

Veriopt

July 6, 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	21
4	Graph Representation	25
4.0.1	Example Graphs	28
5	Data-flow Semantics	29
5.1	Data-flow Tree Representation	30
5.2	Data-flow Tree Evaluation	39
5.3	Data-flow Tree Refinement	40
6	Data-flow Expression-Tree Theorems	41
6.1	Extraction and Evaluation of Expression Trees is Deterministic.	41
6.2	Example Data-flow Optimisations	45
6.3	Monotonicity of Expression Optimization	46
7	Control-flow Semantics	46
7.1	Heap	47
7.2	Intraprocedural Semantics	47
7.3	Interprocedural Semantics	49
7.4	Big-step Execution	50
7.4.1	Heap Testing	51
8	Canonicalization Phase	52
9	Canonicalization Phase	62

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 = (( $-(2^{b-1}) \leq val$ )  $\wedge$  ( $val < (2^b)$ ))

```

```
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-not* :: *Value* \Rightarrow *Value* **where**
intval-not (IntVal32 v) = (IntVal32 (NOT v)) |
intval-not (IntVal64 v) = (IntVal64 (NOT v)) |
intval-not - = UndefVal

fun *intval-negate* :: *Value* \Rightarrow *Value* **where**

```

intval-negate (IntVal32 v) = IntVal32 (- v) |
intval-negate (IntVal64 v) = IntVal64 (- v) |
intval-negate - = UndefVal

```

```

fun intval-abs :: Value ⇒ 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 (231-1)) (IntVal32 (231-1)) = 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*: IN-
 PUT-STATE option) (*ir-values-opt*: INPUT list option) (*ir-virtualObjectMappings-opt*:
 INPUT-STATE list option)
 | *IfNode* (*ir-condition*: INPUT-COND) (*ir-trueSuccessor*: SUCC) (*ir-falseSuccessor*:
 SUCC)
 | *IntegerEqualsNode* (*ir-x*: INPUT) (*ir-y*: INPUT)
 | *IntegerLessThanNode* (*ir-x*: INPUT) (*ir-y*: INPUT)
 | *InvokeNode* (*ir-nid*: ID) (*ir-callTarget*: INPUT-EXT) (*ir-classInit-opt*: IN-
 PUT option) (*ir-stateDuring-opt*: INPUT-STATE option) (*ir-stateAfter-opt*: IN-
 PUT-STATE option) (*ir-next*: SUCC)
 | *InvokeWithExceptionNode* (*ir-nid*: ID) (*ir-callTarget*: INPUT-EXT) (*ir-classInit-opt*:
 INPUT option) (*ir-stateDuring-opt*: INPUT-STATE option) (*ir-stateAfter-opt*: IN-
 PUT-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*: IN-
 PUT-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  $\Rightarrow$  'a list where
  opt-to-list None = [] |
  opt-to-list (Some v) = [v]

```

```

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

```

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

```

fun inputs-of :: IRNode  $\Rightarrow$  ID list where
  inputs-of-AbsNode:
  inputs-of (AbsNode value) = [value] |
  inputs-of-AddNode:
  inputs-of (AddNode x y) = [x, y] |
  inputs-of-AndNode:
  inputs-of (AndNode x y) = [x, y] |
  inputs-of-BEGINNode:
  inputs-of (BeginNode next) = [] |
  inputs-of-BytecodeExceptionNode:
  inputs-of (BytecodeExceptionNode arguments stateAfter next) = arguments @
(opt-to-list stateAfter) |
  inputs-of-ConditionalNode:
  inputs-of (ConditionalNode condition trueValue falseValue) = [condition, true-
Value, falseValue] |
  inputs-of-ConstantNode:
  inputs-of (ConstantNode const) = [] |
  inputs-of-DynamicNewArrayNode:
  inputs-of (DynamicNewArrayNode elementType length0 voidClass stateBefore
next) = [elementType, length0] @ (opt-to-list voidClass) @ (opt-to-list stateBefore)
|

```

inputs-of-EndNode:
inputs-of (EndNode) = [] |
inputs-of-ExceptionObjectNode:
inputs-of (ExceptionObjectNode stateAfter next) = (opt-to-list stateAfter) |
inputs-of-FrameState:
inputs-of (FrameState monitorIds outerFrameState values virtualObjectMappings)
= monitorIds @ (opt-to-list outerFrameState) @ (opt-list-to-list values) @ (opt-list-to-list
virtualObjectMappings) |
inputs-of-IfNode:
inputs-of (IfNode condition trueSuccessor falseSuccessor) = [condition] |
inputs-of-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 state-
Before) |

