

Veriopt

July 23, 2021

Abstract

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

Contents

1	Runtime Values and Arithmetic	3
2	Nodes	8
2.1	Types of Nodes	8
2.2	Hierarchy of Nodes	15
3	Stamp Typing	22
4	Graph Representation	25
4.0.1	Example Graphs	29
5	Data-flow Semantics	29
5.1	Data-flow Tree Representation	31
5.2	Data-flow Tree Evaluation	39
5.3	Data-flow Tree Refinement	41
6	Data-flow Expression-Tree Theorems	41
6.1	Extraction and Evaluation of Expression Trees is Deterministic.	42
6.2	Example Data-flow Optimisations	46
6.3	Monotonicity of Expression Optimization	46
7	Control-flow Semantics	47
7.1	Heap	47
7.2	Intraprocedural Semantics	48
7.3	Interprocedural Semantics	50
7.4	Big-step Execution	51
7.4.1	Heap Testing	52
8	Canonicalization Phase	53
9	Canonicalization Phase	64

1 Runtime Values and Arithmetic

```

theory Values2
  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 |
  FloatVal float |
  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)
```

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  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-sub (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1-v2)) |
  intval-sub (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1-v2)) |
  intval-sub - - = UndefVal

```

```

instantiation Value :: minus
begin

```

```

definition minus-Value :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  minus-Value = intval-sub

```

```

instance <proof>
end

```

```

fun intval-mul :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  intval-mul (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1*v2)) |
  intval-mul (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1*v2)) |
  intval-mul - - = UndefVal

```

```

instantiation Value :: times
begin

```

```

definition times-Value :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  times-Value = intval-mul

```

```

instance <proof>
end

```

```

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

```

```

instantiation Value :: divide
begin

```

```

definition divide-Value :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where
  divide-Value = intval-div

```

```

instance <proof>
end

```

```

fun intval-mod :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value where

```

```

    intval-mod (IntVal32 v1) (IntVal32 v2) = (IntVal32 (word-of-int((sint v1) smod
(sint v2)))) |
    intval-mod (IntVal64 v1) (IntVal64 v2) = (IntVal64 (word-of-int((sint v1) smod
(sint v2)))) |
    intval-mod - - = UndefVal

```

instantiation *Value* :: *modulo*
begin

definition *modulo-Value* :: *Value* \Rightarrow *Value* \Rightarrow *Value* **where**
modulo-Value = *intval-mod*

instance \langle *proof* \rangle
end

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

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

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

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

fun *intval-less-than* :: *Value* \Rightarrow *Value* \Rightarrow *Value* **where**
intval-less-than (IntVal32 v1) (IntVal32 v2) = *bool-to-val* (v1 <_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 ⇒ Value where
  intval-negate (IntVal32 v) = IntVal32 (- v) |
  intval-negate (IntVal64 v) = IntVal64 (- v) |
  intval-negate - = UndefVal

```

```

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

```

```

lemma word-add-sym:
  shows word-of-int v1 + word-of-int v2 = word-of-int v2 + word-of-int v1
  ⟨proof⟩

```

```

lemma intval-add-sym:
  shows intval-add a b = intval-add b a
  ⟨proof⟩

```

```

lemma word-add-assoc:
  shows (word-of-int v1 + word-of-int v2) + word-of-int v3
    = word-of-int v1 + (word-of-int v2 + word-of-int v3)
  ⟨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 IRNodes2
  imports
    Values2
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)
 | *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)
 | *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)

```

| ShortCircuitOrNode (ir-x: INPUT-COND) (ir-y: INPUT-COND)
| SignedDivNode (ir-nid: ID) (ir-x: INPUT) (ir-y: INPUT) (ir-zeroCheck-opt: IN-
PUT-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)
| 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)
| NoNode

| RefNode (ir-ref:ID)

```

```

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

```

```

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

```

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

```

fun inputs-of :: IRNode ⇒ ID list where
  inputs-of-AbsNode:
  inputs-of (AbsNode value) = [value] |
  inputs-of-AddNode:
  inputs-of (AddNode x y) = [x, y] |
  inputs-of-AndNode:
  inputs-of (AndNode x y) = [x, y] |
  inputs-of-BEGINNode:
  inputs-of (BeginNode next) = [] |
  inputs-of-BytecodeExceptionNode:
  inputs-of (BytecodeExceptionNode arguments stateAfter next) = arguments @
  (opt-to-list stateAfter) |
  inputs-of-ConditionalNode:
  inputs-of (ConditionalNode condition trueValue falseValue) = [condition, true-
Value, falseValue] |

```

```

inputs-of-ConstantNode:
inputs-of (ConstantNode const) = [] |
inputs-of-DynamicNewArrayNode:
inputs-of (DynamicNewArrayNode elementType length0 voidClass stateBefore
next) = [elementType, length0] @ (opt-to-list voidClass) @ (opt-to-list stateBefore)
|
inputs-of-EndNode:
inputs-of (EndNode) = [] |
inputs-of-ExceptionObjectNode:
inputs-of (ExceptionObjectNode stateAfter next) = (opt-to-list stateAfter) |
inputs-of-FrameState:
inputs-of (FrameState monitorIds outerFrameState values virtualObjectMappings)
= monitorIds @ (opt-to-list outerFrameState) @ (opt-list-to-list values) @ (opt-list-to-list
virtualObjectMappings) |
inputs-of-IfNode:
inputs-of (IfNode condition trueSuccessor falseSuccessor) = [condition] |
inputs-of-IntegerBelowNode:
inputs-of (IntegerBelowNode x y) = [x, y] |
inputs-of-IntegerEqualsNode:
inputs-of (IntegerEqualsNode x y) = [x, y] |
inputs-of-IntegerLessThanNode:
inputs-of (IntegerLessThanNode x y) = [x, y] |
inputs-of-InvokeNode:
inputs-of (InvokeNode nid0 callTarget classInit stateDuring stateAfter next)
= callTarget # (opt-to-list classInit) @ (opt-to-list stateDuring) @ (opt-to-list
stateAfter) |
inputs-of-InvokeWithExceptionNode:
inputs-of (InvokeWithExceptionNode nid0 callTarget classInit stateDuring stateAfter
next exceptionEdge) = callTarget # (opt-to-list classInit) @ (opt-to-list stateDur-
ing) @ (opt-to-list stateAfter) |
inputs-of-IsNullNode:
inputs-of (IsNullNode value) = [value] |
inputs-of-KillingBeginNode:
inputs-of (KillingBeginNode next) = [] |
inputs-of-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-NegateNode:
inputs-of (*NegateNode* *value*) = [*value*] |
inputs-of-NewArrayNode:
inputs-of (*NewArrayNode* *length0* *stateBefore* *next*) = *length0* # (*opt-to-list* *stateBefore*) |
inputs-of-NewInstanceNode:
inputs-of (*NewInstanceNode* *nid0* *instanceClass* *stateBefore* *next*) = (*opt-to-list* *stateBefore*) |
inputs-of-NotNode:
inputs-of (*NotNode* *value*) = [*value*] |
inputs-of-OrNode:
inputs-of (*OrNode* *x* *y*) = [*x*, *y*] |
inputs-of-ParameterNode:
inputs-of (*ParameterNode* *index*) = [] |
inputs-of-PiNode:
inputs-of (*PiNode* *object* *guard*) = *object* # (*opt-to-list* *guard*) |
inputs-of-ReturnNode:
inputs-of (*ReturnNode* *result* *memoryMap*) = (*opt-to-list* *result*) @ (*opt-to-list* *memoryMap*) |
inputs-of-ShortCircuitOrNode:
inputs-of (*ShortCircuitOrNode* *x* *y*) = [*x*, *y*] |
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-UnwindNode:
inputs-of (*UnwindNode* *exception*) = [*exception*] |
inputs-of-ValuePhiNode:
inputs-of (*ValuePhiNode* *nid* *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-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-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-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-ShortCircuitOrNode:
successors-of (*ShortCircuitOrNode* *x y*) = [] |
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-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-NoNode: successors-of (NoNode) = [] |

