid and their integer value is able to fit within the bit size. This is defined using the *wf-value* function.

— Check that a signed int value does not overflow b bits.

```

fun fits-into-n :: nat  $\Rightarrow$  int  $\Rightarrow$  bool where
  fits-into-n b val = (( $-(2^{b-1}) \leq val$ )  $\wedge$  ( $val < 2^{b-1}$ )))

```

```
fun wf-bool :: Value  $\Rightarrow$  bool where
  wf-bool (IntVal32 v) = (v = 0  $\vee$  v = 1) |
  wf-bool - = False
```

```
fun val-to-bool :: Value  $\Rightarrow$  bool where
  val-to-bool (IntVal32 v) = (v = 1) |
  val-to-bool - = False
```

```
fun bool-to-val :: bool  $\Rightarrow$  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  $\Rightarrow$  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.

```
fun intval-add32 :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-add32 (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1+v2)) |
  intval-add32 - - =.UndefVal
```

```
fun intval-add64 :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-add64 (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1+v2)) |
  intval-add64 - - =.UndefVal
```

```
fun intval-add :: Value  $\Rightarrow$  Value  $\Rightarrow$  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 :: plus
begin
```

```
definition plus-Value :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  plus-Value = intval-add
```

```
instance proof qed
end
```

```
fun intval-sub :: Value  $\Rightarrow$  Value  $\Rightarrow$  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  $\Rightarrow$  Value  $\Rightarrow$  Value where
  minus-Value = intval-sub
```

```
instance proof qed
end
```

```
fun intval-mul :: Value  $\Rightarrow$  Value  $\Rightarrow$  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  $\Rightarrow$  Value  $\Rightarrow$  Value where
  times-Value = intval-mul
```

```
instance proof qed
end
```

```
fun intval-div :: Value  $\Rightarrow$  Value  $\Rightarrow$  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  $\Rightarrow$  Value  $\Rightarrow$  Value where
  divide-Value = intval-div
```

```
instance proof qed
end
```

```
fun intval-mod :: Value  $\Rightarrow$  Value  $\Rightarrow$  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  $\Rightarrow$  Value  $\Rightarrow$  Value where
  modulo-Value = intval-mod
```

```
instance proof qed
end
```

```
fun intval-and :: Value  $\Rightarrow$  Value  $\Rightarrow$  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  $\Rightarrow$  Value  $\Rightarrow$  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  $\Rightarrow$  Value  $\Rightarrow$  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  $\Rightarrow$  Value  $\Rightarrow$  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  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-less-than (IntVal32 v1) (IntVal32 v2) = bool-to-val (v1 < v2) |
  intval-less-than (IntVal64 v1) (IntVal64 v2) = bool-to-val (v1 < v2) |
  intval-less-than - - = UndefVal
```

```

fun intval-below :: Value  $\Rightarrow$  Value  $\Rightarrow$  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  $\Rightarrow$  Value where
  intval-not (IntVal32 v) = (IntVal32 (NOT v)) |
  intval-not (IntVal64 v) = (IntVal64 (NOT v)) |
  intval-not - = UndefVal

```

```

fun intval-negate :: Value  $\Rightarrow$  Value where
  intval-negate (IntVal32 v) = IntVal32 (- v) |
  intval-negate (IntVal64 v) = IntVal64 (- v) |
  intval-negate - = UndefVal

```

```

fun intval-abs :: Value  $\Rightarrow$  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

```

```

lemma [code]: shiftl1 n = n * 2
by (simp add: shiftl1-eq-mult-2)

```

```

lemma [code]: shiftr1 n = n div 2
by (simp add: shiftr1-eq-div-2)

```

```

lemma [code]: sshiftr1 n = word-of-int (sint n div 2)
using sshiftr1-eq by blast

```

```

definition shiftl (infix << 75) where
  shiftl w n = (shiftl1  $\hat{\sim}$  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, lifting) comp-def funpow-swap1 mult.left-commute power-Suc
  shiftl1-eq-mult-2)

```

```

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 *signed-shiftr* (**infix** >> 75) **where**

signed-shiftr *w n* = (*sshiftr1* $\widehat{\sim}$ *n*) *w*

definition *shiftr* (**infix** >>> 75) **where**

shiftr *w n* = (*shiftr1* $\widehat{\sim}$ *n*) *w*

lemma *shiftr-power*[*simp*]: (*x*::('a::len) word) *div* (2^j) = *x* >>> *j*

unfolding *shiftr-def* **apply** (*induction j*)

apply *simp* **unfolding** *funpow-Suc-right*

by (*metis* (*no-types*, *lifting*) *comp-apply* *div-exp-eq* *funpow-swap1* *power-Suc2* *power-add* *power-one-right* *shiftr1-eq-div-2*)

fun *intval-left-shift* :: *Value* \Rightarrow *Value* \Rightarrow *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* \Rightarrow *Value* \Rightarrow *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* \Rightarrow *Value* \Rightarrow *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*

lemma *word-add-sym*:

shows *word-of-int v1* + *word-of-int v2* = *word-of-int v2* + *word-of-int v1*

by *simp*

lemma *intval-add-sym*:

shows *intval-add a b* = *intval-add b a*


```

by (induction a; induction b; auto)

lemma word-add-assoc:
  shows (word-of-int v1 + word-of-int v2) + word-of-int v3
    = word-of-int v1 + (word-of-int v2 + word-of-int v3)
  by simp

lemma intval-bad1 [simp]: intval-add (IntVal32 x) (IntVal64 y) =.UndefVal
  by auto
lemma intval-bad2 [simp]: intval-add (IntVal64 x) (IntVal32 y) =.UndefVal
  by auto

lemma intval-assoc: intval-add32 (intval-add32 x y) z = intval-add32 x (intval-add32
y z)
  apply (induction x)
  apply auto
  apply (induction y)
  apply auto
  apply (induction z)
  by auto

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

lemma intval-add (IntVal32 (231-1)) (IntVal32 (231-1)) = IntVal32 (-2)
  by eval
lemma intval-add (IntVal64 (231-1)) (IntVal64 (231-1)) = 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: IN-
PUT-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: IN-
PUT option) (ir-stateDuring-opt: INPUT-STATE option) (ir-stateAfter-opt: IN-

```

PUT-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, true-
  Value, 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 stateDur-
ing) @ (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 state-
Before) |
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 virtualObjectMap-
pings) = [] |
  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  $\Rightarrow$  bool where
  is-AbstractBeginNode n = ((is-BeginNode n)  $\vee$  (is-BeginStateSplitNode n)  $\vee$ 
    (is-KillingBeginNode n))

fun is-FixedWithNextNode :: IRNode  $\Rightarrow$  bool where
  is-FixedWithNextNode n = ((is-AbstractBeginNode n)  $\vee$  (is-AbstractStateSplit n)
     $\vee$  (is-AccessFieldNode n)  $\vee$  (is-DeoptimizingFixedWithNextNode n))

fun is-WithExceptionNode :: IRNode  $\Rightarrow$  bool where
  is-WithExceptionNode n = ((is-InvokeWithExceptionNode n))

fun is-ControlSplitNode :: IRNode  $\Rightarrow$  bool where
  is-ControlSplitNode n = ((is-IfNode n)  $\vee$  (is-WithExceptionNode n))

fun is-ControlSinkNode :: IRNode  $\Rightarrow$  bool where
  is-ControlSinkNode n = ((is-ReturnNode n)  $\vee$  (is-UnwindNode n))

fun is-AbstractEndNode :: IRNode  $\Rightarrow$  bool where
  is-AbstractEndNode n = ((is-EndNode n)  $\vee$  (is-LoopEndNode n))

fun is-FixedNode :: IRNode  $\Rightarrow$  bool where
  is-FixedNode n = ((is-AbstractEndNode n)  $\vee$  (is-ControlSinkNode n)  $\vee$  (is-ControlSplitNode
    n)  $\vee$  (is-FixedWithNextNode n))

fun is-CallTargetNode :: IRNode  $\Rightarrow$  bool where
  is-CallTargetNode n = ((is-MethodCallTargetNode n))

fun is-ValueNode :: IRNode  $\Rightarrow$  bool where
  is-ValueNode n = ((is-CallTargetNode n)  $\vee$  (is-FixedNode n)  $\vee$  (is-FloatingNode
    n))

fun is-Node :: IRNode  $\Rightarrow$  bool where
  is-Node n = ((is-ValueNode n)  $\vee$  (is-VirtualState n))

fun is-MemoryKill :: IRNode  $\Rightarrow$  bool where
  is-MemoryKill n = ((is-AbstractMemoryCheckpoint n))

fun is-NarrowableArithmeticNode :: IRNode  $\Rightarrow$  bool where
  is-NarrowableArithmeticNode n = ((is-AbsNode n)  $\vee$  (is-AddNode n)  $\vee$  (is-AndNode
    n)  $\vee$  (is-MulNode n)  $\vee$  (is-NegateNode n)  $\vee$  (is-NotNode n)  $\vee$  (is-OrNode n)  $\vee$ 
    (is-ShiftNode n)  $\vee$  (is-SubNode n)  $\vee$  (is-XorNode n))

fun is-AnchoringNode :: IRNode  $\Rightarrow$  bool where
  is-AnchoringNode n = ((is-AbstractBeginNode n))

fun is-DeoptBefore :: IRNode  $\Rightarrow$  bool where
  is-DeoptBefore n = ((is-DeoptimizingFixedWithNextNode n))

fun is-IndirectCanonicalization :: IRNode  $\Rightarrow$  bool where

```

is-IndirectCanonicalization $n = ((is-LogicNode\ n))$

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

fun *is-Invoke* :: *IRNode* \Rightarrow *bool* **where**
is-Invoke $n = ((is-InvokeNode\ n) \vee (is-InvokeWithExceptionNode\ n))$

fun *is-Proxy* :: *IRNode* \Rightarrow *bool* **where**
is-Proxy $n = ((is-ProxyNode\ n))$

fun *is-ValueProxy* :: *IRNode* \Rightarrow *bool* **where**
is-ValueProxy $n = ((is-PiNode\ n) \vee (is-ValueProxyNode\ n))$

fun *is-ValueNodeInterface* :: *IRNode* \Rightarrow *bool* **where**
is-ValueNodeInterface $n = ((is-ValueNode\ n))$

fun *is-ArrayLengthProvider* :: *IRNode* \Rightarrow *bool* **where**
is-ArrayLengthProvider $n = ((is-AbstractNewArrayNode\ n) \vee (is-ConstantNode\ n))$

fun *is-StampInverter* :: *IRNode* \Rightarrow *bool* **where**
is-StampInverter $n = ((is-IntegerConvertNode\ n) \vee (is-NegateNode\ n) \vee (is-NotNode\ n))$

fun *is-GuardingNode* :: *IRNode* \Rightarrow *bool* **where**
is-GuardingNode $n = ((is-AbstractBeginNode\ n))$

fun *is-SingleMemoryKill* :: *IRNode* \Rightarrow *bool* **where**
is-SingleMemoryKill $n = ((is-BytecodeExceptionNode\ n) \vee (is-ExceptionObjectNode\ n) \vee$
 $(is-InvokeNode\ n) \vee (is-InvokeWithExceptionNode\ n) \vee (is-KillingBeginNode\ n) \vee$
 $(is-StartNode\ n))$

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

fun *is-GuardedNode* :: *IRNode* \Rightarrow *bool* **where**
is-GuardedNode $n = ((is-FloatingGuardedNode\ n))$

fun *is-ArithmeticLIRLowerable* :: *IRNode* \Rightarrow *bool* **where**
is-ArithmeticLIRLowerable $n = ((is-AbsNode\ n) \vee (is-BinaryArithmeticNode\ n) \vee$

$(is-IntegerConvertNode\ n) \vee (is-NotNode\ n) \vee (is-ShiftNode\ n) \vee (is-UnaryArithmeticNode\ n))$

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

fun *is-VirtualizableAllocation* :: *IRNode* \Rightarrow *bool* **where**
is-VirtualizableAllocation *n* = ((*is-NewArrayNode* *n*) \vee (*is-NewInstanceNode* *n*))

fun *is-Unary* :: *IRNode* \Rightarrow *bool* **where**
is-Unary *n* = ((*is-LoadFieldNode* *n*) \vee (*is-LogicNegationNode* *n*) \vee (*is-UnaryNode* *n*) \vee (*is-UnaryOpLogicNode* *n*))

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

fun *is-BinaryCommutative* :: *IRNode* \Rightarrow *bool* **where**
is-BinaryCommutative *n* = ((*is-AddNode* *n*) \vee (*is-AndNode* *n*) \vee (*is-IntegerEqualsNode* *n*) \vee (*is-MulNode* *n*) \vee (*is-OrNode* *n*) \vee (*is-XorNode* *n*))

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

fun *is-UncheckedInterfaceProvider* :: *IRNode* \Rightarrow *bool* **where**
is-UncheckedInterfaceProvider *n* = ((*is-InvokeNode* *n*) \vee (*is-InvokeWithExceptionNode* *n*) \vee (*is-LoadFieldNode* *n*) \vee (*is-ParameterNode* *n*))

fun *is-Binary* :: *IRNode* \Rightarrow *bool* **where**
is-Binary *n* = ((*is-BinaryArithmeticNode* *n*) \vee (*is-BinaryNode* *n*) \vee (*is-BinaryOpLogicNode* *n*) \vee (*is-CompareNode* *n*) \vee (*is-FixedBinaryNode* *n*) \vee (*is-ShortCircuitOrNode* *n*))

fun *is-ArithmeticOperation* :: *IRNode* \Rightarrow *bool* **where**
is-ArithmeticOperation *n* = ((*is-BinaryArithmeticNode* *n*) \vee (*is-IntegerConvertNode* *n*) \vee (*is-ShiftNode* *n*) \vee (*is-UnaryArithmeticNode* *n*))

fun *is-ValueNumberable* :: *IRNode* \Rightarrow *bool* **where**
is-ValueNumberable *n* = ((*is-FloatingNode* *n*) \vee (*is-ProxyNode* *n*))

fun *is-Lowerable* :: *IRNode* \Rightarrow *bool* **where**
is-Lowerable *n* = ((*is-AbstractNewObjectNode* *n*) \vee (*is-AccessFieldNode* *n*) \vee (*is-BytecodeExceptionNode* *n*) \vee (*is-ExceptionObjectNode* *n*) \vee (*is-IntegerDivRemNode* *n*) \vee (*is-UnwindNode* *n*))

fun *is-Virtualizable* :: *IRNode* \Rightarrow *bool* **where**
is-Virtualizable *n* = ((*is-IsNullNode* *n*) \vee (*is-LoadFieldNode* *n*) \vee (*is-PiNode* *n*) \vee (*is-StoreFieldNode* *n*) \vee (*is-ValueProxyNode* *n*))

```

fun is-Simplifiable :: IRNode  $\Rightarrow$  bool where
  is-Simplifiable n = ((is-AbstractMergeNode n)  $\vee$  (is-BeginNode n)  $\vee$  (is-IfNode
n)  $\vee$  (is-LoopExitNode n)  $\vee$  (is-MethodCallTargetNode n)  $\vee$  (is-NewArrayNode n))

fun is-StateSplit :: IRNode  $\Rightarrow$  bool where
  is-StateSplit n = ((is-AbstractStateSplit n)  $\vee$  (is-BeginStateSplitNode n)  $\vee$  (is-StoreFieldNode
n))

fun is-ConvertNode :: IRNode  $\Rightarrow$  bool where
  is-ConvertNode n = ((is-IntegerConvertNode n))

```

```

fun is-sequential-node :: IRNode  $\Rightarrow$  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 regardless of their edges. It will return true if both the node parameters are the same node class.

```

fun is-same-ir-node-type :: IRNode  $\Rightarrow$  IRNode  $\Rightarrow$  bool where
is-same-ir-node-type n1 n2 = (
  ((is-AbsNode n1)  $\wedge$  (is-AbsNode n2))  $\vee$ 
  ((is-AddNode n1)  $\wedge$  (is-AddNode n2))  $\vee$ 
  ((is-AndNode n1)  $\wedge$  (is-AndNode n2))  $\vee$ 
  ((is-BeginNode n1)  $\wedge$  (is-BeginNode n2))  $\vee$ 
  ((is-BytecodeExceptionNode n1)  $\wedge$  (is-BytecodeExceptionNode n2))  $\vee$ 
  ((is-ConditionalNode n1)  $\wedge$  (is-ConditionalNode n2))  $\vee$ 
  ((is-ConstantNode n1)  $\wedge$  (is-ConstantNode n2))  $\vee$ 
  ((is-DynamicNewArrayNode n1)  $\wedge$  (is-DynamicNewArrayNode n2))  $\vee$ 
  ((is-EndNode n1)  $\wedge$  (is-EndNode n2))  $\vee$ 
  ((is-ExceptionObjectNode n1)  $\wedge$  (is-ExceptionObjectNode n2))  $\vee$ 
  ((is-FrameState n1)  $\wedge$  (is-FrameState n2))  $\vee$ 
  ((is-IfNode n1)  $\wedge$  (is-IfNode n2))  $\vee$ 
  ((is-IntegerBelowNode n1)  $\wedge$  (is-IntegerBelowNode n2))  $\vee$ 
  ((is-IntegerEqualsNode n1)  $\wedge$  (is-IntegerEqualsNode n2))  $\vee$ 
  ((is-IntegerLessThanNode n1)  $\wedge$  (is-IntegerLessThanNode n2))  $\vee$ 
  ((is-InvokeNode n1)  $\wedge$  (is-InvokeNode n2))  $\vee$ 
  ((is-InvokeWithExceptionNode n1)  $\wedge$  (is-InvokeWithExceptionNode n2))  $\vee$ 
  ((is-IsNullNode n1)  $\wedge$  (is-IsNullNode n2))  $\vee$ 
  ((is-KillingBeginNode n1)  $\wedge$  (is-KillingBeginNode n2))  $\vee$ 
  ((is-LoadFieldNode n1)  $\wedge$  (is-LoadFieldNode n2))  $\vee$ 
  ((is-LogicNegationNode n1)  $\wedge$  (is-LogicNegationNode n2))  $\vee$ 
  ((is-LoopBeginNode n1)  $\wedge$  (is-LoopBeginNode n2))  $\vee$ 

```

```

((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  $\Rightarrow$  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  $\Rightarrow$  Stamp  $\Rightarrow$  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  $\Rightarrow$  Stamp  $\Rightarrow$  bool where
```

```

  neverDistinct stamp1 stamp2 = (asConstant stamp1 = asConstant stamp2  $\wedge$ 
asConstant stamp1  $\neq$  UndefVal)
```

```

fun constantAsStamp :: Value  $\Rightarrow$  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 :: Stamp  $\Rightarrow$  Value  $\Rightarrow$  bool where
```

```

  valid-value (IntegerStamp b l h) (IntVal32 v) = (b=32  $\wedge$  (sint v  $\geq$  l)  $\wedge$  (sint v  $\leq$ 
h)) |
```

```

  valid-value (IntegerStamp b l h) (IntVal64 v) = (b=64  $\wedge$  (sint v  $\geq$  l)  $\wedge$  (sint v  $\leq$ 
h)) |
```

```

  valid-value (VoidStamp) (UndefVal) = True |
```

```

  valid-value (ObjectStamp klass exact nonNull alwaysNull) (ObjRef ref) =
```

```

  (if nonNull then ref $\neq$ None else True) |
```

```

  valid-value stamp val = False
```

— The most common type of stamp within the compiler (apart from the Void-Stamp) 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  $\rightarrow$  (IRNode  $\times$  Stamp) . finite (dom g)}
proof –
  have finite(dom(Map.empty))  $\wedge$  ran Map.empty = {} by auto
  then show ?thesis
    by fastforce
qed
```

setup-lifting *type-definition-IRGraph*

```
lift-definition ids :: IRGraph  $\Rightarrow$  ID set
  is  $\lambda g. \{nid \in dom\ g . \nexists s. g\ nid = (Some\ (NoNode,\ s))\}$  .
```

```
fun with-default :: 'c  $\Rightarrow$  ('b  $\Rightarrow$  'c)  $\Rightarrow$  (('a  $\rightarrow$  'b)  $\Rightarrow$  'a  $\Rightarrow$  'c) where
  with-default def conv = ( $\lambda m\ k.$ 
    (case m k of None  $\Rightarrow$  def | Some v  $\Rightarrow$  conv v))
```

```
lift-definition kind :: IRGraph  $\Rightarrow$  (ID  $\Rightarrow$  IRNode)
  is with-default NoNode fst .
```

```
lift-definition stamp :: IRGraph  $\Rightarrow$  ID  $\Rightarrow$  Stamp
  is with-default IllegalStamp snd .
```

```
lift-definition add-node :: ID  $\Rightarrow$  (IRNode  $\times$  Stamp)  $\Rightarrow$  IRGraph  $\Rightarrow$  IRGraph
  is  $\lambda nid\ k\ g.$  if fst k = NoNode then g else g(nid  $\mapsto$  k) by simp
```

```
lift-definition remove-node :: ID  $\Rightarrow$  IRGraph  $\Rightarrow$  IRGraph
  is  $\lambda nid\ g.$  g(nid := None) by simp
```

```
lift-definition replace-node :: ID  $\Rightarrow$  (IRNode  $\times$  Stamp)  $\Rightarrow$  IRGraph  $\Rightarrow$  IRGraph
  is  $\lambda nid\ k\ g.$  if fst k = NoNode then g else g(nid  $\mapsto$  k) by simp
```

