

# 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

<b>1</b>	<b>Runtime Values and Arithmetic</b>	<b>3</b>
<b>2</b>	<b>Nodes</b>	<b>9</b>
2.1	Types of Nodes . . . . .	9
2.2	Hierarchy of Nodes . . . . .	17
<b>3</b>	<b>Stamp Typing</b>	<b>24</b>
<b>4</b>	<b>Graph Representation</b>	<b>27</b>
4.0.1	Example Graphs . . . . .	31
<b>5</b>	<b>Data-flow Semantics</b>	<b>32</b>
5.1	Data-flow Tree Representation . . . . .	32
5.2	Data-flow Tree Evaluation . . . . .	34
5.3	Data-flow Tree Refinement . . . . .	36
<b>6</b>	<b>Data-flow Expression-Tree Theorems</b>	<b>36</b>
6.1	Extraction and Evaluation of Expression Trees is Deterministic.	37
6.2	Example Data-flow Optimisations . . . . .	42
6.3	Monotonicity of Expression Optimization . . . . .	42
<b>7</b>	<b>Tree to Graph</b>	<b>43</b>
<b>8</b>	<b>Control-flow Semantics</b>	<b>56</b>
8.1	Heap . . . . .	56
8.2	Intraprocedural Semantics . . . . .	56
8.3	Interprocedural Semantics . . . . .	59
8.4	Big-step Execution . . . . .	60
8.4.1	Heap Testing . . . . .	61
<b>9</b>	<b>Properties of Control-flow Semantics</b>	<b>62</b>
<b>10</b>	<b>Proof Infrastructure</b>	<b>63</b>
10.1	Bisimulation . . . . .	63
10.2	Formedness Properties . . . . .	64
10.3	Dynamic Frames . . . . .	65
10.4	Graph Rewriting . . . . .	70
10.5	Stuttering . . . . .	72
<b>11</b>	<b>Canonicalization Phase</b>	<b>73</b>
<b>12</b>	<b>Canonicalization Phase</b>	<b>84</b>

# 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⟩
end
```

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

```
instantiation Value :: minus
begin
```

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

```
instance ⟨proof⟩
end
```

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

```
instantiation Value :: times
begin
```

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

```
instance ⟨proof⟩
end
```

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

```
instantiation Value :: divide
begin
```

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

```
instance <proof>
end
```

```
fun intval-mod :: Value ⇒ Value ⇒ Value where
  intval-mod (IntVal32 v1) (IntVal32 v2) = (IntVal32 (word-of-int((sint v1) smod
(sint v2)))) |
  intval-mod (IntVal64 v1) (IntVal64 v2) = (IntVal64 (word-of-int((sint v1) smod
(sint v2)))) |
  intval-mod - - = UndefVal
```

```
instantiation Value :: modulo
begin
```

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

```
instance <proof>
end
```

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

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

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

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

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

```

fun intval-below :: Value  $\Rightarrow$  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]: shiftrl1 n = n * 2
  <proof>

```

```

lemma [code]: shiftr1 n = n div 2
  <proof>

```

```

lemma [code]: sshiftr1 n = word-of-int (sint n div 2)
  <proof>

```

```

definition shiftrl (infix << 75) where
  shiftrl w n = (shiftrl1  $\hat{\hat{}}$  n) w

```

```

lemma shiftrl-power[simp]: (x::('a::len) word) * ( $2^j$ ) = x << j
  <proof>

```

```

lemma (x::('a::len) word) * ( $(2^j) + 1$ ) = x << j + x
  <proof>

```

```

lemma (x::('a::len) word) * ( $(2^j) - 1$ ) = x << j - x
  <proof>

```

```

lemma (x::('a::len) word) * ( $(2^j) + (2^k)$ ) = x << j + x << k
  <proof>

```

```

lemma (x::('a::len) word) * ( $(2^j) - (2^k)$ ) = x << j - x << k
  <proof>

```

```

definition signed-shiftr (infix >> 75) where

```

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

**definition** *shiftr* (**infix**  $>>>$  75) **where**  
*shiftr*  $w\ n = (\text{shiftr1} \ \widehat{\sim} \ n)\ w$

**lemma** *shiftr-power*[simp]:  $(x::('a::\text{len})\ \text{word})\ \text{div}\ (2 \wedge j) = x >>> j$   
 $\langle \text{proof} \rangle$

**fun** *intval-left-shift* ::  $\text{Value} \Rightarrow \text{Value} \Rightarrow \text{Value}$  **where**  
*intval-left-shift* (*IntVal32*  $v1$ ) (*IntVal32*  $v2$ ) = *IntVal32* ( $v1 << \text{unat}\ (v2\ \text{AND}\ 0x1f)$ ) |  
*intval-left-shift* (*IntVal64*  $v1$ ) (*IntVal64*  $v2$ ) = *IntVal64* ( $v1 << \text{unat}\ (v2\ \text{AND}\ 0x3f)$ ) |  
*intval-left-shift* - - = *UndefVal*

**fun** *intval-right-shift* ::  $\text{Value} \Rightarrow \text{Value} \Rightarrow \text{Value}$  **where**  
*intval-right-shift* (*IntVal32*  $v1$ ) (*IntVal32*  $v2$ ) = *IntVal32* ( $v1 >> \text{unat}\ (v2\ \text{AND}\ 0x1f)$ ) |  
*intval-right-shift* (*IntVal64*  $v1$ ) (*IntVal64*  $v2$ ) = *IntVal64* ( $v1 >> \text{unat}\ (v2\ \text{AND}\ 0x3f)$ ) |  
*intval-right-shift* - - = *UndefVal*

**fun** *intval-uright-shift* ::  $\text{Value} \Rightarrow \text{Value} \Rightarrow \text{Value}$  **where**  
*intval-uright-shift* (*IntVal32*  $v1$ ) (*IntVal32*  $v2$ ) = *IntVal32* ( $v1 >>> \text{unat}\ (v2\ \text{AND}\ 0x1f)$ ) |  
*intval-uright-shift* (*IntVal64*  $v1$ ) (*IntVal64*  $v2$ ) = *IntVal64* ( $v1 >>> \text{unat}\ (v2\ \text{AND}\ 0x3f)$ ) |  
*intval-uright-shift* - - = *UndefVal*

**lemma** *word-add-sym*:  
**shows**  $\text{word-of-int}\ v1 + \text{word-of-int}\ v2 = \text{word-of-int}\ v2 + \text{word-of-int}\ v1$   
 $\langle \text{proof} \rangle$

**lemma** *intval-add-sym*:  
**shows**  $\text{intval-add}\ a\ b = \text{intval-add}\ b\ a$   
 $\langle \text{proof} \rangle$

**lemma** *word-add-assoc*:  
**shows**  $(\text{word-of-int}\ v1 + \text{word-of-int}\ v2) + \text{word-of-int}\ v3$   
 $= \text{word-of-int}\ v1 + (\text{word-of-int}\ v2 + \text{word-of-int}\ v3)$



*<proof>*

**lemma** *intval-bad1* [simp]: *intval-add* (*IntVal32* *x*) (*IntVal64* *y*) = *UndefVal*  
*<proof>*

**lemma** *intval-bad2* [simp]: *intval-add* (*IntVal64* *x*) (*IntVal32* *y*) = *UndefVal*  
*<proof>*

**lemma** *intval-assoc*: *intval-add32* (*intval-add32* *x y*) *z* = *intval-add32* *x* (*intval-add32* *y z*)  
*<proof>*

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

**lemma** *intval-add* (*IntVal32* ( $2^{31}-1$ )) (*IntVal32* ( $2^{31}-1$ )) = *IntVal32* ( $-2$ )  
*<proof>*

**lemma** *intval-add* (*IntVal64* ( $2^{31}-1$ )) (*IntVal64* ( $2^{31}-1$ )) = *IntVal64* 4294967294  
*<proof>*

**end**

## 2 Nodes

### 2.1 Types of Nodes

**theory** *IRNodes*  
  **imports**  
    *Values*  
**begin**

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

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

The *inputs\_of* and *successors\_of* functions partition those labelled references into input edges and successor edges of a node.

To identify each Node, we use a simple natural number index. Zero is always the start node in a graph. For human readability, within nodes we write

INPUT (or special case thereof) instead of ID for input edges, and SUCC instead of ID for control-flow successor edges. Optional edges are handled as "INPUT option" etc.

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

**datatype** (*discs-sels*) *IRNode* =  
*AbsNode* (*ir-value*: *INPUT*)  
 | *AddNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)  
 | *AndNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)  
 | *BeginNode* (*ir-next*: *SUCC*)  
 | *BytecodeExceptionNode* (*ir-arguments*: *INPUT* list) (*ir-stateAfter-opt*: *INPUT-STATE* option) (*ir-next*: *SUCC*)  
 | *ConditionalNode* (*ir-condition*: *INPUT-COND*) (*ir-trueValue*: *INPUT*) (*ir-falseValue*: *INPUT*)  
 | *ConstantNode* (*ir-const*: *Value*)  
 | *DynamicNewArrayNode* (*ir-elementType*: *INPUT*) (*ir-length*: *INPUT*) (*ir-voidClass-opt*: *INPUT* option) (*ir-stateBefore-opt*: *INPUT-STATE* option) (*ir-next*: *SUCC*)  
 | *EndNode*  
 | *ExceptionObjectNode* (*ir-stateAfter-opt*: *INPUT-STATE* option) (*ir-next*: *SUCC*)  
  
 | *FrameState* (*ir-monitorIds*: *INPUT-ASSOC* list) (*ir-outerFrameState-opt*: *INPUT-STATE* option) (*ir-values-opt*: *INPUT* list option) (*ir-virtualObjectMappings-opt*: *INPUT-STATE* list option)  
 | *IfNode* (*ir-condition*: *INPUT-COND*) (*ir-trueSuccessor*: *SUCC*) (*ir-falseSuccessor*: *SUCC*)  
 | *IntegerBelowNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)  
 | *IntegerEqualsNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)  
 | *IntegerLessThanNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)  
 | *InvokeNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT* option) (*ir-stateDuring-opt*: *INPUT-STATE* option) (*ir-stateAfter-opt*: *INPUT-STATE* option) (*ir-next*: *SUCC*)  
 | *InvokeWithExceptionNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT* option) (*ir-stateDuring-opt*: *INPUT-STATE* option) (*ir-stateAfter-opt*: *INPUT-STATE* option) (*ir-next*: *SUCC*) (*ir-exceptionEdge*: *SUCC*)  
 | *IsNullNode* (*ir-value*: *INPUT*)  
 | *KillingBeginNode* (*ir-next*: *SUCC*)  
 | *LeftShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)  
 | *LoadFieldNode* (*ir-nid*: *ID*) (*ir-field*: *string*) (*ir-object-opt*: *INPUT* option) (*ir-next*: *SUCC*)  
 | *LogicNegationNode* (*ir-value*: *INPUT-COND*)  
 | *LoopBeginNode* (*ir-ends*: *INPUT-ASSOC* list) (*ir-overflowGuard-opt*: *INPUT-GUARD*

*option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)  
 | *LoopEndNode* (*ir-loopBegin*: *INPUT-ASSOC*)  
 | *LoopExitNode* (*ir-loopBegin*: *INPUT-ASSOC*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)  
 | *MergeNode* (*ir-ends*: *INPUT-ASSOC list*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)  
 | *MethodCallTargetNode* (*ir-targetMethod*: *string*) (*ir-arguments*: *INPUT list*)  
 | *MulNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)  
 | *NarrowNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)  
 | *NegateNode* (*ir-value*: *INPUT*)  
 | *NewArrayNode* (*ir-length*: *INPUT*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)  
 | *NewInstanceNode* (*ir-nid*: *ID*) (*ir-instanceClass*: *string*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)  
 | *NotNode* (*ir-value*: *INPUT*)  
 | *OrNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)  
 | *ParameterNode* (*ir-index*: *nat*)  
 | *PiNode* (*ir-object*: *INPUT*) (*ir-guard-opt*: *INPUT-GUARD option*)  
 | *ReturnNode* (*ir-result-opt*: *INPUT option*) (*ir-memoryMap-opt*: *INPUT-EXT option*)  
 | *RightShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)  
 | *ShortCircuitOrNode* (*ir-x*: *INPUT-COND*) (*ir-y*: *INPUT-COND*)  
 | *SignExtendNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)  
 | *SignedDivNode* (*ir-nid*: *ID*) (*ir-x*: *INPUT*) (*ir-y*: *INPUT*) (*ir-zeroCheck-opt*: *INPUT-GUARD option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)  
 | *SignedRemNode* (*ir-nid*: *ID*) (*ir-x*: *INPUT*) (*ir-y*: *INPUT*) (*ir-zeroCheck-opt*: *INPUT-GUARD option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)  
 | *StartNode* (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)  
 | *StoreFieldNode* (*ir-nid*: *ID*) (*ir-field*: *string*) (*ir-value*: *INPUT*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-object-opt*: *INPUT option*) (*ir-next*: *SUCC*)  
 | *SubNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)  
 | *UnsignedRightShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)  
 | *UnwindNode* (*ir-exception*: *INPUT*)  
 | *ValuePhiNode* (*ir-nid*: *ID*) (*ir-values*: *INPUT list*) (*ir-merge*: *INPUT-ASSOC*)  
 | *ValueProxyNode* (*ir-value*: *INPUT*) (*ir-loopExit*: *INPUT-ASSOC*)  
 | *XorNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)  
 | *ZeroExtendNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)  
 | *NoNode*  
 | *RefNode* (*ir-ref*: *ID*)

**fun** *opt-to-list* :: 'a *option*  $\Rightarrow$  'a *list* **where**  
*opt-to-list* *None* = [] |

$opt\text{-}to\text{-}list\ (Some\ v) = [v]$