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

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

end

2.2 Hierarchy of Nodes

theory *IRNodeHierarchy*
imports *IRNodes2*
begin

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

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

These functions have been automatically generated from the compiler.

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

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

fun *is-AbstractMergeNode :: IRNode ⇒ bool where*
is-AbstractMergeNode n = ((is-LoopBeginNode n) ∨ (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-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-AccessFieldNode :: IRNode  $\Rightarrow$  bool where
  is-AccessFieldNode n = ((is-LoadFieldNode n)  $\vee$  (is-StoreFieldNode 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-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-FloatingGuardedNode :: IRNode  $\Rightarrow$  bool where
  is-FloatingGuardedNode n = ((is-PiNode 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-UnaryArithmeticNode 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-BinaryNode :: IRNode  $\Rightarrow$  bool where
  is-BinaryNode n = ((is-BinaryArithmeticNode n))

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

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

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

```

```

n) ∨ (is-ConstantNode n) ∨ (is-FloatingGuardedNode n) ∨ (is-LogicNode n) ∨
(is-PhiNode n) ∨ (is-ProxyNode n) ∨ (is-UnaryNode n))

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

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

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

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

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

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

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

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

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

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

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

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

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

fun is-ValueNodeInterface :: IRNode ⇒ bool where

```

```

is-ValueNodeInterface n = ((is-ValueNode n))

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

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

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

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

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

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

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

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

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

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

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

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

```

```

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

fun is-UncheckedInterfaceProvider :: IRNode ⇒ bool where
  is-UncheckedInterfaceProvider n = ((is-InvokeNode n) ∨ (is-InvokeWithExceptionNode
    n) ∨ (is-LoadFieldNode n) ∨ (is-ParameterNode n))

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

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

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

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

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

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

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

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

```

The following convenience function is useful in determining if two *IRNodes*

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

fun *is-same-ir-node-type* :: *IRNode* \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)  $\wedge$  (is-PiNode n2))  $\vee$ 
  ((is-ReturnNode n1)  $\wedge$  (is-ReturnNode n2))  $\vee$ 
  ((is-ShortCircuitOrNode n1)  $\wedge$  (is-ShortCircuitOrNode n2))  $\vee$ 
  ((is-SignedDivNode n1)  $\wedge$  (is-SignedDivNode n2))  $\vee$ 
  ((is-StartNode n1)  $\wedge$  (is-StartNode n2))  $\vee$ 
  ((is-StoreFieldNode n1)  $\wedge$  (is-StoreFieldNode n2))  $\vee$ 
  ((is-SubNode n1)  $\wedge$  (is-SubNode n2))  $\vee$ 
  ((is-UnwindNode n1)  $\wedge$  (is-UnwindNode n2))  $\vee$ 
  ((is-ValuePhiNode n1)  $\wedge$  (is-ValuePhiNode n2))  $\vee$ 
  ((is-ValueProxyNode n1)  $\wedge$  (is-ValueProxyNode n2))  $\vee$ 
  ((is-XorNode n1)  $\wedge$  (is-XorNode n2)))
```

end

3 Stamp Typing

```
theory Stamp2
  imports Values2
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  $\Rightarrow$  (int  $\times$  int) where
  bit-bounds bits = (((2  $\wedge$  bits) div 2) * -1, ((2  $\wedge$  bits) div 2) - 1)
```

— A stamp which includes the full range of the type

```
fun unrestricted-stamp :: Stamp  $\Rightarrow$  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* \Rightarrow *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* \Rightarrow *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* *type* *exactType* *nonNull* *alwaysNull*) |
empty-stamp *stamp* = *IllegalStamp*

fun *is-stamp-empty* :: *Stamp* \Rightarrow *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* \Rightarrow *Stamp* \Rightarrow *Stamp* **where**
meet *VoidStamp* *VoidStamp* = *VoidStamp* |
meet (*IntegerStamp* *b1* *l1* *u1*) (*IntegerStamp* *b2* *l2* *u2*) = (
 if *b1* \neq *b2* *then* *IllegalStamp* *else*
 (*IntegerStamp* *b1* (*min* *l1* *l2*) (*max* *u1* *u2*))
) |

meet (*KlassPointerStamp* *nn1* *an1*) (*KlassPointerStamp* *nn2* *an2*) = (
 KlassPointerStamp (*nn1* \wedge *nn2*) (*an1* \wedge *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* (*min* *l1* *l2*) (*max* *u1* *u2*))
)

```

    if b1 ≠ b2 then IllegalStamp else
    (IntegerStamp b1 (max l1 l2) (min u1 u2))
  ) |

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

```

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

```

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

```

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

```

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

```

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

```

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

```

```

fun constantAsStamp :: Value ⇒ Stamp where

```

```

  constantAsStamp (IntVal32 v) = (IntegerStamp (nat 32) (sint v) (sint v)) |
  constantAsStamp (IntVal64 v) = (IntegerStamp (nat 64) (sint v) (sint v)) |

```

```

  constantAsStamp - = IllegalStamp

```

— Define when a runtime value is valid for a stamp

```

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

```



```

h)) |
  valid-value (IntegerStamp b l h) (IntVal64 v) = (b=64 ∧ (sint v ≥ l) ∧ (sint v ≤
h)) |

  valid-value (VoidStamp) (UndefVal) = True |
  valid-value (ObjectStamp klass exact nonNull alwaysNull) (ObjRef ref) =
    (if nonNull then ref≠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
    Stamp2
    HOL-Library.FSet
    HOL.Relation
  begin

```

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

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