```
lift-definition as-list :: IRGraph  $\Rightarrow$  (ID  $\times$  IRNode  $\times$  Stamp) list
  is  $\lambda g.$  map ( $\lambda k. (k, the\ (g\ k))$ ) (sorted-list-of-set (dom g)) .
```

```
fun no-node :: (ID  $\times$  (IRNode  $\times$  Stamp)) list  $\Rightarrow$  (ID  $\times$  (IRNode  $\times$  Stamp)) list
where
  no-node g = filter ( $\lambda n. fst\ (snd\ n) \neq NoNode$ ) g
```

```
lift-definition irgraph :: (ID  $\times$  (IRNode  $\times$  Stamp)) list  $\Rightarrow$  IRGraph
  is map-of  $\circ$  no-node
  by (simp add: finite-dom-map-of)
```

```
definition as-set :: IRGraph  $\Rightarrow$  (ID  $\times$  (IRNode  $\times$  Stamp)) set where
```

$as\text{-}set\ g = \{(n, kind\ g\ n, stamp\ g\ n) \mid n . n \in ids\ g\}$

definition *domain-subtraction* :: 'a set \Rightarrow ('a \times 'b) set \Rightarrow ('a \times 'b) set
 (infix ≤ 30) **where**
domain-subtraction s r = $\{(x, y) . (x, y) \in r \wedge x \notin s\}$

notation (*latex*)
domain-subtraction (- \triangleleft -)

code-datatype *irgraph*

fun *filter-none* **where**
filter-none g = $\{nid \in dom\ g . \nexists s. g\ nid = (Some\ (NoNode, s))\}$

lemma *no-node-clears*:
 $res = no\text{-}node\ xs \longrightarrow (\forall x \in set\ res. fst\ (snd\ x) \neq NoNode)$
by *simp*

lemma *dom-eq*:
assumes $\forall x \in set\ xs. fst\ (snd\ x) \neq NoNode$
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* \Rightarrow *ID* \Rightarrow *ID* set **where**
inputs g nid = set (inputs-of (kind g nid))

— Get the successor set of a given node ID

fun *succ* :: *IRGraph* \Rightarrow *ID* \Rightarrow *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* \Rightarrow *ID* rel **where**
input-edges g = $(\bigcup i \in ids\ g. \{(i, j) \mid j. j \in (inputs\ g\ i)\})$

— 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]*: $x \in \text{ids } g \longleftrightarrow \text{kind } g \ x \neq \text{NoNode}$

proof –

```

  have that:  $x \in \text{ids } g \longrightarrow \text{kind } g \ x \neq \text{NoNode}$ 
  using ids.rep-eq kind.rep-eq by force
  have  $\text{kind } g \ x \neq \text{NoNode} \longrightarrow x \in \text{ids } g$ 
  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  $\text{nid} \notin \text{ids } g$ 
  shows  $\text{kind } g \ \text{nid} = \text{NoNode}$ 
  using assms ids-some by blast

```

lemma *valid-creation[simp]*:

```

  finite (dom g)  $\longleftrightarrow \text{Rep-IRGraph } (\text{Abs-IRGraph } g) = g$ 
  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)  $\longrightarrow$  ids (Abs-IRGraph g) = {nid  $\in$  dom g .  $\nexists$  s. g
nid = Some (NoNode, s)}
  using ids.rep-eq by simp

lemma [simp]: finite (dom g)  $\longrightarrow$  kind (Abs-IRGraph g) = ( $\lambda$ x . (case g x of None
 $\Rightarrow$  NoNode | Some n  $\Rightarrow$  fst n))
  by (simp add: kind.rep-eq)

lemma [simp]: finite (dom g)  $\longrightarrow$  stamp (Abs-IRGraph g) = ( $\lambda$ x . (case g x of
None  $\Rightarrow$  IllegalStamp | Some n  $\Rightarrow$  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) = ( $\lambda$ nid. (case (map-of (no-node g)) nid of None
 $\Rightarrow$  NoNode | Some n  $\Rightarrow$  fst n))
  using irgraph.rep-eq kind.transfer kind.rep-eq by auto

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

lemma map-of-upd: (map-of g)(k  $\mapsto$  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 re-
place-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  $\longrightarrow$ 
  (if k  $\neq$  NoNode then kind gup nid = k  $\wedge$  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  $\longrightarrow$  kind gup nid = NoNode  $\wedge$  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  $\wedge$  k  $\neq$  NoNode  $\longrightarrow$  kind gup nid = k  $\wedge$  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  $\longrightarrow$  ( $\forall$  n  $\in$  (ids g - {nid}) . n  $\in$  ids g  $\wedge$  n  $\in$  ids
  gup  $\wedge$  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

5 Data-flow Semantics

```

theory IRTreeEval
  imports
    Graph.Values
    Graph.Stamp
    HOL-Library.Word
begin

```

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

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

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

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

```

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

```

```

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

```

```

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

```

```

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

```

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*

5.2 Data-flow Tree Evaluation

fun *unary-eval* :: *IRUnaryOp* \Rightarrow *Value* \Rightarrow *Value* **where**
unary-eval *UnaryAbs* *v* = *intval-abs* *v* |
unary-eval *UnaryNeg* *v* = *intval-negate* *v* |
unary-eval *UnaryNot* *v* = *intval-not* *v* |
unary-eval *UnaryLogicNegation* (*IntVal32* *v1*) = (if *v1* = 0 then (*IntVal32* 1) else
(*IntVal32* 0)) |
unary-eval *op* *v1* = *UndefVal*

fun *bin-eval* :: *IRBinaryOp* \Rightarrow *Value* \Rightarrow *Value* \Rightarrow *Value* **where**
bin-eval *BinAdd* *v1* *v2* = *intval-add* *v1* *v2* |
bin-eval *BinMul* *v1* *v2* = *intval-mul* *v1* *v2* |
bin-eval *BinSub* *v1* *v2* = *intval-sub* *v1* *v2* |
bin-eval *BinAnd* *v1* *v2* = *intval-and* *v1* *v2* |
bin-eval *BinOr* *v1* *v2* = *intval-or* *v1* *v2* |
bin-eval *BinXor* *v1* *v2* = *intval-xor* *v1* *v2* |
bin-eval *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* \Rightarrow *Value* \Rightarrow *bool* **where**
 $\llbracket \text{value} \neq \text{UndefVal} \rrbracket \Longrightarrow \text{not-undef-or-fail value value}$

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

inductive

evaltree :: *MapState* \Rightarrow *Params* \Rightarrow *IRExpr* \Rightarrow *Value* \Rightarrow *bool* ($\llbracket _, _ \rrbracket \vdash _ \mapsto _$ 55)
for *m p* **where**

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

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

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

$v \neq \text{UndefVal}$
 $\implies [m,p] \vdash (\text{ConditionalExpr } ce \text{ te } fe) \mapsto v \mid$

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

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

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

$$\begin{array}{c}
\frac{\text{valid-value } (\text{constantAsStamp } c) \ c}{[m,p] \vdash \text{ConstantExpr } c \mapsto c} \\
\\
\frac{i < |p| \quad \text{valid-value } s \ p[i]}{[m,p] \vdash \text{ParameterExpr } i \ s \mapsto p[i]} \\
\\
\frac{[m,p] \vdash ce \mapsto \text{cond} \quad \text{branch} = (\text{if } \text{IRTreeEval.val-to-bool } \text{cond} \ \text{then } te \ \text{else } fe) \quad [m,p] \vdash \text{branch} \mapsto v \quad v \neq \text{UndefVal}}{[m,p] \vdash \text{ConditionalExpr } ce \ te \ fe \mapsto v} \\
\\
\frac{[m,p] \vdash xe \mapsto v \quad \text{result} = \text{unary-eval } op \ v \quad \text{result} \neq \text{UndefVal}}{[m,p] \vdash \text{UnaryExpr } op \ xe \mapsto \text{result}} \\
\\
\frac{[m,p] \vdash ye \mapsto y \quad [m,p] \vdash xe \mapsto x \quad \text{result} = \text{bin-eval } op \ x \ y \quad \text{result} \neq \text{UndefVal}}{[m,p] \vdash \text{BinaryExpr } op \ xe \ ye \mapsto \text{result}} \\
\\
\frac{val = m \ n \quad \text{valid-value } s \ val}{[m,p] \vdash \text{LeafExpr } n \ s \mapsto val}
\end{array}$$

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

inductive

$\text{evaltrees} :: \text{MapState} \Rightarrow \text{Params} \Rightarrow \text{IRExpr list} \Rightarrow \text{Value list} \Rightarrow \text{bool } ([-,] \vdash - \mapsto_L$

- 55)
for $m\ p$ **where**

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

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

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

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

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

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

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

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

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

definition
 $lt\text{-}expr\text{-}def\ [simp]: (e1 < e2) \longleftrightarrow (e1 \leq e2 \wedge \neg (e1 \doteq e2))$

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

end

6 Data-flow Expression-Tree Theorems

```
theory IRTreeEvalThms
  imports
    TreeToGraph
    HOL-Eisbach.Eisbach
begin
```

6.1 Extraction and Evaluation of Expression Trees is Deterministic.

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

named-theorems *rep*

```
lemma rep-constant [rep]:
   $g \vdash n \simeq e \implies$ 
   $\text{kind } g \ n = \text{ConstantNode } c \implies$ 
   $e = \text{ConstantExpr } c$ 
  by (induction rule: rep.induct; auto)
```

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

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

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

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

lemma *rep-negate* [rep]:

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

lemma *rep-logicnegation* [rep]:

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

lemma *rep-add* [rep]:

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

lemma *rep-sub* [rep]:

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

lemma *rep-mul* [rep]:

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

lemma *rep-and* [rep]:

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

lemma *rep-or* [rep]:

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

lemma *rep-xor* [rep]:

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

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

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

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

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

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

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

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

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

method *solve-det* **uses** *node* =
 (match *node* **in** *kind* - - = *node* - **for** *node* \implies
 (match *rep* **in** *r*: - \implies - = *node* - \implies - \implies
 (match *IRNode.inject* **in** *i*: (*node* - = *node* -) = - \implies
 (match *RepE* **in** *e*: - \implies ($\bigwedge x. - = \text{node } x \implies -$) \implies - \implies
 (*metis i e r*) |
 match *node* **in** *kind* - - = *node* - - **for** *node* \implies


```

    <match rep in r: - ==> - = node - - ==> - ==>
    <match IRNode.inject in i: (node - - = node - -) = - ==>
    <match RepE in e: - ==> (Λx y. - = node x y ==> -) ==> - ==>
    <metis i e r>>> |
  match node in kind - - = node - - - for node ==>
  <match rep in r: - ==> - = node - - - ==> - ==>
  <match IRNode.inject in i: (node - - - = node - - -) = - ==>
  <match RepE in e: - ==> (Λx y z. - = node x y z ==> -) ==> - ==>
  <metis i e r>>> |
  match node in kind - - = node - - - for node ==>
  <match rep in r: - ==> - = node - - - ==> - ==>
  <match IRNode.inject in i: (node - - - = node - - -) = - ==>
  <match RepE in e: - ==> (Λx. - = node - - x ==> -) ==> - ==>
  <metis i e r>>>)

```

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 using rep-parameter by auto
next
  case (ConditionalNode n c t f ce te fe)
  then show ?case
    by (solve-det node: ConditionalNode)
next
  case (AbsNode n x xe)
  then show ?case
    by (solve-det node: AbsNode)
next
  case (NotNode n x xe)
  then show ?case
    by (solve-det node: NotNode)
next
  case (NegateNode n x xe)
  then show ?case
    by (solve-det node: NegateNode)
next
  case (LogicNegationNode n x xe)
  then show ?case
    by (solve-det node: LogicNegationNode)
next
  case (AddNode n x y xe ye)
  then show ?case
    by (solve-det node: AddNode)

```

```

next
  case (MulNode n x y xe ye)
  then show ?case
    by (solve-det node: MulNode)
next
  case (SubNode n x y xe ye)
  then show ?case
    by (solve-det node: SubNode)
next
  case (AndNode n x y xe ye)
  then show ?case
    by (solve-det node: AndNode)
next
  case (OrNode n x y xe ye)
  then show ?case
    by (solve-det node: OrNode)
next
  case (XorNode n x y xe ye)
  then show ?case
    by (solve-det node: XorNode)
next
  case (IntegerBelowNode n x y xe ye)
  then show ?case
    by (solve-det node: IntegerBelowNode)
next
  case (IntegerEqualsNode n x y xe ye)
  then show ?case
    by (solve-det node: IntegerEqualsNode)
next
  case (IntegerLessThanNode n x y xe ye)
  then show ?case
    by (solve-det node: IntegerLessThanNode)
next
  case (NarrowNode n x xe)
  then show ?case
    by (metis IRNode.inject(28) NarrowNodeE rep-narrow)
next
  case (SignExtendNode n x xe)
  then show ?case
    using SignExtendNodeE rep-sign-extend IRNode.inject(39)
    by (metis IRNode.inject(39) SignExtendNodeE rep-sign-extend)
next
  case (ZeroExtendNode n x xe)
  then show ?case
    by (metis IRNode.inject(50) ZeroExtendNodeE rep-zero-extend)
next
  case (LeafNode n s)
  then show ?case using rep-load-field LeafNodeE by blast
qed

```

```

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

```

```

lemma evalDet:
   $[m,p] \vdash e \mapsto v1 \implies$ 
   $[m,p] \vdash e \mapsto v2 \implies$ 
   $v1 = v2$ 
apply (induction arbitrary: v2 rule: evaltree.induct)
by (elim EvalTreeE; auto)+

```

```

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

```

```

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

```

```

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

```

A valid value cannot be *UndefVal*.

```

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

```

```

lemma valid-VoidStamp[elim]:
  shows valid-value VoidStamp val  $\implies$ 
    val = UndefVal
  using valid-value.simps by (metis IRTreeEval.val-to-bool.cases)

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

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

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

TODO: could we prove that expression evaluation never returns UndefVal?  

  But this might require restricting unary and binary operators to be total...

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

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

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

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

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

```

lemma *valid32* [*simp*]:
assumes *valid-value* (*IntegerStamp* 32 *lo hi*) *val*
shows $\exists v. (val = (IntVal32\ v) \wedge lo \leq sint\ v \wedge sint\ v \leq hi)$
using *assms valid-int32* **by** *force*

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

experiment begin

lemma *int-stamp-implies-valid-value*:

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

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

lemma *valid32or64-both*:

assumes *valid-value* (*IntegerStamp* *b lox hix*) *x*
and *valid-value* (*IntegerStamp* *b loy hiy*) *y*
shows $(\exists v1\ v2. x = IntVal32\ v1 \wedge y = IntVal32\ v2) \vee (\exists v3\ v4. x = IntVal64\ v3 \wedge y = IntVal64\ v4)$
using *assms valid32or64 valid32 valid-value.elims(2) valid-value.simps(1)* **by** *metis*

6.2 Example Data-flow Optimisations

```

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

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

```

```

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

```

6.3 Monotonicity of Expression Optimization

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

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

```

lemma mono-unary:
  assumes e ≥ 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  $(BinaryExpr\ op\ x\ y) \geq (BinaryExpr\ op\ x'\ y')$ 
  using BinaryExpr assms by auto

lemma mono-conditional:
  assumes  $ce \geq ce'$ 
  assumes  $te \geq te'$ 
  assumes  $fe \geq fe'$ 
  shows  $(ConditionalExpr\ ce\ te\ fe) \geq (ConditionalExpr\ ce'\ te'\ fe')$ 
proof (simp only: le-expr-def; (rule allI)+; rule impI)
  fix  $m\ p\ v$ 
  assume  $a: [m,p] \vdash ConditionalExpr\ ce\ te\ fe \mapsto v$ 
  then obtain  $cond$  where  $ce: [m,p] \vdash ce \mapsto cond$  by auto
  then have  $ce': [m,p] \vdash ce' \mapsto cond$  using assms by auto
  define  $branch$  where  $b: branch = (if\ val\text{-}to\text{-}bool\ cond\ then\ te\ else\ fe)$ 
  define  $branch'$  where  $b': branch' = (if\ val\text{-}to\text{-}bool\ cond\ then\ te'\ else\ fe')$ 
  then have  $[m,p] \vdash branch \mapsto v$  using  $a\ b\ ce\ evalDet$  by blast
  then have  $[m,p] \vdash branch' \mapsto v$  using assms  $b\ b'$  by auto
  then show  $[m,p] \vdash ConditionalExpr\ ce'\ te'\ fe' \mapsto v$ 
    using  $ConditionalExpr\ ce'\ b'$ 
    using  $a$  by blast
qed

end

```

7 Tree to Graph

```

theory TreeToGraph
  imports
    Semantics.IRTreeEval
    Graph.IRGraph
  begin

  fun find-node-and-stamp :: IRGraph  $\Rightarrow$  (IRNode  $\times$  Stamp)  $\Rightarrow$  ID option where
    find-node-and-stamp  $g\ (n,s) =$ 
      find  $(\lambda i. kind\ g\ i = n \wedge stamp\ g\ i = s)$  (sorted-list-of-set(ids  $g$ ))

  export-code find-node-and-stamp

  fun is-preevaluated :: IRNode  $\Rightarrow$  bool where
    is-preevaluated (InvokeNode  $n\ -\ -\ -\ -$ ) = True |
    is-preevaluated (InvokeWithExceptionNode  $n\ -\ -\ -\ -$ ) = True |

```

$is_preevaluated \ (NewInstanceNode \ n \ - \ -) = True \mid$
 $is_preevaluated \ (LoadFieldNode \ n \ - \ -) = True \mid$
 $is_preevaluated \ (SignedDivNode \ n \ - \ - \ - \ -) = True \mid$
 $is_preevaluated \ (SignedRemNode \ n \ - \ - \ - \ -) = True \mid$
 $is_preevaluated \ (ValuePhiNode \ n \ - \ -) = True \mid$
 $is_preevaluated \ - = False$

inductive

$rep :: IRGraph \Rightarrow ID \Rightarrow IRExpr \Rightarrow bool \ (- \vdash \ - \simeq \ - \ 55)$
for g where

ConstantNode:

$\llbracket kind \ g \ n = ConstantNode \ c \rrbracket$
 $\implies g \vdash n \simeq (ConstantExpr \ c) \mid$

ParameterNode:

$\llbracket kind \ g \ n = ParameterNode \ i;$
 $\quad stamp \ g \ n = s \rrbracket$
 $\implies g \vdash n \simeq (ParameterExpr \ i \ s) \mid$

ConditionalNode:

$\llbracket kind \ g \ n = ConditionalNode \ c \ t \ f;$
 $\quad g \vdash c \simeq ce;$
 $\quad g \vdash t \simeq te;$
 $\quad g \vdash f \simeq fe \rrbracket$
 $\implies g \vdash n \simeq (ConditionalExpr \ ce \ te \ fe) \mid$

AbsNode:

$\llbracket kind \ g \ n = AbsNode \ x;$
 $\quad g \vdash x \simeq xe \rrbracket$
 $\implies g \vdash n \simeq (UnaryExpr \ UnaryAbs \ xe) \mid$

NotNode:

$\llbracket kind \ g \ n = NotNode \ x;$
 $\quad g \vdash x \simeq xe \rrbracket$
 $\implies g \vdash n \simeq (UnaryExpr \ UnaryNot \ xe) \mid$

NegateNode:

$\llbracket kind \ g \ n = NegateNode \ x;$
 $\quad g \vdash x \simeq xe \rrbracket$
 $\implies g \vdash n \simeq (UnaryExpr \ UnaryNeg \ xe) \mid$

LogicNegationNode:

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

AddNode:

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

MulNode:

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

SubNode:

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

AndNode:

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

OrNode:

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

XorNode:

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

IntegerBelowNode:

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

IntegerEqualsNode:

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

IntegerLessThanNode:

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

NarrowNode:

$\llbracket \text{kind } g \ n = \text{NarrowNode } \text{inputBits } \text{resultBits } x; \\ g \vdash x \simeq xe \rrbracket \\ \implies g \vdash n \simeq (\text{UnaryExpr } (\text{UnaryNarrow } \text{inputBits } \text{resultBits}) \ xe) \mid$

SignExtendNode:

$\llbracket \text{kind } g \ n = \text{SignExtendNode } \text{inputBits } \text{resultBits } x; \\ g \vdash x \simeq xe \rrbracket \\ \implies g \vdash n \simeq (\text{UnaryExpr } (\text{UnarySignExtend } \text{inputBits } \text{resultBits}) \ xe) \mid$

ZeroExtendNode:

$\llbracket \text{kind } g \ n = \text{ZeroExtendNode } \text{inputBits } \text{resultBits } x; \\ g \vdash x \simeq xe \rrbracket \\ \implies g \vdash n \simeq (\text{UnaryExpr } (\text{UnaryZeroExtend } \text{inputBits } \text{resultBits}) \ xe) \mid$

LeafNode:

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

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

inductive

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

RepNil:

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

RepCons:

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

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

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

$$\begin{array}{c}
\frac{\text{kind } g \ n = \text{ParameterNode } i \quad \text{stamp } g \ n = s}{g \vdash n \simeq \text{ParameterExpr } i \ s} \\
\\
\frac{\text{kind } g \ n = \text{AbsNode } x \quad g \vdash x \simeq xe}{g \vdash n \simeq \text{UnaryExpr } \text{UnaryAbs } xe} \\
\\
\frac{\text{kind } g \ n = \text{AddNode } x \ y \quad g \vdash x \simeq xe \quad g \vdash y \simeq ye}{g \vdash n \simeq \text{BinaryExpr } \text{BinAdd } xe \ ye} \\
\\
\frac{\text{kind } g \ n = \text{MulNode } x \ y \quad g \vdash x \simeq xe \quad g \vdash y \simeq ye}{g \vdash n \simeq \text{BinaryExpr } \text{BinMul } xe \ ye} \\
\\
\frac{\text{kind } g \ n = \text{SubNode } x \ y \quad g \vdash x \simeq xe \quad g \vdash y \simeq ye}{g \vdash n \simeq \text{BinaryExpr } \text{BinSub } xe \ ye} \\
\\
\frac{\text{is-preevaluated } (\text{kind } g \ n) \quad \text{stamp } g \ n = s}{g \vdash n \simeq \text{LeafExpr } n \ s}
\end{array}$$

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

fun *stamp-unary* :: *IRUnaryOp* \Rightarrow *Stamp* \Rightarrow *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* \Rightarrow *Stamp* \Rightarrow *Stamp* \Rightarrow *Stamp* **where**
stamp-binary op (*IntegerStamp* *b1 lo1 hi1*) (*IntegerStamp* *b2 lo2 hi2*) =
 (case *op* \in *fixed-32* of *True* \Rightarrow *unrestricted-stamp* (*IntegerStamp* 32 *lo1 hi1*) |
False \Rightarrow
 (if (*b1* = *b2*) then *unrestricted-stamp* (*IntegerStamp* *b1 lo1 hi1*) else *IllegalStamp*)) |

stamp-binary op - - = *IllegalStamp*

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

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