inputs-of-NewInstanceNode:
inputs-of (*NewInstanceNode* *nid0* *instanceClass* *stateBefore* *next*) = (*opt-to-list* *stateBefore*) |
inputs-of-NotNode:
inputs-of (*NotNode* *value*) = [*value*] |
inputs-of-OrNode:
inputs-of (*OrNode* *x* *y*) = [*x*, *y*] |
inputs-of-ParameterNode:
inputs-of (*ParameterNode* *index*) = [] |
inputs-of-PiNode:
inputs-of (*PiNode* *object* *guard*) = *object* # (*opt-to-list* *guard*) |
inputs-of-ReturnNode:
inputs-of (*ReturnNode* *result* *memoryMap*) = (*opt-to-list* *result*) @ (*opt-to-list* *memoryMap*) |
inputs-of-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 virtualObjectMappings*) = [] |
successors-of-IfNode:
successors-of (*IfNode* *condition trueSuccessor falseSuccessor*) = [*trueSuccessor*, *falseSuccessor*] |
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) = []  $\wedge$  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  $\Rightarrow$  bool where
    is-EndNode EndNode = True |
    is-EndNode - = False

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

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

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

fun is-AbstractBeginNode :: IRNode  $\Rightarrow$  bool where
    is-AbstractBeginNode n = ((is-BeginNode n)  $\vee$  (is-BeginStateSplitNode n)  $\vee$  (is-KillingBeginNode n))

fun is-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 $n = ((is-AbsNode\ n) \vee (is-NegateNode\ n) \vee (is-NotNode\ n))$

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*))

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

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

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

```

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

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

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

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

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

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

fun is-IndirectCanonicalization :: IRNode  $\Rightarrow$  bool where
  is-IndirectCanonicalization n = ((is-LogicNode n))

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

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

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

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

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

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

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

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

```

is-GuardingNode $n = ((\text{is-AbstractBeginNode } n))$

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

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

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

fun *is-ArithmeticLIRLowerable* :: *IRNode* \Rightarrow *bool* **where**
is-ArithmeticLIRLowerable $n = ((\text{is-AbsNode } n) \vee (\text{is-BinaryArithmeticNode } n) \vee (\text{is-NotNode } n) \vee (\text{is-UnaryArithmeticNode } n))$

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

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

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

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

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

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

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

fun *is-Binary* :: *IRNode* \Rightarrow *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* ⇒ *IRNode* ⇒ *bool* **where**
is-same-ir-node-type *n1* *n2* = (
 ((*is-AbsNode* *n1*) ∧ (*is-AbsNode* *n2*)) ∨
 ((*is-AddNode* *n1*) ∧ (*is-AddNode* *n2*)) ∨
 ((*is-AndNode* *n1*) ∧ (*is-AndNode* *n2*)) ∨
 ((*is-BeginNode* *n1*) ∧ (*is-BeginNode* *n2*)) ∨
 ((*is-BytecodeExceptionNode* *n1*) ∧ (*is-BytecodeExceptionNode* *n2*)) ∨
 ((*is-ConditionalNode* *n1*) ∧ (*is-ConditionalNode* *n2*)) ∨

```

((is-ConstantNode n1) ∧ (is-ConstantNode n2)) ∨
((is-DynamicNewArrayNode n1) ∧ (is-DynamicNewArrayNode n2)) ∨
((is-EndNode n1) ∧ (is-EndNode n2)) ∨
((is-ExceptionObjectNode n1) ∧ (is-ExceptionObjectNode n2)) ∨
((is-FrameState n1) ∧ (is-FrameState n2)) ∨
((is-IfNode n1) ∧ (is-IfNode n2)) ∨
((is-IntegerEqualsNode n1) ∧ (is-IntegerEqualsNode n2)) ∨
((is-IntegerLessThanNode n1) ∧ (is-IntegerLessThanNode n2)) ∨
((is-InvokeNode n1) ∧ (is-InvokeNode n2)) ∨
((is-InvokeWithExceptionNode n1) ∧ (is-InvokeWithExceptionNode n2)) ∨
((is-IsNullNode n1) ∧ (is-IsNullNode n2)) ∨
((is-KillingBeginNode n1) ∧ (is-KillingBeginNode n2)) ∨
((is-LoadFieldNode n1) ∧ (is-LoadFieldNode n2)) ∨
((is-LogicNegationNode n1) ∧ (is-LogicNegationNode n2)) ∨
((is-LoopBeginNode n1) ∧ (is-LoopBeginNode n2)) ∨
((is-LoopEndNode n1) ∧ (is-LoopEndNode n2)) ∨
((is-LoopExitNode n1) ∧ (is-LoopExitNode n2)) ∨
((is-MergeNode n1) ∧ (is-MergeNode n2)) ∨
((is-MethodCallTargetNode n1) ∧ (is-MethodCallTargetNode n2)) ∨
((is-MulNode n1) ∧ (is-MulNode n2)) ∨
((is-NegateNode n1) ∧ (is-NegateNode n2)) ∨
((is-NewArrayNode n1) ∧ (is-NewArrayNode n2)) ∨
((is-NewInstanceNode n1) ∧ (is-NewInstanceNode n2)) ∨
((is-NotNode n1) ∧ (is-NotNode n2)) ∨
((is-OrNode n1) ∧ (is-OrNode n2)) ∨
((is-ParameterNode n1) ∧ (is-ParameterNode n2)) ∨
((is-PiNode n1) ∧ (is-PiNode n2)) ∨
((is-ReturnNode n1) ∧ (is-ReturnNode n2)) ∨
((is-ShortCircuitOrNode n1) ∧ (is-ShortCircuitOrNode n2)) ∨
((is-SignedDivNode n1) ∧ (is-SignedDivNode n2)) ∨
((is-StartNode n1) ∧ (is-StartNode n2)) ∨
((is-StoreFieldNode n1) ∧ (is-StoreFieldNode n2)) ∨
((is-SubNode n1) ∧ (is-SubNode n2)) ∨
((is-UnwindNode n1) ∧ (is-UnwindNode n2)) ∨
((is-ValuePhiNode n1) ∧ (is-ValuePhiNode n2)) ∨
((is-ValueProxyNode n1) ∧ (is-ValueProxyNode n2)) ∨
((is-XorNode n1) ∧ (is-XorNode n2))