```

typedef IRGraph = {g :: ID → (IRNode × Stamp) . finite (dom g)}
  <proof>

```

```

setup-lifting type-definition-IRGraph

```

```

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

```

```

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

```

```

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

```

```

lift-definition stamp :: IRGraph ⇒ ID ⇒ Stamp
  is with-default IllegalStamp snd <proof>

```

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

lift-definition *remove-node* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph$
is $\lambda nid\ g.$ $g(nid := None)$ *<proof>*

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

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

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

lift-definition *irgraph* :: $(ID \times (IRNode \times Stamp))\ list \Rightarrow IRGraph$
is *map-of* $\circ no-node$
<proof>

code-datatype *irgraph*

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

lemma *no-node-clears*:
 $res = no-node\ xs \longrightarrow (\forall x \in set\ res. fst\ (snd\ x) \neq NoNode)$
<proof>

lemma *dom-eq*:
assumes $\forall x \in set\ xs. fst\ (snd\ x) \neq NoNode$
shows *filter-none* (*map-of* xs) = *dom* (*map-of* xs)
<proof>

lemma *fil-eq*:
filter-none (*map-of* (*no-node* xs)) = *set* (*map* *fst* (*no-node* xs))
<proof>

lemma *irgraph[code]*: *ids* (*irgraph* m) = *set* (*map* *fst* (*no-node* m))
<proof>

lemma *[code]*: *Rep-IRGraph* (*irgraph* m) = *map-of* (*no-node* m)
<proof>

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**

$\text{succ } g \text{ nid} = \text{set } (\text{successors-of } (\text{kind } g \text{ nid}))$
 — Gives a relation between node IDs - between a node and its input nodes
fun *input-edges* :: *IRGraph* \Rightarrow *ID rel* **where**
 input-edges *g* = $(\bigcup i \in \text{ids } g. \{(i,j) | j. j \in (\text{inputs } g \ i)\})$
 — 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* = $\{j. j \in \text{ids } g \wedge (j, \text{nid}) \in \text{input-edges } g\}$
fun *successor-edges* :: *IRGraph* \Rightarrow *ID rel* **where**
 successor-edges *g* = $(\bigcup i \in \text{ids } g. \{(i,j) | j. j \in (\text{succ } g \ i)\})$
fun *predecessors* :: *IRGraph* \Rightarrow *ID* \Rightarrow *ID set* **where**
 predecessors *g* *nid* = $\{j. j \in \text{ids } g \wedge (j, \text{nid}) \in \text{successor-edges } g\}$
fun *nodes-of* :: *IRGraph* \Rightarrow (*IRNode* \Rightarrow *bool*) \Rightarrow *ID set* **where**
 nodes-of *g* *sel* = $\{\text{nid} \in \text{ids } g. \text{sel } (\text{kind } g \text{ nid})\}$
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* = $\{n \in (\text{usages } g \text{ nid}). f (\text{kind } g \ n)\}$

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

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

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

lemma [simp]: $\text{finite } (\text{ids } g)$
 <proof>

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

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

$\langle \text{proof} \rangle$

lemma $[simp]$: $\text{finite } (\text{dom } g) \longrightarrow \text{kind } (\text{Abs-IRGraph } g) = (\lambda x . (\text{case } g \ x \text{ of } \text{None} \Rightarrow \text{NoNode} \mid \text{Some } n \Rightarrow \text{fst } n))$
 $\langle \text{proof} \rangle$

lemma $[simp]$: $\text{finite } (\text{dom } g) \longrightarrow \text{stamp } (\text{Abs-IRGraph } g) = (\lambda x . (\text{case } g \ x \text{ of } \text{None} \Rightarrow \text{IllegalStamp} \mid \text{Some } n \Rightarrow \text{snd } n))$
 $\langle \text{proof} \rangle$

lemma $[simp]$: $\text{ids } (\text{irgraph } g) = \text{set } (\text{map } \text{fst } (\text{no-node } g))$
 $\langle \text{proof} \rangle$

lemma $[simp]$: $\text{kind } (\text{irgraph } g) = (\lambda \text{nid} . (\text{case } (\text{map-of } (\text{no-node } g)) \ \text{nid} \text{ of } \text{None} \Rightarrow \text{NoNode} \mid \text{Some } n \Rightarrow \text{fst } n))$
 $\langle \text{proof} \rangle$

lemma $[simp]$: $\text{stamp } (\text{irgraph } g) = (\lambda \text{nid} . (\text{case } (\text{map-of } (\text{no-node } g)) \ \text{nid} \text{ of } \text{None} \Rightarrow \text{IllegalStamp} \mid \text{Some } n \Rightarrow \text{snd } n))$
 $\langle \text{proof} \rangle$

lemma map-of-upd : $(\text{map-of } g)(k \mapsto v) = (\text{map-of } ((k, v) \# g))$
 $\langle \text{proof} \rangle$

lemma $[\text{code}]$: $\text{replace-node } \text{nid } k \ (\text{irgraph } g) = (\text{irgraph } ((\text{nid}, k) \# g))$
 $\langle \text{proof} \rangle$

lemma $[\text{code}]$: $\text{add-node } \text{nid } k \ (\text{irgraph } g) = (\text{irgraph } (((\text{nid}, k) \# g)))$
 $\langle \text{proof} \rangle$

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

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

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

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

$gup \wedge kind\ g\ n = kind\ gup\ n$
 $\langle proof \rangle$

4.0.1 Example Graphs

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

definition *start-end-graph* :: *IRGraph* **where**

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

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

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

definition *eg2-sq* :: *IRGraph* **where**

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

value *input-edges* *eg2-sq*

value *usages* *eg2-sq* 1

end

5 Data-flow Semantics

theory *IRTreeEval*

imports

Graph.IRGraph

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 *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*)

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* \Rightarrow *ID* \Rightarrow *ID list* **where**
phi-list g nid =
 (*filter* ($\lambda x.$ (*is-PhiNode* (*kind g x*))))
 (*sorted-list-of-set* (*usages g nid*)))

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

fun *phi-inputs* :: *IRGraph* \Rightarrow *nat* \Rightarrow *ID list* \Rightarrow *ID list* **where**
phi-inputs g i nodes = (*map* ($\lambda n.$ (*inputs-of (kind g n)*))!(*i + 1*)) *nodes*)

fun *set-phis* :: *ID list* \Rightarrow *Value list* \Rightarrow *MapState* \Rightarrow *MapState* **where**
set-phis [] [] *m* = *m* |
set-phis (nid # xs) (v # vs) m = (*set-phis xs vs (m(nid := v))*) |
set-phis [] (*v # vs*) *m* = *m* |
set-phis (x # xs) [] m = *m*

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*

5.1 Data-flow Tree Representation

datatype *IRUnaryOp* =
 UnaryAbs
 | *UnaryNeg*
 | *UnaryNot*
 | *UnaryLogicNegation*

datatype *IRBinaryOp* =
 BinAdd
 | *BinMul*
 | *BinSub*
 | *BinAnd*
 | *BinOr*
 | *BinXor*
 | *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)
 | *ConstantExpr* (*ir-const*: *Value*)

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

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

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

inductive

$rep :: IRGraph \Rightarrow ID \Rightarrow IRExpr \Rightarrow bool \ (- \vdash - \triangleright - \ 55)$

for g where

ConstantNode:

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

ParameterNode:

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

ConditionalNode:

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

AbsNode:

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

NotNode:

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

NegateNode:

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

LogicNegationNode:

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

AddNode:

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

MulNode:

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

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

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

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

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

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

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

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

LeafNode:
 $\llbracket \text{is-preevaluated } (\text{kind } g \ n);$

$stamp\ g\ n = s$
 $\implies g \vdash n \triangleright (LeafExpr\ n\ s)$

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

inductive

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

RepNil:
 $g \vdash [] \triangleright_L []$

RepCons:
 $\llbracket g \vdash x \triangleright xe;$
 $g \vdash xs \triangleright_L xse \rrbracket$
 $\implies g \vdash x \# xs \triangleright_L xe \# xse$

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

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

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

$$\frac{kind\ g\ n = AbsNode\ x \quad g \vdash x \triangleright xe}{g \vdash n \triangleright UnaryExpr\ UnaryAbs\ xe}$$

$$\frac{kind\ g\ n = AddNode\ x\ y \quad g \vdash x \triangleright xe \quad g \vdash y \triangleright ye}{g \vdash n \triangleright BinaryExpr\ BinAdd\ xe\ ye}$$

$$\frac{kind\ g\ n = MulNode\ x\ y \quad g \vdash x \triangleright xe \quad g \vdash y \triangleright ye}{g \vdash n \triangleright BinaryExpr\ BinMul\ xe\ ye}$$

$$\frac{kind\ g\ n = SubNode\ x\ y \quad g \vdash x \triangleright xe \quad g \vdash y \triangleright ye}{g \vdash n \triangleright BinaryExpr\ BinSub\ xe\ ye}$$

$$\frac{is-preevaluated\ (kind\ g\ n) \quad stamp\ g\ n = s}{g \vdash n \triangleright LeafExpr\ n\ s}$$

values {*t*. *eg2-sq* $\vdash 4 \triangleright t$ }

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

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

stamp-binary op - - = IllegalStamp

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

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

fun unary-node :: IRUnaryOp ⇒ ID ⇒ IRNode where
  unary-node UnaryAbs v = AbsNode v |
  unary-node UnaryNot v = NotNode v |
  unary-node UnaryNeg v = NegateNode v |
  unary-node UnaryLogicNegation v = LogicNegationNode v

fun bin-node :: IRBinaryOp ⇒ ID ⇒ ID ⇒ IRNode where
  bin-node BinAdd x y = AddNode x y |
  bin-node BinMul x y = MulNode x y |
  bin-node BinSub x y = SubNode x y |
  bin-node BinAnd x y = AndNode x y |
  bin-node BinOr x y = OrNode x y |
  bin-node BinXor x y = XorNode x y |
  bin-node BinIntegerEquals x y = IntegerEqualsNode x y |
  bin-node BinIntegerLessThan x y = IntegerLessThanNode x y |
  bin-node BinIntegerBelow x y = IntegerBelowNode x y

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