```
fun opt-list-to-list :: 'a list option  $\Rightarrow$  '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  $\Rightarrow$  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*:  $\text{successors-of } (\text{RefNode } \text{ref}) = [\text{ref}]$

**lemma** *inputs-of* (*FrameState* *x* (*Some* *y*) (*Some* *z*) *None*) = *x* @ [*y*] @ *z*  
 <proof>

**lemma** *successors-of* (*FrameState* *x* (*Some* *y*) (*Some* *z*) *None*) = []  
 <proof>

**lemma** *inputs-of* (*IfNode* *c* *t* *f*) = [*c*]  
 <proof>

**lemma** *successors-of* (*IfNode* *c* *t* *f*) = [*t*, *f*]  
 <proof>

**lemma** *inputs-of* (*EndNode*) = []  $\wedge$  *successors-of* (*EndNode*) = []  
 <proof>

**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*  $\Rightarrow$  *bool* **where**  
   *is-EndNode* *EndNode* = *True* |  
   *is-EndNode* - = *False*

**fun** *is-VirtualState* :: *IRNode*  $\Rightarrow$  *bool* **where**  
   *is-VirtualState* *n* = ((*is-FrameState* *n*))

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

```

fun is-ShiftNode :: IRNode  $\Rightarrow$  bool where
  is-ShiftNode n = ((is-LeftShiftNode n)  $\vee$  (is-RightShiftNode n)  $\vee$  (is-UnsignedRightShiftNode
n))

fun is-BinaryNode :: IRNode  $\Rightarrow$  bool where
  is-BinaryNode n = ((is-BinaryArithmeticNode n)  $\vee$  (is-ShiftNode n))

fun is-AbstractLocalNode :: IRNode  $\Rightarrow$  bool where
  is-AbstractLocalNode n = ((is-ParameterNode n))

fun is-IntegerConvertNode :: IRNode  $\Rightarrow$  bool where
  is-IntegerConvertNode n = ((is-NarrowNode n)  $\vee$  (is-SignExtendNode n)  $\vee$ 
(is-ZeroExtendNode n))

fun is-UnaryArithmeticNode :: IRNode  $\Rightarrow$  bool where
  is-UnaryArithmeticNode n = ((is-AbsNode n)  $\vee$  (is-NegateNode n)  $\vee$  (is-NotNode
n))

fun is-UnaryNode :: IRNode  $\Rightarrow$  bool where
  is-UnaryNode n = ((is-IntegerConvertNode n)  $\vee$  (is-UnaryArithmeticNode n))

fun is-PhiNode :: IRNode  $\Rightarrow$  bool where
  is-PhiNode n = ((is-ValuePhiNode n))

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

fun is-UnaryOpLogicNode :: IRNode  $\Rightarrow$  bool where
  is-UnaryOpLogicNode n = ((is-IsNullNode n))

fun is-IntegerLowerThanNode :: IRNode  $\Rightarrow$  bool where
  is-IntegerLowerThanNode n = ((is-IntegerBelowNode n)  $\vee$  (is-IntegerLessThanNode
n))

fun is-CompareNode :: IRNode  $\Rightarrow$  bool where
  is-CompareNode n = ((is-IntegerEqualsNode n)  $\vee$  (is-IntegerLowerThanNode n))

fun is-BinaryOpLogicNode :: IRNode  $\Rightarrow$  bool where
  is-BinaryOpLogicNode n = ((is-CompareNode n))

fun is-LogicNode :: IRNode  $\Rightarrow$  bool where
  is-LogicNode n = ((is-BinaryOpLogicNode n)  $\vee$  (is-LogicNegationNode n)  $\vee$ 
(is-ShortCircuitOrNode n)  $\vee$  (is-UnaryOpLogicNode n))

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

fun is-FloatingNode :: IRNode  $\Rightarrow$  bool where
  is-FloatingNode n = ((is-AbstractLocalNode n)  $\vee$  (is-BinaryNode n)  $\vee$  (is-ConditionalNode

```

$n) \vee (is-ConstantNode\ n) \vee (is-FloatingGuardedNode\ n) \vee (is-LogicNode\ n) \vee (is-PhiNode\ n) \vee (is-ProxyNode\ n) \vee (is-UnaryNode\ n))$

**fun** *is-AccessFieldNode* :: *IRNode*  $\Rightarrow$  *bool* **where**  
*is-AccessFieldNode* *n* = ((*is-LoadFieldNode* *n*)  $\vee$  (*is-StoreFieldNode* *n*))

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

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

**fun** *is-IntegerDivRemNode* :: *IRNode*  $\Rightarrow$  *bool* **where**  
*is-IntegerDivRemNode* *n* = ((*is-SignedDivNode* *n*)  $\vee$  (*is-SignedRemNode* *n*))

**fun** *is-FixedBinaryNode* :: *IRNode*  $\Rightarrow$  *bool* **where**  
*is-FixedBinaryNode* *n* = ((*is-IntegerDivRemNode* *n*))

**fun** *is-DeoptimizingFixedWithNextNode* :: *IRNode*  $\Rightarrow$  *bool* **where**  
*is-DeoptimizingFixedWithNextNode* *n* = ((*is-AbstractNewObjectNode* *n*)  $\vee$  (*is-FixedBinaryNode* *n*))

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

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

**fun** *is-AbstractMergeNode* :: *IRNode*  $\Rightarrow$  *bool* **where**  
*is-AbstractMergeNode* *n* = ((*is-LoopBeginNode* *n*)  $\vee$  (*is-MergeNode* *n*))

**fun** *is-BeginStateSplitNode* :: *IRNode*  $\Rightarrow$  *bool* **where**  
*is-BeginStateSplitNode* *n* = ((*is-AbstractMergeNode* *n*)  $\vee$  (*is-ExceptionObjectNode* *n*)  $\vee$  (*is-LoopExitNode* *n*)  $\vee$  (*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))

```

$n) \vee (is\text{-}UnaryOpLogicNode\ n))$

**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 irregardless of their edges. It will return true if both the node parameters are the same node class.

```

fun is-same-ir-node-type :: IRNode  $\Rightarrow$  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)  $\wedge$  (is-LoopEndNode n2))  $\vee$ 
  ((is-LoopExitNode n1)  $\wedge$  (is-LoopExitNode n2))  $\vee$ 
  ((is-MergeNode n1)  $\wedge$  (is-MergeNode n2))  $\vee$ 
  ((is-MethodCallTargetNode n1)  $\wedge$  (is-MethodCallTargetNode n2))  $\vee$ 
  ((is-MulNode n1)  $\wedge$  (is-MulNode n2))  $\vee$ 
  ((is-NegateNode n1)  $\wedge$  (is-NegateNode n2))  $\vee$ 
  ((is-NewArrayNode n1)  $\wedge$  (is-NewArrayNode n2))  $\vee$ 
  ((is-NewInstanceNode n1)  $\wedge$  (is-NewInstanceNode n2))  $\vee$ 
  ((is-NotNode n1)  $\wedge$  (is-NotNode n2))  $\vee$ 
  ((is-OrNode n1)  $\wedge$  (is-OrNode n2))  $\vee$ 
  ((is-ParameterNode n1)  $\wedge$  (is-ParameterNode n2))  $\vee$ 

```

```

((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  $\wedge$  nn2) (an1  $\wedge$  an2)
) |
meet (MethodPointersStamp nn1 an1) (MethodPointersStamp nn2 an2) = (
  MethodPointersStamp (nn1  $\wedge$  nn2) (an1  $\wedge$  an2)
) |
meet s1 s2 = IllegalStamp

```

— Calculate the join stamp of two stamps

```

fun join :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  Stamp where
  join VoidStamp VoidStamp = VoidStamp |
  join (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1  $\neq$  b2 then IllegalStamp else
    (IntegerStamp b1 (max l1 l2) (min u1 u2))
  ) |
  join (KlassPointerStamp nn1 an1) (KlassPointerStamp nn2 an2) = (
    if ((nn1  $\vee$  nn2)  $\wedge$  (an1  $\vee$  an2))
    then (empty-stamp (KlassPointerStamp nn1 an1))
    else (KlassPointerStamp (nn1  $\vee$  nn2) (an1  $\vee$  an2))
  ) |
  join (MethodCountersPointerStamp nn1 an1) (MethodCountersPointerStamp nn2
an2) = (
    if ((nn1  $\vee$  nn2)  $\wedge$  (an1  $\vee$  an2))
    then (empty-stamp (MethodCountersPointerStamp nn1 an1))
    else (MethodCountersPointerStamp (nn1  $\vee$  nn2) (an1  $\vee$  an2))
  ) |
  join (MethodPointersStamp nn1 an1) (MethodPointersStamp nn2 an2) = (
    if ((nn1  $\vee$  nn2)  $\wedge$  (an1  $\vee$  an2))
    then (empty-stamp (MethodPointersStamp nn1 an1))
    else (MethodPointersStamp (nn1  $\vee$  nn2) (an1  $\vee$  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  $\mapsto$  (IRNode  $\times$  Stamp) . finite (dom g)}
 $\langle proof \rangle$ 
```