```

end

3 Stamp Typing

```

theory 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
"" True True False) |
  empty-stamp stamp = IllegalStamp

```

```

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

  is-stamp-empty x = False

```

— Calculate the meet stamp of two stamps

```

fun meet :: Stamp ⇒ Stamp ⇒ Stamp where
  meet VoidStamp VoidStamp = VoidStamp |
  meet (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1 ≠ b2 then IllegalStamp else
    (IntegerStamp b1 (min l1 l2) (max u1 u2))
  ) |

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

```

— Calculate the join stamp of two stamps

```

fun join :: Stamp ⇒ Stamp ⇒ Stamp where
  join VoidStamp VoidStamp = VoidStamp |
  join (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1 ≠ b2 then IllegalStamp else
    (IntegerStamp b1 (max l1 l2) (min u1 u2))
  ) |

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

```

```

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

```

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

```

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

```

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

```

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

```

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

```

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

```

```

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

  constantAsStamp - = IllegalStamp

```

— Define when a runtime value is valid for a stamp

```

fun valid-value :: 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* \rightarrow (*IRNode* \times *Stamp*) . *finite* (*dom g*)}
 \langle *proof* \rangle

setup-lifting *type-definition-IRGraph*

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

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

lift-definition *kind* :: *IRGraph* \Rightarrow (*ID* \Rightarrow *IRNode*)
is *with-default* *NoNode* *fst* \langle *proof* \rangle

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

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

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

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

lift-definition *as-list* :: *IRGraph* \Rightarrow (*ID* \times *IRNode* \times *Stamp*) *list*

```

is  $\lambda g. \text{map } (\lambda k. (k, \text{the } (g\ k))) (\text{sorted-list-of-set } (\text{dom } g)) \langle \text{proof} \rangle$ 

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

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

code-datatype irgraph

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

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

lemma dom-eq:
  assumes  $\forall x \in \text{set } xs. \text{fst } (\text{snd } x) \neq \text{NoNode}$ 
  shows filter-none  $(\text{map-of } xs) = \text{dom } (\text{map-of } xs)$ 
 $\langle \text{proof} \rangle$ 

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

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

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

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

fun filtered-inputs :: IRGraph \Rightarrow ID \Rightarrow (IRNode \Rightarrow bool) \Rightarrow ID list **where**
 filtered-inputs g nid f = filter (f \circ (kind g)) (inputs-of (kind g nid))
fun filtered-successors :: IRGraph \Rightarrow ID \Rightarrow (IRNode \Rightarrow bool) \Rightarrow ID list **where**
 filtered-successors g nid f = filter (f \circ (kind g)) (successors-of (kind g nid))
fun filtered-usages :: IRGraph \Rightarrow ID \Rightarrow (IRNode \Rightarrow bool) \Rightarrow ID set **where**
 filtered-usages g nid f = {n \in (usages g nid). f (kind g n)}

fun is-empty :: IRGraph \Rightarrow bool **where**
 is-empty g = (ids g = {})

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

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

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

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

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

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

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

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

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

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

$\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

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

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 calculated 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 = (λx . *UndefVal*)

```

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

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

fun find-index :: 'a ⇒ 'a list ⇒ nat where
  find-index - [] = 0 |
  find-index v (x # xs) = (if (x=v) then 0 else find-index v xs + 1)

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

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

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

fun set-phis :: ID list ⇒ Value list ⇒ MapState ⇒ MapState where
  set-phis [] [] m = m |
  set-phis (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 ⇒ (IRNode × Stamp) ⇒ ID option where
  find-node-and-stamp g (n,s) =
    find (λi. kind g i = n ∧ stamp g i = s) (sorted-list-of-set(ids g))

export-code find-node-and-stamp