```

```

fun bin-eval :: IRBinaryOp ⇒ Value ⇒ Value ⇒ Value where
  bin-eval BinAdd v1 v2 = intval-add v1 v2 |
  bin-eval BinMul v1 v2 = intval-mul v1 v2 |
  bin-eval BinSub v1 v2 = intval-sub v1 v2 |
  bin-eval BinAnd v1 v2 = intval-and v1 v2 |
  bin-eval BinOr v1 v2 = intval-or v1 v2 |
  bin-eval BinXor v1 v2 = intval-xor v1 v2 |
  bin-eval 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 fresh-id :: IRGraph ⇒ ID ⇒ bool where
  nid ∉ ids g ⇒ fresh-id g nid

```

```

code-pred fresh-id ⟨proof⟩

```

```

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

```

```

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

```

```

export-code get-fresh-id

```

```

value get-fresh-id eg2-sq

```

```

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

```

```

inductive

```

```

  unrep :: IRGraph ⇒ IRExpr ⇒ (IRGraph × ID) ⇒ bool (- ◁ - ∼ - 55)

```

```

and

```

```

  unrepList :: IRGraph ⇒ IRExpr list ⇒ (IRGraph × ID list) ⇒ bool (- ◁L - ∼ - 55)

```

```

where

```

```

  ConstantNodeSame:

```

```

  [[find-node-and-stamp g (ConstantNode c, constantAsStamp c) = Some nid]]
    ⇒ g ◁ (ConstantExpr c) ∼ (g, nid) |

```

```

  ConstantNodeNew:

```

```

  [[find-node-and-stamp g (ConstantNode c, constantAsStamp c) = None;
    nid = get-fresh-id g;
    g' = add-node nid (ConstantNode c, constantAsStamp c) g]]
    ⇒ g ◁ (ConstantExpr c) ∼ (g', nid) |

```

```

  ParameterNodeSame:

```

```

  [[find-node-and-stamp g (ParameterNode i, s) = Some nid]]
    ⇒ g ◁ (ParameterExpr i s) ∼ (g, nid) |

```

ParameterNodeNew:

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

ConditionalNodeSame:

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

ConditionalNodeNew:

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

UnaryNodeSame:

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

UnaryNodeNew:

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

BinaryNodeSame:

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

BinaryNodeNew:

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

AllLeafNodes:

stamp g $nid = s$

$\implies g \triangleleft (LeafExpr\ nid\ s) \rightsquigarrow (g, nid) \mid$

UnrepNil:

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

UnrepCons:

$\llbracket g \triangleleft xe \rightsquigarrow (g2, x) \rrbracket$

$g2 \triangleleft_L xes \rightsquigarrow (g3, xs)$

$\implies g \triangleleft_L (xe \# xes) \rightsquigarrow (g3, x \# xs)$

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

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

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

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

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

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

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

$$\frac{\begin{array}{c} g \triangleleft_L [ce, te, fe] \rightsquigarrow (g2, [c, t, f]) \quad s' = \text{meet } (\text{stamp } g2\ t) (\text{stamp } g2\ f) \\ \text{find-node-and-stamp } g2 \text{ (ConditionalNode } c\ t\ f, s') = \text{Some } nid \end{array}}{g \triangleleft \text{ConditionalExpr } ce\ te\ fe \rightsquigarrow (g2, nid)}$$

$$\frac{\begin{array}{c} g \triangleleft_L [ce, te, fe] \rightsquigarrow (g2, [c, t, f]) \quad s' = \text{meet } (\text{stamp } g2\ t) (\text{stamp } g2\ f) \\ \text{find-node-and-stamp } g2 \text{ (ConditionalNode } c\ t\ f, s') = \text{None} \\ nid = \text{get-fresh-id } g2 \quad g' = \text{add-node } nid \text{ (ConditionalNode } c\ t\ f, s') \end{array} \quad g2}{g \triangleleft \text{ConditionalExpr } ce\ te\ fe \rightsquigarrow (g', nid)}$$

$$\frac{\begin{array}{c} g \triangleleft_L [xe, ye] \rightsquigarrow (g2, [x, y]) \quad s' = \text{stamp-binary } op \text{ (stamp } g2\ x) (\text{stamp } g2\ y) \\ \text{find-node-and-stamp } g2 \text{ (bin-node } op\ x\ y, s') = \text{Some } nid \end{array}}{g \triangleleft \text{BinaryExpr } op\ xe\ ye \rightsquigarrow (g2, nid)}$$

$$\begin{array}{c}
\frac{
\begin{array}{l}
g \triangleleft_L [xe, ye] \rightsquigarrow (g2, [x, y]) \quad s' = \text{stamp-binary op } (\text{stamp } g2 \ x) (\text{stamp } g2 \ y) \\
\quad \text{find-node-and-stamp } g2 \ (\text{bin-node op } x \ y, s') = \text{None} \\
nid = \text{get-fresh-id } g2 \quad g' = \text{add-node } nid \ (\text{bin-node op } x \ y, s') \ g2
\end{array}
}{g \triangleleft \text{BinaryExpr op } xe \ ye \rightsquigarrow (g', nid)} \\
\\
\frac{
\begin{array}{l}
g \triangleleft xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op } (\text{stamp } g2 \ x) \\
\text{find-node-and-stamp } g2 \ (\text{unary-node op } x, s') = \text{Some } nid
\end{array}
}{g \triangleleft \text{UnaryExpr op } xe \rightsquigarrow (g2, nid)} \\
\\
\frac{
\begin{array}{l}
g \triangleleft xe \rightsquigarrow (g2, x) \quad s' = \text{stamp-unary op } (\text{stamp } g2 \ x) \\
\text{find-node-and-stamp } g2 \ (\text{unary-node op } x, s') = \text{None} \\
nid = \text{get-fresh-id } g2 \quad g' = \text{add-node } nid \ (\text{unary-node op } x, s') \ g2
\end{array}
}{g \triangleleft \text{UnaryExpr op } xe \rightsquigarrow (g', nid)} \\
\\
\frac{
\text{stamp } g \ nid = s
}{g \triangleleft \text{LeafExpr } nid \ s \rightsquigarrow (g, nid)}
\end{array}$$

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

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

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

5.2 Data-flow Tree Evaluation

inductive

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

ConstantExpr:

$\llbracket c \neq \text{UndefVal} \rrbracket$
 $\implies [m, p] \vdash (\text{ConstantExpr } c) \mapsto c \mid$

ParameterExpr:

$\llbracket \text{valid-value } s \ (p!i) \rrbracket$
 $\implies [m, p] \vdash (\text{ParameterExpr } i \ s) \mapsto p!i \mid$

ConditionalExpr:

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

UnaryExpr:

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

BinaryExpr:

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

LeafExpr:

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

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

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

[show-steps, show-mode-inference, show-intermediate-results]
evaltree <proof>

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:

$\llbracket [m,p] \vdash x \mapsto xval;$
 $[m,p] \vdash yy \mapsto_L yyval \rrbracket$


```

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

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

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

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

```

5.3 Data-flow Tree Refinement

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

definition *equiv-exprs* :: *IRExpr* \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*
 $\langle \text{proof} \rangle$

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

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

instance $\langle \text{proof} \rangle$
end

end

6 Data-flow Expression-Tree Theorems

theory *IRTreeEvalThms*
imports

Semantics.IRTreeEval
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.

lemma *rep-constant*:

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

lemma *rep-parameter*:

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

lemma *rep-conditional*:

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

lemma *rep-abs*:

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

lemma *rep-not*:

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

lemma *rep-negate*:

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

lemma *rep-logicnegation*:

$$\begin{aligned} g \vdash n \triangleright e &\implies \\ \text{kind } g \ n &= \text{LogicNegationNode } x \implies \end{aligned}$$

$(\exists xe. e = \text{UnaryExpr UnaryLogicNegation } xe)$
 $\langle \text{proof} \rangle$

lemma *rep-add*:

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

lemma *rep-sub*:

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

lemma *rep-mul*:

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

lemma *rep-and*:

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

lemma *rep-or*:

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

lemma *rep-xor*:

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

lemma *rep-integer-below*:

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

lemma *rep-integer-equals*:

$g \vdash n \triangleright e \implies$
 $\text{kind } g \ n = \text{IntegerEqualsNode } x \ y \implies$
 $(\exists xe \ ye. e = \text{BinaryExpr BinIntegerEquals } xe \ ye)$

$\langle \text{proof} \rangle$

lemma *rep-integer-less-than*:

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

lemma *rep-load-field*:

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

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

lemma *repDet*:

shows $(g \vdash n \triangleright e1) \implies (g \vdash n \triangleright e2) \implies e1 = e2$
 $\langle \text{proof} \rangle$

lemma *repAllDet*:

$g \vdash xs \triangleright_L e1 \implies$
 $g \vdash xs \triangleright_L e2 \implies$
 $e1 = e2$
 $\langle \text{proof} \rangle$

lemma *evalDet*:

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

lemma *evalAllDet*:

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

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* \implies

$val = \text{UndefVal}$
 $\langle \text{proof} \rangle$

lemma *valid-ObjStamp[elim]*:
shows *valid-value* (*ObjectStamp* *klass exact nonNull alwaysNull*) *val* \implies
 $(\exists v. val = \text{ObjRef } v)$
 $\langle \text{proof} \rangle$

lemma *valid-int32[elim]*:
shows *valid-value* (*IntegerStamp* 32 *l h*) *val* \implies
 $(\exists v. val = \text{IntVal32 } v)$
 $\langle \text{proof} \rangle$

lemma *valid-int64[elim]*:
shows *valid-value* (*IntegerStamp* 64 *l h*) *val* \implies
 $(\exists v. val = \text{IntVal64 } v)$
 $\langle \text{proof} \rangle$