```
setup-lifting type-definition-IRGraph
```

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

**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*  $\langle \text{proof} \rangle$

**lift-definition** *stamp* :: *IRGraph*  $\Rightarrow$  *ID*  $\Rightarrow$  *Stamp*  
**is** *with-default* *IllegalStamp snd*  $\langle \text{proof} \rangle$

**lift-definition** *add-node* :: *ID*  $\Rightarrow$  (*IRNode*  $\times$  *Stamp*)  $\Rightarrow$  *IRGraph*  $\Rightarrow$  *IRGraph*  
**is**  $\lambda nid k g. \text{if } fst \ k = \text{NoNode} \text{ then } g \text{ else } g(nid \mapsto k) \langle \text{proof} \rangle$

**lift-definition** *remove-node* :: *ID*  $\Rightarrow$  *IRGraph*  $\Rightarrow$  *IRGraph*  
**is**  $\lambda nid g. g(nid := \text{None}) \langle \text{proof} \rangle$

**lift-definition** *replace-node* :: *ID*  $\Rightarrow$  (*IRNode*  $\times$  *Stamp*)  $\Rightarrow$  *IRGraph*  $\Rightarrow$  *IRGraph*  
**is**  $\lambda nid k g. \text{if } fst \ k = \text{NoNode} \text{ then } g \text{ else } g(nid \mapsto k) \langle \text{proof} \rangle$

**lift-definition** *as-list* :: *IRGraph*  $\Rightarrow$  (*ID*  $\times$  *IRNode*  $\times$  *Stamp*) *list*  
**is**  $\lambda g. \text{map } (\lambda k. (k, \text{the } (g \ k))) \ (\text{sorted-list-of-set } (\text{dom } g)) \langle \text{proof} \rangle$

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

**lift-definition** *irgraph* :: (*ID*  $\times$  (*IRNode*  $\times$  *Stamp*)) *list*  $\Rightarrow$  *IRGraph*  
**is** *map-of*  $\circ$  *no-node*  
 $\langle \text{proof} \rangle$

**definition** *as-set* :: *IRGraph*  $\Rightarrow$  (*ID*  $\times$  (*IRNode*  $\times$  *Stamp*)) *set* **where**  
*as-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**  $\trianglelefteq$  30) **where**  
*domain-subtraction* *s r* =  $\{(x, y) . (x, y) \in r \wedge x \notin s\}$

**notation** (*latex*)  
*domain-subtraction* ( $- \trianglelefteq -$ )

**code-datatype** *irgraph*

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

**lemma** *no-node-clears*:

$res = no\text{-}node\ xs \longrightarrow (\forall x \in set\ res. fst\ (snd\ x) \neq NoNode)$   
 $\langle proof \rangle$

**lemma** *dom-eq*:

**assumes**  $\forall x \in set\ xs. fst\ (snd\ x) \neq NoNode$   
**shows**  $filter\ none\ (map\ of\ xs) = dom\ (map\ of\ xs)$   
 $\langle proof \rangle$

**lemma** *fil-eq*:

$filter\ none\ (map\ of\ (no\text{-}node\ xs)) = set\ (map\ fst\ (no\text{-}node\ xs))$   
 $\langle proof \rangle$

**lemma** *irgraph[code]*:  $ids\ (irgraph\ m) = set\ (map\ fst\ (no\text{-}node\ m))$

$\langle proof \rangle$

**lemma** *[code]*:  $Rep\ IRGraph\ (irgraph\ m) = map\ of\ (no\text{-}node\ m)$

$\langle proof \rangle$

**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**

$input\ edges\ g = (\bigcup i \in ids\ g. \{(i,j) | j. j \in (inputs\ g\ i)\})$

— Find all the nodes in the graph that have nid as an input - the usages of nid

**fun** *usages* ::  $IRGraph \Rightarrow ID \Rightarrow ID\ set$  **where**

$usages\ g\ nid = \{j. j \in ids\ g \wedge (j,nid) \in input\ edges\ g\}$

**fun** *successor-edges* ::  $IRGraph \Rightarrow ID\ rel$  **where**

$successor\ edges\ g = (\bigcup i \in ids\ g. \{(i,j) | j. j \in (succ\ g\ i)\})$

**fun** *predecessors* ::  $IRGraph \Rightarrow ID \Rightarrow ID\ set$  **where**

$predecessors\ g\ nid = \{j. j \in ids\ g \wedge (j,nid) \in successor\ edges\ g\}$

**fun** *nodes-of* ::  $IRGraph \Rightarrow (IRNode \Rightarrow bool) \Rightarrow ID\ set$  **where**

$nodes\ of\ g\ sel = \{nid \in ids\ g. sel\ (kind\ g\ nid)\}$

**fun** *edge* ::  $(IRNode \Rightarrow 'a) \Rightarrow ID \Rightarrow IRGraph \Rightarrow 'a$  **where**

$edge\ sel\ nid\ g = sel\ (kind\ g\ nid)$

**fun** *filtered-inputs* ::  $IRGraph \Rightarrow ID \Rightarrow (IRNode \Rightarrow bool) \Rightarrow ID\ list$  **where**

$filtered\ inputs\ g\ nid\ f = filter\ (f \circ (kind\ g))\ (inputs\ of\ (kind\ g\ nid))$

**fun** *filtered-successors* ::  $IRGraph \Rightarrow ID \Rightarrow (IRNode \Rightarrow bool) \Rightarrow ID\ list$  **where**

$filtered\ successors\ g\ nid\ f = filter\ (f \circ (kind\ g))\ (successors\ of\ (kind\ g\ nid))$

**fun** *filtered-usages* ::  $IRGraph \Rightarrow ID \Rightarrow (IRNode \Rightarrow bool) \Rightarrow ID\ set$  **where**

$filtered\ usages\ g\ nid\ f = \{n \in (usages\ g\ nid). f\ (kind\ g\ n)\}$

**fun** *is-empty* ::  $IRGraph \Rightarrow bool$  **where**

$is\ empty\ g = (ids\ g = \{\})$

**fun** *any-usage* ::  $IRGraph \Rightarrow ID \Rightarrow ID$  **where**

*any-usage*  $g$   $nid = hd$  (*sorted-list-of-set* (*usages*  $g$   $nid$ ))

**lemma** *ids-some*[*simp*]:  $x \in ids\ g \longleftrightarrow kind\ g\ x \neq NoNode$   
*<proof>*

**lemma** *not-in-g*:  
**assumes**  $nid \notin ids\ g$   
**shows**  $kind\ g\ nid = NoNode$   
*<proof>*

**lemma** *valid-creation*[*simp*]:  
 $finite\ (dom\ g) \longleftrightarrow Rep-IRGraph\ (Abs-IRGraph\ g) = g$   
*<proof>*

**lemma** [*simp*]:  $finite\ (ids\ g)$   
*<proof>*

**lemma** [*simp*]:  $finite\ (ids\ (irgraph\ g))$   
*<proof>*

**lemma** [*simp*]:  $finite\ (dom\ g) \longrightarrow ids\ (Abs-IRGraph\ g) = \{nid \in dom\ g . \nexists s. g\ nid = Some\ (NoNode, s)\}$   
*<proof>*

**lemma** [*simp*]:  $finite\ (dom\ g) \longrightarrow kind\ (Abs-IRGraph\ g) = (\lambda x . (case\ g\ x\ of\ None \Rightarrow NoNode \mid Some\ n \Rightarrow fst\ n))$   
*<proof>*

**lemma** [*simp*]:  $finite\ (dom\ g) \longrightarrow stamp\ (Abs-IRGraph\ g) = (\lambda x . (case\ g\ x\ of\ None \Rightarrow IllegalStamp \mid Some\ n \Rightarrow snd\ n))$   
*<proof>*

**lemma** [*simp*]:  $ids\ (irgraph\ g) = set\ (map\ fst\ (no-node\ g))$   
*<proof>*

**lemma** [*simp*]:  $kind\ (irgraph\ g) = (\lambda nid. (case\ (map-of\ (no-node\ g))\ nid\ of\ None \Rightarrow NoNode \mid Some\ n \Rightarrow fst\ n))$   
*<proof>*

**lemma** [*simp*]:  $stamp\ (irgraph\ g) = (\lambda nid. (case\ (map-of\ (no-node\ g))\ nid\ of\ None \Rightarrow IllegalStamp \mid Some\ n \Rightarrow snd\ n))$   
*<proof>*

**lemma** *map-of-upd*:  $(map-of\ g)(k \mapsto v) = (map-of\ ((k, v) \# g))$   
*<proof>*

**lemma** [*code*]:  $replace-node\ nid\ k\ (irgraph\ g) = (irgraph\ ((nid, k) \# g))$   
*<proof>*

**lemma** *[code]: add-node nid k (irgraph g) = (irgraph (((nid, k) # g)))*  
*<proof>*

**lemma** *add-node-lookup:*  
 $gup = \text{add-node } nid \ (k, s) \ g \longrightarrow$   
 $(\text{if } k \neq \text{NoNode} \text{ then } \text{kind } gup \ nid = k \wedge \text{stamp } gup \ nid = s \text{ else } \text{kind } gup \ nid = \text{kind } g \ nid)$   
*<proof>*

**lemma** *remove-node-lookup:*  
 $gup = \text{remove-node } nid \ g \longrightarrow \text{kind } gup \ nid = \text{NoNode} \wedge \text{stamp } gup \ nid = \text{IllegalStamp}$   
*<proof>*

**lemma** *replace-node-lookup[simp]:*  
 $gup = \text{replace-node } nid \ (k, s) \ g \wedge k \neq \text{NoNode} \longrightarrow \text{kind } gup \ nid = k \wedge \text{stamp } gup \ nid = s$   
*<proof>*

**lemma** *replace-node-unchanged:*  
 $gup = \text{replace-node } nid \ (k, s) \ g \longrightarrow (\forall \ n \in (\text{ids } g - \{nid\}) . n \in \text{ids } g \wedge n \in \text{ids } gup \wedge \text{kind } g \ n = \text{kind } gup \ n)$   
*<proof>*

#### 4.0.1 Example Graphs

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

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

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*  
 $\text{eg2-sq} = \text{irgraph } [$   
 $(0, \text{StartNode } \text{None } 5, \text{VoidStamp}),$   
 $(1, \text{ParameterNode } 0, \text{default-stamp}),$   
 $(4, \text{MulNode } 1 \ 1, \text{default-stamp}),$   
 $(5, \text{ReturnNode } (\text{Some } 4) \ \text{None}, \text{default-stamp})$   
 $]$

**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 }
  ⟨proof⟩

```

## 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**  
 $\llbracket \text{value} \neq \text{UndefVal} \rrbracket \implies \text{not-undef-or-fail value value}$

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

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

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

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

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

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

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

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

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

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

**inductive**

$evaltrees :: MapState \Rightarrow Params \Rightarrow IRExpr\ list \Rightarrow Value\ list \Rightarrow bool\ ([-,] \vdash - \mapsto_L$   
- 55)

**for  $m\ p$  where**

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

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

**code-pred** (*modes: i  $\Rightarrow$  i  $\Rightarrow$  i  $\Rightarrow$  o  $\Rightarrow$  bool as evalTs*)  
*evaltrees*  $\langle proof \rangle$

### 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*  $\Rightarrow$  *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*  
 $\langle proof \rangle$

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**  $\langle proof \rangle$

**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*]:

$$\begin{aligned} g \vdash n \simeq e &\implies \\ \text{kind } g \ n = \text{ConstantNode } c &\implies \\ e = \text{ConstantExpr } c & \\ \langle \text{proof} \rangle \end{aligned}$$

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

$$\begin{aligned} g \vdash n \simeq e &\implies \\ \text{kind } g \ n = \text{ParameterNode } i &\implies \\ (\exists s. e = \text{ParameterExpr } i \ s) & \\ \langle \text{proof} \rangle \end{aligned}$$

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

$$\begin{aligned} g \vdash n \simeq e &\implies \\ \text{kind } g \ n = \text{ConditionalNode } c \ t \ f &\implies \\ (\exists ce \ te \ fe. e = \text{ConditionalExpr } ce \ te \ fe) & \\ \langle \text{proof} \rangle \end{aligned}$$

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

$$\begin{aligned} g \vdash n \simeq e &\implies \\ \text{kind } g \ n = \text{AbsNode } x &\implies \\ (\exists xe. e = \text{UnaryExpr } \text{UnaryAbs } xe) & \\ \langle \text{proof} \rangle \end{aligned}$$

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

$$\begin{aligned} g \vdash n \simeq e &\implies \\ \text{kind } g \ n = \text{NotNode } x &\implies \\ (\exists xe. e = \text{UnaryExpr } \text{UnaryNot } xe) & \\ \langle \text{proof} \rangle \end{aligned}$$

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

$$\begin{aligned} g \vdash n \simeq e &\implies \\ \text{kind } g \ n = \text{NegateNode } x &\implies \\ (\exists xe. e = \text{UnaryExpr } \text{UnaryNeg } xe) & \\ \langle \text{proof} \rangle \end{aligned}$$

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

$$\begin{aligned} g \vdash n \simeq e &\implies \\ \text{kind } g \ n = \text{LogicNegationNode } x &\implies \\ (\exists xe. e = \text{UnaryExpr } \text{UnaryLogicNegation } xe) & \\ \langle \text{proof} \rangle \end{aligned}$$

**lemma** *rep-add* [*rep*]:

$g \vdash n \simeq e \implies$   
 $kind\ g\ n = AddNode\ x\ y \implies$   
 $(\exists\ xe\ ye. e = BinaryExpr\ BinAdd\ xe\ ye)$   
 $\langle proof \rangle$

**lemma** *rep-sub* [*rep*]:

$g \vdash n \simeq e \implies$   
 $kind\ g\ n = SubNode\ x\ y \implies$   
 $(\exists\ xe\ ye. e = BinaryExpr\ BinSub\ xe\ ye)$   
 $\langle proof \rangle$

**lemma** *rep-mul* [*rep*]:

$g \vdash n \simeq e \implies$   
 $kind\ g\ n = MulNode\ x\ y \implies$   
 $(\exists\ xe\ ye. e = BinaryExpr\ BinMul\ xe\ ye)$   
 $\langle proof \rangle$

**lemma** *rep-and* [*rep*]:

$g \vdash n \simeq e \implies$   
 $kind\ g\ n = AndNode\ x\ y \implies$   
 $(\exists\ xe\ ye. e = BinaryExpr\ BinAnd\ xe\ ye)$   
 $\langle proof \rangle$

**lemma** *rep-or* [*rep*]:

$g \vdash n \simeq e \implies$   
 $kind\ g\ n = OrNode\ x\ y \implies$   
 $(\exists\ xe\ ye. e = BinaryExpr\ BinOr\ xe\ ye)$   
 $\langle proof \rangle$