```

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

```

```

fun bin-node :: IRBinaryOp  $\Rightarrow$  ID  $\Rightarrow$  ID  $\Rightarrow$  IRNode where
  bin-node BinAdd x y = AddNode x y |
  bin-node BinMul x y = MulNode x y |
  bin-node BinSub x y = SubNode x y |
  bin-node BinAnd x y = AndNode x y |
  bin-node BinOr x y = OrNode x y |
  bin-node BinXor x y = XorNode x y |
  bin-node 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  $\Rightarrow$  int64  $\Rightarrow$  Value where
  choose-32-64 bits val =
    (if bits = 32
     then (IntVal32 (ucast val))
     else (IntVal64 (val)))

```

```

inductive fresh-id :: IRGraph  $\Rightarrow$  ID  $\Rightarrow$  bool where
  n  $\notin$  ids g  $\Longrightarrow$  fresh-id g n

```

```

code-pred fresh-id .

```

```

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

```

```

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

```

```

export-code get-fresh-id

```

```

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

```

inductive

unrep :: *IRGraph* \Rightarrow *IRExpr* \Rightarrow (*IRGraph* \times *ID*) \Rightarrow *bool* (- \triangleleft - \rightsquigarrow - 55)
and
unrepList :: *IRGraph* \Rightarrow *IRExpr list* \Rightarrow (*IRGraph* \times *ID list*) \Rightarrow *bool* (- \triangleleft_L - \rightsquigarrow - 55)
where

ConstantNodeSame:

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

ConstantNodeNew:

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

ParameterNodeSame:

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

ParameterNodeNew:

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

ConditionalNodeSame:

$\llbracket g \triangleleft_L [ce, te, fe] \rightsquigarrow (g2, [c, t, f]);$
 $s' = \text{meet (stamp } g2 \text{ } t) \text{ (stamp } g2 \text{ } f);$
 $\text{find-node-and-stamp } g2 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{Some } n \rrbracket$
 $\implies g \triangleleft (\text{ConditionalExpr } ce \text{ } te \text{ } fe) \rightsquigarrow (g2, n) \mid$

ConditionalNodeNew:

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

UnaryNodeSame:

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

UnaryNodeNew:

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

BinaryNodeSame:

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

BinaryNodeNew:

$$\begin{aligned} & \llbracket g \triangleleft_L [xe, ye] \rightsquigarrow (g2, [x, y]); \\ & \quad s' = \text{stamp-binary } op \ (\text{stamp } g2 \ x) \ (\text{stamp } g2 \ y); \\ & \quad \text{find-node-and-stamp } g2 \ (\text{bin-node } op \ x \ y, s') = \text{None}; \\ & \quad n = \text{get-fresh-id } g2; \\ & \quad g' = \text{add-node } n \ (\text{bin-node } op \ x \ y, s') \ g2 \rrbracket \\ & \implies g \triangleleft (\text{BinaryExpr } op \ xe \ ye) \rightsquigarrow (g', n) \mid \end{aligned}$$

AllLeafNodes:

$$\begin{aligned} & \text{stamp } g \ n = s \\ & \implies g \triangleleft (\text{LeafExpr } n \ s) \rightsquigarrow (g, n) \mid \end{aligned}$$

UnrepNil:

$$g \triangleleft_L [] \rightsquigarrow (g, []) \mid$$

UnrepCons:

$$\begin{aligned} & \llbracket g \triangleleft xe \rightsquigarrow (g2, x); \\ & \quad g2 \triangleleft_L xes \rightsquigarrow (g3, xs) \rrbracket \\ & \implies g \triangleleft_L (xe \# xes) \rightsquigarrow (g3, x \# xs) \end{aligned}$$

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

unrep .

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

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

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

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

$$\begin{array}{c}
\frac{\text{find-node-and-stamp } g \text{ (ParameterNode } i, s) = \text{None} \quad n = \text{get-fresh-id } g \quad g' = \text{add-node } n \text{ (ParameterNode } i, s) \text{ } g}{g \triangleleft \text{ParameterExpr } i \text{ } s \rightsquigarrow (g', n)} \\
\\
\frac{g \triangleleft_L [ce, te, fe] \rightsquigarrow (g2, [c, t, f]) \quad s' = \text{meet (stamp } g2 \text{ } t) \text{ (stamp } g2 \text{ } f) \quad \text{find-node-and-stamp } g2 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{Some } n}{g \triangleleft \text{ConditionalExpr } ce \text{ } te \text{ } fe \rightsquigarrow (g2, n)} \\
\\
\frac{g \triangleleft_L [ce, te, fe] \rightsquigarrow (g2, [c, t, f]) \quad s' = \text{meet (stamp } g2 \text{ } t) \text{ (stamp } g2 \text{ } f) \quad \text{find-node-and-stamp } g2 \text{ (ConditionalNode } c \text{ } t \text{ } f, s') = \text{None} \quad n = \text{get-fresh-id } g2 \quad g' = \text{add-node } n \text{ (ConditionalNode } c \text{ } t \text{ } f, s') \text{ } g2}{g \triangleleft \text{ConditionalExpr } ce \text{ } te \text{ } fe \rightsquigarrow (g', n)} \\
\\
\frac{g \triangleleft_L [xe, ye] \rightsquigarrow (g2, [x, y]) \quad s' = \text{stamp-binary op (stamp } g2 \text{ } x) \text{ (stamp } g2 \text{ } y) \quad \text{find-node-and-stamp } g2 \text{ (bin-node op } x \text{ } y, s') = \text{Some } n}{g \triangleleft \text{BinaryExpr op } xe \text{ } ye \rightsquigarrow (g2, n)} \\
\\
\frac{g \triangleleft_L [xe, ye] \rightsquigarrow (g2, [x, y]) \quad s' = \text{stamp-binary op (stamp } g2 \text{ } x) \text{ (stamp } g2 \text{ } y) \quad \text{find-node-and-stamp } g2 \text{ (bin-node op } x \text{ } y, s') = \text{None} \quad n = \text{get-fresh-id } g2 \quad g' = \text{add-node } n \text{ (bin-node op } x \text{ } y, s') \text{ } g2}{g \triangleleft \text{BinaryExpr op } xe \text{ } ye \rightsquigarrow (g', n)} \\
\\
\frac{g \triangleleft xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op (stamp } g2 \text{ } x) \quad \text{find-node-and-stamp } g2 \text{ (unary-node op } x, s') = \text{Some } n}{g \triangleleft \text{UnaryExpr op } xe \rightsquigarrow (g2, n)} \\
\\
\frac{g \triangleleft xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op (stamp } g2 \text{ } x) \quad \text{find-node-and-stamp } g2 \text{ (unary-node op } x, s') = \text{None} \quad n = \text{get-fresh-id } g2 \quad g' = \text{add-node } n \text{ (unary-node op } x, s') \text{ } g2}{g \triangleleft \text{UnaryExpr op } xe \rightsquigarrow (g', n)} \\
\\
\frac{\text{stamp } g \text{ } n = s}{g \triangleleft \text{LeafExpr } n \text{ } s \rightsquigarrow (g, n)}
\end{array}$$

definition *sq-param0* :: *IRExpr* **where**

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

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

definition *encodeeval* :: *IRGraph* \Rightarrow *MapState* \Rightarrow *Params* \Rightarrow *ID* \Rightarrow *Value* \Rightarrow *bool*
 $([\cdot, \cdot, \cdot] \vdash - \mapsto - \text{ } 50)$
where

$encodeeval\ g\ m\ p\ n\ v = (\exists\ e. (g \vdash n \simeq e) \wedge ([m,p] \vdash e \mapsto v))$

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

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

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

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

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

end
theory $TreeToGraphThms$
imports
 $TreeToGraph$
 $IRTreeEvalThms$
 $HOL\text{-}Eisbach.Eisbach$
begin

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

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

lemma $mono\text{-}not$:

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

lemma *mono-negate*:

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

lemma *mono-logic-negation*:

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

lemma *mono-narrow*:

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

lemma *mono-sign-extend*:

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

lemma *mono-zero-extend*:

assumes $\text{kind } g1 \ n = \text{ZeroExtendNode } ib \ rb \ x \wedge \text{kind } g2 \ n = \text{ZeroExtendNode } ib \ rb \ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$

shows $e1 \geq e2$
 using *assms mono-unary repDet ZeroExtendNode*
 by *metis*

lemma *mono-conditional-graph:*

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

lemma *mono-add:*

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

lemma *mono-mul:*

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

lemma *encodes-contains:*

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

lemma *no-encoding:*

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

```

lemma not-excluded-keep-type:
  assumes  $n \in \text{ids } g1$ 
  assumes  $n \notin \text{excluded}$ 
  assumes  $(\text{excluded} \sqsubseteq \text{as-set } g1) \subseteq \text{as-set } g2$ 
  shows  $\text{kind } g1 \ n = \text{kind } g2 \ n \wedge \text{stamp } g1 \ n = \text{stamp } g2 \ n$ 
  using assms unfolding as-set-def domain-subtraction-def by blast

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

lemma graph-semantics-preservation:
  assumes  $a: e1' \geq e2'$ 
  assumes  $b: (\{n'\} \sqsubseteq \text{as-set } g1) \subseteq \text{as-set } g2$ 
  assumes  $c: g1 \vdash n' \simeq e1'$ 
  assumes  $d: g2 \vdash n' \simeq e2'$ 
  shows graph-refinement  $g1 \ g2$ 
  unfolding graph-refinement-def
  apply (rule allI) apply (rule impI) apply (rule allI) apply (rule impI)
proof -
  fix  $n \ e1$ 
  assume  $e: n \in \text{ids } g1$ 
  assume  $f: (g1 \vdash n \simeq e1)$ 

  show  $\exists \ e2. (g2 \vdash n \simeq e2) \wedge e1 \geq e2$ 
  proof (cases  $n = n'$ )
    case True
      have  $g: e1 = e1'$  using  $c \ f \ \text{True} \ \text{repDet}$  by simp
      have  $h: (g2 \vdash n \simeq e2') \wedge e1' \geq e2'$ 
        using  $\text{True } a \ d$  by blast
      then show ?thesis
        using  $g$  by blast
    next
      case False
      have  $n \notin \{n'\}$ 
        using False by simp
      then have  $i: \text{kind } g1 \ n = \text{kind } g2 \ n \wedge \text{stamp } g1 \ n = \text{stamp } g2 \ n$ 
        using not-excluded-keep-type
        using  $b \ e$  by presburger
      show ?thesis using  $f \ i$ 
      proof (induction  $e1$ )
        case (ConstantNode  $n \ c$ )

```

```

    then show ?case
      by (metis eq-refl rep.ConstantNode)
  next
    case (ParameterNode n i s)
    then show ?case
      by (metis eq-refl rep.ParameterNode)
  next
    case (ConditionalNode n c t f ce1 te1 fe1)
    have k:  $g1 \vdash n \simeq \text{ConditionalExpr } ce1 \ te1 \ fe1$  using f ConditionalNode
      by (simp add: ConditionalNode.hyps(2) rep.ConditionalNode)
    obtain cn tn fn where l:  $\text{kind } g1 \ n = \text{ConditionalNode } cn \ tn \ fn$ 
      using ConditionalNode.hyps(1) by blast
    then have mc:  $g1 \vdash cn \simeq ce1$ 
      using ConditionalNode.hyps(1) ConditionalNode.hyps(2) by fastforce
    from l have mt:  $g1 \vdash tn \simeq te1$ 
      using ConditionalNode.hyps(1) ConditionalNode.hyps(3) by fastforce
    from l have mf:  $g1 \vdash fn \simeq fe1$ 
      using ConditionalNode.hyps(1) ConditionalNode.hyps(4) by fastforce
    then show ?case
    proof -
      have  $g1 \vdash cn \simeq ce1$  using mc by simp
      have  $g1 \vdash tn \simeq te1$  using mt by simp
      have  $g1 \vdash fn \simeq fe1$  using mf by simp
      have cer:  $\exists \ ce2. (g2 \vdash cn \simeq ce2) \wedge ce1 \geq ce2$ 
        using ConditionalNode
        using a b c d l no-encoding not-excluded-keep-type repDet singletonD
        by (metis-node-eq-ternary ConditionalNode)
      have ter:  $\exists \ te2. (g2 \vdash tn \simeq te2) \wedge te1 \geq te2$ 
        using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
        singletonD
        by (metis-node-eq-ternary ConditionalNode)
      have  $\exists \ fe2. (g2 \vdash fn \simeq fe2) \wedge fe1 \geq fe2$ 
        using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
        singletonD
        by (metis-node-eq-ternary ConditionalNode)
      then have  $\exists \ ce2 \ te2 \ fe2. (g2 \vdash n \simeq \text{ConditionalExpr } ce2 \ te2 \ fe2) \wedge$ 
        ConditionalExpr ce1 te1 fe1  $\geq \text{ConditionalExpr } ce2 \ te2 \ fe2$ 
      using ConditionalNode.prem1 l mono-conditional rep.ConditionalNode cer
      ter
      by (smt (verit) IRTreeEvalThms.mono-conditional)
    then show ?thesis
      by meson
  qed
next
  case (AbsNode n x xe1)
  have k:  $g1 \vdash n \simeq \text{UnaryExpr } \text{UnaryAbs } xe1$  using f AbsNode
    by (simp add: AbsNode.hyps(2) rep.AbsNode)
  obtain xn where l:  $\text{kind } g1 \ n = \text{AbsNode } xn$ 
    using AbsNode.hyps(1) by blast

```

```

then have m:  $g1 \vdash xn \simeq xe1$ 
  using AbsNode.hyps(1) AbsNode.hyps(2) by fastforce
then show ?case
proof (cases  $xn = n'$ )
  case True
  then have n:  $xe1 = e1'$  using c m repDet by simp
  then have ev:  $g2 \vdash n \simeq \text{UnaryExpr UnaryAbs } e2'$  using AbsNode.hyps(1)
l m n
    using AbsNode.premis True d rep.AbsNode by simp
  then have r:  $\text{UnaryExpr UnaryAbs } e1' \geq \text{UnaryExpr UnaryAbs } e2'$ 
    by (meson a mono-unary)
  then show ?thesis using ev r
    by (metis n)
next
  case False
  have  $g1 \vdash xn \simeq xe1$  using m by simp
  have  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
    using AbsNode
  using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
  by (metis-node-eq-unary AbsNode)
  then have  $\exists xe2. (g2 \vdash n \simeq \text{UnaryExpr UnaryAbs } xe2) \wedge \text{UnaryExpr}$ 
UnaryAbs  $xe1 \geq \text{UnaryExpr UnaryAbs } xe2$ 
    by (metis AbsNode.premis l mono-unary rep.AbsNode)
  then show ?thesis
    by meson
qed
next
  case (NotNode n x xe1)
  have k:  $g1 \vdash n \simeq \text{UnaryExpr UnaryNot } xe1$  using f NotNode
    by (simp add: NotNode.hyps(2) rep.NotNode)
  obtain xn where l: kind  $g1 \ n = \text{NotNode } xn$ 
    using NotNode.hyps(1) by blast
  then have m:  $g1 \vdash xn \simeq xe1$ 
    using NotNode.hyps(1) NotNode.hyps(2) by fastforce
  then show ?case
proof (cases  $xn = n'$ )
  case True
  then have n:  $xe1 = e1'$  using c m repDet by simp
  then have ev:  $g2 \vdash n \simeq \text{UnaryExpr UnaryNot } e2'$  using NotNode.hyps(1)
l m n
    using NotNode.premis True d rep.NotNode by simp
  then have r:  $\text{UnaryExpr UnaryNot } e1' \geq \text{UnaryExpr UnaryNot } e2'$ 
    by (meson a mono-unary)
  then show ?thesis using ev r
    by (metis n)
next
  case False
  have  $g1 \vdash xn \simeq xe1$  using m by simp
  have  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 

```

```

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

```

```

    using LogicNegationNode.hyps(1) LogicNegationNode.hyps(2) by fastforce
  then show ?case
  proof (cases  $xn = n'$ )
    case True
    then have  $n: xe1 = e1'$  using  $c m repDet$  by simp
    then have  $ev: g2 \vdash n \simeq UnaryExpr UnaryLogicNegation e2'$  using
      LogicNegationNode.hyps(1)  $l m n$ 
    using LogicNegationNode.prem1 True  $d rep.LogicNegationNode$  by simp
    then have  $r: UnaryExpr UnaryLogicNegation e1' \geq UnaryExpr UnaryLog-$ 
       $icNegation e2'$ 
    by (meson a mono-unary)
    then show ?thesis using  $ev r$ 
    by (metis  $n$ )
  next
  case False
  have  $g1 \vdash xn \simeq xe1$  using  $m$  by simp
  have  $\exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
  using LogicNegationNode
  using False i b l not-excluded-keep-type singletonD no-encoding
  by (metis-node-eq-unary LogicNegationNode)
  then have  $\exists xe2. (g2 \vdash n \simeq UnaryExpr UnaryLogicNegation xe2) \wedge$ 
     $UnaryExpr UnaryLogicNegation xe1 \geq UnaryExpr UnaryLogicNegation xe2$ 
  by (metis LogicNegationNode.prem1 l mono-unary rep.LogicNegationNode)
  then show ?thesis
  by meson
qed
next
case (AddNode  $n x y xe1 ye1$ )
have  $k: g1 \vdash n \simeq BinaryExpr BinAdd xe1 ye1$  using  $f AddNode$ 
  by (simp add: AddNode.hyps(2) rep.AddNode)
obtain  $xn yn$  where  $l: kind g1 n = AddNode xn yn$ 
  using AddNode.hyps(1) by blast
then have  $mx: g1 \vdash xn \simeq xe1$ 
  using AddNode.hyps(1) AddNode.hyps(2) by fastforce
from  $l$  have  $my: g1 \vdash yn \simeq ye1$ 
  using AddNode.hyps(1) AddNode.hyps(3) by fastforce
then show ?case
proof -
  have  $g1 \vdash xn \simeq xe1$  using  $mx$  by simp
  have  $g1 \vdash yn \simeq ye1$  using  $my$  by simp
  have  $xer: \exists xe2. (g2 \vdash xn \simeq xe2) \wedge xe1 \geq xe2$ 
  using AddNode
  using a b c d l no-encoding not-excluded-keep-type repDet singletonD
  by (metis-node-eq-binary AddNode)
  have  $\exists ye2. (g2 \vdash yn \simeq ye2) \wedge ye1 \geq ye2$ 
  using AddNode
  using a b c d l no-encoding not-excluded-keep-type repDet singletonD
  by (metis-node-eq-binary AddNode)
  then have  $\exists xe2 ye2. (g2 \vdash n \simeq BinaryExpr BinAdd xe2 ye2) \wedge BinaryExpr$ 

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

      qed
    next
      case (LeafNode n s)
    then show ?case
      by (metis eq-refl rep.LeanNode)
    qed
  qed
qed

```

definition *maximal-sharing*:

$$\text{maximal-sharing } g = (\forall \ n1 \ n2 . n1 \in \text{ids } g \wedge n2 \in \text{ids } g \longrightarrow \\ (\forall \ e. (g \vdash n1 \simeq e) \wedge (g \vdash n2 \simeq e) \longrightarrow n1 = n2))$$

lemma *tree-to-graph-rewriting*:

```

  e1 ≥ e2
  ∧ (g1 ⊢ n ≃ e1) ∧ maximal-sharing g1
  ∧ ({n} ⊆ as-set g1) ⊆ as-set g2
  ∧ (g2 ⊢ n ≃ e2) ∧ maximal-sharing g2
  ⇒ graph-refinement g1 g2
  using graph-semantics-preservation
  by auto

```

declare *[[simp-trace]]*

lemma *equal-refines*:

```

  fixes e1 e2 :: IRExp
  assumes e1 = e2
  shows e1 ≥ e2
  using assms
  by simp

```

declare *[[simp-trace=false]]*

lemma *subset-implies-evals*:

```

  assumes as-set g1 ⊆ as-set g2
  shows (g1 ⊢ n ≃ e) ⇒ (g2 ⊢ n ≃ e)
proof (induction e arbitrary: n)
  case (UnaryExpr op e)
  then have n ∈ ids g1
    using no-encoding by force
  then have kind g1 n = kind g2 n
    using assms unfolding as-set-def
    by blast
  then show ?case using UnaryExpr UnaryRepE
    by (smt (verit, ccfv-threshold) AbsNode LogicNegationNode NarrowNode NegateNode
      NotNode SignExtendNode ZeroExtendNode)

```