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 $\exists v. val = (\text{IntVal32 } v)$
 $\langle \text{proof} \rangle$

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

lemma *default-stamp [simp]*: *default-stamp* = *IntegerStamp* 32 (-2147483648)
 2147483647
 $\langle \text{proof} \rangle$

lemma *valid32 [simp]*:
assumes *valid-value* (*IntegerStamp* 32 *lo hi*) *val*
shows $\exists v. (val = (\text{IntVal32 } v) \wedge lo \leq \text{sint } v \wedge \text{sint } v \leq hi)$
 $\langle \text{proof} \rangle$

lemma *valid64 [simp]*:
assumes *valid-value* (*IntegerStamp* 64 *lo hi*) *val*
shows $\exists v. (val = (\text{IntVal64 } v) \wedge lo \leq \text{sint } v \wedge \text{sint } v \leq hi)$
 $\langle \text{proof} \rangle$

lemma *int-stamp-implies-valid-value*:
 $[m, p] \vdash \text{expr} \mapsto val \implies$

valid-value (*stamp-expr* *expr*) *val*
 ⟨*proof*⟩

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

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

6.2 Example Data-flow Optimisations

lemma *a0a-helper* [*simp*]:
assumes *a*: *valid-value* (*IntegerStamp* 32 *lo hi*) *v*
shows *intval-add* *v* (*IntVal32* 0) = *v*
 ⟨*proof*⟩

lemma *a0a*: (*BinaryExpr* *BinAdd* (*LeafExpr* 1 *default-stamp*) (*ConstantExpr* (*IntVal32* 0)))
 ≤ (*LeafExpr* 1 *default-stamp*) (**is** ?*L* ≤ ?*R*)
 ⟨*proof*⟩

lemma *xyx-y-helper* [*simp*]:
assumes *valid-value* (*IntegerStamp* 32 *lox hix*) *x*
assumes *valid-value* (*IntegerStamp* 32 *loy hiy*) *y*
shows *intval-add* *x* (*intval-sub* *y* *x*) = *y*
 ⟨*proof*⟩

lemma *xyx-y*:
 (*BinaryExpr* *BinAdd*
 (*LeafExpr* *x* (*IntegerStamp* 32 *lox hix*))
 (*BinaryExpr* *BinSub*
 (*LeafExpr* *y* (*IntegerStamp* 32 *loy hiy*))
 (*LeafExpr* *x* (*IntegerStamp* 32 *lox hix*))))
 ≤ (*LeafExpr* *y* (*IntegerStamp* 32 *loy hiy*))
 ⟨*proof*⟩

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 $e \leq e'$

shows $(UnaryExpr\ op\ e) \leq (UnaryExpr\ op\ e')$

<proof>

lemma *mono-binary*:

assumes $x \leq x'$

assumes $y \leq y'$

shows $(BinaryExpr\ op\ x\ y) \leq (BinaryExpr\ op\ x'\ y')$

<proof>

lemma *mono-conditional*:

assumes $ce \leq ce'$

assumes $te \leq te'$

assumes $fe \leq fe'$

shows $(ConditionalExpr\ ce\ te\ fe) \leq (ConditionalExpr\ ce'\ te'\ fe')$

<proof>

end

7 Control-flow Semantics

theory *IRStepObj*

imports

IRTreeEval

begin

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

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

type-synonym FieldRefHeap = (string, objref) DynamicHeap

definition new-heap :: ('a, 'b) DynamicHeap where
  new-heap = ((λf. λp. UndefVal), 0)

```

7.2 Intraprocedural Semantics

Intraprocedural semantics are given as a small-step semantics.

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

```

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

```

SequentialNode:

```

[[is-sequential-node (kind g nid);
  nid' = (successors-of (kind g nid))!0]]
⇒ g, p ⊢ (nid, m, h) → (nid', m, h) |

```

IfNode:

```

[[kind g nid = (IfNode cond tb fb);
  g ⊢ cond ▷ condE;
  [m, p] ⊢ condE ↦ val;
  nid' = (if val-to-bool val then tb else fb)]
⇒ g, p ⊢ (nid, m, h) → (nid', m, h) |

```

EndNodes:

```

[[is-AbstractEndNode (kind g nid);
  merge = any-usage g nid;
  is-AbstractMergeNode (kind g merge);

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

  m' = set-phis phis vs m]
⇒ g, p ⊢ (nid, m, h) → (merge, m', h) |

```


NewInstanceNode:

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

LoadFieldNode:

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

SignedDivNode:

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

SignedRemNode:

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

StaticLoadFieldNode:

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

StoreFieldNode:

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

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

StaticStoreFieldNode:

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

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

7.3 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:

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

InvokeNodeStep:

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

callTarget = *ir-callTarget* (*kind* *g* *nid*);

kind *g* *callTarget* = (*MethodCallTargetNode* *targetMethod* *arguments*);

Some *targetGraph* = *P* *targetMethod*;

m' = *new-map-state*;

g \vdash *arguments* \triangleright_L *argsE*;

[*m*, *p*] \vdash *argsE* \mapsto_L *p*]

$$\Longrightarrow P \vdash ((g, nid, m, p) \# stk, h) \longrightarrow ((targetGraph, 0, m', p') \# (g, nid, m, p) \# stk, h)$$

|

ReturnNode:

$$\llbracket kind\ g\ nid = (ReturnNode\ (Some\ expr)\ -) ;$$

g \vdash *expr* \triangleright *e*;

[*m*, *p*] \vdash *e* \mapsto *v*;

cm' = *cm*(*cnid* := *v*);

cnid' = (*successors-of* (*kind* *cg* *cnid*))!0]

$$\Longrightarrow P \vdash ((g, nid, m, p) \# (cg, cnid, cm, cp) \# stk, h) \longrightarrow ((cg, cnid', cm', cp) \# stk, h) \mid$$

ReturnNodeVoid:

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

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

UnwindNode:

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

$g \vdash \text{exception} \triangleright \text{exceptionE};$
 $[m, p] \vdash \text{exceptionE} \mapsto e;$

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

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

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

7.4 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*

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

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

for *P*

where

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

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

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

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

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

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

$\Rightarrow (IRGraph \times ID \times MapState \times Params) \text{ list} \times FieldRefHeap$
 $\Rightarrow \text{nat}$
 $\Rightarrow (IRGraph \times ID \times MapState \times Params) \text{ list} \times FieldRefHeap$
 $\Rightarrow \text{bool}$

($\vdash \dashv \rightarrow * - * -$)

where

$\llbracket n > 0; \quad$
 $p \vdash s \longrightarrow s';$
 $\text{exec-debug } p \ s' \ (n - 1) \ s'' \rrbracket$
 $\implies \text{exec-debug } p \ s \ n \ s'' \mid$

$\llbracket n = 0 \rrbracket$

$\implies \text{exec-debug } p \ s \ n \ s$

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

7.4.1 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**

field-sq = "sq"

definition *eg3-sq* :: *IRGraph* **where**

eg3-sq = *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 $\{h\text{-load-field } \text{field-sq } \text{None} \ (prod.snd \ res)$

$\mid res. (\lambda x. \text{Some } eg3\text{-sq}) \vdash ([(eg3\text{-sq}, 0, \text{new-map-state}, p3), (eg3\text{-sq}, 0,$
 $\text{new-map-state}, p3)], \text{new-heap}) \rightarrow * 3 * res\}$

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-load-field field-sq (Some 0) (prod.snd res) | res.
        (λx. Some eg4-sq) ⊢ [(eg4-sq, 0, new-map-state, p3), (eg4-sq, 0,
new-map-state, p3)], new-heap) →*4* res}