**lemma** *rep-xor* [*rep*]:

$g \vdash n \simeq e \implies$   
 $kind\ g\ n = XorNode\ x\ y \implies$   
 $(\exists\ xe\ ye. e = BinaryExpr\ BinXor\ xe\ ye)$   
 $\langle proof \rangle$

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

$g \vdash n \simeq e \implies$   
 $kind\ g\ n = IntegerBelowNode\ x\ y \implies$   
 $(\exists\ xe\ ye. e = BinaryExpr\ BinIntegerBelow\ xe\ ye)$   
 $\langle proof \rangle$

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

$g \vdash n \simeq e \implies$   
 $kind\ g\ n = IntegerEqualsNode\ x\ y \implies$   
 $(\exists\ xe\ ye. e = BinaryExpr\ BinIntegerEquals\ xe\ ye)$   
 $\langle proof \rangle$

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

$g \vdash n \simeq e \implies$   
 $kind\ g\ n = IntegerLessThanNode\ x\ y \implies$   
 $(\exists\ xe\ ye. e = BinaryExpr\ BinIntegerLessThan\ xe\ ye)$   
 $\langle proof \rangle$

**lemma** *rep-narrow* [rep]:

$g \vdash n \simeq e \implies$   
 $kind\ g\ n = NarrowNode\ ib\ rb\ x \implies$   
 $(\exists\ x. e = UnaryExpr\ (UnaryNarrow\ ib\ rb)\ x)$   
 $\langle proof \rangle$

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

$g \vdash n \simeq e \implies$   
 $kind\ g\ n = SignExtendNode\ ib\ rb\ x \implies$   
 $(\exists\ x. e = UnaryExpr\ (UnarySignExtend\ ib\ rb)\ x)$   
 $\langle proof \rangle$

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

$g \vdash n \simeq e \implies$   
 $kind\ g\ n = ZeroExtendNode\ ib\ rb\ x \implies$   
 $(\exists\ x. e = UnaryExpr\ (UnaryZeroExtend\ ib\ rb)\ x)$   
 $\langle proof \rangle$

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

$g \vdash n \simeq e \implies$   
 $is-preevaluated\ (kind\ g\ n) \implies$   
 $(\exists\ s. e = LeafExpr\ n\ s)$   
 $\langle proof \rangle$

**method** *solve-det* **uses** *node* =

$\langle match\ node\ in\ kind\ -\ - = node\ -\ for\ node \Rightarrow$   
 $\langle match\ rep\ in\ r: - \implies - = node\ - \implies - \Rightarrow$   
 $\langle match\ IRNode.inject\ in\ i: (node\ - = node\ -) = - \Rightarrow$   
 $\langle match\ RepE\ in\ e: - \implies (\bigwedge x. - = node\ x \implies -) \implies - \Rightarrow$   
 $\langlemetis\ i\ e\ r\rangle\rangle\rangle\ |\$   
 $match\ node\ in\ kind\ -\ - = node\ -\ -\ for\ node \Rightarrow$   
 $\langle match\ rep\ in\ r: - \implies - = node\ -\ - \implies - \Rightarrow$   
 $\langle match\ IRNode.inject\ in\ i: (node\ -\ - = node\ -\ -) = - \Rightarrow$   
 $\langle match\ RepE\ in\ e: - \implies (\bigwedge x\ y. - = node\ x\ y \implies -) \implies - \Rightarrow$   
 $\langlemetis\ i\ e\ r\rangle\rangle\rangle\ |\$   
 $match\ node\ in\ kind\ -\ - = node\ -\ -\ -\ for\ node \Rightarrow$   
 $\langle match\ rep\ in\ r: - \implies - = node\ -\ -\ - \implies - \Rightarrow$   
 $\langle match\ IRNode.inject\ in\ i: (node\ -\ -\ - = node\ -\ -\ -) = - \Rightarrow$   
 $\langle match\ RepE\ in\ e: - \implies (\bigwedge x\ y\ z. - = node\ x\ y\ z \implies -) \implies - \Rightarrow$   
 $\langlemetis\ i\ e\ r\rangle\rangle\rangle\ |\$   
 $match\ node\ in\ kind\ -\ - = node\ -\ -\ -\ for\ node \Rightarrow$   
 $\langle match\ rep\ in\ r: - \implies - = node\ -\ -\ - \implies - \Rightarrow$

$\langle \text{match } IRNode.inject \text{ in } i: (node \text{ --- } = node \text{ --- }) = - \Rightarrow$   
 $\langle \text{match } RepE \text{ in } e: - \Rightarrow (\bigwedge x. - = node \text{ --- } x \Rightarrow -) \Rightarrow - \Rightarrow$   
 $\langle \text{metis } i \text{ e } r \rangle \rangle \rangle$

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

**lemma** *repDet*:

**shows**  $(g \vdash n \simeq e1) \Rightarrow (g \vdash n \simeq e2) \Rightarrow e1 = e2$   
 $\langle \text{proof} \rangle$

**lemma** *repAllDet*:

$g \vdash xs \simeq_L e1 \Rightarrow$   
 $g \vdash xs \simeq_L e2 \Rightarrow$   
 $e1 = e2$   
 $\langle \text{proof} \rangle$

**lemma** *evalDet*:

$[m,p] \vdash e \mapsto v1 \Rightarrow$   
 $[m,p] \vdash e \mapsto v2 \Rightarrow$   
 $v1 = v2$   
 $\langle \text{proof} \rangle$

**lemma** *evalAllDet*:

$[m,p] \vdash e \mapsto_L v1 \Rightarrow$   
 $[m,p] \vdash e \mapsto_L v2 \Rightarrow$   
 $v1 = v2$   
 $\langle \text{proof} \rangle$

**lemma** *encodeEvalDet*:

$[g,m,p] \vdash e \mapsto v1 \Rightarrow$   
 $[g,m,p] \vdash e \mapsto v2 \Rightarrow$   
 $v1 = v2$   
 $\langle \text{proof} \rangle$

**lemma** *graphDet*:  $([g,m,p] \vdash nid \mapsto v1) \wedge ([g,m,p] \vdash nid \mapsto v2) \Rightarrow v1 = v2$   
 $\langle \text{proof} \rangle$

A valid value cannot be *UndefVal*.

**lemma** *valid-not-undef*:

**assumes** *a1*: *valid-value s val*  
**assumes** *a2*:  $s \neq VoidStamp$   
**shows**  $val \neq UndefVal$   
 $\langle \text{proof} \rangle$

**lemma** *valid-VoidStamp[elim]*:

**shows** *valid-value VoidStamp val*  $\Rightarrow$   
 $val = UndefVal$



$\langle proof \rangle$

**lemma** *valid-ObjStamp[elim]*:

**shows** *valid-value* (*ObjectStamp* *klass exact nonNull alwaysNull*) *val*  $\implies$   
( $\exists v. val = \text{ObjRef } v$ )

$\langle proof \rangle$

**lemma** *valid-int32[elim]*:

**shows** *valid-value* (*IntegerStamp* 32 *l h*) *val*  $\implies$   
( $\exists v. val = \text{IntVal32 } v$ )

$\langle proof \rangle$

**lemma** *valid-int64[elim]*:

**shows** *valid-value* (*IntegerStamp* 64 *l h*) *val*  $\implies$   
( $\exists v. val = \text{IntVal64 } v$ )

$\langle proof \rangle$

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**  $\exists v. val = (\text{IntVal32 } v)$

$\langle proof \rangle$

**lemma** *leafint64*:

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

$\langle proof \rangle$

**lemma** *default-stamp [simp]*: *default-stamp* = *IntegerStamp* 32 (-2147483648)  
2147483647

$\langle proof \rangle$

**lemma** *valid32 [simp]*:

**assumes** *valid-value* (*IntegerStamp* 32 *lo hi*) *val*  
**shows**  $\exists v. (val = (\text{IntVal32 } v) \wedge lo \leq \text{sint } v \wedge \text{sint } v \leq hi)$

$\langle proof \rangle$

**lemma** *valid64 [simp]*:

**assumes** *valid-value* (*IntegerStamp* 64 *lo hi*) *val*  
**shows**  $\exists v. (val = (\text{IntVal64 } v) \wedge lo \leq \text{sint } v \wedge \text{sint } v \leq hi)$

$\langle proof \rangle$

**experiment begin**

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

$[m, p] \vdash \text{expr} \mapsto val \implies$

$\text{valid-value } (\text{stamp-expr } \text{expr}) \text{ val}$   
 $\langle \text{proof} \rangle$   
**end**

**lemma** *valid32or64*:  
**assumes**  $\text{valid-value } (\text{IntegerStamp } b \text{ lo hi}) \ x$   
**shows**  $(\exists \ v1. (x = \text{IntVal32 } v1)) \vee (\exists \ v2. (x = \text{IntVal64 } v2))$   
 $\langle \text{proof} \rangle$

**lemma** *valid32or64-both*:  
**assumes**  $\text{valid-value } (\text{IntegerStamp } b \text{ lox hix}) \ x$   
**and**  $\text{valid-value } (\text{IntegerStamp } b \text{ loy hiy}) \ y$   
**shows**  $(\exists \ v1 \ v2. x = \text{IntVal32 } v1 \wedge y = \text{IntVal32 } v2) \vee (\exists \ v3 \ v4. x = \text{IntVal64 } v3 \wedge y = \text{IntVal64 } v4)$   
 $\langle \text{proof} \rangle$

## 6.2 Example Data-flow Optimisations

**lemma** *a0a-helper* [simp]:  
**assumes**  $a: \text{valid-value } (\text{IntegerStamp } 32 \text{ lo hi}) \ v$   
**shows**  $\text{intval-add } v (\text{IntVal32 } 0) = v$   
 $\langle \text{proof} \rangle$

**lemma** *a0a*:  $(\text{BinaryExpr BinAdd } (\text{LeafExpr } 1 \text{ default-stamp}) (\text{ConstantExpr } (\text{IntVal32 } 0)))$   
 $\geq (\text{LeafExpr } 1 \text{ default-stamp})$   
 $\langle \text{proof} \rangle$

**lemma** *xyx-y-helper* [simp]:  
**assumes**  $\text{valid-value } (\text{IntegerStamp } 32 \text{ lox hix}) \ x$   
**assumes**  $\text{valid-value } (\text{IntegerStamp } 32 \text{ loy hiy}) \ y$   
**shows**  $\text{intval-add } x (\text{intval-sub } y \ x) = y$   
 $\langle \text{proof} \rangle$

**lemma** *xyx-y*:  
 $(\text{BinaryExpr BinAdd } (\text{LeafExpr } x (\text{IntegerStamp } 32 \text{ lox hix}))$   
 $(\text{BinaryExpr BinSub } (\text{LeafExpr } y (\text{IntegerStamp } 32 \text{ loy hiy}))$   
 $(\text{LeafExpr } x (\text{IntegerStamp } 32 \text{ lox hix}))))$   
 $\geq (\text{LeafExpr } y (\text{IntegerStamp } 32 \text{ loy hiy}))$   
 $\langle \text{proof} \rangle$

## 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**  $e \geq e'$   
**shows**  $(UnaryExpr\ op\ e) \geq (UnaryExpr\ op\ e')$   
 $\langle proof \rangle$

**lemma** *mono-binary*:

**assumes**  $x \geq x'$   
**assumes**  $y \geq y'$   
**shows**  $(BinaryExpr\ op\ x\ y) \geq (BinaryExpr\ op\ x'\ y')$   
 $\langle proof \rangle$

**lemma** *mono-conditional*:

**assumes**  $ce \geq ce'$   
**assumes**  $te \geq te'$   
**assumes**  $fe \geq fe'$   
**shows**  $(ConditionalExpr\ ce\ te\ fe) \geq (ConditionalExpr\ ce'\ te'\ fe')$   
 $\langle proof \rangle$

**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* |  
*is-preevaluated* (*LoadFieldNode* *n* - - -) = *True* |  
*is-preevaluated* (*SignedDivNode* *n* - - - - -) = *True* |

$is\_preevaluated \ (SignedRemNode \ n \ - \ - \ - \ -) = True \mid$   
 $is\_preevaluated \ (ValuePhiNode \ n \ - \ -) = True \mid$   
 $is\_preevaluated \ - = False$

### 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:*

$\llbracket kind \ g \ n = AddNode \ x \ y;$

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

*MulNode:*

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

*SubNode:*

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

*AndNode:*

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

*OrNode:*

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

*XorNode:*

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

*IntegerBelowNode:*

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

*IntegerEqualsNode:*

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

*IntegerLessThanNode:*

$$\begin{aligned}
&\llbracket \text{kind } g \ n = \text{IntegerLessThanNode } x \ y; \\
&\quad g \vdash x \simeq xe;
\end{aligned}$$