```

next
  case (BinaryExpr op e1 e2)
  then have  $n \in \text{ids } g1$ 
    using no-encoding by force
  then have  $\text{kind } g1 \ n = \text{kind } g2 \ n$ 
    using assms unfolding as-set-def
    by blast
  then show ?case using BinaryExpr BinaryRepE
    by (smt (verit, ccfv-threshold) AddNode MulNode SubNode AndNode OrNode
XorNode IntegerBelowNode IntegerEqualsNode IntegerLessThanNode)
next
  case (ConditionalExpr e1 e2 e3)
  then have  $n \in \text{ids } g1$ 
    using no-encoding by force
  then have  $\text{kind } g1 \ n = \text{kind } g2 \ n$ 
    using assms unfolding as-set-def
    by blast
  then show ?case using ConditionalExpr ConditionalExprE
    by (smt (verit, best) ConditionalNode ConditionalNodeE)
next
  case (ConstantExpr x)
  then have  $n \in \text{ids } g1$ 
    using no-encoding by force
  then have  $\text{kind } g1 \ n = \text{kind } g2 \ n$ 
    using assms unfolding as-set-def
    by blast
  then show ?case using ConstantExpr ConstantExprE
    by (metis ConstantNode ConstantNodeE)
next
  case (ParameterExpr x1 x2)
  then have in-g1:  $n \in \text{ids } g1$ 
    using no-encoding by force
  then have kinds:  $\text{kind } g1 \ n = \text{kind } g2 \ n$ 
    using assms unfolding as-set-def
    by blast
  from in-g1 have stamps:  $\text{stamp } g1 \ n = \text{stamp } g2 \ n$ 
    using assms unfolding as-set-def
    by blast
  from kinds stamps show ?case using ParameterExpr ParameterExprE
    by (metis ParameterNode ParameterNodeE)
next
  case (LeafExpr nid s)
  then have in-g1:  $n \in \text{ids } g1$ 
    using no-encoding by force
  then have kinds:  $\text{kind } g1 \ n = \text{kind } g2 \ n$ 
    using assms unfolding as-set-def
    by blast
  from in-g1 have stamps:  $\text{stamp } g1 \ n = \text{stamp } g2 \ n$ 
    using assms unfolding as-set-def

```



```

    by blast
  from kinds stamps show ?case using LeafExpr LeafExprE LeafNode
  by (smt (z3) IRExpr.distinct(29) IRExpr.simps(16) IRExpr.simps(28) rep.simps)
next
case (ConstantVar x)
then have in-g1:  $n \in \text{ids } g1$ 
  using no-encoding by force
then have kinds:  $\text{kind } g1 \ n = \text{kind } g2 \ n$ 
  using assms unfolding as-set-def
  by blast
from in-g1 have stamps:  $\text{stamp } g1 \ n = \text{stamp } g2 \ n$ 
  using assms unfolding as-set-def
  by blast
from kinds stamps show ?case using ConstantVar
  using rep.simps by blast
next
case (VariableExpr x s)
then have in-g1:  $n \in \text{ids } g1$ 
  using no-encoding by force
then have kinds:  $\text{kind } g1 \ n = \text{kind } g2 \ n$ 
  using assms unfolding as-set-def
  by blast
from in-g1 have stamps:  $\text{stamp } g1 \ n = \text{stamp } g2 \ n$ 
  using assms unfolding as-set-def
  by blast
from kinds stamps show ?case using VariableExpr
  using rep.simps by blast
qed

```

```

lemma subset-refines:
  assumes  $\text{as-set } g1 \subseteq \text{as-set } g2$ 
  shows graph-refinement  $g1 \ g2$ 
proof -
  have  $\text{ids } g1 \subseteq \text{ids } g2$  using assms unfolding as-set-def
  by blast
show ?thesis unfolding graph-refinement-def
  apply (rule allI) apply (rule impI) apply (rule allI) apply (rule impI)
proof -
  fix  $n \ e1$ 
  assume 1:  $n \in \text{ids } g1$ 
  assume 2:  $g1 \vdash n \simeq e1$ 

  show  $\exists e2. (g2 \vdash n \simeq e2) \wedge e1 \geq e2$ 
  using assms 1 2 using subset-implies-evals
  by (meson equal-refines)
qed
qed

```

```

lemma graph-construction:
   $e1 \geq e2$ 
   $\wedge$  as-set  $g1 \subseteq$  as-set  $g2 \wedge$  maximal-sharing  $g1$ 
   $\wedge$  ( $g2 \vdash n \simeq e2$ )  $\wedge$  maximal-sharing  $g2$ 
   $\implies$  ( $g2 \vdash n \sqsubseteq e1$ )  $\wedge$  graph-refinement  $g1$   $g2$ 
  using subset-refines
  by (meson encodeeval-def graph-represents-expression-def le-expr-def)

end

```

8 Control-flow Semantics

```

theory IRStepObj
  imports
    TreeToGraph
begin

```

8.1 Heap

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

```

type-synonym ('a, 'b) Heap = 'a  $\Rightarrow$  'b  $\Rightarrow$  Value
type-synonym Free = nat
type-synonym ('a, 'b) DynamicHeap = ('a, 'b) Heap  $\times$  Free

fun h-load-field :: 'a  $\Rightarrow$  'b  $\Rightarrow$  ('a, 'b) DynamicHeap  $\Rightarrow$  Value where
  h-load-field  $f$   $r$  ( $h, n$ ) =  $h$   $f$   $r$ 

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

fun h-new-inst :: ('a, 'b) DynamicHeap  $\Rightarrow$  ('a, 'b) DynamicHeap  $\times$  Value where
  h-new-inst ( $h, n$ ) = (( $h, n+1$ ), (ObjRef (Some  $n$ )))

type-synonym FieldRefHeap = (string, objref) DynamicHeap

definition new-heap :: ('a, 'b) DynamicHeap where
  new-heap = (( $\lambda f. \lambda p. \text{UndefVal}$ ), 0)

```

8.2 Intraprocedural Semantics

```

fun find-index :: 'a  $\Rightarrow$  'a list  $\Rightarrow$  nat where
  find-index - [] = 0 |

```

```

find-index v (x # xs) = (if (x=v) then 0 else find-index v xs + 1)

fun phi-list :: IRGraph ⇒ 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;

```

$$\begin{aligned}
& m' = \text{set-phis } \text{phis} \text{ vs } m \\
& \implies g, p \vdash (nid, m, h) \rightarrow (\text{merge}, m', h) \mid
\end{aligned}$$

NewInstanceNode:

$$\begin{aligned}
& \llbracket \text{kind } g \text{ nid} = (\text{NewInstanceNode } nid \text{ f obj } nid') \rrbracket; \\
& (h', \text{ref}) = h\text{-new-inst } h; \\
& m' = m(nid := \text{ref}) \\
& \implies g, p \vdash (nid, m, h) \rightarrow (nid', m', h') \mid
\end{aligned}$$

LoadFieldNode:

$$\begin{aligned}
& \llbracket \text{kind } g \text{ nid} = (\text{LoadFieldNode } nid \text{ f (Some obj) } nid') \rrbracket; \\
& g \vdash \text{obj} \simeq \text{objE}; \\
& [m, p] \vdash \text{objE} \mapsto \text{ObjRef } \text{ref}; \\
& h\text{-load-field } f \text{ ref } h = v; \\
& m' = m(nid := v) \\
& \implies g, p \vdash (nid, m, h) \rightarrow (nid', m', h) \mid
\end{aligned}$$

SignedDivNode:

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

SignedRemNode:

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

StaticLoadFieldNode:

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

StoreFieldNode:

$$\begin{aligned}
& \llbracket \text{kind } g \text{ nid} = (\text{StoreFieldNode } nid \text{ f newval - (Some obj) } nid') \rrbracket; \\
& g \vdash \text{newval} \simeq \text{newvalE}; \\
& g \vdash \text{obj} \simeq \text{objE}; \\
& [m, p] \vdash \text{newvalE} \mapsto \text{val};
\end{aligned}$$

$$\begin{aligned}
& [m, p] \vdash \text{obj}E \mapsto \text{ObjRef } \text{ref}; \\
& h' = h\text{-store-field } f \text{ ref val } h; \\
& m' = m(\text{nid} := \text{val}) \\
\implies & g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m', h') \mid
\end{aligned}$$

StaticStoreFieldNode:

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

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

8.3 Interprocedural Semantics

type-synonym *Signature* = *string*

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

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

(\vdash - \longrightarrow - 55)

for *P* **where**

Lift:

$$\begin{aligned}
& \llbracket g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m', h') \rrbracket \\
& \implies P \vdash ((g, \text{nid}, m, p) \# \text{stk}, h) \longrightarrow ((g, \text{nid}', m', p) \# \text{stk}, h') \mid
\end{aligned}$$

InvokeNodeStep:

$\llbracket \text{is-Invoke } (\text{kind } g \text{ nid}) \rrbracket$;

$$\begin{aligned}
& \text{callTarget} = \text{ir-callTarget } (\text{kind } g \text{ nid}); \\
& \text{kind } g \text{ callTarget} = (\text{MethodCallTargetNode } \text{targetMethod } \text{arguments}); \\
& \text{Some } \text{targetGraph} = P \text{ targetMethod}; \\
& m' = \text{new-map-state}; \\
& g \vdash \text{arguments} \simeq_L \text{args}E; \\
& [m, p] \vdash \text{args}E \mapsto_L p \\
& \implies P \vdash ((g, \text{nid}, m, p) \# \text{stk}, h) \longrightarrow ((\text{targetGraph}, 0, m', p') \# (g, \text{nid}, m, p) \# \text{stk}, h)
\end{aligned}$$

|

ReturnNode:

$$\begin{aligned}
& \llbracket \text{kind } g \text{ nid} = (\text{ReturnNode } (\text{Some } \text{expr}) -) \rrbracket; \\
& g \vdash \text{expr} \simeq e; \\
& [m, p] \vdash e \mapsto v;
\end{aligned}$$

$\text{cm}' = \text{cm}(\text{cnid} := v)$;

$$cnid' = (successors-of (kind\ cg\ cnid))!0 \\ \Rightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk, h) \longrightarrow ((cg,cnid',cm',cp)\#stk, h) \mid$$

ReturnNodeVoid:

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

$$cnid' = (successors-of (kind\ cg\ cnid))!0 \\ \Rightarrow P \vdash ((g,nid,m,p)\#(cg,cnid,cm,cp)\#stk, h) \longrightarrow ((cg,cnid',cm',cp)\#stk, h) \mid$$

UnwindNode:

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

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

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

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

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

8.4 Big-step Execution

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

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

inductive *exec* :: *Program*

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

($- \vdash - \mid - \longrightarrow^* - \mid -$)

for *P*

where

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

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

$$exec\ P\ (((g',nid',m',p')\#ys),h')\ l'\ next-state\ l'' \\ \Rightarrow exec\ P\ (((g,nid,m,p)\#xs),h)\ l\ next-state\ l''$$

$$\mid \\ \llbracket P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h') \rrbracket;$$

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

inductive *exec-debug* :: *Program*
 $\Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times FieldRefHeap$
 $\Rightarrow nat$
 $\Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times FieldRefHeap$
 $\Rightarrow bool$
 $(\vdash \longrightarrow * - * -)$
where
 $\llbracket n > 0; \quad p \vdash s \longrightarrow s'; \quad exec_debug\ p\ s'\ (n - 1)\ s'' \rrbracket$
 $\implies exec_debug\ p\ s\ n\ s'' \mid$
 $\llbracket n = 0 \rrbracket$
 $\implies exec_debug\ p\ s\ n\ s$
code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *exec-debug* .

8.4.1 Heap Testing

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

values $\{(prod.fst(prod.snd\ (prod.snd\ (hd\ (prod.fst\ res))))\ 0 \mid res. (\lambda x. Some\ eg2\ sq) \vdash [(eg2\ sq, 0, new\ map\ state, p3), (eg2\ sq, 0, new\ map\ state, p3)], new\ heap) \rightarrow *2* res\}$

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

definition $eg4\text{-}sq :: IRGraph$ **where**
 $eg4\text{-}sq = irgraph$ [
 (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 { $h\text{-load-field field-sq (Some 0) (prod.snd res) \mid res.$
 $(\lambda x. \text{Some } eg4\text{-}sq) \vdash [(eg4\text{-}sq, 0, \text{new-map-state}, p3), (eg4\text{-}sq, 0,$
 $\text{new-map-state}, p3)], \text{new-heap}) \rightarrow^* 4^* res\}$

end

9 Properties of Control-flow Semantics

theory *IRStepThms*

imports

IRStepObj

IRTreeEvalThms

begin

We prove that within the same graph, a configuration triple will always transition to the same subsequent configuration. Therefore, our step semantics is deterministic.

theorem *stepDet*:

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

proof (*induction rule: step.induct*)

case (*SequentialNode nid next m h*)

have *notif*: $\neg(is\text{-IfNode } (kind\ g\ nid))$

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

by (*metis is-IfNode-def*)

have *notend*: $\neg(is\text{-AbstractEndNode } (kind\ g\ nid))$

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

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

have *notnew*: $\neg(is\text{-NewInstanceNode } (kind\ g\ nid))$

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

by (*metis is-NewInstanceNode-def*)

have *notload*: $\neg(is\text{-LoadFieldNode } (kind\ g\ nid))$


```

    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 iphis 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:  $\neg$ (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:  $\neg$ (is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notend:  $\neg$ (is-AbstractEndNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notif:  $\neg$ (is-IfNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notref:  $\neg$ (is-RefNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notload:  $\neg$ (is-LoadFieldNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notstore:  $\neg$ (is-StoreFieldNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
have notdivrem:  $\neg$ (is-IntegerDivRemNode (kind g nid))
  using is-AbstractMergeNode.simps
  by (simp add: NewInstanceNode.hyps(1))
from notseq notend notif notref notload notstore notdivrem
show ?case using NewInstanceNode step.cases
  by (smt (z3) IRNode.disc(1028) IRNode.disc(2270) IRNode.discI(11) IRN-
ode.distinct(2311) IRNode.distinct(2313) IRNode.inject(31) Pair-inject)
next
case (LoadFieldNode nid f obj nxt m ref h v m')
then have notseq:  $\neg$ (is-sequential-node (kind g nid))
  using is-sequential-node.simps is-AbstractMergeNode.simps
  by (simp add: LoadFieldNode.hyps(1))
have notend:  $\neg$ (is-AbstractEndNode (kind g nid))
  using is-AbstractEndNode.simps
  by (simp add: LoadFieldNode.hyps(1))
have notdivrem:  $\neg$ (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 nrt 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 nrt 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 nrt 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 Static-

```

```

StoreFieldNode.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:  $\neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: SignedDivNode.hyps(1))
  have notend:  $\neg(\text{is-AbstractEndNode } (\text{kind } g \text{ nid}))$ 
    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:  $\neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: SignedRemNode.hyps(1))
  have notend:  $\neg(\text{is-AbstractEndNode } (\text{kind } g \text{ nid}))$ 
    using is-AbstractEndNode.simps
    by (simp add: SignedRemNode.hyps(1))
  from notseq notend
  show ?case using SignedRemNode step.cases repDet evalDet
    by (smt (z3) IRNode.distinct(1093) IRNode.distinct(1741) IRNode.distinct(2313)
IRNode.distinct(2601) IRNode.distinct(2627) IRNode.inject(41) Pair-inject)
qed

```

lemma *stepRefNode*:

```

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

```

lemma *IfNodeStepCases*:

```

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

```

lemma *IfNodeSeq*:

```

shows kind g nid = IfNode cond tb fb  $\longrightarrow \neg(\text{is-sequential-node } (\text{kind } g \text{ nid}))$ 
unfolding is-sequential-node.simps by simp

```

lemma *IfNodeCond*:

```

assumes kind g nid = IfNode cond tb fb
assumes g, p  $\vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m, h)$ 
shows  $\exists \text{ condE } v. ((g \vdash \text{cond} \simeq \text{condE}) \wedge ([m, p] \vdash \text{condE} \mapsto v))$ 

```

using *assms*(2,1) **by** (*induct* (*nid*,*m*,*h*) (*nid'*,*m*,*h*) *rule: step.induct; auto*)

lemma *step-in-ids*:

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

shows $nid \in ids\ g$

using *assms* **apply** (*induct* (*nid*, *m*, *h*) (*nid'*, *m'*, *h'*) *rule: step.induct*)

using *is-sequential-node.simps*(45) *not-in-g*

apply *simp*

apply (*metis is-sequential-node.simps*(53))

using *ids-some*

using *IRNode.distinct*(1113) **apply** *presburger*

using *EndNodes*(1) *is-AbstractEndNode.simps is-EndNode.simps*(45) *ids-some*

apply (*metis IRNode.disc*(1218) *is-EndNode.simps*(52))

by *simp+*

end

10 Proof Infrastructure

10.1 Bisimulation

theory *Bisimulation*

imports

Stuttering

begin

inductive *weak-bisimilar* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$

($- . - \sim -$) **for** *nid* **where**

$\llbracket \forall P'. (g\ m\ p\ h \vdash nid \rightsquigarrow P') \longrightarrow (\exists Q'. (g'\ m\ p\ h \vdash nid \rightsquigarrow Q') \wedge P' = Q');$

$\forall Q'. (g'\ m\ p\ h \vdash nid \rightsquigarrow Q') \longrightarrow (\exists P'. (g\ m\ p\ h \vdash nid \rightsquigarrow P') \wedge P' = Q') \rrbracket$

$\implies nid . g \sim g'$

A strong bisimulation between no-op transitions

inductive *strong-noop-bisimilar* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$

($- | - \sim -$) **for** *nid* **where**

$\llbracket \forall P'. (g, p \vdash (nid, m, h) \rightarrow P') \longrightarrow (\exists Q'. (g', p \vdash (nid, m, h) \rightarrow Q') \wedge P' = Q');$

$\forall Q'. (g', p \vdash (nid, m, h) \rightarrow Q') \longrightarrow (\exists P'. (g, p \vdash (nid, m, h) \rightarrow P') \wedge P' = Q') \rrbracket$

$\implies nid | g \sim g'$

lemma *lockstep-strong-bisimulation*:

assumes $g' = \text{replace-node } nid\ node\ g$

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

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

shows $nid | g \sim g'$

```

using assms(2) assms(3) stepDet strong-noop-bisimilar.simps by metis

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

end

```

10.2 Formedness Properties

```

theory Form
imports
  Semantics.TreeToGraph
begin

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

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

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

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

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

lemmas wf-folds =
  wf-graph.simps
  wf-start-def

```

wf-closed-def
wf-phis-def
wf-ends-def

fun *wf-stamps* :: *IRGraph* \Rightarrow *bool* **where**
wf-stamps *g* = (\forall *n* \in *ids* *g* .
 $(\forall$ *v m p e* . (*g* \vdash *n* \simeq *e*) \wedge (*[m, p]* \vdash *e* \mapsto *v*) \longrightarrow *valid-value* (*stamp-expr* *e*) *v*))

fun *wf-stamp* :: *IRGraph* \Rightarrow (*ID* \Rightarrow *Stamp*) \Rightarrow *bool* **where**
wf-stamp *g s* = (\forall *n* \in *ids* *g* .
 $(\forall$ *v m p e* . (*g* \vdash *n* \simeq *e*) \wedge (*[m, p]* \vdash *e* \mapsto *v*) \longrightarrow *valid-value* (*s* *n*) *v*))

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

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

fun *wf-logic-node-inputs* :: *IRGraph* \Rightarrow *ID* \Rightarrow *bool* **where**
wf-logic-node-inputs *g n* =
 $(\forall$ *inp* \in *set* (*inputs-of* (*kind* *g n*)) . (\forall *v m p* . (*[g, m, p]* \vdash *inp* \mapsto *v*) \longrightarrow *wf-bool* *v*))

fun *wf-values* :: *IRGraph* \Rightarrow *bool* **where**
wf-values *g* = (\forall *n* \in *ids* *g* .
 $(\forall$ *v m p* . (*[g, m, p]* \vdash *n* \mapsto *v*) \longrightarrow
 $(\textit{is-LogicNode}$ (*kind* *g n*) \longrightarrow
 $\textit{wf-bool}$ *v* \wedge *wf-logic-node-inputs* *g n*)))

end

10.3 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* \Rightarrow *IRGraph* \Rightarrow *IRGraph* \Rightarrow *bool* **where**
unchanged *ns g1 g2* = (\forall *n* . *n* \in *ns* \longrightarrow
 $(\textit{n} \in \textit{ids } g1 \wedge \textit{n} \in \textit{ids } g2 \wedge \textit{kind } g1 \textit{ n} = \textit{kind } g2 \textit{ n} \wedge \textit{stamp } g1 \textit{ n} = \textit{stamp } g2 \textit{ n})$)

fun *changeonly* :: *ID set* \Rightarrow *IRGraph* \Rightarrow *IRGraph* \Rightarrow *bool* **where**
changeonly ns g1 g2 = (\forall *n* . *n* \in *ids g1* \wedge *n* \notin *ns* \longrightarrow
(*n* \in *ids g1* \wedge *n* \in *ids g2* \wedge *kind g1 n* = *kind g2 n* \wedge *stamp g1 n* = *stamp g2 n*))

lemma *node-unchanged*:
assumes *unchanged ns g1 g2*
assumes *nid* \in *ns*
shows *kind g1 nid* = *kind g2 nid*
using *assms* **by** *auto*

lemma *other-node-unchanged*:
assumes *changeonly ns g1 g2*
assumes *nid* \in *ids g1*
assumes *nid* \notin *ns*
shows *kind g1 nid* = *kind g2 nid*
using *assms*
using *changeonly.simps* **by** *blast*

Some notation for input nodes used

inductive *eval-uses*:: *IRGraph* \Rightarrow *ID* \Rightarrow *ID* \Rightarrow *bool*
for *g* **where**

use0: *nid* \in *ids g*
 \implies *eval-uses g nid nid* |

use-inp: *nid'* \in *inputs g n*
 \implies *eval-uses g nid nid'* |

use-trans: \llbracket *eval-uses g nid nid'*;
eval-uses g nid' nid'' \rrbracket
 \implies *eval-uses g nid nid''*

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

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

lemma *not-in-g-inputs*:
assumes *nid* \notin *ids g*
shows *inputs g nid* = {}
proof –

have *k*: *kind g nid* = *NoNode* **using** *assms not-in-g* **by** *blast*
then show *?thesis* **by** (*simp add: k*)

qed

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

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

lemma *inp-in-g*:
 assumes $n \in \text{inputs } g \text{ nid}$
 shows $\text{nid} \in \text{ids } g$
proof –
 have $\text{inputs } g \text{ nid} \neq \{\}$
 using *assms*
 by (*metis empty-iff empty-set*)
 then have $\text{kind } g \text{ nid} \neq \text{NoNode}$
 using *not-in-g-inputs*
 using *ids-some* by *blast*
 then show ?thesis
 using *not-in-g*
 by *metis*

qed

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

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

using *unchanged.simps* by *blast*
qed

lemma *stamp-unchanged*:
 assumes $nid \in ids\ g1$
 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 $child \in inputs\ g1\ nid$
 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 $nid' \in ids\ g$
 shows *eval-uses* *g* *nid* *nid'* $\longleftrightarrow nid' \in us$ (is ?*P* \longleftrightarrow ?*Q*)
 using *assms* *eval-usages.simps*
 by (*simp* *add: ids.rep-eq*)

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

lemma *inputs-are-usages*:
 assumes $nid' \in inputs\ g\ nid$
 assumes $nid' \in ids\ g$
 shows $nid' \in eval-usages\ g\ nid$
 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 $nid' \in ids\ g$
 shows $nid' \in eval-usages\ g\ nid$
 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 $ls \subseteq ids\ g$
 shows $ls \subseteq us$
 using *inputs-are-usages* *eval-usages*
 using *assms*(1) *assms*(2) *assms*(3) by *blast*