```

end

8 Canonicalization Phase

theory CanonicalizationTree

imports

Semantics.IRTreeEval

begin

fun is-neutral :: IRBinaryOp ⇒ Value ⇒ bool **where**

```

is-neutral BinMul (IntVal32 x) = (x = 1) |
is-neutral BinMul (IntVal64 x) = (x = 1) |

```

```

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

```

```

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

```

```

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

```

is-neutral - - = False

fun is-zero :: IRBinaryOp ⇒ Value ⇒ bool **where**

```

is-zero BinMul (IntVal32 x) = (x = 0) |
is-zero BinMul (IntVal64 x) = (x = 0) |

```

is-zero - - = False

fun int-to-value :: Value ⇒ int ⇒ Value **where**

$int\text{-}to\text{-}value\ (IntVal32\ -)\ y = (IntVal32\ (word\text{-}of\text{-}int\ y))\ |\$
 $int\text{-}to\text{-}value\ (IntVal64\ -)\ y = (IntVal64\ (word\text{-}of\text{-}int\ y))\ |\$
 $int\text{-}to\text{-}value\ -\ =\ UndefVal$

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

binary-const-fold:

$\llbracket x = (ConstantExpr\ val1);$
 $y = (ConstantExpr\ val2);$
 $val = bin\text{-}eval\ op\ val1\ val2;$
 $val \neq UndefVal \rrbracket$
 $\implies CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ (ConstantExpr\ val)\ |$

binary-fold-yneutral:

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

binary-fold-yzero32:

$\llbracket y = ConstantExpr\ c;$
 $is\text{-}zero\ op\ c;$
 $stampx = stamp\text{-}expr\ x;$
 $stampy = stamp\text{-}expr\ y;$
 $stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
 $stp\text{-}bits\ stampx = 32;$
 $is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy \rrbracket$
 $\implies CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ (ConstantExpr\ (IntVal32\ 0))\ |$

binary-fold-yzero64:

$\llbracket y = ConstantExpr\ c;$
 $is\text{-}zero\ op\ c;$
 $stampx = stamp\text{-}expr\ x;$
 $stampy = stamp\text{-}expr\ y;$
 $stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
 $stp\text{-}bits\ stampx = 64;$
 $is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy \rrbracket$
 $\implies CanonicalizeBinaryOp\ (BinaryExpr\ op\ x\ y)\ (ConstantExpr\ (IntVal64\ 0))$

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

unary-const-fold:

$\llbracket val' = unary\text{-}eval\ op\ val;$
 $val' \neq UndefVal \rrbracket$
 $\implies CanonicalizeUnaryOp\ (UnaryExpr\ op\ (ConstantExpr\ val))\ (ConstantExpr\ val')$

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

mul-negate32:

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

mul-negate64:

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

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

add-xsub:

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

add-y-sub:

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

add-xnegate:

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

add-ynegate:

$\llbracket ny = (\text{UnaryExpr UnaryNeg } y);$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{is-IntegerStamp stampx} \wedge \text{is-IntegerStamp stampy};$
 $\text{stp-bits stampx} = \text{stp-bits stampy} \rrbracket$

$\implies \text{CanonicalizeAdd } (\text{BinaryExpr BinAdd } x \text{ } ny) (\text{BinaryExpr BinSub } x \text{ } y)$

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

sub-same32:

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

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

sub-left-add1:

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

sub-left-add2:

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

sub-left-sub:

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

sub-right-add1:

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

$\implies \text{CanonicalizeSub } (\text{BinaryExpr BinSub } a \ y) \ (\text{UnaryExpr UnaryNeg } b) \mid$

sub-right-add2:

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

sub-right-sub:

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

sub-xzero32:

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

sub-xzero64:

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

sub-y-negate:

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

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

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

negate-sub:

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

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

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

abs-neg:

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

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

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

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

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

and-demorgans:

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

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

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

or-demorgans:

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

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:

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

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:

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

int-equals-right-contains-left3:

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

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

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

cond-eq:

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

condition-bounds-x:

$\llbracket c = (\text{BinaryExpr BinIntegerLessThan } x \ y);$
 $\text{stampx} = \text{stamp-expr } x;$
 $\text{stampy} = \text{stamp-expr } y;$
 $\text{stpi-upper stampx} \leq \text{stpi-lower stampy};$

$stp\text{-}bits\ stampx = stp\text{-}bits\ stampy;$
 $is\text{-}IntegerStamp\ stampx \wedge is\text{-}IntegerStamp\ stampy$
 $\implies CanonicalizeConditional\ (ConditionalExpr\ c\ x\ y)\ x\ |$

condition-bounds-y:

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

negate-condition:

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

const-true:

$\llbracket c = ConstantExpr\ val;$
 $val\text{-}to\text{-}bool\ val$
 $\implies CanonicalizeConditional\ (ConditionalExpr\ c\ t\ f)\ t\ |$

const-false:

$\llbracket c = ConstantExpr\ val;$
 $\neg(val\text{-}to\text{-}bool\ val)$
 $\implies CanonicalizeConditional\ (ConditionalExpr\ c\ t\ f)\ f$

inductive *CanonicalizationStep* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
BinaryNode:
 $\llbracket CanonicalizeBinaryOp\ expr\ expr' \rrbracket$

$\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$

UnaryNode:

$\llbracket \text{CanonicalizeUnaryOp } \text{expr } \text{expr}' \rrbracket$
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$

NegateNode:

$\llbracket \text{CanonicalizeNegate } \text{expr } \text{expr}' \rrbracket$
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$

NotNode:

$\llbracket \text{CanonicalizeNegate } \text{expr } \text{expr}' \rrbracket$
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$

AddNode:

$\llbracket \text{CanonicalizeAdd } \text{expr } \text{expr}' \rrbracket$
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$