$g \vdash y \simeq ye$   
 $\implies g \vdash n \simeq (\text{BinaryExpr BinIntegerLessThan } xe \ ye) \mid$

*NarrowNode:*

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

*SignExtendNode:*

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

*ZeroExtendNode:*

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

*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*  $\langle \text{proof} \rangle$

**inductive**

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

*RepNil:*

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

*RepCons:*

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

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

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

$$\frac{\text{kind } g \ n = \text{ParameterNode } i \quad \text{stamp } g \ n = s}{g \vdash n \simeq \text{ParameterExpr } i \ s}$$

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

**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 Illegal-*  
*Stamp)) |*

*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  $\implies$  fresh-id g n

```

**code-pred** fresh-id  $\langle$ proof $\rangle$

**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**



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

*ConstantNodeSame:*

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

*ConstantNodeNew:*

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

*ParameterNodeSame:*

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

*ParameterNodeNew:*

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

*ConditionalNodeSame:*

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

*ConditionalNodeNew:*

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

*UnaryNodeSame:*

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

*UnaryNodeNew:*

$\llbracket g \triangleleft xe \rightsquigarrow (g2, \ x);$   
 $s' = stamp\text{-}unary \ op \ (stamp \ g2 \ x);$

$find\_node\_and\_stamp\ g2\ (unary\_node\ op\ x,\ s') = None;$   
 $n = get\_fresh\_id\ g2;$   
 $g' = add\_node\ n\ (unary\_node\ op\ x,\ s')\ g2$   
 $\implies g \triangleleft (UnaryExpr\ op\ xe) \rightsquigarrow (g', n) \mid$

*BinaryNodeSame:*

$\llbracket g \triangleleft_L [xe, ye] \rightsquigarrow (g2, [x, y]);$   
 $s' = stamp\_binary\ op\ (stamp\ g2\ x)\ (stamp\ g2\ y);$   
 $find\_node\_and\_stamp\ g2\ (bin\_node\ op\ x\ y,\ s') = Some\ n \rrbracket$   
 $\implies g \triangleleft (BinaryExpr\ op\ xe\ ye) \rightsquigarrow (g2, n) \mid$

*BinaryNodeNew:*

$\llbracket g \triangleleft_L [xe, ye] \rightsquigarrow (g2, [x, y]);$   
 $s' = stamp\_binary\ op\ (stamp\ g2\ x)\ (stamp\ g2\ y);$   
 $find\_node\_and\_stamp\ g2\ (bin\_node\ op\ x\ y,\ s') = None;$   
 $n = get\_fresh\_id\ g2;$   
 $g' = add\_node\ n\ (bin\_node\ op\ x\ y,\ s')\ g2$   
 $\implies g \triangleleft (BinaryExpr\ op\ xe\ ye) \rightsquigarrow (g', n) \mid$

*AllLeafNodes:*

$stamp\ g\ n = s$   
 $\implies g \triangleleft (LeafExpr\ n\ s) \rightsquigarrow (g, n) \mid$

*UnrepNil:*

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

*UnrepCons:*

$\llbracket g \triangleleft xe \rightsquigarrow (g2, x);$   
 $g2 \triangleleft_L xes \rightsquigarrow (g3, xs) \rrbracket$   
 $\implies g \triangleleft_L (xe \# xes) \rightsquigarrow (g3, x \# xs)$

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

*unrep*  $\langle proof \rangle$

**code-pred** (*modes*:  $i \Rightarrow i \Rightarrow o \Rightarrow bool$  as *unrepListE*) *unrepList*  $\langle proof \rangle$

$find\_node\_and\_stamp\ g\ (ConstantNode\ c,\ constantAsStamp\ c) = Some\ n$   
 $\hline g \triangleleft ConstantExpr\ c \rightsquigarrow (g, n)$

$find\_node\_and\_stamp\ g\ (ConstantNode\ c,\ constantAsStamp\ c) = None$   
 $n = get\_fresh\_id\ g \quad g' = add\_node\ n\ (ConstantNode\ c,\ constantAsStamp\ c)\ g$   
 $\hline g \triangleleft ConstantExpr\ c \rightsquigarrow (g', n)$

$find\_node\_and\_stamp\ g\ (ParameterNode\ i,\ s) = Some\ n$   
 $\hline g \triangleleft ParameterExpr\ i\ s \rightsquigarrow (g, n)$

$$\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**  $graph\text{-}refinement$ :  
 $graph\text{-}refinement\ g1\ g2 \implies (\forall\ n\ m\ p\ v. n \in ids\ g1 \longrightarrow ([g1, m, p] \vdash n \mapsto v) \longrightarrow ([g2, m, p] \vdash n \mapsto v))$   
 $\langle proof \rangle$

**definition**  $graph\text{-}represents\text{-}expression :: IRGraph \Rightarrow ID \Rightarrow IRExpr \Rightarrow bool$   
 $(- \vdash - \triangleleft -\ 50)$   
**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**  $e1 \geq e2$   
 $\langle proof \rangle$

**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**  $e1 \geq e2$   
 $\langle \text{proof} \rangle$

**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**  $e1 \geq e2$   
 $\langle \text{proof} \rangle$

**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**  $e1 \geq e2$   
 $\langle \text{proof} \rangle$

**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**  $e1 \geq e2$   
 $\langle \text{proof} \rangle$

**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**  $e1 \geq e2$   
 $\langle \text{proof} \rangle$

**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**  $e1 \geq e2$   
 $\langle \text{proof} \rangle$

**lemma** *mono-conditional-graph*:

**assumes**  $\text{kind } g1 \ n = \text{ConditionalNode } c \ t \ f \wedge \text{kind } g2 \ n = \text{ConditionalNode } c \ t \ f$   
**assumes**  $(g1 \vdash c \simeq ce1) \wedge (g2 \vdash c \simeq ce2)$   
**assumes**  $(g1 \vdash t \simeq te1) \wedge (g2 \vdash t \simeq te2)$   
**assumes**  $(g1 \vdash f \simeq fe1) \wedge (g2 \vdash f \simeq fe2)$   
**assumes**  $ce1 \geq ce2 \wedge te1 \geq te2 \wedge fe1 \geq fe2$   
**assumes**  $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$   
**shows**  $e1 \geq e2$   
 $\langle \text{proof} \rangle$

**lemma** *mono-add*:

**assumes**  $\text{kind } g1 \ n = \text{AddNode } x \ y \wedge \text{kind } g2 \ n = \text{AddNode } x \ y$   
**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**  $e1 \geq e2$   
 $\langle \text{proof} \rangle$

**lemma** *mono-mul*:

**assumes**  $\text{kind } g1 \ n = \text{MulNode } x \ y \wedge \text{kind } g2 \ n = \text{MulNode } x \ y$   
**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**  $e1 \geq e2$   
 $\langle \text{proof} \rangle$

**lemma** *encodes-contains*:

$g \vdash n \simeq e \implies$   
 $\text{kind } g \ n \neq \text{NoNode}$   
 $\langle \text{proof} \rangle$

**lemma** *no-encoding*:

**assumes**  $n \notin \text{ids } g$   
**shows**  $\neg(g \vdash n \simeq e)$   
 $\langle \text{proof} \rangle$

**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**  $\text{kind } g1 \ n = \text{kind } g2 \ n \wedge \text{stamp } g1 \ n = \text{stamp } g2 \ n$   
 $\langle \text{proof} \rangle$

**method** *metis-node-eq-unary* **for**  $\text{node} :: 'a \Rightarrow \text{IRNode} =$

$(\text{match } \text{IRNode.inject} \text{ in } i: (\text{node } - = \text{node } -) = - \implies$   
 $\langle \text{metis } i \rangle)$

```

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

```

```

lemma graph-antics-preservation:
  assumes a:  $e1' \geq e2'$ 
  assumes b: ( $\{n'\} \trianglelefteq \text{as-set } g1$ )  $\subseteq$  as-set g2
  assumes c:  $g1 \vdash n' \simeq e1'$ 
  assumes d:  $g2 \vdash n' \simeq e2'$ 
  shows graph-refinement g1 g2
   $\langle$ proof $\rangle$ 

```

```

definition maximal-sharing:
  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:
   $e1 \geq e2$ 
   $\wedge$  ( $g1 \vdash n \simeq e1$ )  $\wedge$  maximal-sharing g1
   $\wedge$  ( $\{n\} \trianglelefteq \text{as-set } g1$ )  $\subseteq$  as-set g2
   $\wedge$  ( $g2 \vdash n \simeq e2$ )  $\wedge$  maximal-sharing g2
   $\implies$  graph-refinement g1 g2
   $\langle$ proof $\rangle$ 

```

```

declare [[simp-trace]]
lemma equal-refines:
  fixes e1 e2 :: IRExpr
  assumes  $e1 = e2$ 
  shows  $e1 \geq e2$ 
   $\langle$ proof $\rangle$ 
declare [[simp-trace=false]]

```

```

lemma subset-implies-evals:
  assumes as-set g1  $\subseteq$  as-set g2
  shows ( $g1 \vdash n \simeq e$ )  $\implies$  ( $g2 \vdash n \simeq e$ )
   $\langle$ proof $\rangle$ 

```

```

lemma subset-refines:
  assumes as-set g1  $\subseteq$  as-set g2
  shows graph-refinement g1 g2

```

$\langle proof \rangle$

**lemma** *graph-construction*:

$e1 \geq e2$   
 $\wedge as\text{-}set\ g1 \subseteq as\text{-}set\ g2 \wedge maximal\text{-}sharing\ g1$   
 $\wedge (g2 \vdash n \simeq e2) \wedge maximal\text{-}sharing\ g2$   
 $\implies (g2 \vdash n \leq e1) \wedge graph\text{-}refinement\ g1\ g2$   
 $\langle proof \rangle$

**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*  $= ((\lambda f. \lambda p. Undefined), 0)$

### 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;

```

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

*NewInstanceNode:*

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

*LoadFieldNode:*

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

*SignedDivNode:*

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

*SignedRemNode:*

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

*StaticLoadFieldNode:*

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

*StoreFieldNode:*

$$\begin{aligned}
& \llbracket \text{kind } g \text{ nid} = (\text{StoreFieldNode } nid \text{ f newval - (Some obj) } nid') ; \\
& \quad g \vdash \text{newval} \simeq \text{newvalE} ; \\
& \quad g \vdash \text{obj} \simeq \text{objE} ;
\end{aligned}$$

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

*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*  $\langle \text{proof} \rangle$

### 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:*

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

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

*ReturnNodeVoid:*

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

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

*UnwindNode:*

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

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

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

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

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

## 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**

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

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

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

|

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

$$\text{has-return } m';$$

$$l' = (l @ \llbracket (g, \text{nid}, m, p) \rrbracket)$$

$$\implies \text{exec } P \llbracket ((g, \text{nid}, m, p) \# xs), h \rrbracket l \llbracket ((g', \text{nid}', m', p') \# ys), h' \rrbracket l'$$
**code-pred** (modes:  $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow \text{bool}$  as *Exec*) *exec*  $\langle \text{proof} \rangle$

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

### 8.4.1 Heap Testing

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

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

**definition** *field-sq* :: *string* **where**  
 $\text{field-sq} = "sq"$

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

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

$new\_map\_state, p3)], new\_heap) \rightarrow *3* res\}$

**definition**  $eg4\_sq :: IRGraph$  **where**

```

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

```

**values**  $\{h\text{-load-field field-sq (Some 0) (prod.snd res) \mid res.}$

$(\lambda x. \text{Some } eg4\_sq) \vdash ([ (eg4\_sq, 0, new\_map\_state, p3), (eg4\_sq, 0, new\_map\_state, p3)], new\_heap) \rightarrow *4* res\}$

**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**  $stepDet$ :

$$(g, p \vdash (nid, m, h) \rightarrow next) \implies$$

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

$\langle proof \rangle$

**lemma**  $stepRefNode$ :

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

$\langle proof \rangle$

**lemma**  $IfNodeStepCases$ :

**assumes**  $kind\ g\ nid = IfNode\ cond\ tb\ fb$

**assumes**  $g \vdash cond \simeq condE$

**assumes**  $[m, p] \vdash condE \mapsto v$

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

**shows**  $nid' \in \{tb, fb\}$   
 $\langle proof \rangle$

**lemma** *IfNodeSeq*:

**shows**  $kind\ g\ nid = IfNode\ cond\ tb\ fb \longrightarrow \neg(is\_sequential\_node\ (kind\ g\ nid))$   
 $\langle proof \rangle$