```

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

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

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

lemma transitive-kind-same:
  assumes unchanged (eval-usages  $g1 \text{ } nid$ )  $g1 \text{ } g2$ 
  shows  $\forall \text{ } nid' \in (\text{eval-usages } g1 \text{ } nid) . \text{kind } g1 \text{ } nid' = \text{kind } g2 \text{ } nid'$ 
  using assms
  by (meson unchanged.elims(1))

theorem stay-same-encoding:
  assumes nc: unchanged (eval-usages  $g1 \text{ } nid$ )  $g1 \text{ } g2$ 
  assumes  $g1: g1 \vdash nid \simeq e$ 
  assumes wf: wf-graph  $g1$ 
  shows  $g2 \vdash nid \simeq e$ 
proof –
  have dom:  $nid \in \text{ids } g1$ 
  using  $g1$  encode-in-ids by simp
  show ?thesis
using  $g1 \text{ } nc \text{ } wf \text{ } dom$  proof (induction e rule: rep.induct)
  case (ConstantNode  $n \text{ } c$ )
  then have  $\text{kind } g2 \text{ } n = \text{ConstantNode } c$ 
  using dom nc kind-unchanged
  by metis
  then show ?case using rep.ConstantNode
  by presburger
next
  case (ParameterNode  $n \text{ } i \text{ } s$ )
  then have  $\text{kind } g2 \text{ } n = \text{ParameterNode } i$ 
  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.premis(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.premis(1) AbsNode.premis(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.premis(1) NotNode.premis(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.premis(1) NegateNode.premis(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 Logic-
NegationNode.hyps(1) LogicNegationNode.hyps(2) LogicNegationNode.premis(1) (kind
g2 n = LogicNegationNode x) child-unchanged encode-in-ids inputs.simps list.set-intros(1)
local.wf rep.LogicNegationNode)
next
  case (AddNode n x y xe ye)
  then have kind g2 n = AddNode x y
  using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
  using inputs-of-LogicNegationNode inputs-of-are-usages
  by (metis AddNode.hyps(1) AddNode.hyps(2) AddNode.hyps(3) IRNodes.inputs-of-AddNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case
  by (metis AddNode.IH(1) AddNode.IH(2) AddNode.hyps(1) AddNode.hyps(2)
AddNode.hyps(3) AddNode.premis(1) IRNodes.inputs-of-AddNode (kind g2 n = AddNode
x y) child-unchanged encode-in-ids in-set-member inputs.simps local.wf member-rec(1)
rep.AddNode)
next
  case (MulNode n x y xe ye)
  then have kind g2 n = MulNode x y
  using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
  using inputs-of-LogicNegationNode inputs-of-are-usages
  by (metis MulNode.hyps(1) MulNode.hyps(2) MulNode.hyps(3) IRNodes.inputs-of-MulNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using MulNode inputs-of-MulNode
  by (metis (kind g2 n = MulNode x y) child-unchanged inputs.simps list.set-intros(1)
rep.MulNode set-subset-Cons subset-iff unchanged.elims(2))
next
  case (SubNode n x y xe ye)
  then have kind g2 n = SubNode x y
  using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
  using inputs-of-LogicNegationNode inputs-of-are-usages
  by (metis SubNode.hyps(1) SubNode.hyps(2) SubNode.hyps(3) IRNodes.inputs-of-SubNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)

```

```

then show ?case using SubNode inputs-of-SubNode
  by (metis (kind g2 n = SubNode x y) child-member child-unchanged encode-in-ids
ids-some member-rec(1) rep.SubNode)
next
  case (AndNode n x y xe ye)
  then have kind g2 n = AndNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-LogicNegationNode inputs-of-are-usages
  by (metis AndNode.hyps(1) AndNode.hyps(2) AndNode.hyps(3) IRNodes.inputs-of-AndNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using AndNode inputs-of-AndNode
    by (metis (kind g2 n = AndNode x y) child-unchanged inputs.simps list.set-intros(1)
rep.AndNode set-subset-Cons subset-iff unchanged.elims(2))
next
  case (OrNode n x y xe ye)
  then have kind g2 n = OrNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-OrNode inputs-of-are-usages
  by (metis OrNode.hyps(1) OrNode.hyps(2) OrNode.hyps(3) IRNodes.inputs-of-OrNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using OrNode inputs-of-OrNode
    by (metis (kind g2 n = OrNode x y) child-member child-unchanged encode-in-ids
ids-some member-rec(1) rep.OrNode)
next
  case (XorNode n x y xe ye)
  then have kind g2 n = XorNode x y
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-XorNode inputs-of-are-usages
  by (metis XorNode.hyps(1) XorNode.hyps(2) XorNode.hyps(3) IRNodes.inputs-of-XorNode
encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using XorNode inputs-of-XorNode
    by (metis (kind g2 n = XorNode x y) child-member child-unchanged encode-in-ids
ids-some member-rec(1) rep.XorNode)
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) IntegerEqual-
sNode.hyps(3) IRNodes.inputs-of-IntegerEqualsNode encode-in-ids in-mono inputs.simps
inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using IntegerEqualsNode inputs-of-IntegerEqualsNode
    by (metis ⟨kind g2 n = IntegerEqualsNode x y⟩ child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.IntegerEqualsNode)
next
case (IntegerLessThanNode n x y xe ye)
then have kind g2 n = IntegerLessThanNode x y
  using kind-unchanged by metis
then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
  using inputs-of-IntegerLessThanNode inputs-of-are-usages
  by (metis IntegerLessThanNode.hyps(1) IntegerLessThanNode.hyps(2) Inte-
gerLessThanNode.hyps(3) IRNodes.inputs-of-IntegerLessThanNode encode-in-ids in-mono
inputs.simps inputs-are-usages list.set-intros(1) set-subset-Cons)
  then show ?case using IntegerLessThanNode inputs-of-IntegerLessThanNode
    by (metis ⟨kind g2 n = IntegerLessThanNode x y⟩ child-member child-unchanged
encode-in-ids ids-some member-rec(1) rep.IntegerLessThanNode)
next
case (NarrowNode n ib rb x xe)
then have kind g2 n = NarrowNode ib rb x
  using kind-unchanged by metis
then have x ∈ eval-usages g1 n
  using inputs-of-NarrowNode inputs-of-are-usages
  by (metis NarrowNode.hyps(1) NarrowNode.hyps(2) IRNodes.inputs-of-NarrowNode
encode-in-ids inputs.simps inputs-are-usages list.set-intros(1))
  then show ?case using NarrowNode inputs-of-NarrowNode
    by (metis ⟨kind g2 n = NarrowNode ib rb x⟩ child-unchanged inputs.elims
list.set-intros(1) rep.NarrowNode unchanged.simps)
next
case (SignExtendNode n ib rb x xe)
then have kind g2 n = SignExtendNode ib rb x
  using kind-unchanged by metis
then have x ∈ eval-usages g1 n
  using inputs-of-SignExtendNode inputs-of-are-usages
  by (metis SignExtendNode.hyps(1) SignExtendNode.hyps(2) encode-in-ids in-
puts.simps inputs-are-usages list.set-intros(1))
  then show ?case using SignExtendNode inputs-of-SignExtendNode
    by (metis ⟨kind g2 n = SignExtendNode ib rb x⟩ child-member-in child-unchanged
in-set-member list.set-intros(1) rep.SignExtendNode unchanged.elims(2))
next
case (ZeroExtendNode n ib rb x xe)
then have kind g2 n = ZeroExtendNode ib rb x
  using kind-unchanged by metis

```

```

then have  $x \in \text{eval-usages } g1 \ n$ 
  using  $\text{inputs-of-ZeroExtendNode inputs-of-are-usages}$ 
  by ( $\text{metis ZeroExtendNode.hyps(1) ZeroExtendNode.hyps(2) IRNodes.inputs-of-ZeroExtendNode}$ 
 $\text{encode-in-ids inputs.simps inputs-are-usages list.set-intros(1)}$ )
  then show  $?case$  using  $\text{ZeroExtendNode inputs-of-ZeroExtendNode}$ 
  by ( $\text{metis } \langle \text{kind } g2 \ n = \text{ZeroExtendNode ib rb } x \rangle \text{ child-member-in child-unchanged}$ 
 $\text{member-rec(1) rep.ZeroExtendNode unchanged.simps}$ )
next
  case ( $\text{LeafNode } n \ s$ )
  then show  $?case$ 
    by ( $\text{metis kind-unchanged rep.LeafNode stamp-unchanged}$ )
qed
qed

```

```

theorem stay-same:
  assumes  $nc: \text{unchanged } (\text{eval-usages } g1 \ nid) \ g1 \ g2$ 
  assumes  $g1: [g1, m, p] \vdash nid \mapsto v1$ 
  assumes  $wf: wf\text{-graph } g1$ 
  shows  $[g2, m, p] \vdash nid \mapsto v1$ 
proof –
  have  $nid: nid \in \text{ids } g1$ 
    using  $g1 \text{ eval-in-ids}$  by  $\text{simp}$ 
  then have  $nid \in \text{eval-usages } g1 \ nid$ 
    using  $\text{eval-usages-self}$  by  $\text{blast}$ 
  then have  $\text{kind-same: kind } g1 \ nid = \text{kind } g2 \ nid$ 
    using  $nc \text{ node-unchanged}$  by  $\text{blast}$ 
  obtain  $e$  where  $e: (g1 \vdash nid \simeq e) \wedge ([m, p] \vdash e \mapsto v1)$ 
    using  $\text{encodeeval-def } g1$ 
    by  $\text{auto}$ 
  then have  $\text{val: } [m, p] \vdash e \mapsto v1$ 
    using  $g1 \text{ encodeeval-def}$ 
    by  $\text{simp}$ 
  then show  $?thesis$  using  $e \ nid \ nc$ 
    unfolding  $\text{encodeeval-def}$ 
proof ( $\text{induct } e \ v1 \text{ arbitrary: nid rule: evaltree.induct}$ )
  case ( $\text{ConstantExpr } c$ )
  then show  $?case$ 
    by ( $\text{metis ConstantNode ConstantNodeE kind-unchanged}$ )
next
  case ( $\text{ParameterExpr } i \ s$ )
  have  $g2 \vdash nid \simeq \text{ParameterExpr } i \ s$ 
    using  $\text{stay-same-encoding ParameterExpr}$ 
    by ( $\text{meson local.wf}$ )
  then show  $?case$  using  $\text{evaltree.ParameterExpr}$ 
    by ( $\text{meson ParameterExpr.hyps}$ )
next
  case ( $\text{ConditionalExpr } ce \ \text{cond branch } te \ fe \ v$ )

```



```

    then have  $g2 \vdash nid \simeq \text{ConditionalExpr } ce \ te \ fe$ 
    using  $\text{ConditionalExpr.prem}(1) \ \text{ConditionalExpr.prem}(3) \ \text{local.wf stay-same-encoding}$ 
  by presburger
    then show ?case
      by (metis  $\text{ConditionalExpr.prem}(1)$ )
  next
    case ( $\text{UnaryExpr } xe \ v \ op$ )
    then show ?case
      using  $\text{local.wf stay-same-encoding}$  by blast
  next
    case ( $\text{BinaryExpr } xe \ x \ ye \ y \ op$ )
    then show ?case
      using  $\text{local.wf stay-same-encoding}$  by blast
  next
    case ( $\text{LeafExpr } val \ nid \ s$ )
    then show ?case
      by (metis  $\text{local.wf stay-same-encoding}$ )
  qed
qed

```

```

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

```

```

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

```

```

lemma add-node-unchanged:
  assumes  $new \notin \text{ids } g$ 
  assumes  $nid \in \text{ids } g$ 
  assumes  $gup = \text{add-node } new \ k \ g$ 
  assumes  $\text{wf-graph } g$ 
  shows  $\text{unchanged } (\text{eval-usages } g \ nid) \ g \ gup$ 
proof -
  have  $new \notin (\text{eval-usages } g \ nid)$  using  $\text{assms}$ 
  using  $\text{eval-usages.simps}$  by blast
  then have  $\text{changeonly } \{new\} \ g \ gup$ 
  using  $\text{assms add-changed}$  by blast
  then show ?thesis using  $\text{assms add-node-def disjoint-change}$ 
  using  $\text{Diff-insert-absorb}$  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'))
   $\longleftrightarrow$  eval-uses g nid nid'
using use0 use-inp use-trans
by (meson eval-uses.simps)

lemma wf-use-ids:
  assumes wf-graph g
  assumes nid ∈ 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

```

10.4 Graph Rewriting

theory

Rewrites

imports

IRGraphFrames

Stuttering

begin

fun *replace-usages* :: *ID* \Rightarrow *ID* \Rightarrow *IRGraph* \Rightarrow *IRGraph* **where**
replace-usages *nid* *nid'* *g* = *replace-node* *nid* (*RefNode* *nid'*, *stamp* *g* *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* \in *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* \in *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* \Rightarrow *ID* **where**

nextNid *g* = (*Max* (*ids* *g*)) + 1

lemma *max-plus-one*:

fixes *c* :: *ID* *set*

shows $\llbracket \text{finite } c; c \neq \{\} \rrbracket \implies (\text{Max } c) + 1 \notin c$

by (*meson* *Max-gr-iff* *less-add-one* *less-irrefl*)

lemma *ids-finite*:

finite (*ids* *g*)

by *simp*

lemma *nextNidNotIn*:

ids *g* $\neq \{\} \longrightarrow \text{nextNid } g \notin \text{ids } g$

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 (IRTreeEval.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 (constantAsStamp (IntVal32 1)) (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 (IRTreeEval.bool-to-val True)⟩ encodeeval-def truedef
    by metis
  from if' c' show ?thesis using IfNode
    by (metis (no-types, hide-lams) IRTreeEval.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  $\bigwedge c. \text{ConstantNode } c \neq \text{NoNode}$  by simp
ultimately have kind g' (nextNid g) = ConstantNode (IRTreeEval.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 (constantAsStamp (IntVal32 0)) (IntVal32 0)
  unfolding constantAsStamp.simps valid-value.simps
  using nat-numeral by blast
then have  $[g', m, p] \vdash \text{nextNid } g \mapsto \text{IntVal32 } 0$ 
  by (metis ConstantExpr ConstantNode  $\langle \text{kind } g' (\text{nextNid } g) = \text{ConstantNode } (\text{IRTreeEval.bool-to-val False}) \rangle$  encodeeval-def falsedef)
from if' c' show ?thesis using IfNode
  by (metis (no-types, hide-lams) IRTreeEval.val-to-bool.simps(1)  $\langle [g', m, p] \vdash \text{nextNid } g \mapsto \text{IntVal32 } 0 \rangle$  encodeeval-def)
qed

```

lemma *diff-forall*:

```

assumes  $\forall n \in \text{ids } g - \{nid\}. \text{cond } n$ 
shows  $\forall n. n \in \text{ids } g \wedge n \notin \{nid\} \longrightarrow \text{cond } n$ 
by (meson Diff-iff assms)

```

lemma *replace-node-changeonly*:

```

assumes  $g' = \text{replace-node } nid \text{ node } g$ 
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  $g' = \text{add-node } nid \text{ node } g$ 
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  $\neg(\text{is-IfNode } (\text{kind } g \text{ nid}))$ 
shows  $g = \text{constantCondition } b \text{ nid } (\text{kind } g \text{ nid}) g$ 
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  $const = \text{val-to-bool } v$ 
assumes  $g' = \text{constantCondition } const \text{ ifcond } (\text{kind } g \text{ ifcond}) \ g$ 
shows  $\exists \text{nid}' . (g \ m \ p \ h \vdash \text{ifcond} \rightsquigarrow \text{nid}') \longleftrightarrow (g' \ m \ p \ h \vdash \text{ifcond} \rightsquigarrow \text{nid}')$ 
proof (cases const)
  case True
    have  $\text{ifstep}: g, p \vdash (\text{ifcond}, m, h) \rightarrow (t, m, h)$ 
      by (meson IfNode True assms(1) assms(2) assms(3) encodeeval-def)
    have  $\text{ifstep}': g', p \vdash (\text{ifcond}, m, h) \rightarrow (t, m, h)$ 
      using constantConditionTrue
      using True assms(1) assms(4) by presburger
    from  $\text{ifstep} \ \text{ifstep}'$  show ?thesis
      using StutterStep by blast
  next
    case False
      have  $\text{ifstep}: g, p \vdash (\text{ifcond}, m, h) \rightarrow (f, m, h)$ 
        by (meson IfNode False assms(1) assms(2) assms(3) encodeeval-def)
      have  $\text{ifstep}': g', p \vdash (\text{ifcond}, m, h) \rightarrow (f, m, h)$ 
        using constantConditionFalse
        using False assms(1) assms(4) by presburger
      from  $\text{ifstep} \ \text{ifstep}'$  show ?thesis
        using StutterStep by blast
  qed
end

```

10.5 Stuttering

```

theory Stuttering
imports
  Semantics.IRStepThms
begin

```

```

inductive stutter:: IRGraph  $\Rightarrow$  MapState  $\Rightarrow$  Params  $\Rightarrow$  FieldRefHeap  $\Rightarrow$  ID  $\Rightarrow$ 
ID  $\Rightarrow$  bool (- - -  $\vdash$  -  $\rightsquigarrow$  - 55)
  for  $g \ m \ p \ h$  where

```

```

  StutterStep:
   $\llbracket g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m, h) \rrbracket$ 
   $\implies g \ m \ p \ h \vdash \text{nid} \rightsquigarrow \text{nid}' \mid$ 

```

```

  Transitive:
   $\llbracket g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}'', m, h);$ 
   $g \ m \ p \ h \vdash \text{nid}'' \rightsquigarrow \text{nid}' \rrbracket$ 
   $\implies g \ m \ p \ h \vdash \text{nid} \rightsquigarrow \text{nid}'$ 

```

```

lemma stuttering-successor:
  assumes  $(g, p \vdash (\text{nid}, m, h) \rightarrow (\text{nid}', m, h))$ 
  shows  $\{P'. (g \ m \ p \ h \vdash \text{nid} \rightsquigarrow P')\} = \{\text{nid}'\} \cup \{\text{nid}'' . (g \ m \ p \ h \vdash \text{nid}' \rightsquigarrow \text{nid}'')\}$ 
proof –

```

```

have nextin:  $nid' \in \{P'. (g \ m \ p \ h \vdash \text{nid} \rightsquigarrow P')\}$ 
using assms StutterStep by blast
have nextsubset:  $\{nid''. (g \ m \ p \ h \vdash \text{nid}' \rightsquigarrow \text{nid}'')\} \subseteq \{P'. (g \ m \ p \ h \vdash \text{nid} \rightsquigarrow P')\}$ 
by (metis Collect-mono assms stutter.Transitive)
have  $\forall n \in \{P'. (g \ m \ p \ h \vdash \text{nid} \rightsquigarrow P')\} . n = \text{nid}' \vee n \in \{nid''. (g \ m \ p \ h \vdash \text{nid}' \rightsquigarrow \text{nid}'')\}$ 
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