MulNode:

$\llbracket \text{CanonicalizeMul } \text{expr } \text{expr}' \rrbracket$
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$

SubNode:

$\llbracket \text{CanonicalizeSub } \text{expr } \text{expr}' \rrbracket$
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$

AndNode:

$\llbracket \text{CanonicalizeSub } \text{expr } \text{expr}' \rrbracket$
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$

OrNode:

$\llbracket \text{CanonicalizeSub } \text{expr } \text{expr}' \rrbracket$
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$

IntegerEqualsNode:

$\llbracket \text{CanonicalizeIntegerEquals } \text{expr } \text{expr}' \rrbracket$
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}' \mid$

ConditionalNode:

$\llbracket \text{CanonicalizeConditional } \text{expr } \text{expr}' \rrbracket$
 $\implies \text{CanonicalizationStep } \text{expr } \text{expr}'$

code-pred (modes: $i \Rightarrow o \Rightarrow \text{bool}$) *CanonicalizeBinaryOp* $\langle \text{proof} \rangle$
code-pred (modes: $i \Rightarrow o \Rightarrow \text{bool}$) *CanonicalizeUnaryOp* $\langle \text{proof} \rangle$
code-pred (modes: $i \Rightarrow o \Rightarrow \text{bool}$) *CanonicalizeNegate* $\langle \text{proof} \rangle$
code-pred (modes: $i \Rightarrow o \Rightarrow \text{bool}$) *CanonicalizeNot* $\langle \text{proof} \rangle$
code-pred (modes: $i \Rightarrow o \Rightarrow \text{bool}$) *CanonicalizeAdd* $\langle \text{proof} \rangle$
code-pred (modes: $i \Rightarrow o \Rightarrow \text{bool}$) *CanonicalizeSub* $\langle \text{proof} \rangle$
code-pred (modes: $i \Rightarrow o \Rightarrow \text{bool}$) *CanonicalizeMul* $\langle \text{proof} \rangle$

```

code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizeAnd ⟨proof⟩
code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizeIntegerEquals ⟨proof⟩
code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizeConditional ⟨proof⟩

code-pred (modes:  $i \Rightarrow o \Rightarrow \text{bool}$ ) CanonicalizationStep ⟨proof⟩

end

```

9 Canonicalization Phase

```

theory CanonicalizationTreeProofs
  imports
    CanonicalizationTree
    Semantics.IRTreeEvalThms
begin

```

lemma *neutral-rewrite-helper*:

```

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

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

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

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

and valid-value (IntegerStamp 32 lo hi)  $x \Longrightarrow \text{intval-or } x \text{ (IntVal32 (0))} = x$ 
and valid-value (IntegerStamp 64 lo hi)  $x \Longrightarrow \text{intval-or } x \text{ (IntVal64 (0))} = x$ 
⟨proof⟩

```

lemma *CanonicalizeBinaryProof*:

```

assumes CanonicalizeBinaryOp before after
assumes  $[m, p] \vdash \text{before} \mapsto \text{res}$ 
assumes  $[m, p] \vdash \text{after} \mapsto \text{res}'$ 
shows  $\text{res} = \text{res}'$ 
⟨proof⟩

```

lemma *CanonicalizeUnaryProof*:

```

assumes CanonicalizeUnaryOp before after
assumes  $[m, p] \vdash \text{before} \mapsto \text{res}$ 
assumes  $[m, p] \vdash \text{after} \mapsto \text{res}'$ 
shows  $\text{res} = \text{res}'$ 
⟨proof⟩

```

lemma *mul-rewrite-helper*:

```

shows valid-value (IntegerStamp 32 lo hi)  $x \Longrightarrow \text{intval-mul } x \text{ (IntVal32 (-1))} =$ 

```


intval-negate x
and *valid-value (IntegerStamp 64 lo hi) x \implies intval-mul x (IntVal64 (-1)) =*
intval-negate x
<proof>

lemma *CanonicalizeMulProof:*
assumes *CanonicalizeMul before after*
assumes $[m, p] \vdash \text{before} \mapsto \text{res}$
assumes $[m, p] \vdash \text{after} \mapsto \text{res}'$
shows $\text{res} = \text{res}'$
<proof>

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

shows *intval-add (intval-sub x y) y = x*
and *intval-add x (intval-sub y x) = y*
and *intval-add (intval-negate x) y = intval-sub y x*
and *intval-add x (intval-negate y) = intval-sub x y*
<proof>

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 $\text{res} = \text{res}'$
<proof>

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

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

and $\text{intval-sub } x (\text{intval-sub } x y) = y$
and $\text{intval-sub } x (\text{intval-negate } y) = \text{intval-add } x y$
 $\langle \text{proof} \rangle$

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

lemma *negate-negate-helper*:
assumes $\text{valid-value } (\text{IntegerStamp } b \text{ lox hix}) x$
shows $\text{intval-negate } (\text{intval-negate } x) = x$
 $\langle \text{proof} \rangle$

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

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 $\bigwedge x :: 32 \text{ word. } \neg \neg x < s \ 0 \wedge \neg x < s \ 0 \implies 2 * x = 0$
and $\bigwedge x :: 64 \text{ word. } \neg \neg x < s \ 0 \wedge \neg x < s \ 0 \implies 2 * x = 0$
 $\langle \text{proof} \rangle$

lemma *abs-rewrite-helper*:
assumes *valid-value* (*IntegerStamp* *b lox hix*) *x*
shows *intval-abs* (*intval-negate* *x*) = *intval-abs* *x*
and *intval-abs* (*intval-abs* *x*) = *intval-abs* *x*
 $\langle proof \rangle$

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

lemma *not-rewrite-helper*:
assumes *valid-value* (*IntegerStamp* *b lox hix*) *x*
shows *intval-not* (*intval-not* *x*) = *x*
 $\langle proof \rangle$

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

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 \textit{intval-and} *x y* = *x*
and $x = y \implies \textit{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 $\textit{res} = \textit{res}'$
 $\langle proof \rangle$

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 $\textit{res} = \textit{res}'$

$\langle \text{proof} \rangle$

lemma *stamps-touch-but-not-less-than-implies-equal:*

$\llbracket \text{valid-value stamp} x;$
 $\text{valid-value stamp} y;$
 $\text{is-IntegerStamp stamp} x \wedge \text{is-IntegerStamp stamp} y;$
 $\text{stpi-upper stamp} x = \text{stpi-lower stamp} y;$
 $\neg \text{val-to-bool}(\text{intval-less-than } x \ y) \rrbracket \implies x = y$
 $\langle \text{proof} \rangle$

lemma *disjoint-stamp-implies-less-than:*

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

lemma *CanonicalizeConditionalProof:*

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

end