**lemma** *IfNodeCond*:

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

**lemma** *step-in-ids*:

**assumes**  $g, p \vdash (nid, m, h) \rightarrow (nid', m', h')$   
**shows**  $nid \in ids\ g$   
 $\langle proof \rangle$

**end**

## 10 Proof Infrastructure

### 10.1 Bisimulation

**theory** *Bisimulation*

**imports**

*Stuttering*

**begin**

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

$(- \cdot - \sim -)$  **for**  $nid$  **where**  
 $\llbracket \forall P'. (g\ m\ p\ h \vdash nid \rightsquigarrow P') \longrightarrow (\exists Q'. (g'\ m\ p\ h \vdash nid \rightsquigarrow Q') \wedge P' = Q');$   
 $\forall Q'. (g'\ m\ p\ h \vdash nid \rightsquigarrow Q') \longrightarrow (\exists P'. (g\ m\ p\ h \vdash nid \rightsquigarrow P') \wedge P' = Q') \rrbracket$   
 $\implies nid \cdot g \sim g'$

A strong bisimulation between no-op transitions

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

$(- \mid - \sim -)$  **for**  $nid$  **where**  
 $\llbracket \forall P'. (g, p \vdash (nid, m, h) \rightarrow P') \longrightarrow (\exists Q'. (g', p \vdash (nid, m, h) \rightarrow Q') \wedge P' = Q');$   
 $\forall Q'. (g', p \vdash (nid, m, h) \rightarrow Q') \longrightarrow (\exists P'. (g, p \vdash (nid, m, h) \rightarrow P') \wedge P' = Q') \rrbracket$   
 $\implies nid \mid g \sim g'$

**lemma** *lockstep-strong-bisimulation*:

**assumes**  $g' = replace\_node\ nid\ node\ g$

**assumes**  $g, p \vdash (nid, m, h) \rightarrow (nid', m, h)$   
**assumes**  $g', p \vdash (nid, m, h) \rightarrow (nid', m, h)$   
**shows**  $nid \mid g \sim g'$   
 $\langle proof \rangle$

**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**  $nid \mid g \sim g'$   
 $\langle proof \rangle$

**end**

## 10.2 Formedness Properties

**theory** *Form*

**imports**

*Semantics.TreeToGraph*

**begin**

**definition** *wf-start* **where**

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

**definition** *wf-closed* **where**

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

**definition** *wf-phs* **where**

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

**definition** *wf-ends* **where**

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

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

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

**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*  
*<proof>*

**lemma** *wf-eg2-sq: wf-graph eg2-sq*  
*<proof>*

**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*  $\longrightarrow$

$$(n \in \text{ids } g1 \wedge n \in \text{ids } g2 \wedge \text{kind } g1 \ n = \text{kind } g2 \ n \wedge \text{stamp } g1 \ n = \text{stamp } g2 \ n))$$

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

**lemma** *node-unchanged*:  
**assumes** *unchanged ns g1 g2*  
**assumes** *nid*  $\in$  *ns*  
**shows** *kind g1 nid* = *kind g2 nid*  
 $\langle \text{proof} \rangle$

**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*  
 $\langle \text{proof} \rangle$

Some notation for input nodes used

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

*use0*: *nid*  $\in$  *ids g*  
 $\implies \text{eval-uses } g \text{ nid nid} \mid$

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

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

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

**lemma** *eval-usages-self*:  
**assumes** *nid*  $\in$  *ids g*  
**shows** *nid*  $\in$  *eval-usages g nid*  
 $\langle \text{proof} \rangle$

**lemma** *not-in-g-inputs*:  
**assumes** *nid*  $\notin$  *ids g*  
**shows** *inputs g nid* =  $\{\}$   
 $\langle \text{proof} \rangle$

**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**  $\text{child} \in \text{inputs } g \text{ nid}$   
 $\langle \text{proof} \rangle$

**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**  $\text{child} \in \text{inputs } g \text{ nid}$   
 $\langle \text{proof} \rangle$

**lemma** *inp-in-g*:  
**assumes**  $n \in \text{inputs } g \text{ nid}$   
**shows**  $\text{nid} \in \text{ids } g$   
 $\langle \text{proof} \rangle$

**lemma** *inp-in-g-wf*:  
**assumes**  $\text{wf-graph } g$   
**assumes**  $n \in \text{inputs } g \text{ nid}$   
**shows**  $n \in \text{ids } g$   
 $\langle \text{proof} \rangle$

**lemma** *kind-unchanged*:  
**assumes**  $\text{nid} \in \text{ids } g1$   
**assumes**  $\text{unchanged } (\text{eval-usages } g1 \text{ nid}) \text{ } g1 \text{ } g2$   
**shows**  $\text{kind } g1 \text{ nid} = \text{kind } g2 \text{ nid}$   
 $\langle \text{proof} \rangle$

**lemma** *stamp-unchanged*:  
**assumes**  $\text{nid} \in \text{ids } g1$   
**assumes**  $\text{unchanged } (\text{eval-usages } g1 \text{ nid}) \text{ } g1 \text{ } g2$   
**shows**  $\text{stamp } g1 \text{ nid} = \text{stamp } g2 \text{ nid}$   
 $\langle \text{proof} \rangle$

**lemma** *child-unchanged*:  
**assumes**  $\text{child} \in \text{inputs } g1 \text{ nid}$   
**assumes**  $\text{unchanged } (\text{eval-usages } g1 \text{ nid}) \text{ } g1 \text{ } g2$   
**shows**  $\text{unchanged } (\text{eval-usages } g1 \text{ child}) \text{ } g1 \text{ } g2$   
 $\langle \text{proof} \rangle$

**lemma** *eval-usages*:  
**assumes**  $us = \text{eval-usages } g \text{ nid}$   
**assumes**  $\text{nid}' \in \text{ids } g$   
**shows**  $\text{eval-uses } g \text{ nid } \text{nid}' \longleftrightarrow \text{nid}' \in us \text{ (is } ?P \longleftrightarrow ?Q)$

$\langle \text{proof} \rangle$

**lemma** *inputs-are-uses*:

**assumes**  $\text{id}' \in \text{inputs } g \text{ nid}$

**shows**  $\text{eval-uses } g \text{ nid id}'$

$\langle \text{proof} \rangle$

**lemma** *inputs-are-usages*:

**assumes**  $\text{id}' \in \text{inputs } g \text{ nid}$

**assumes**  $\text{id}' \in \text{ids } g$

**shows**  $\text{id}' \in \text{eval-usages } g \text{ nid}$

$\langle \text{proof} \rangle$

**lemma** *inputs-of-are-usages*:

**assumes**  $\text{List.member } (\text{inputs-of } (\text{kind } g \text{ nid})) \text{ id}'$

**assumes**  $\text{id}' \in \text{ids } g$

**shows**  $\text{id}' \in \text{eval-usages } g \text{ nid}$

$\langle \text{proof} \rangle$

**lemma** *usage-includes-inputs*:

**assumes**  $us = \text{eval-usages } g \text{ nid}$

**assumes**  $ls = \text{inputs } g \text{ nid}$

**assumes**  $ls \subseteq \text{ids } g$

**shows**  $ls \subseteq us$

$\langle \text{proof} \rangle$

**lemma** *elim-inp-set*:

**assumes**  $k = \text{kind } g \text{ nid}$

**assumes**  $k \neq \text{NoNode}$

**assumes**  $\text{child} \in \text{set } (\text{inputs-of } k)$

**shows**  $\text{child} \in \text{inputs } g \text{ nid}$

$\langle \text{proof} \rangle$

**lemma** *encode-in-ids*:

**assumes**  $g \vdash \text{id} \simeq e$

**shows**  $\text{id} \in \text{ids } g$

$\langle \text{proof} \rangle$

**lemma** *eval-in-ids*:

**assumes**  $[g, m, p] \vdash \text{id} \mapsto v$

**shows**  $\text{id} \in \text{ids } g$

$\langle \text{proof} \rangle$

**lemma** *transitive-kind-same*:

**assumes**  $\text{unchanged } (\text{eval-usages } g1 \text{ nid}) \text{ } g1 \text{ } g2$

**shows**  $\forall \text{id}' \in (\text{eval-usages } g1 \text{ nid}) . \text{kind } g1 \text{ id}' = \text{kind } g2 \text{ id}'$

$\langle \text{proof} \rangle$

**theorem** *stay-same-encoding*:

**assumes**  $nc$ : *unchanged* (*eval-usages*  $g1$   $nid$ )  $g1$   $g2$   
**assumes**  $g1$ :  $g1 \vdash nid \simeq e$   
**assumes**  $wf$ : *wf-graph*  $g1$   
**shows**  $g2 \vdash nid \simeq e$   
 $\langle proof \rangle$

**theorem** *stay-same*:  
**assumes**  $nc$ : *unchanged* (*eval-usages*  $g1$   $nid$ )  $g1$   $g2$   
**assumes**  $g1$ :  $[g1, m, p] \vdash nid \mapsto v1$   
**assumes**  $wf$ : *wf-graph*  $g1$   
**shows**  $[g2, m, p] \vdash nid \mapsto v1$   
 $\langle proof \rangle$

**lemma** *add-changed*:  
**assumes**  $gup = \text{add-node } new \ k \ g$   
**shows** *changeonly*  $\{new\} \ g \ gup$   
 $\langle proof \rangle$

**lemma** *disjoint-change*:  
**assumes** *changeonly*  $change \ g \ gup$   
**assumes**  $nochange = ids \ g - change$   
**shows** *unchanged*  $nochange \ g \ gup$   
 $\langle proof \rangle$

**lemma** *add-node-unchanged*:  
**assumes**  $new \notin ids \ g$   
**assumes**  $nid \in ids \ g$   
**assumes**  $gup = \text{add-node } new \ k \ g$   
**assumes** *wf-graph*  $g$   
**shows** *unchanged* (*eval-usages*  $g$   $nid$ )  $g \ gup$   
 $\langle proof \rangle$

**lemma** *eval-uses-imp*:  
 $((nid' \in ids \ g \wedge nid = nid') \vee nid' \in inputs \ g \ nid \vee (\exists nid'' . eval\text{-}uses \ g \ nid \ nid'' \wedge eval\text{-}uses \ g \ nid'' \ nid')) \longleftrightarrow eval\text{-}uses \ g \ nid \ nid'$   
 $\langle proof \rangle$

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

```

lemma no-external-use:
  assumes wf-graph g
  assumes nid' ∉ ids g
  assumes nid ∈ ids g
  shows  $\neg(\text{eval-uses } g \text{ } nid \text{ } nid')$ 
   $\langle \text{proof} \rangle$ 

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'
   $\langle \text{proof} \rangle$ 

lemma replace-usages-changeonly:
  assumes nid ∈ ids g
  assumes g' = replace-usages nid nid' g
  shows changeonly {nid} g g'
   $\langle \text{proof} \rangle$ 

lemma replace-usages-unchanged:
  assumes nid ∈ ids g
  assumes g' = replace-usages nid nid' g
  shows unchanged (ids g - {nid}) g g'
   $\langle \text{proof} \rangle$ 

fun nextNid :: IRGraph  $\Rightarrow$  ID where
  nextNid g = (Max (ids g)) + 1

lemma max-plus-one:
  fixes c :: ID set
  shows  $\llbracket \text{finite } c; c \neq \{\} \rrbracket \implies (\text{Max } c) + 1 \notin c$ 
   $\langle \text{proof} \rangle$ 

lemma ids-finite:

```

*finite* (*ids g*)  
 ⟨*proof*⟩

**lemma** *nextNidNotIn*:  
*ids g* ≠ {} ⟶ *nextNid g* ∉ *ids g*  
 ⟨*proof*⟩

**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*⟩

**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*⟩

**lemma** *diff-forall*:  
**assumes** ∀ *n* ∈ *ids g* - {*nid*}. *cond n*  
**shows** ∀ *n*. *n* ∈ *ids g* ∧ *n* ∉ {*nid*} ⟶ *cond n*  
 ⟨*proof*⟩

**lemma** *replace-node-changeonly*:  
**assumes** *g'* = *replace-node nid node g*  
**shows** *changeonly* {*nid*} *g g'*  
 ⟨*proof*⟩

**lemma** *add-node-changeonly*:  
**assumes** *g'* = *add-node nid node g*  
**shows** *changeonly* {*nid*} *g g'*  
 ⟨*proof*⟩