```

11 Canonicalization Phase

```

theory CanonicalizationTree
imports
  Semantics.TreeToGraph
  Semantics.IRTreeEval
begin

```

```

fun is-idempotent-binary :: IRBinaryOp  $\Rightarrow$  bool where
is-idempotent-binary BinAnd = True |
is-idempotent-binary BinOr  = True |
is-idempotent-binary -      = False

```

```

fun is-idempotent-unary :: IRUnaryOp  $\Rightarrow$  bool where
is-idempotent-unary UnaryAbs = True |
is-idempotent-unary -      = False

```

```

fun is-self-inverse :: IRUnaryOp  $\Rightarrow$  bool where
is-self-inverse UnaryNeg = True |
is-self-inverse UnaryNot = True |
is-self-inverse UnaryLogicNegation = True |
is-self-inverse -      = False

```

```

fun is-neutral :: IRBinaryOp  $\Rightarrow$  Value  $\Rightarrow$  bool where

is-neutral BinAdd (IntVal32 x) = (x = 0) |
is-neutral BinAdd (IntVal64 x) = (x = 0) |

is-neutral BinSub (IntVal32 x) = (x = 0) |

```


is-neutral BinSub (IntVal64 x) = (x = 0) |

is-neutral BinMul (IntVal32 x) = (x = 1) |

is-neutral BinMul (IntVal64 x) = (x = 1) |

is-neutral BinAnd (IntVal32 x) = (x = 1) |

is-neutral BinAnd (IntVal64 x) = (x = 1) |

is-neutral BinOr (IntVal32 x) = (x = 0) |

is-neutral BinOr (IntVal64 x) = (x = 0) |

is-neutral BinXor (IntVal32 x) = (x = 0) |

is-neutral BinXor (IntVal64 x) = (x = 0) |

is-neutral - - = False

fun *is-annihilator* :: *IRBinaryOp* \Rightarrow *Value* \Rightarrow *bool* **where**

is-annihilator BinMul (IntVal32 x) = (x = 0) |

is-annihilator BinMul (IntVal64 x) = (x = 0) |

is-annihilator BinAnd (IntVal32 x) = (x = 0) |

is-annihilator BinAnd (IntVal64 x) = (x = 0) |

is-annihilator BinOr (IntVal32 x) = (x = 1) |

is-annihilator BinOr (IntVal64 x) = (x = 1) |

is-annihilator - - = False

fun *int-to-value* :: *Value* \Rightarrow *int* \Rightarrow *Value* **where**

int-to-value (IntVal32 -) y = (IntVal32 (word-of-int y)) |

int-to-value (IntVal64 -) y = (IntVal64 (word-of-int y)) |

int-to-value - - = UndefVal

inductive *CanonicalizeBinaryOp* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**

binary-const-fold:

$\llbracket x = (\text{ConstantExpr } \text{val1});$

$y = (\text{ConstantExpr } \text{val2});$

$\text{val} = \text{bin-eval } \text{op } \text{val1 } \text{val2};$

$\text{val} \neq \text{UndefVal} \rrbracket$

$\implies \text{CanonicalizeBinaryOp } (\text{BinaryExpr } \text{op } x \ y) \ (\text{ConstantExpr } \text{val}) \mid$

binary-fold-yneutral:

$\llbracket y = (\text{ConstantExpr } c);$

$\text{is-neutral } \text{op } c;$

$\text{stamp } x = \text{stamp-expr } x;$

$\text{stamp } y = \text{stamp-expr } y;$

$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
 $is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy$
 $\implies CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ x \mid$

$binary\text{-}fold\text{-}yzero32:$
 $\llbracket y = ConstantExpr\ c;$
 $is\text{-}annihilator\ op\ c;$
 $stampx = stamp\text{-}expr\ x;$
 $stampy = stamp\text{-}expr\ y;$
 $stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
 $stp\text{-}bits\ stampx = 32;$
 $is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy$
 $\implies CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ (ConstantExpr\ c) \mid$

$binary\text{-}fold\text{-}yzero64:$
 $\llbracket y = ConstantExpr\ c;$
 $is\text{-}annihilator\ op\ c;$
 $stampx = stamp\text{-}expr\ x;$
 $stampy = stamp\text{-}expr\ y;$
 $stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
 $stp\text{-}bits\ stampx = 64;$
 $is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy$
 $\implies CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ (ConstantExpr\ c) \mid$

$binary\text{-}idempotent:$
 $\llbracket is\text{-}idempotent\text{-}binary\ op \rrbracket$
 $\implies CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ x)\ x$

inductive $CanonicalizeUnaryOp :: IRExpr \Rightarrow IRExpr \Rightarrow bool$ **where**
 $unary\text{-}const\text{-}fold:$
 $\llbracket val' = unary\text{-}eval\ op\ val;$
 $val' \neq UndefinedVal \rrbracket$
 $\implies CanonicalizeUnaryOp\ (UnaryExpr\ op\ (ConstantExpr\ val))\ (ConstantExpr\ val')$

inductive $CanonicalizeMul :: IRExpr \Rightarrow IRExpr \Rightarrow bool$ **where**

$mul\text{-}negate32:$
 $\llbracket y = ConstantExpr\ (IntVal32\ (-1));$
 $stamp\text{-}expr\ x = IntegerStamp\ 32\ lo\ hi \rrbracket$
 $\implies CanonicalizeMul\ (BinaryExpr\ BinMul\ x\ y)\ (UnaryExpr\ UnaryNeg\ x) \mid$
 $mul\text{-}negate64:$
 $\llbracket y = ConstantExpr\ (IntVal64\ (-1));$
 $stamp\text{-}expr\ x = IntegerStamp\ 64\ lo\ hi \rrbracket$
 $\implies CanonicalizeMul\ (BinaryExpr\ BinMul\ x\ y)\ (UnaryExpr\ UnaryNeg\ x)$

inductive $CanonicalizeAdd :: IRExpr \Rightarrow IRExpr \Rightarrow bool$ **where**
 $add\text{-}xsub:$

$\llbracket x = (\text{BinaryExpr BinSub } a \ y);$
 $\text{stamp}_a = \text{stamp-expr } a;$
 $\text{stamp}_y = \text{stamp-expr } y;$
 $\text{is-IntegerStamp } \text{stamp}_a \wedge \text{is-IntegerStamp } \text{stamp}_y;$
 $\text{stp-bits } \text{stamp}_a = \text{stp-bits } \text{stamp}_y \rrbracket$
 $\implies \text{CanonicalizeAdd } (\text{BinaryExpr BinAdd } x \ y) \ a \mid$

add-ysub:

$\llbracket y = (\text{BinaryExpr BinSub } a \ x);$
 $\text{stamp}_a = \text{stamp-expr } a;$
 $\text{stamp}_x = \text{stamp-expr } x;$
 $\text{is-IntegerStamp } \text{stamp}_a \wedge \text{is-IntegerStamp } \text{stamp}_x;$
 $\text{stp-bits } \text{stamp}_a = \text{stp-bits } \text{stamp}_x \rrbracket$
 $\implies \text{CanonicalizeAdd } (\text{BinaryExpr BinAdd } x \ y) \ a \mid$

add-xnegate:

$\llbracket nx = (\text{UnaryExpr UnaryNeg } x);$
 $\text{stamp}_x = \text{stamp-expr } x;$
 $\text{stamp}_y = \text{stamp-expr } y;$
 $\text{is-IntegerStamp } \text{stamp}_x \wedge \text{is-IntegerStamp } \text{stamp}_y;$
 $\text{stp-bits } \text{stamp}_x = \text{stp-bits } \text{stamp}_y \rrbracket$
 $\implies \text{CanonicalizeAdd } (\text{BinaryExpr BinAdd } nx \ y) \ (\text{BinaryExpr BinSub } y \ x) \mid$

add-ynegate:

$\llbracket ny = (\text{UnaryExpr UnaryNeg } y);$
 $\text{stamp}_x = \text{stamp-expr } x;$
 $\text{stamp}_y = \text{stamp-expr } y;$
 $\text{is-IntegerStamp } \text{stamp}_x \wedge \text{is-IntegerStamp } \text{stamp}_y;$
 $\text{stp-bits } \text{stamp}_x = \text{stp-bits } \text{stamp}_y \rrbracket$
 $\implies \text{CanonicalizeAdd } (\text{BinaryExpr BinAdd } x \ ny) \ (\text{BinaryExpr BinSub } x \ y)$

inductive *CanonicalizeSub* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**

sub-same32:

$\llbracket \text{stamp}_x = \text{stamp-expr } x;$
 $\text{stamp}_x = \text{IntegerStamp } 32 \text{ lo } hi \rrbracket$
 $\implies \text{CanonicalizeSub } (\text{BinaryExpr BinSub } x \ x) \ (\text{ConstantExpr } (\text{IntVal32 } 0)) \mid$
sub-same64:

$\llbracket \text{stamp}_x = \text{stamp-expr } x;$
 $\text{stamp}_x = \text{IntegerStamp } 64 \text{ lo } hi \rrbracket$
 $\implies \text{CanonicalizeSub } (\text{BinaryExpr BinSub } x \ x) \ (\text{ConstantExpr } (\text{IntVal64 } 0)) \mid$

sub-left-add1:

$$\begin{aligned} & \llbracket x = (\text{BinaryExpr BinAdd } a \ b); \\ & \quad \text{stamp}_a = \text{stamp-expr } a; \\ & \quad \text{stamp}_b = \text{stamp-expr } b; \\ & \quad \text{is-IntegerStamp stamp}_a \wedge \text{is-IntegerStamp stamp}_b; \\ & \quad \text{stp-bits stamp}_a = \text{stp-bits stamp}_b \rrbracket \\ & \implies \text{CanonicalizeSub } (\text{BinaryExpr BinSub } x \ b) \ a \mid \end{aligned}$$

sub-left-add2:

$$\begin{aligned} & \llbracket x = (\text{BinaryExpr BinAdd } a \ b); \\ & \quad \text{stamp}_a = \text{stamp-expr } a; \\ & \quad \text{stamp}_b = \text{stamp-expr } b; \\ & \quad \text{is-IntegerStamp stamp}_a \wedge \text{is-IntegerStamp stamp}_b; \\ & \quad \text{stp-bits stamp}_a = \text{stp-bits stamp}_b \rrbracket \\ & \implies \text{CanonicalizeSub } (\text{BinaryExpr BinSub } x \ a) \ b \mid \end{aligned}$$

sub-left-sub:

$$\begin{aligned} & \llbracket x = (\text{BinaryExpr BinSub } a \ b); \\ & \quad \text{stamp}_a = \text{stamp-expr } a; \\ & \quad \text{stamp}_b = \text{stamp-expr } b; \\ & \quad \text{is-IntegerStamp stamp}_a \wedge \text{is-IntegerStamp stamp}_b; \\ & \quad \text{stp-bits stamp}_a = \text{stp-bits stamp}_b \rrbracket \\ & \implies \text{CanonicalizeSub } (\text{BinaryExpr BinSub } x \ a) \ (\text{UnaryExpr UnaryNeg } b) \mid \end{aligned}$$

sub-right-add1:

$$\begin{aligned} & \llbracket y = (\text{BinaryExpr BinAdd } a \ b); \\ & \quad \text{stamp}_a = \text{stamp-expr } a; \\ & \quad \text{stamp}_b = \text{stamp-expr } b; \\ & \quad \text{is-IntegerStamp stamp}_a \wedge \text{is-IntegerStamp stamp}_b; \\ & \quad \text{stp-bits stamp}_a = \text{stp-bits stamp}_b \rrbracket \\ & \implies \text{CanonicalizeSub } (\text{BinaryExpr BinSub } a \ y) \ (\text{UnaryExpr UnaryNeg } b) \mid \end{aligned}$$

sub-right-add2:

$$\begin{aligned} & \llbracket y = (\text{BinaryExpr BinAdd } a \ b); \\ & \quad \text{stamp}_a = \text{stamp-expr } a; \\ & \quad \text{stamp}_b = \text{stamp-expr } b; \\ & \quad \text{is-IntegerStamp stamp}_a \wedge \text{is-IntegerStamp stamp}_b; \\ & \quad \text{stp-bits stamp}_a = \text{stp-bits stamp}_b \rrbracket \\ & \implies \text{CanonicalizeSub } (\text{BinaryExpr BinSub } b \ y) \ (\text{UnaryExpr UnaryNeg } a) \mid \end{aligned}$$

sub-right-sub:

$$\llbracket y = (\text{BinaryExpr BinSub } a \ b);$$

$stamp_a = stamp_expr\ a;$
 $stamp_b = stamp_expr\ b;$
 $is_IntegerStamp\ stamp_a \wedge is_IntegerStamp\ stamp_b;$
 $stp_bits\ stamp_a = stp_bits\ stamp_b$
 $\implies CanonicalizeSub\ (BinaryExpr\ BinSub\ a\ y)\ b\ |$

$sub_xzero32:$
 $\llbracket stamp_x = stamp_expr\ x;$
 $\quad stamp_x = IntegerStamp\ 32\ lo\ hi \rrbracket$
 $\implies CanonicalizeSub\ (BinaryExpr\ BinSub\ (ConstantExpr\ (IntVal32\ 0))\ x)$
 $(UnaryExpr\ UnaryNeg\ x)\ |$
 $sub_xzero64:$
 $\llbracket stamp_x = stamp_expr\ x;$
 $\quad stamp_x = IntegerStamp\ 64\ lo\ hi \rrbracket$
 $\implies CanonicalizeSub\ (BinaryExpr\ BinSub\ (ConstantExpr\ (IntVal64\ 0))\ x)$
 $(UnaryExpr\ UnaryNeg\ x)\ |$

$sub_y_negate:$

$\llbracket nb = (UnaryExpr\ UnaryNeg\ b);$
 $\quad stamp_a = stamp_expr\ a;$
 $\quad stamp_b = stamp_expr\ b;$
 $\quad is_IntegerStamp\ stamp_a \wedge is_IntegerStamp\ stamp_b;$
 $\quad stp_bits\ stamp_a = stp_bits\ stamp_b$
 $\implies CanonicalizeSub\ (BinaryExpr\ BinSub\ a\ nb)\ (BinaryExpr\ BinAdd\ a\ b)$

inductive $CanonicalizeNegate :: IRExpr \Rightarrow IRExpr \Rightarrow bool$ **where**
 $negate_negate:$

$\llbracket nx = (UnaryExpr\ UnaryNeg\ x);$
 $\quad is_IntegerStamp\ (stamp_expr\ x) \rrbracket$
 $\implies CanonicalizeNegate\ (UnaryExpr\ UnaryNeg\ nx)\ x\ |$

$negate_sub:$

$\llbracket e = (BinaryExpr\ BinSub\ x\ y);$
 $\quad stamp_x = stamp_expr\ x;$
 $\quad stamp_y = stamp_expr\ y;$
 $\quad is_IntegerStamp\ stamp_x \wedge is_IntegerStamp\ stamp_y;$
 $\quad stp_bits\ stamp_x = stp_bits\ stamp_y$
 $\implies CanonicalizeNegate\ (UnaryExpr\ UnaryNeg\ e)\ (BinaryExpr\ BinSub\ y\ x)$

inductive $CanonicalizeAbs :: IRExpr \Rightarrow IRExpr \Rightarrow bool$ **where**

abs-abs:

$\llbracket ax = (\text{UnaryExpr } \text{UnaryAbs } x);$
 $\text{is-IntegerStamp } (\text{stamp-expr } x) \rrbracket$
 $\implies \text{CanonicalizeAbs } (\text{UnaryExpr } \text{UnaryAbs } ax) \text{ } ax \mid$

abs-neg:

$\llbracket nx = (\text{UnaryExpr } \text{UnaryNeg } x);$
 $\text{is-IntegerStamp } (\text{stamp-expr } x) \rrbracket$
 $\implies \text{CanonicalizeAbs } (\text{UnaryExpr } \text{UnaryAbs } nx) (\text{UnaryExpr } \text{UnaryAbs } x)$

inductive *CanonicalizeNot* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
not-not:

$\llbracket nx = (\text{UnaryExpr } \text{UnaryNot } x);$
 $\text{is-IntegerStamp } (\text{stamp-expr } x) \rrbracket$
 $\implies \text{CanonicalizeNot } (\text{UnaryExpr } \text{UnaryNot } nx) \text{ } x$

inductive *CanonicalizeAnd* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
and-same:

$\llbracket \text{is-IntegerStamp } (\text{stamp-expr } x) \rrbracket$
 $\implies \text{CanonicalizeAnd } (\text{BinaryExpr } \text{BinAnd } x \text{ } x) \text{ } x \mid$

and-demorgans:

$\llbracket nx = (\text{UnaryExpr } \text{UnaryNot } x);$
 $ny = (\text{UnaryExpr } \text{UnaryNot } y);$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{is-IntegerStamp } \text{stampx} \wedge \text{is-IntegerStamp } \text{stampy};$
 $\text{stp-bits } \text{stampx} = \text{stp-bits } \text{stampy} \rrbracket$
 $\implies \text{CanonicalizeAnd } (\text{BinaryExpr } \text{BinAnd } nx \text{ } ny) (\text{UnaryExpr } \text{UnaryNot } (\text{BinaryExpr } \text{BinOr } x \text{ } y))$

inductive *CanonicalizeOr* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
or-same:

$\llbracket \text{is-IntegerStamp } (\text{stamp-expr } x) \rrbracket$
 $\implies \text{CanonicalizeOr } (\text{BinaryExpr } \text{BinOr } x \text{ } x) \text{ } x \mid$

or-demorgans:

$\llbracket nx = (\text{UnaryExpr } \text{UnaryNot } x);$

$ny = (\text{UnaryExpr } \text{UnaryNot } y);$
 $stampx = \text{stamp-expr } x;$
 $stampy = \text{stamp-expr } y;$
 $is\text{-IntegerStamp } stampx \wedge is\text{-IntegerStamp } stampy;$
 $stp\text{-bits } stampx = stp\text{-bits } stampy$
 $\implies \text{CanonicalizeOr } (\text{BinaryExpr } \text{BinOr } nx \ ny) (\text{UnaryExpr } \text{UnaryNot } (\text{BinaryExpr } \text{BinAnd } x \ y))$

inductive *CanonicalizeIntegerEquals* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
int-equals-same:

$\llbracket x = y \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr } \text{BinIntegerEquals } x \ y) (\text{ConstantExpr } (\text{IntVal32 } 1)) \mid$

int-equals-distinct:
 $\llbracket \text{alwaysDistinct } (\text{stamp-expr } x) (\text{stamp-expr } y) \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr } \text{BinIntegerEquals } x \ y) (\text{ConstantExpr } (\text{IntVal32 } 0)) \mid$

int-equals-add-first-both-same:
 $\llbracket \text{left} = (\text{BinaryExpr } \text{BinAdd } x \ y);$
 $\text{right} = (\text{BinaryExpr } \text{BinAdd } x \ z) \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr } \text{BinIntegerEquals } \text{left } \text{right}) (\text{BinaryExpr } \text{BinIntegerEquals } y \ z) \mid$

int-equals-add-first-second-same:
 $\llbracket \text{left} = (\text{BinaryExpr } \text{BinAdd } x \ y);$
 $\text{right} = (\text{BinaryExpr } \text{BinAdd } z \ x) \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr } \text{BinIntegerEquals } \text{left } \text{right}) (\text{BinaryExpr } \text{BinIntegerEquals } y \ z) \mid$

int-equals-add-second-first-same:
 $\llbracket \text{left} = (\text{BinaryExpr } \text{BinAdd } y \ x);$
 $\text{right} = (\text{BinaryExpr } \text{BinAdd } x \ z) \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr } \text{BinIntegerEquals } \text{left } \text{right}) (\text{BinaryExpr } \text{BinIntegerEquals } y \ z) \mid$

int-equals-add-second-both--same:

$\llbracket \text{left} = (\text{BinaryExpr } \text{BinAdd } y \ x);$
 $\text{right} = (\text{BinaryExpr } \text{BinAdd } z \ x) \rrbracket$

$\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr BinIntegerEquals left right}) (\text{BinaryExpr BinIntegerEquals } y \ z) \mid$

int-equals-sub-first-both-same:

$\llbracket \text{left} = (\text{BinaryExpr BinSub } x \ y);$
 $\text{right} = (\text{BinaryExpr BinSub } x \ z) \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr BinIntegerEquals left right}) (\text{BinaryExpr BinIntegerEquals } y \ z) \mid$

int-equals-sub-second-both-same:

$\llbracket \text{left} = (\text{BinaryExpr BinSub } y \ x);$
 $\text{right} = (\text{BinaryExpr BinSub } z \ x) \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr BinIntegerEquals left right}) (\text{BinaryExpr BinIntegerEquals } y \ z) \mid$

int-equals-left-contains-right1:

$\llbracket \text{left} = (\text{BinaryExpr BinAdd } x \ y);$
 $\text{zero} = (\text{ConstantExpr (IntVal32 0)}) \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr BinIntegerEquals left } x) (\text{BinaryExpr BinIntegerEquals } y \ \text{zero}) \mid$

int-equals-left-contains-right2:

$\llbracket \text{left} = (\text{BinaryExpr BinAdd } x \ y);$
 $\text{zero} = (\text{ConstantExpr (IntVal32 0)}) \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr BinIntegerEquals left } y) (\text{BinaryExpr BinIntegerEquals } x \ \text{zero}) \mid$

int-equals-right-contains-left1:

$\llbracket \text{right} = (\text{BinaryExpr BinAdd } x \ y);$
 $\text{zero} = (\text{ConstantExpr (IntVal32 0)}) \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr BinIntegerEquals } x \ \text{right}) (\text{BinaryExpr BinIntegerEquals } y \ \text{zero}) \mid$

int-equals-right-contains-left2:

$\llbracket \text{right} = (\text{BinaryExpr BinAdd } x \ y);$
 $\text{zero} = (\text{ConstantExpr (IntVal32 0)}) \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr BinIntegerEquals } y \ \text{right}) (\text{BinaryExpr BinIntegerEquals } x \ \text{zero}) \mid$

int-equals-left-contains-right3:

$\llbracket \text{left} = (\text{BinaryExpr BinSub } x \ y);$
 $\text{zero} = (\text{ConstantExpr } (\text{IntVal32 } 0)) \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr BinIntegerEquals left } x) (\text{BinaryExpr BinIntegerEquals } y \ \text{zero}) \mid$

int-equals-right-contains-left3:

$\llbracket \text{right} = (\text{BinaryExpr BinSub } x \ y);$
 $\text{zero} = (\text{ConstantExpr } (\text{IntVal32 } 0)) \rrbracket$
 $\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr BinIntegerEquals } x \ \text{right}) (\text{BinaryExpr BinIntegerEquals } y \ \text{zero})$

inductive *CanonicalizeConditional* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
eq-branches:

$\llbracket t = f \rrbracket$
 $\implies \text{CanonicalizeConditional } (\text{ConditionalExpr } c \ t \ f) \ t \mid$

cond-eq:

$\llbracket c = (\text{BinaryExpr BinIntegerEquals } x \ y);$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy};$
 $\text{stp-bits stampx} = \text{stp-bits stampy} \rrbracket$
 $\implies \text{CanonicalizeConditional } (\text{ConditionalExpr } c \ x \ y) \ y \mid$

condition-bounds-x:

$\llbracket c = (\text{BinaryExpr BinIntegerLessThan } x \ y);$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{stpi-upper stampx} \leq \text{stpi-lower stampy};$
 $\text{stp-bits stampx} = \text{stp-bits stampy};$
 $\text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy} \rrbracket$
 $\implies \text{CanonicalizeConditional } (\text{ConditionalExpr } c \ x \ y) \ x \mid$

condition-bounds-y:

$\llbracket c = (\text{BinaryExpr BinIntegerLessThan } x \ y);$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{stpi-upper stampx} \leq \text{stpi-lower stampy};$
 $\text{stp-bits stampx} = \text{stp-bits stampy};$
 $\text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy} \rrbracket$
 $\implies \text{CanonicalizeConditional } (\text{ConditionalExpr } c \ y \ x) \ y \mid$

negate-condition:

```

[[nc = (UnaryExpr UnaryLogicNegation c);
 stampc = stamp-expr c;
 stampc = IntegerStamp 32 lo hi;
 stampx = stamp-expr x;
 stampy = stamp-expr y;
 stp-bits stampx = stp-bits stampy;
 is-IntegerStamp stampx ∧ is-IntegerStamp stampy]]
⇒ CanonicalizeConditional (ConditionalExpr nc x y) (ConditionalExpr c y x)
|

```

const-true:

```

[[c = ConstantExpr val;
 val-to-bool val]]
⇒ CanonicalizeConditional (ConditionalExpr c t f) t |

```

const-false:

```

[[c = ConstantExpr val;
 ¬(val-to-bool val)]]
⇒ CanonicalizeConditional (ConditionalExpr c t f) f

```

inductive CanonicalizationStep :: IRExpr ⇒ IRExpr ⇒ bool **where**

```

BinaryNode:
[[CanonicalizeBinaryOp expr expr']]
⇒ CanonicalizationStep expr expr' |

```

```

UnaryNode:
[[CanonicalizeUnaryOp expr expr']]
⇒ CanonicalizationStep expr expr' |

```

```

NegateNode:
[[CanonicalizeNegate expr expr']]
⇒ CanonicalizationStep expr expr' |

```

```

NotNode:
[[CanonicalizeNegate expr expr']]
⇒ CanonicalizationStep expr expr' |

```

```

AddNode:
[[CanonicalizeAdd expr expr']]
  ⇒ CanonicalizationStep expr expr' |

MulNode:
[[CanonicalizeMul expr expr']]
  ⇒ CanonicalizationStep expr expr' |

SubNode:
[[CanonicalizeSub expr expr']]
  ⇒ CanonicalizationStep expr expr' |

AndNode:
[[CanonicalizeSub expr expr']]
  ⇒ CanonicalizationStep expr expr' |

OrNode:
[[CanonicalizeSub expr expr']]
  ⇒ CanonicalizationStep expr expr' |

IntegerEqualsNode:
[[CanonicalizeIntegerEquals expr expr']]
  ⇒ CanonicalizationStep expr expr' |

ConditionalNode:
[[CanonicalizeConditional expr expr']]
  ⇒ CanonicalizationStep expr expr'

code-pred (modes: i ⇒ o ⇒ bool) CanonicalizeBinaryOp .
code-pred (modes: i ⇒ o ⇒ bool) CanonicalizeUnaryOp .
code-pred (modes: i ⇒ o ⇒ bool) CanonicalizeNegate .
code-pred (modes: i ⇒ o ⇒ bool) CanonicalizeNot .
code-pred (modes: i ⇒ o ⇒ bool) CanonicalizeAdd .
code-pred (modes: i ⇒ o ⇒ bool) CanonicalizeSub .
code-pred (modes: i ⇒ o ⇒ bool) CanonicalizeMul .
code-pred (modes: i ⇒ o ⇒ bool) CanonicalizeAnd .
code-pred (modes: i ⇒ o ⇒ bool) CanonicalizeIntegerEquals .
code-pred (modes: i ⇒ o ⇒ bool) CanonicalizeConditional .

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

end

```

12 Canonicalization Phase

```

theory CanonicalizationTreeProofs
  imports
    CanonicalizationTree
    Semantics.TreeToGraph

```

Semantics.IRTreeEvalThms
begin

lemma *neutral-rewrite-helper:*

shows *valid-value* (*IntegerStamp* 32 lo hi) $x \implies \text{intval-mul } x \text{ (IntVal32 (1))} = x$
and *valid-value* (*IntegerStamp* 64 lo hi) $x \implies \text{intval-mul } x \text{ (IntVal64 (1))} = x$

and *valid-value* (*IntegerStamp* 32 lo hi) $x \implies \text{intval-add } x \text{ (IntVal32 (0))} = x$
and *valid-value* (*IntegerStamp* 64 lo hi) $x \implies \text{intval-add } x \text{ (IntVal64 (0))} = x$

and *valid-value* (*IntegerStamp* 32 lo hi) $x \implies \text{intval-sub } x \text{ (IntVal32 (0))} = x$
and *valid-value* (*IntegerStamp* 64 lo hi) $x \implies \text{intval-sub } x \text{ (IntVal64 (0))} = x$

and *valid-value* (*IntegerStamp* 32 lo hi) $x \implies \text{intval-xor } x \text{ (IntVal32 (0))} = x$
and *valid-value* (*IntegerStamp* 64 lo hi) $x \implies \text{intval-xor } x \text{ (IntVal64 (0))} = x$

and *valid-value* (*IntegerStamp* 32 lo hi) $x \implies \text{intval-or } x \text{ (IntVal32 (0))} = x$
and *valid-value* (*IntegerStamp* 64 lo hi) $x \implies \text{intval-or } x \text{ (IntVal64 (0))} = x$
using *valid32or64-both* **by** *fastforce+*

lemma *annihilator-rewrite-helper:*

shows *valid-value* (*IntegerStamp* 32 lo hi) $x \implies \text{intval-mul } x \text{ (IntVal32 0)} = \text{IntVal32 0}$
and *valid-value* (*IntegerStamp* 64 lo hi) $x \implies \text{intval-mul } x \text{ (IntVal64 0)} = \text{IntVal64 0}$

and *valid-value* (*IntegerStamp* 32 lo hi) $x \implies \text{intval-and } x \text{ (IntVal32 0)} = \text{IntVal32 0}$
and *valid-value* (*IntegerStamp* 64 lo hi) $x \implies \text{intval-and } x \text{ (IntVal64 0)} = \text{IntVal64 0}$

and *valid-value* (*IntegerStamp* 32 lo hi) $x \implies \text{intval-or } x \text{ (IntVal32 (-1))} = \text{IntVal32 (-1)}$
and *valid-value* (*IntegerStamp* 64 lo hi) $x \implies \text{intval-or } x \text{ (IntVal64 (-1))} = \text{IntVal64 (-1)}$
using *valid32or64-both*
apply *auto*
apply (*metis intval-mul.simps(1) mult-zero-right valid32*)
by *fastforce+*

lemma *idempotent-rewrite-helper:*

shows *valid-value* (*IntegerStamp* 32 lo hi) $x \implies \text{intval-and } x x = x$
and *valid-value* (*IntegerStamp* 64 lo hi) $x \implies \text{intval-and } x x = x$

and *valid-value* (*IntegerStamp* 32 lo hi) $x \implies \text{intval-or } x x = x$
and *valid-value* (*IntegerStamp* 64 lo hi) $x \implies \text{intval-or } x x = x$
using *valid32or64-both*
apply *auto*
by *fastforce+*

```

value size (v::32 word)

lemma signed-int-bottom32: -(((2::int) ^ 31)) ≤ sint (v::int32)
proof -
  have size v = 32 apply (cases v; auto) sorry
  then show ?thesis
    using sint-range-size sorry
qed

lemma signed-int-top32: (2 ^ 31) - 1 ≥ sint (v::int32)
proof -
  have size v = 32 sorry
  then show ?thesis
    using sint-range-size sorry
qed

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

value size (v::64 word)

lemma signed-int-bottom64: -(((2::int) ^ 63)) ≤ sint (v::int64)
proof -
  have size v = 64 apply (cases v; auto) sorry
  then show ?thesis
    using sint-range-size sorry
qed

lemma signed-int-top64: (2 ^ 63) - 1 ≥ sint (v::int64)
proof -
  have size v = 64 sorry
  then show ?thesis
    using sint-range-size sorry
qed

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

```

```

lemma upper-bounds-equiv64:  $(2::\text{int}) \wedge 63 = (2::\text{int}) \wedge 64 \text{ div } 2$ 
  by simp

lemma bit-bounds-min64:  $((\text{fst } (\text{bit-bounds } 64))) \leq (\text{sint } (v::\text{int64}))$ 
  unfolding bit-bounds.simps fst-def using signed-int-bottom64 lower-bounds-equiv64
  by auto

lemma bit-bounds-max64:  $((\text{snd } (\text{bit-bounds } 64))) \geq (\text{sint } (v::\text{int64}))$ 
  unfolding bit-bounds.simps fst-def using signed-int-top64 upper-bounds-equiv64
  by auto

lemma unrestricted-32bit-always-valid:
  valid-value (unrestricted-stamp (IntegerStamp 32 lo hi)) (IntVal32 v)
  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 (unrestricted-stamp (IntegerStamp 64 lo hi)) (IntVal64 v)
  using valid-value.simps(2) bit-bounds-min64 bit-bounds-max64
  using unrestricted-stamp.simps(2) by presburger

lemma unary-undef:  $\text{val} = \text{UndefVal} \implies \text{unary-eval op val} = \text{UndefVal}$ 
  by (cases op; auto)

lemma unary-obj:  $\text{val} = \text{ObjRef } x \implies \text{unary-eval op val} = \text{UndefVal}$ 
  by (cases op; auto)

lemma unary-eval-implies-valud-value:
  assumes  $[m,p] \vdash \text{expr} \mapsto \text{val}$ 
  assumes  $\text{result} = \text{unary-eval op val}$ 
  assumes  $\text{result} \neq \text{UndefVal}$ 
  assumes valid-value (stamp-expr expr) val
  shows valid-value (stamp-expr (UnaryExpr op expr)) result
proof –
  have is-IntVal:  $\exists x y. \text{result} = \text{IntVal32 } x \vee \text{result} = \text{IntVal64 } y$ 
    using assms(2,3) apply (cases op; auto; cases val; auto)
    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:  $\text{stamp-expr expr} = (\text{IntegerStamp } b \text{ lo hi})$ 
    using is-IntegerStamp-def by auto
  then have  $\text{stamp-expr } (\text{UnaryExpr op expr}) = \text{unrestricted-stamp } (\text{IntegerStamp } b \text{ lo hi})$ 

```

```

    using stamp-expr.simps(1) stamp-unary.simps(1) by presburger
  from stamp-expr-def have bit32:  $b = 32 \implies \exists x. \text{result} = \text{IntVal32 } x$ 
    using assms(2,3,4) by (cases op; auto; cases val; auto)
  from stamp-expr-def have bit64:  $b = 64 \implies \exists x. \text{result} = \text{IntVal64 } x$ 
    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:  $v1 = \text{UndefVal} \vee v2 = \text{UndefVal} \implies \text{bin-eval op } v1 \ v2 = \text{UndefVal}$ 
  by (cases op; auto)

lemma binary-obj:  $v1 = \text{ObjRef } x \vee v2 = \text{ObjRef } y \implies \text{bin-eval op } v1 \ v2 = \text{UndefVal}$ 
  by (cases op; auto)

lemma binary-eval-bits-equal:
  assumes  $\text{result} = \text{bin-eval op } val1 \ val2$ 
  assumes  $\text{result} \neq \text{UndefVal}$ 
  assumes  $\text{valid-value } (\text{IntegerStamp } b1 \ lo1 \ hi1) \ val1$ 
  assumes  $\text{valid-value } (\text{IntegerStamp } b2 \ lo2 \ hi2) \ val2$ 
  shows  $b1 = b2$ 
  using assms
  by (cases op; cases val1; cases val2; auto)

lemma binary-eval-values:
  assumes  $\exists x \ y. \text{result} = \text{IntVal32 } x \vee \text{result} = \text{IntVal64 } y$ 
  assumes  $\text{result} = \text{bin-eval op } val1 \ val2$ 
  shows  $\exists x32 \ x64 \ y32 \ y64. \text{val1} = \text{IntVal32 } x32 \wedge \text{val2} = \text{IntVal32 } y32 \vee \text{val1} = \text{IntVal64 } x64 \wedge \text{val2} = \text{IntVal64 } y64$ 
  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-valud-value:
  assumes  $[m,p] \vdash \text{expr1} \mapsto \text{val1}$ 
  assumes  $[m,p] \vdash \text{expr2} \mapsto \text{val2}$ 
  assumes  $\text{result} = \text{bin-eval op } val1 \ val2$ 
  assumes  $\text{result} \neq \text{UndefVal}$ 
  assumes  $\text{valid-value } (\text{stamp-expr expr1}) \ val1$ 
  assumes  $\text{valid-value } (\text{stamp-expr expr2}) \ val2$ 
  shows  $\text{valid-value } (\text{stamp-expr } (\text{BinaryExpr op expr1 expr2})) \ \text{result}$ 
proof -
  have is-IntVal:  $\exists x \ y. \text{result} = \text{IntVal32 } x \vee \text{result} = \text{IntVal64 } y$ 

```

```

    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  $\exists x32\ x64\ y32\ y64. (val1 = \text{IntVal32 } x32 \wedge val2 = \text{IntVal32 } y32) \vee (val1$ 
=  $\text{IntVal64 } x64 \wedge val2 = \text{IntVal64 } y64)$ 
    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  $\in$  fixed-32 of True  $\Rightarrow$  unrestricted-stamp (IntegerStamp 32 lo1 hi1)|
False  $\Rightarrow$  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  $\Longrightarrow \exists x. \text{result} = \text{IntVal32 } x$ 
    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  $\wedge op \notin \text{fixed-32} \Longrightarrow \exists x\ y. \text{result} =$ 
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  $\in$  fixed-32  $\Longrightarrow \exists x\ y. \text{result} = \text{IntVal32 } x$ 
    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  $\in$  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  $\langle \exists x32\ x64\ y32\ y64. \text{val1} = \text{IntVal32 } x32 \wedge \text{val2} = \text{IntVal32 } y32 \vee \text{val1}$ 
 $= \text{IntVal64 } x64 \wedge \text{val2} = \text{IntVal64 } y64 \rangle$  assms(5) apply auto[1]
using fixed by force
qed

lemma stamp-meet-is-valid:
assumes valid-value stamp1 val  $\vee$  valid-value stamp2 val
assumes meet stamp1 stamp2  $\neq$  IllegalStamp
shows valid-value (meet stamp1 stamp2) val
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  $\langle b = b2 \rangle$  stamp2-def by blast
have min: sint x  $\geq$  min lo lo2
using val-def
using IntegerStamp assms(1)
using stamp2-def by force
have max: sint x  $\leq$  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
    by (metis meet.simps(13) meet.simps(14) meet.simps(65) meet.simps(67) unre-
stricted-stamp.cases valid-value.simps(10) valid-value.simps(11) valid-value.simps(16)
valid-value.simps(9))
next
  case (MethodCountersPointerStamp x41 x42)
  then show ?thesis using assms
    by (metis meet.simps(20) meet.simps(21) meet.simps(24) meet.simps(67) unre-
stricted-stamp.cases valid-value.simps(10) valid-value.simps(11) valid-value.simps(16)
valid-value.simps(9))
next
  case (MethodPointersStamp x51 x52)
  then show ?thesis using assms
    by (smt (z3) is-stamp-empty.elims(1) meet.simps(27) meet.simps(28) meet.simps(65)
meet.simps(67) valid-value.simps(10) valid-value.simps(11) valid-value.simps(16)
valid-value.simps(9))
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-valud-value:
  assumes  $[m, p] \vdash \text{cond} \mapsto \text{condv}$ 
  assumes  $\text{expr} = (\text{if } \text{IRTreeEval.val-to-bool } \text{condv} \text{ then } \text{expr1} \text{ else } \text{expr2})$ 
  assumes  $[m, p] \vdash \text{expr} \mapsto \text{val}$ 
  assumes  $\text{val} \neq \text{UndefVal}$ 
  assumes  $\text{valid-value } (\text{stamp-expr } \text{cond}) \text{ condv}$ 
  assumes  $\text{valid-value } (\text{stamp-expr } \text{expr}) \text{ val}$ 
  shows  $\text{valid-value } (\text{stamp-expr } (\text{ConditionalExpr } \text{cond } \text{expr1 } \text{expr2})) \text{ val}$ 
proof –
  have  $\text{meet } (\text{stamp-expr } \text{expr1}) (\text{stamp-expr } \text{expr2}) \neq \text{IllegalStamp}$ 
  using assms apply (cases stamp-expr expr; auto)
  using valid-VoidStamp apply blast sorry
  then show ?thesis using stamp-meet-is-valid using stamp-expr.simps(6)
  using assms(2) assms(6) by presburger
qed

```

```

lemma stamp-implies-valid-value:
  assumes  $[m, p] \vdash \text{expr} \mapsto \text{val}$ 
  shows  $\text{valid-value } (\text{stamp-expr } \text{expr}) \text{ val}$ 
  using assms proof (induction expr val)
case (UnaryExpr expr val result op)
  then show ?case using unary-eval-implies-valud-value by simp
next
  case (BinaryExpr expr1 val1 expr2 val2 result op)
  then show ?case using binary-eval-implies-valud-value by simp
next
  case (ConditionalExpr cond condv expr expr1 expr2 val)
  then show ?case using conditional-eval-implies-valud-value by simp
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 CanonicalizeBinaryProof:
  assumes CanonicalizeBinaryOp before after
  assumes  $[m, p] \vdash \text{before} \mapsto \text{res}$ 
  assumes  $[m, p] \vdash \text{after} \mapsto \text{res}'$ 
  shows  $\text{res} = \text{res}'$ 
  using assms
proof (induct rule: CanonicalizeBinaryOp.induct)
  case (binary-const-fold x val1 y val2 val op)
  then show ?case by auto
next

```

```

case (binary-fold-yneutral y c op stampx x stampy)
obtain xval where x-eval:  $[m, p] \vdash x \mapsto xval$ 
  using binary-fold-yneutral.prems(2) by auto
then have bin-eval op xval c = xval
  using neutral-rewrite-helper binary-fold-yneutral.hyps(2-3,6-) stamp-implies-valid-value
is-IntegerStamp-def
  sorry
then show ?case
  by (metis binary-fold-yneutral.hyps(1) binary-fold-yneutral.prems(1) binary-fold-yneutral.prems(2)
x-eval
    BinaryExprE ConstantExprE evalDet)
next
case (binary-fold-yzero32 y c op stampx x stampy)
obtain xval where x-eval:  $[m, p] \vdash x \mapsto xval$ 
  using binary-fold-yzero32.prems(1) by auto
then have bin-eval op xval c = c
  using annihilator-rewrite-helper binary-fold-yzero32.hyps stamp-implies-valid-value
is-IntegerStamp-def
  sorry
then show ?case
  by (metis BinaryExprE ConstantExprE binary-fold-yzero32.hyps(1) binary-fold-yzero32.prems(1)
binary-fold-yzero32.prems(2) evalDet x-eval)

next
case (binary-fold-yzero64 y c op stampx x stampy)
obtain xval where x-eval:  $[m, p] \vdash x \mapsto xval$ 
  using binary-fold-yzero64.prems(1) by auto
then have bin-eval op xval c = c
  using annihilator-rewrite-helper
  sorry
then show ?case
  by (metis BinaryExprE ConstantExprE binary-fold-yzero64.hyps(1)
binary-fold-yzero64.prems(1) binary-fold-yzero64.prems(2) evalDet x-eval)

next
case (binary-idempotent op x)
obtain xval where x-eval:  $[m, p] \vdash x \mapsto xval$ 
  using binary-idempotent.prems(1) by auto
then have bin-eval op xval xval = xval
  using idempotent-rewrite-helper binary-idempotent.hyps
  sorry
then show ?case
  by (metis BinaryExprE binary-idempotent.prems(1) binary-idempotent.prems(2)
evalDet x-eval)

qed

lemma CanonicalizeUnaryProof:
  assumes CanonicalizeUnaryOp before after