**lemma** *constantConditionNoEffect*:  
**assumes** ¬(*is-IfNode* (*kind g nid*))  
**shows** *g* = *constantCondition b nid* (*kind g nid*) *g*  
 ⟨*proof*⟩

**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)
  <proof>

```

**lemma** *constantCondition-changeonly*:

```

  assumes nid ∈ ids g
  assumes g' = constantCondition b nid (kind g nid) g
  shows changeonly {nid} g g'
<proof>

```

**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>

```

**lemma** *constantConditionValid*:

```

  assumes kind g ifcond = IfNode cond t f
  assumes [g, m, p] ⊢ cond ↦ v
  assumes const = val-to-bool v
  assumes g' = constantCondition const ifcond (kind g ifcond) g
  shows ∃ nid'. (g m p h ⊢ ifcond ∼ nid') ⟷ (g' m p h ⊢ ifcond ∼ nid')
<proof>

```

**end**

## 10.5 Stuttering

**theory** *Stuttering*  
**imports**  
*Semantics.IRStepThms*  
**begin**

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

*StutterStep*:

$$\llbracket g, p \vdash (nid, m, h) \rightarrow (nid', m, h) \rrbracket \\ \implies g \ m \ p \ h \vdash nid \rightsquigarrow nid'$$

*Transitive*:

$$\llbracket g, p \vdash (nid, m, h) \rightarrow (nid'', m, h); \\ g \ m \ p \ h \vdash nid'' \rightsquigarrow nid' \rrbracket \\ \implies g \ m \ p \ h \vdash nid \rightsquigarrow nid'$$

**lemma** *stuttering-successor*:



```

    assumes  $(g, p \vdash (nid, m, h) \rightarrow (nid', m, h))$ 
    shows  $\{P'. (g \ m \ p \ h \vdash nid \rightsquigarrow P')\} = \{nid'\} \cup \{nid''. (g \ m \ p \ h \vdash nid' \rightsquigarrow nid'')\}$ 
  <proof>

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*:

$\llbracket y = (\text{ConstantExpr } c);$   
 $\text{is-neutral } \text{op } c;$   
 $\text{stampx} = \text{stamp-expr } x;$   
 $\text{stampy} = \text{stamp-expr } y;$   
 $\text{stp-bits } \text{stampx} = \text{stp-bits } \text{stampy};$   
 $\text{is-IntegerStamp } \text{stampx} \wedge \text{is-IntegerStamp } \text{stampy} \rrbracket$   
 $\implies \text{CanonicalizeBinaryOp } (\text{BinaryExpr } \text{op } x \ y) x \mid$

*binary-fold-yzero32*:

$\llbracket y = \text{ConstantExpr } c;$   
 $\text{is-annihilator } \text{op } c;$   
 $\text{stampx} = \text{stamp-expr } x;$

*stampy* = *stamp-expr* *y*;  
*stp-bits* *stampx* = *stp-bits* *stampy*;  
*stp-bits* *stampx* = 32;  
*is-IntegerStamp* *stampx*  $\wedge$  *is-IntegerStamp* *stampy*]  
 $\implies$  *CanonicalizeBinaryOp* (*BinaryExpr* *op* *x* *y*) (*ConstantExpr* *c*) |

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

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

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

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

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

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

$\llbracket x = (\text{BinaryExpr } \text{BinSub } a \text{ } y);$   
*stampa* = *stamp-expr* *a*;  
*stampy* = *stamp-expr* *y*;  
*is-IntegerStamp* *stampa*  $\wedge$  *is-IntegerStamp* *stampy*;  
*stp-bits* *stampa* = *stp-bits* *stampy*]  
 $\implies$  *CanonicalizeAdd* (*BinaryExpr* *BinAdd* *x* *y*) *a* |

*add-ysub:*

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

*add-xnegate:*

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

*add-ynegate:*

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

**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:*

$\llbracket x = (\text{BinaryExpr BinAdd } a \ b);$   
 $\text{stamp}_a = \text{stamp-expr } a;$   
 $\text{stamp}_b = \text{stamp-expr } b;$   
 $\text{is-IntegerStamp stamp}_a \wedge \text{is-IntegerStamp stamp}_b;$   
 $\text{stp-bits stamp}_a = \text{stp-bits stamp}_b \rrbracket$

$$\implies \text{CanonicalizeSub } (\text{BinaryExpr BinSub } x \ b) \ a \mid$$

*sub-left-add2:*

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

*sub-left-sub:*

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

*sub-right-add1:*

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

*sub-right-add2:*

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

*sub-right-sub:*

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

*sub-xzero32:*  
 $\llbracket \text{stamp}x = \text{stamp-expr } x;$   
 $\text{stamp}x = \text{IntegerStamp } 32 \text{ lo } hi \rrbracket$   
 $\implies \text{CanonicalizeSub } (\text{BinaryExpr } \text{BinSub } (\text{ConstantExpr } (\text{IntVal32 } 0)) \text{ } x)$   
 $(\text{UnaryExpr } \text{UnaryNeg } x) \mid$   
*sub-xzero64:*  
 $\llbracket \text{stamp}x = \text{stamp-expr } x;$   
 $\text{stamp}x = \text{IntegerStamp } 64 \text{ lo } hi \rrbracket$   
 $\implies \text{CanonicalizeSub } (\text{BinaryExpr } \text{BinSub } (\text{ConstantExpr } (\text{IntVal64 } 0)) \text{ } x)$   
 $(\text{UnaryExpr } \text{UnaryNeg } x) \mid$

*sub-y-negate:*

$\llbracket nb = (\text{UnaryExpr } \text{UnaryNeg } b);$   
 $\text{stamp}a = \text{stamp-expr } a;$   
 $\text{stamp}b = \text{stamp-expr } b;$   
 $is\text{-IntegerStamp } \text{stamp}a \wedge is\text{-IntegerStamp } \text{stamp}b;$   
 $stp\text{-bits } \text{stamp}a = stp\text{-bits } \text{stamp}b \rrbracket$   
 $\implies \text{CanonicalizeSub } (\text{BinaryExpr } \text{BinSub } a \text{ } nb) (\text{BinaryExpr } \text{BinAdd } a \text{ } b)$

**inductive** *CanonicalizeNegate* :: *IRExpr*  $\Rightarrow$  *IRExpr*  $\Rightarrow$  *bool* **where**  
*negate-negate:*

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

*negate-sub:*

$\llbracket e = (\text{BinaryExpr } \text{BinSub } x \text{ } y);$   
 $\text{stamp}x = \text{stamp-expr } x;$   
 $\text{stamp}y = \text{stamp-expr } y;$   
 $is\text{-IntegerStamp } \text{stamp}x \wedge is\text{-IntegerStamp } \text{stamp}y;$   
 $stp\text{-bits } \text{stamp}x = stp\text{-bits } \text{stamp}y \rrbracket$   
 $\implies \text{CanonicalizeNegate } (\text{UnaryExpr } \text{UnaryNeg } e) (\text{BinaryExpr } \text{BinSub } y \text{ } x)$

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

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

*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*:

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

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

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

*and-demorgans*:

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

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

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

*or-demorgans*:

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

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

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

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

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

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

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

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

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



*BinIntegerEquals y z*) |

*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:*

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

*int-equals-right-contains-left3:*

$\llbracket \text{right} = (\text{BinaryExpr BinSub } x \ y);$

$zero = (ConstantExpr (IntVal32 0))$   
 $\implies CanonicalizeIntegerEquals (BinaryExpr BinIntegerEquals x right) (BinaryExpr BinIntegerEquals y zero)$

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

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

*cond-eq*:

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

*condition-bounds-x*:

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

*condition-bounds-y*:

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

*negate-condition*:

$\llbracket nc = (UnaryExpr UnaryLogicNegation c);$   
 $stampc = stamp\text{-}expr\ c;$   
 $stampc = IntegerStamp\ 32\ lo\ hi;$   
 $stampx = stamp\text{-}expr\ x;$   
 $stampy = stamp\text{-}expr\ y;$   
 $stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$

$$\begin{aligned} & \text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy} \\ \implies & \text{CanonicalizeConditional } (\text{ConditionalExpr nc } x \ y) \ (\text{ConditionalExpr c } y \ x) \end{aligned}$$

|

*const-true:*

$$\begin{aligned} & \llbracket c = \text{ConstantExpr val}; \\ & \quad \text{val-to-bool val} \rrbracket \\ \implies & \text{CanonicalizeConditional } (\text{ConditionalExpr c t f}) \ t \mid \end{aligned}$$

*const-false:*

$$\begin{aligned} & \llbracket c = \text{ConstantExpr val}; \\ & \quad \neg(\text{val-to-bool val}) \rrbracket \\ \implies & \text{CanonicalizeConditional } (\text{ConditionalExpr c t f}) \ f \end{aligned}$$

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

*BinaryNode:*

$$\begin{aligned} & \llbracket \text{CanonicalizeBinaryOp expr expr'} \rrbracket \\ \implies & \text{CanonicalizationStep expr expr'} \mid \end{aligned}$$

*UnaryNode:*

$$\begin{aligned} & \llbracket \text{CanonicalizeUnaryOp expr expr'} \rrbracket \\ \implies & \text{CanonicalizationStep expr expr'} \mid \end{aligned}$$

*NegateNode:*

$$\begin{aligned} & \llbracket \text{CanonicalizeNegate expr expr'} \rrbracket \\ \implies & \text{CanonicalizationStep expr expr'} \mid \end{aligned}$$

*NotNode:*

$$\begin{aligned} & \llbracket \text{CanonicalizeNegate expr expr'} \rrbracket \\ \implies & \text{CanonicalizationStep expr expr'} \mid \end{aligned}$$

*AddNode:*

$$\begin{aligned} & \llbracket \text{CanonicalizeAdd expr expr'} \rrbracket \\ \implies & \text{CanonicalizationStep expr expr'} \mid \end{aligned}$$

*MulNode:*

$$\begin{aligned} & \llbracket \text{CanonicalizeMul expr expr'} \rrbracket \\ \implies & \text{CanonicalizationStep expr expr'} \mid \end{aligned}$$

```

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 <proof>
code-pred (modes: i => o => bool) CanonicalizeUnaryOp <proof>
code-pred (modes: i => o => bool) CanonicalizeNegate <proof>
code-pred (modes: i => o => bool) CanonicalizeNot <proof>
code-pred (modes: i => o => bool) CanonicalizeAdd <proof>
code-pred (modes: i => o => bool) CanonicalizeSub <proof>
code-pred (modes: i => o => bool) CanonicalizeMul <proof>
code-pred (modes: i => o => bool) CanonicalizeAnd <proof>
code-pred (modes: i => o => bool) CanonicalizeIntegerEquals <proof>
code-pred (modes: i => o => bool) CanonicalizeConditional <proof>

code-pred (modes: i => o => bool) CanonicalizationStep <proof>

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 ==> intval-mul x (IntVal32 (1)) = x
  and valid-value (IntegerStamp 64 lo hi) x ==> intval-mul x (IntVal64 (1)) = x

  and valid-value (IntegerStamp 32 lo hi) x ==> intval-add x (IntVal32 (0)) = x

```

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

**lemma** *annihilator-rewrite-helper*:

**shows**  $\text{valid-value } (\text{IntegerStamp } 32 \text{ lo hi}) \ x \implies \text{intval-mul } x \ (\text{IntVal32 } 0) = \text{IntVal32 } 0$   
**and**  $\text{valid-value } (\text{IntegerStamp } 64 \text{ lo hi}) \ x \implies \text{intval-mul } x \ (\text{IntVal64 } 0) = \text{IntVal64 } 0$   
**and**  $\text{valid-value } (\text{IntegerStamp } 32 \text{ lo hi}) \ x \implies \text{intval-and } x \ (\text{IntVal32 } 0) = \text{IntVal32 } 0$   
**and**  $\text{valid-value } (\text{IntegerStamp } 64 \text{ lo hi}) \ x \implies \text{intval-and } x \ (\text{IntVal64 } 0) = \text{IntVal64 } 0$   
**and**  $\text{valid-value } (\text{IntegerStamp } 32 \text{ lo hi}) \ x \implies \text{intval-or } x \ (\text{IntVal32 } (-1)) = \text{IntVal32 } (-1)$   
**and**  $\text{valid-value } (\text{IntegerStamp } 64 \text{ lo hi}) \ x \implies \text{intval-or } x \ (\text{IntVal64 } (-1)) = \text{IntVal64 } (-1)$   
 $\langle \text{proof} \rangle$

**lemma** *idempotent-rewrite-helper*:

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

**value** *size* ( $v::32 \text{ word}$ )

**lemma** *signed-int-bottom32*:  $-(((2::\text{int}) \wedge 31)) \leq \text{sint } (v::\text{int32})$   
 $\langle \text{proof} \rangle$

**lemma** *signed-int-top32*:  $(2 \wedge 31) - 1 \geq \text{sint } (v::\text{int32})$   
 $\langle \text{proof} \rangle$

**lemma** *lower-bounds-equiv32*:  $-(((2::\text{int}) \wedge 31)) = (2::\text{int}) \wedge 32 \text{ div } 2 * -1$   
 $\langle \text{proof} \rangle$

**lemma** *upper-bounds-equiv32*:  $(2::\text{int}) \wedge 31 = (2::\text{int}) \wedge 32 \text{ div } 2$

$\langle \text{proof} \rangle$

**lemma** *bit-bounds-min32*:  $((\text{fst } (\text{bit-bounds } 32))) \leq (\text{sint } (v::\text{int32}))$   
 $\langle \text{proof} \rangle$

**lemma** *bit-bounds-max32*:  $((\text{snd } (\text{bit-bounds } 32))) \geq (\text{sint } (v::\text{int32}))$   
 $\langle \text{proof} \rangle$

**value** *size*  $(v::64 \text{ word})$

**lemma** *signed-int-bottom64*:  $-(((2::\text{int}) \wedge 63)) \leq \text{sint } (v::\text{int64})$   
 $\langle \text{proof} \rangle$

**lemma** *signed-int-top64*:  $(2 \wedge 63) - 1 \geq \text{sint } (v::\text{int64})$   
 $\langle \text{proof} \rangle$

**lemma** *lower-bounds-equiv64*:  $-(((2::\text{int}) \wedge 63)) = (2::\text{int}) \wedge 64 \text{ div } 2 * - 1$   
 $\langle \text{proof} \rangle$

**lemma** *upper-bounds-equiv64*:  $(2::\text{int}) \wedge 63 = (2::\text{int}) \wedge 64 \text{ div } 2$   
 $\langle \text{proof} \rangle$

**lemma** *bit-bounds-min64*:  $((\text{fst } (\text{bit-bounds } 64))) \leq (\text{sint } (v::\text{int64}))$   
 $\langle \text{proof} \rangle$

**lemma** *bit-bounds-max64*:  $((\text{snd } (\text{bit-bounds } 64))) \geq (\text{sint } (v::\text{int64}))$   
 $\langle \text{proof} \rangle$

**lemma** *unrestricted-32bit-always-valid*:  
*valid-value*  $(\text{unrestricted-stamp } (\text{IntegerStamp } 32 \text{ lo hi})) (\text{IntVal32 } v)$   
 $\langle \text{proof} \rangle$

**lemma** *unrestricted-64bit-always-valid*:  
*valid-value*  $(\text{unrestricted-stamp } (\text{IntegerStamp } 64 \text{ lo hi})) (\text{IntVal64 } v)$   
 $\langle \text{proof} \rangle$

**lemma** *unary-undef*:  $\text{val} = \text{UndefVal} \implies \text{unary-eval op val} = \text{UndefVal}$   
 $\langle \text{proof} \rangle$

**lemma** *unary-obj*:  $\text{val} = \text{ObjRef } x \implies \text{unary-eval op val} = \text{UndefVal}$   
 $\langle \text{proof} \rangle$

**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*  $(\text{stamp-expr expr}) \text{ val}$   
**shows** *valid-value*  $(\text{stamp-expr } (\text{UnaryExpr op expr})) \text{ result}$   
 $\langle \text{proof} \rangle$

**lemma** *binary-undef*:  $v1 = \text{UndefVal} \vee v2 = \text{UndefVal} \implies \text{bin-eval op } v1 \ v2 = \text{UndefVal}$   
 ⟨proof⟩

**lemma** *binary-obj*:  $v1 = \text{ObjRef } x \vee v2 = \text{ObjRef } y \implies \text{bin-eval op } v1 \ v2 = \text{UndefVal}$   
 ⟨proof⟩

**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  $b1 = b2$   
 ⟨proof⟩

**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  $\exists x32 \ x64 \ y32 \ y64. \text{val1} = \text{IntVal32 } x32 \wedge \text{val2} = \text{IntVal32 } y32 \vee \text{val1} = \text{IntVal64 } x64 \wedge \text{val2} = \text{IntVal64 } y64$   
 ⟨proof⟩

**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⟩

**lemma** *stamp-meet-is-valid*:  
 assumes  $\text{valid-value stamp1 } val \vee \text{valid-value stamp2 } val$   
 assumes  $\text{meet stamp1 stamp2} \neq \text{IllegalStamp}$   
 shows  $\text{valid-value } (\text{meet stamp1 stamp2}) \ val$   
 ⟨proof⟩

**lemma** *conditional-eval-implies-valud-value*:  
 assumes  $[m, p] \vdash \text{cond} \mapsto \text{condv}$   
 assumes  $\text{expr} = (\text{if } \text{IRTreeEval.val-to-bool condv} \text{ then expr1 else expr2})$   
 assumes  $[m, p] \vdash \text{expr} \mapsto \text{val}$   
 assumes  $\text{val} \neq \text{UndefVal}$   
 assumes  $\text{valid-value } (\text{stamp-expr cond}) \ \text{condv}$   
 assumes  $\text{valid-value } (\text{stamp-expr expr}) \ val$   
 shows  $\text{valid-value } (\text{stamp-expr } (\text{ConditionalExpr cond expr1 expr2})) \ val$

$\langle proof \rangle$

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

**assumes**  $[m, p] \vdash expr \mapsto val$   
**shows** *valid-value* (*stamp-expr* *expr*) *val*  
 $\langle proof \rangle$

**lemma** *CanonicalizeBinaryProof*:

**assumes** *CanonicalizeBinaryOp before after*  
**assumes**  $[m, p] \vdash before \mapsto res$   
**assumes**  $[m, p] \vdash after \mapsto res'$   
**shows**  $res = res'$   
 $\langle proof \rangle$

**lemma** *CanonicalizeUnaryProof*:

**assumes** *CanonicalizeUnaryOp before after*  
**assumes**  $[m, p] \vdash before \mapsto res$   
**assumes**  $[m, p] \vdash after \mapsto res'$   
**shows**  $res = res'$   
 $\langle proof \rangle$

**lemma** *mul-rewrite-helper*:

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

**lemma** *CanonicalizeMulProof*:

**assumes** *CanonicalizeMul before after*  
**assumes**  $[m, p] \vdash before \mapsto res$   
**assumes**  $[m, p] \vdash after \mapsto res'$   
**shows**  $res = res'$   
 $\langle proof \rangle$

**lemma** *add-rewrites-helper*:

**assumes** *valid-value* (*IntegerStamp* *b lox hix*)  $x$   
**and** *valid-value* (*IntegerStamp* *b 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**  $\text{intval-add } x \ (\text{intval-negate } y) = \text{intval-sub } x \ y$   
 $\langle proof \rangle$



**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**  $\text{res} = \text{res}'$   
 $\langle \text{proof} \rangle$

**lemma** *sub-rewrites-helper*:  
**assumes** *valid-value (IntegerStamp b lox hix) x*  
**and** *valid-value (IntegerStamp b loy hiy) y*  
  
**shows**  $\text{intval-sub } (\text{intval-add } x \ y) \ y = x$   
**and**  $\text{intval-sub } (\text{intval-add } x \ y) \ x = y$   
**and**  $\text{intval-sub } (\text{intval-sub } x \ y) \ x = \text{intval-negate } y$   
**and**  $\text{intval-sub } x \ (\text{intval-add } x \ y) = \text{intval-negate } y$   
**and**  $\text{intval-sub } y \ (\text{intval-add } x \ y) = \text{intval-negate } x$   
**and**  $\text{intval-sub } x \ (\text{intval-sub } x \ y) = y$   
**and**  $\text{intval-sub } x \ (\text{intval-negate } y) = \text{intval-add } x \ y$   
 $\langle \text{proof} \rangle$

**lemma** *sub-single-rewrites-helper*:  
**assumes** *valid-value (IntegerStamp b lox hix) x*  
**shows**  $b = 32 \implies \text{intval-sub } x \ x = \text{IntVal32 } 0$   
**and**  $b = 64 \implies \text{intval-sub } x \ x = \text{IntVal64 } 0$   
**and**  $b = 32 \implies \text{intval-sub } (\text{IntVal32 } 0) \ x = \text{intval-negate } x$   
**and**  $b = 64 \implies \text{intval-sub } (\text{IntVal64 } 0) \ x = \text{intval-negate } x$   
 $\langle \text{proof} \rangle$

**lemma** *CanonicalizeSubProof*:  
**assumes** *CanonicalizeSub before after*  
**assumes**  $[m, p] \vdash \text{before} \mapsto \text{res}$   
**assumes**  $[m, p] \vdash \text{after} \mapsto \text{res}'$   
**shows**  $\text{res} = \text{res}'$   
 $\langle \text{proof} \rangle$

**lemma** *negate-xsuby-helper*:  
**assumes** *valid-value (IntegerStamp b lox hix) x*  
**and** *valid-value (IntegerStamp b loy hiy) y*  
**shows**  $\text{intval-negate } (\text{intval-sub } x \ y) = \text{intval-sub } y \ x$

$\langle \text{proof} \rangle$

**lemma** *negate-negate-helper*:

**assumes** *valid-value* (*IntegerStamp* *b lox hix*) *x*

**shows** *intval-negate* (*intval-negate* *x*) = *x*

$\langle \text{proof} \rangle$

**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** *res* = *res'*

$\langle \text{proof} \rangle$

**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**  $\bigwedge x :: 32 \text{ word. } \neg - x < s \ 0 \wedge \neg x < s \ 0 \implies 2 * x = 0$

**and**  $\bigwedge x :: 64 \text{ word. } \neg - x < s \ 0 \wedge \neg x < s \ 0 \implies 2 * x = 0$

$\langle \text{proof} \rangle$

**lemma** *abs-abs-is-abs*:

**assumes** *valid-value* (*IntegerStamp* *b lox hix*) *x*

**shows** *intval-abs* (*intval-abs* *x*) = *intval-abs* *x*

$\langle \text{proof} \rangle$

**lemma** *abs-neg-is-neg*:

**assumes** *valid-value* (*IntegerStamp* *b lox hix*) *x*

**shows** *intval-abs* (*intval-negate* *x*) = *intval-abs* *x*

$\langle \text{proof} \rangle$

**lemma** *not-rewrite-helper*:

**assumes** *valid-value* (*IntegerStamp* *b lox hix*) *x*

**shows** *intval-not* (*intval-not* *x*) = *x*

$\langle \text{proof} \rangle$

**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** *res* = *res'*

$\langle \text{proof} \rangle$

**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*  $\implies$  *intval-and* *x y* = *x*  
**and** *x* = *y*  $\implies$  *intval-or* *x y* = *x*  
 $\langle$ *proof* $\rangle$

**lemma** *CanonicalizeAndProof*:  
**assumes** *CanonicalizeAnd* *before after*  
**assumes**  $[m, p] \vdash \textit{before} \mapsto \textit{res}$   
**assumes**  $[m, p] \vdash \textit{after} \mapsto \textit{res}'$   
**shows** *res* = *res'*  
 $\langle$ *proof* $\rangle$

**lemma** *CanonicalizeOrProof*:  
**assumes** *CanonicalizeOr* *before after*  
**assumes**  $[m, p] \vdash \textit{before} \mapsto \textit{res}$   
**assumes**  $[m, p] \vdash \textit{after} \mapsto \textit{res}'$   
**shows** *res* = *res'*  
 $\langle$ *proof* $\rangle$

**lemma** *stamps-touch-but-not-less-than-implies-equal*:  
 $\llbracket \textit{valid-value stampx } x;$   
 $\textit{valid-value stampy } y;$   
 $\textit{is-IntegerStamp stampx} \wedge \textit{is-IntegerStamp stampy};$   
 $\textit{stpi-upper stampx} = \textit{stpi-lower stampy};$   
 $\neg \textit{val-to-bool} (\textit{intval-less-than } x y) \rrbracket \implies x = y$   
 $\langle$ *proof* $\rangle$

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

**lemma** *CanonicalizeConditionalProof*:  
**assumes** *CanonicalizeConditional* *before after*  
**assumes**  $[m, p] \vdash \textit{before} \mapsto \textit{res}$   
**assumes**  $[m, p] \vdash \textit{after} \mapsto \textit{res}'$   
**shows** *res* = *res'*  
 $\langle$ *proof* $\rangle$

**end**