```

```

    assumes  $[m, p] \vdash \text{before} \mapsto \text{res}$ 
    assumes  $[m, p] \vdash \text{after} \mapsto \text{res}'$ 
    shows  $\text{res} = \text{res}'$ 
    using assms
  proof (induct rule: CanonicalizeUnaryOp.induct)
    case (unary-const-fold  $\text{val}' \text{ op val}$ )
    then show ?case by auto
  qed

lemma mul-rewrite-helper:
  shows  $\text{valid-value } (\text{IntegerStamp } 32 \text{ lo hi}) \ x \implies \text{intval-mul } x \ (\text{IntVal32 } (-1)) = \text{intval-negate } x$ 
  and  $\text{valid-value } (\text{IntegerStamp } 64 \text{ lo hi}) \ x \implies \text{intval-mul } x \ (\text{IntVal64 } (-1)) = \text{intval-negate } x$ 
  using valid32or64-both by fastforce+

lemma CanonicalizeMulProof:
  assumes CanonicalizeMul before after
  assumes  $[m, p] \vdash \text{before} \mapsto \text{res}$ 
  assumes  $[m, p] \vdash \text{after} \mapsto \text{res}'$ 
  shows  $\text{res} = \text{res}'$ 
  using assms
  proof (induct rule: CanonicalizeMul.induct)
    case (mul-negate32  $y \ x \text{ lo hi}$ )
    then show ?case
      using ConstantExprE BinaryExprE bin-eval.simps evalDet mul-rewrite-helper stamp-implies-valid-value
      by (auto; metis)
    next
    case (mul-negate64  $y \ x \text{ lo hi}$ )
    then show ?case
      using ConstantExprE BinaryExprE bin-eval.simps evalDet mul-rewrite-helper stamp-implies-valid-value
      by (auto; metis)
  qed

lemma add-rewrites-helper:
  assumes  $\text{valid-value } (\text{IntegerStamp } b \text{ lox hix}) \ x$ 
  and  $\text{valid-value } (\text{IntegerStamp } b \text{ loy hiy}) \ y$ 

  shows  $\text{intval-add } (\text{intval-sub } x \ y) \ y = x$ 
  and  $\text{intval-add } x \ (\text{intval-sub } y \ x) = y$ 
  and  $\text{intval-add } (\text{intval-negate } x) \ y = \text{intval-sub } y \ x$ 
  and  $\text{intval-add } x \ (\text{intval-negate } y) = \text{intval-sub } x \ y$ 
  using valid32or64-both assms by fastforce+

```

```

lemma CanonicalizeAddProof:
  assumes CanonicalizeAdd before after
  assumes  $[m, p] \vdash \text{before} \mapsto \text{res}$ 
  assumes  $[m, p] \vdash \text{after} \mapsto \text{res}'$ 
  shows  $\text{res} = \text{res}'$ 
  using assms
proof (induct rule: CanonicalizeAdd.induct)
  case (add-xsub x a y stampa stampy)
  then show ?case
    by (metis BinaryExprE Stamp.collapse(1) bin-eval.simps(1) bin-eval.simps(3)
      evalDet stamp-implies-valid-value intval-add-sym add-rewrites-helper(1))
  next
  case (add-ysub y a x stampa stampx)
  then show ?case
    by (metis is-IntegerStamp-def add-ysub.hyps add-ysub.premis evalDet BinaryExprE Stamp.sel(1)
      bin-eval.simps(1) bin-eval.simps(3) stamp-implies-valid-value intval-add-sym
      add-rewrites-helper(2))
  next
  case (add-xnegate nx x stampx stampy y)
  then show ?case
    by (smt (verit, del-insts) BinaryExprE Stamp.sel(1) UnaryExprE add-rewrites-helper(4)
      bin-eval.simps(1) bin-eval.simps(3) evalDet stamp-implies-valid-value int-
      val-add-sym is-IntegerStamp-def unary-eval.simps(2))
  next
  case (add-ynegate ny y stampx x stampy)
  then show ?case
    by (smt (verit) BinaryExprE Stamp.sel(1) UnaryExprE add-rewrites-helper(4)
      bin-eval.simps(1)
      bin-eval.simps(3) evalDet stamp-implies-valid-value is-IntegerStamp-def
      unary-eval.simps(2))
qed

```

```

lemma sub-rewrites-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  and valid-value (IntegerStamp b loy hiy) y

  shows intval-sub (intval-add x y) y = x
  and intval-sub (intval-add x y) x = y
  and intval-sub (intval-sub x y) x = intval-negate y

```

```

and   intval-sub x (intval-add x y) = intval-negate y
and   intval-sub y (intval-add x y) = intval-negate x
and   intval-sub x (intval-sub x y) = y
and   intval-sub x (intval-negate y) = intval-add x y
using valid32or64-both assms by fastforce+

```

lemma *sub-single-rewrites-helper*:

```

assumes valid-value (IntegerStamp b lox hix) x
shows   b = 32  $\implies$  intval-sub x x = IntVal32 0
and     b = 64  $\implies$  intval-sub x x = IntVal64 0
and     b = 32  $\implies$  intval-sub (IntVal32 0) x = intval-negate x
and     b = 64  $\implies$  intval-sub (IntVal64 0) x = intval-negate x
using valid32or64-both assms by fastforce+

```

lemma *CanonicalizeSubProof*:

```

assumes CanonicalizeSub before after
assumes [m, p]  $\vdash$  before  $\mapsto$  res
assumes [m, p]  $\vdash$  after  $\mapsto$  res'
shows   res = res'
using assms
proof (induct rule: CanonicalizeSub.induct)
  case (sub-same32 stampx x lo hi)
  show ?case
    using ConstantExprE BinaryExprE bin-eval.simps evalDet sub-same32.prem
    sub-single-rewrites-helper
    stamp-implies-valid-value sub-same32.hyps(1) sub-same32.hyps(2)
    by (auto; metis)
  next
  case (sub-same64 stampx x lo hi)
  show ?case
    using ConstantExprE BinaryExprE bin-eval.simps evalDet sub-same64.prem
    sub-single-rewrites-helper
    stamp-implies-valid-value sub-same64.hyps(1) sub-same64.hyps(2)
    by (auto; metis)
  next
  case (sub-left-add1 x a b stampa stampb)
  then show ?case
    by (metis BinaryExprE Stamp.collapse(1) bin-eval.simps(1) bin-eval.simps(3)
    evalDet
    stamp-implies-valid-value sub-rewrites-helper(1))
  next
  case (sub-left-add2 x a b stampa stampb)
  then show ?case
    by (metis BinaryExprE Stamp.collapse(1) bin-eval.simps(1) bin-eval.simps(3)
    evalDet
    stamp-implies-valid-value sub-rewrites-helper(2))
  next

```

```

    case (sub-left-sub x a b stampa stampb)
  then show ?case
    by (smt (verit) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(3)
evalDet
      stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(3) unary-eval.simps(2))
next
  case (sub-right-add1 y a b stampa stampb)
  then show ?case
    by (smt (verit) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(1)
bin-eval.simps(3) evalDet
      stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(4) unary-eval.simps(2))
next
  case (sub-right-add2 y a b stampa stampb)
  then show ?case
    by (smt (verit) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(1)
bin-eval.simps(3) evalDet
      stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(5) unary-eval.simps(2))
next
  case (sub-right-sub y a b stampa stampb)
  then show ?case
    by (metis BinaryExprE Stamp.sel(1) bin-eval.simps(3) evalDet
      stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(6))
next
  case (sub-xzero32 stampx x lo hi)
  then show ?case
    using ConstantExprE BinaryExprE bin-eval.simps evalDet sub-xzero32.premis
sub-single-rewrites-helper
      stamp-implies-valid-value sub-xzero32.hyps(1) sub-xzero32.hyps(2)
    by (auto; metis)
next
  case (sub-xzero64 stampx x lo hi)
  then show ?case
    using ConstantExprE BinaryExprE bin-eval.simps evalDet sub-xzero64.premis
sub-single-rewrites-helper
      stamp-implies-valid-value sub-xzero64.hyps(1) sub-xzero64.hyps(2)
    by (auto; metis)
next
  case (sub-y-negate nb b stampa a stampb)
  then show ?case
    by (smt (verit, best) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(1)
bin-eval.simps(3) evalDet
      stamp-implies-valid-value is-IntegerStamp-def sub-rewrites-helper(7) unary-eval.simps(2))
qed

```

lemma *negate-xsuby-helper*:

```

  assumes valid-value (IntegerStamp b lox hix) x
  and valid-value (IntegerStamp b loy hiy) y
  shows intval-negate (intval-sub x y) = intval-sub y x

```



```

using valid32or64-both assms by fastforce

lemma negate-negate-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  shows intval-negate (intval-negate x) = x
  using valid32or64 assms by fastforce

lemma CanonicalizeNegateProof:
  assumes CanonicalizeNegate before after
  assumes  $[m, p] \vdash \text{before} \mapsto \text{res}$ 
  assumes  $[m, p] \vdash \text{after} \mapsto \text{res}'$ 
  shows  $\text{res} = \text{res}'$ 
  using assms
proof (induct rule: CanonicalizeNegate.induct)
  case (negate-negate nx x)
  thus ?case
    by (metis UnaryExprE evalDet stamp-implies-valid-value is-IntegerStamp-def
        negate-negate-helper unary-eval.simps(2))
next
  case (negate-sub e x y stampx stampy)
  thus ?case
    by (smt (verit) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(3)
        evalDet stamp-implies-valid-value
        is-IntegerStamp-def negate-xsuby-helper unary-eval.simps(2))
qed

lemma word-helper:
  shows  $\bigwedge x :: 32 \text{ word. } \neg(\neg x < s \ 0 \wedge x < s \ 0)$ 
  and  $\bigwedge x :: 64 \text{ word. } \neg(\neg x < s \ 0 \wedge x < s \ 0)$ 
  and  $\bigwedge x :: 32 \text{ word. } \neg \neg x < s \ 0 \wedge \neg x < s \ 0 \implies 2 * x = 0$ 
  and  $\bigwedge x :: 64 \text{ word. } \neg \neg x < s \ 0 \wedge \neg x < s \ 0 \implies 2 * x = 0$ 
  apply (case-tac[!]) x
  apply auto+
  sorry

lemma abs-abs-is-abs:
  assumes valid-value (IntegerStamp b lox hix) x
  shows intval-abs (intval-abs x) = intval-abs x
  using word-helper
  by (metis assms intval-abs.simps(1) intval-abs.simps(2) valid32or64-both)

lemma abs-neg-is-neg:
  assumes valid-value (IntegerStamp b lox hix) x
  shows intval-abs (intval-negate x) = intval-abs x
  apply (case-tac[!]) x

```

```

using word-helper apply auto+
done

```

```

lemma not-rewrite-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  shows intval-not (intval-not x) = x
  using valid32or64 assms by fastforce+

```

```

lemma CanonicalizeNotProof:
  assumes CanonicalizeNot before after
  assumes  $[m, p] \vdash \text{before} \mapsto \text{res}$ 
  assumes  $[m, p] \vdash \text{after} \mapsto \text{res}'$ 
  shows  $\text{res} = \text{res}'$ 
  using assms
proof (induct rule: CanonicalizeNot.induct)
  case (not-not nx x)
  then show ?case
    by (metis UnaryExprE evalDet is-IntegerStamp-def not-rewrite-helper
        stamp-implies-valid-value unary-eval.simps(3))
qed

```

```

lemma demorgans-rewrites-helper:
  assumes valid-value (IntegerStamp b lox hix) x
  and      valid-value (IntegerStamp b loy hiy) y

  shows intval-and (intval-not x) (intval-not y) = intval-not (intval-or x y)
  and   intval-or (intval-not x) (intval-not y) = intval-not (intval-and x y)
  and    $x = y \implies \text{intval-and } x \ y = x$ 
  and    $x = y \implies \text{intval-or } x \ y = x$ 
  using valid32or64-both assms by fastforce+

```

```

lemma CanonicalizeAndProof:
  assumes CanonicalizeAnd before after
  assumes  $[m, p] \vdash \text{before} \mapsto \text{res}$ 
  assumes  $[m, p] \vdash \text{after} \mapsto \text{res}'$ 
  shows  $\text{res} = \text{res}'$ 
  using assms
proof (induct rule: CanonicalizeAnd.induct)
  case (and-same x)
  then show ?case
    by (metis BinaryExprE bin-eval.simps(4) demorgans-rewrites-helper(3) evalDet
        stamp-implies-valid-value is-IntegerStamp-def)
next
  case (and-demorgans nx x ny y stampx stampy)
  then show ?case
    by (smt (z3) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(4) bin-eval.simps(5))

```

demorgans-rewrites-helper(1) evalDet stamp-implies-valid-value is-IntegerStamp-def
 unary-eval.simps(3))
 qed

lemma CanonicalizeOrProof:
 assumes CanonicalizeOr before after
 assumes $[m, p] \vdash \text{before} \mapsto \text{res}$
 assumes $[m, p] \vdash \text{after} \mapsto \text{res}'$
 shows $\text{res} = \text{res}'$
 using assms
proof (induct rule: CanonicalizeOr.induct)
 case (or-same x)
 then show ?case
 by (metis BinaryExprE bin-eval.simps(5) demorgans-rewrites-helper(4) evalDet
 stamp-implies-valid-value is-IntegerStamp-def)
 next
 case (or-demorgans nx x ny y stampx stampy)
 then show ?case
 by (smt (z3) BinaryExprE Stamp.sel(1) UnaryExprE bin-eval.simps(4) bin-eval.simps(5)
 demorgans-rewrites-helper(2)
 evalDet stamp-implies-valid-value is-IntegerStamp-def unary-eval.simps(3))
 qed

lemma stamps-touch-but-not-less-than-implies-equal:
 $\llbracket \text{valid-value stampx } x;$
 $\text{valid-value stampy } y;$
 $\text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy};$
 $\text{stpi-upper stampx} = \text{stpi-lower stampy};$
 $\neg \text{val-to-bool}(\text{intval-less-than } x \ y) \rrbracket \implies x = y$
 using valid32or64-both intval-equals.simps(1-2) intval-less-than.simps(1-2) val-to-bool.simps(1)
 sorry

lemma disjoint-stamp-implies-less-than:
 $\llbracket \text{valid-value stampx } x;$
 $\text{valid-value stampy } y;$
 $\text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy};$
 $\text{stpi-upper stampx} < \text{stpi-lower stampy} \rrbracket$
 $\implies \text{val-to-bool}(\text{intval-less-than } x \ y)$
 sorry

lemma CanonicalizeConditionalProof:
 assumes CanonicalizeConditional before after
 assumes $[m, p] \vdash \text{before} \mapsto \text{res}$
 assumes $[m, p] \vdash \text{after} \mapsto \text{res}'$
 shows $\text{res} = \text{res}'$
 using assms
proof (induct rule: CanonicalizeConditional.induct)
 case (eq-branches t f c)

```

    then show ?case using evalDet by auto
next
  case (cond-eq c x y stampx stampy)
  obtain xval where xeval: [m,p] ⊢ x ↦ xval
    using cond-eq.hyps(1) cond-eq.prem(1) by blast
  obtain yval where yeval: [m,p] ⊢ y ↦ yval
    using cond-eq.prem(2) by auto
  show ?case proof (cases xval = yval)
    case True
    then show ?thesis
    by (smt (verit, ccfv-threshold) ConditionalExprE cond-eq.prem(1) cond-eq.prem(2)
evalDet xeval yeval)
  next
  case False
  then have ¬(val-to-bool(intval-equals xval yval))
  using ConstantExpr Value.distinct(9) valid-value.simps stamp-implies-valid-value
  apply (cases intval-equals xval yval)
  using IRTreeEval.val-to-bool.simps(2) apply presburger sorry
  then have res = yval
  by (smt (verit, ccfv-threshold) BinaryExprE ConditionalExprE bin-eval.simps(10)
cond-eq.hyps(1) cond-eq.prem(1) evalDet xeval yeval)
  then show ?thesis
  using cond-eq.prem(1) cond-eq.prem(2) xeval yeval evalDet by auto
qed
next
  case (condition-bounds-x c x y stampx stampy)
  obtain xval where xeval: [m,p] ⊢ x ↦ xval
    using condition-bounds-x.prem(2) by auto
  obtain yval where yeval: [m,p] ⊢ y ↦ yval
    using condition-bounds-x.hyps(1) condition-bounds-x.prem(1) by blast
  then show ?case proof (cases val-to-bool(intval-less-than xval yval))
    case True
    then show ?thesis
    by (smt (verit, best) BinaryExprE ConditionalExprE bin-eval.simps(11) con-
dition-bounds-x.hyps(1) condition-bounds-x.prem(1) condition-bounds-x.prem(2)
evalDet xeval yeval)
  next
  case False
  then have stpi-upper stampx = stpi-lower stampy
  by (metis False condition-bounds-x.hyps(4) order.not-eq-order-implies-strict
disjoint-stamp-implies-less-than condition-bounds-x.hyps(2) condition-bounds-x.hyps(3)
condition-bounds-x.hyps(6)
stamp-implies-valid-value xeval yeval)
  then have (xval = yval)
  by (metis False condition-bounds-x.hyps(2-3,6) stamp-implies-valid-value
stamps-touch-but-not-less-than-implies-equal xeval yeval)
  then have res = xval ∧ res' = xval
  using ConditionalExprE condition-bounds-x.prem(1) ⟨[m,p] ⊢ x ↦ res'⟩
evalDet xeval yeval

```

```

      by force
    then show ?thesis by simp
  qed
next
case (condition-bounds-y c x y stampx stampy)
obtain xval where xeval:  $[m,p] \vdash x \mapsto xval$ 
  using condition-bounds-y.hyps(1) condition-bounds-y.prem(1) by auto
obtain yval where yeval:  $[m,p] \vdash y \mapsto yval$ 
  using condition-bounds-y.hyps(1) condition-bounds-y.prem(1) by blast
then show ?case proof (cases val-to-bool(intval-less-than xval yval))
  case True
  then show ?thesis
    by (smt (verit, best) BinaryExprE ConditionalExprE bin-eval.simps(11) condition-bounds-y.hyps(1) condition-bounds-y.prem(1) condition-bounds-y.prem(2) evalDet xeval yeval)
  next
  case False
  have stpi-upper stampx = stpi-lower stampy
    by (metis False condition-bounds-y.hyps(4) order.not-eq-order-implies-strict disjoint-stamp-implies-less-than condition-bounds-y.hyps(2) condition-bounds-y.hyps(3) condition-bounds-y.hyps(6) stamp-implies-valid-value xeval yeval)
  then have (xval = yval)
    by (metis False condition-bounds-y.hyps(2-3,6) stamp-implies-valid-value stamps-touch-but-not-less-than-implies-equal xeval yeval)
  then have res = yval  $\wedge$  res' = yval
    using ConditionalExprE condition-bounds-y.prem(1)  $\langle [m,p] \vdash y \mapsto res' \rangle$ 
  evalDet xeval yeval
  by force
  then show ?thesis by simp
  qed
next
case (negate-condition nc c stampc lo hi stampx x stampy y)
obtain cval where ceval:  $[m,p] \vdash c \mapsto cval$ 
  using negate-condition.prem(2) by auto
obtain ncval where nceval:  $[m,p] \vdash nc \mapsto ncval$ 
  using negate-condition.prem(1) negate-condition.prem(2) by blast
then show ?case using assms proof (cases (val-to-bool ncval))
  case True
  obtain xval where xeval:  $[m,p] \vdash x \mapsto xval$ 
  by (metis (full-types) ConditionalExprE nceval evalDet True negate-condition.prem(1))
  then have res = xval
  by (metis (full-types) ConditionalExprE True evalDet nceval negate-condition.prem(1))
  have  $c \neq nc$ 
  by (simp add: negate-condition.hyps(1))
  then have  $\neg(\text{val-to-bool } cval)$ 
  by (metis IRTreeEval.val-to-bool.elims(2) IRTreeEval.val-to-bool.simps(1) True UnaryExprE ceval evalDet nceval negate-condition.hyps(1) unary-eval.simps(4))
  then have res' = xval

```

```

    using nceval ceval True negate-condition(1) negate-condition(9)
    by (metis (full-types) ConditionalExprE evalDet xeval)
  then show ?thesis
    by (simp add: (res = xval))
next
case False
obtain yval where yeval: [m,p] ⊢ y ↦ yval
by (metis (full-types) ConditionalExprE nceval evalDet False negate-condition.prem(1))
then have res = yval
    using False nceval negate-condition.prem(1) evaltree.ConditionalExpr yeval
evalDet
    by (metis (full-types) ConditionalExprE)
moreover have val-to-bool(cval)
by (metis False UnaryExprE ceval nceval negate-condition.hyps(1-3) unary-eval.simps(4)
    IRTreeEval.val-to-bool.simps(1) evalDet IRTreeEval.bool-to-val.simps(2)
    stamp-implies-valid-value valid-int32 zero-neq-one)
moreover have res' = yval
    using calculation(2) ceval negate-condition.prem evaltree.ConditionalExpr
yeval evalDet unary-eval.simps(4)
    by (metis (full-types) ConditionalExprE)
ultimately show ?thesis by simp
qed
next
case (const-true c val t f)
then show ?case using evalDet by auto
next
case (const-false c val t f)
then show ?case using evalDet by auto
qed
end

```