```

5.1 Data-flow Tree Representation

```

datatype IRUnaryOp =
  UnaryAbs
  | UnaryNeg
  | UnaryNot
  | UnaryLogicNegation

datatype IRBinaryOp =

```

```

    BinAdd
  | BinMul
  | BinSub
  | BinAnd
  | BinOr
  | BinXor
  | BinIntegerEquals
  | BinIntegerLessThan

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 ⇒ ID ⇒ IRExpr ⇒ bool (- ⊢ - ▷ - 55)
  for g where

  ConstantNode:
  [[kind g n = ConstantNode c]]
  ⇒ g ⊢ n ▷ (ConstantExpr c) |

  ParameterNode:
  [[kind g n = ParameterNode i;
  stamp g n = s]]
  ⇒ g ⊢ n ▷ (ParameterExpr i s) |

```

ConditionalNode:

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

AbsNode:

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

NotNode:

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

NegateNode:

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

LogicNegationNode:

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

AddNode:

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

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:

$\llbracket \text{kind } g \ n = \text{AndNode } x \ y; \\$

$g \vdash x \triangleright xe;$
 $g \vdash y \triangleright ye \parallel$
 $\implies g \vdash n \triangleright (BinaryExpr \text{ BinAnd } xe \ ye) \mid$

OrNode:

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

XorNode:

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

IntegerEqualsNode:

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

IntegerLessThanNode:

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

LeafNode:

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

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:

$g \vdash [] \triangleright_L [] \mid$

RepCons:

$\llbracket g \vdash x \triangleright xe;$
 $g \vdash xs \triangleright_L xse \parallel$
 $\implies g \vdash x \# xs \triangleright_L xe \# xse$

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

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

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

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

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

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

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

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

fun *bin-eval* :: *IRBinaryOp* \Rightarrow *Value* \Rightarrow *Value* \Rightarrow *Value* **where**
bin-eval *BinAdd* *v1 v2* = *intval-add* *v1 v2* |
bin-eval *BinMul* *v1 v2* = *intval-mul* *v1 v2* |
bin-eval *BinSub* *v1 v2* = *intval-sub* *v1 v2* |
bin-eval *BinAnd* *v1 v2* = *intval-and* *v1 v2* |
bin-eval *BinOr* *v1 v2* = *intval-or* *v1 v2* |
bin-eval *BinXor* *v1 v2* = *intval-xor* *v1 v2* |
bin-eval *BinIntegerEquals* *v1 v2* = *intval-equals* *v1 v2* |
bin-eval *BinIntegerLessThan* *v1 v2* = *intval-less-than* *v1 v2*

inductive *fresh-id* :: *IRGraph* \Rightarrow *ID* \Rightarrow *bool* **where**
nid \notin *ids* *g* \Longrightarrow *fresh-id* *g* *nid*

code-pred *fresh-id* *<proof>*

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

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

export-code *get-fresh-id*

value *get-fresh-id* *eg2-sq*

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

inductive

unrep :: *IRGraph* \Rightarrow *IRExpr* \Rightarrow (*IRGraph* \times *ID*) \Rightarrow *bool* (*-* \triangleleft *-* \rightsquigarrow *-* 55)

and

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

where

ConstantNodeSame:

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

ConstantNodeNew:

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

ParameterNodeSame:

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

ParameterNodeNew:

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

ConditionalNodeSame:

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

ConditionalNodeNew:

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

UnaryNodeSame:

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

UnaryNodeNew:

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

BinaryNodeSame:

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

BinaryNodeNew:

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

AllLeafNodes:

$\text{stamp } g \ nid = s$
 $\implies g \triangleleft (\text{LeafExpr } nid \ s) \rightsquigarrow (g, nid) \mid$

UnrepNil:

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

UnrepCons:

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

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

unrep <proof>

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

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

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

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

ConditionalExpr:

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

UnaryExpr:

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

BinaryExpr:

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

LeafExpr:

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

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

$$\frac{[m,p] \vdash xe \mapsto x \quad [m,p] \vdash ye \mapsto y}{[m,p] \vdash \text{BinaryExpr } op \ x \ ye \mapsto \text{bin-eval } op \ x \ y}$$

$$\frac{val = m \ \text{nid} \quad \text{valid-value } s \ val}{[m,p] \vdash \text{LeafExpr } \text{nid } s \mapsto val}$$

code-pred (*modes: i ⇒ i ⇒ i ⇒ o ⇒ bool as evalT*)
[show-steps, show-mode-inference, show-intermediate-results]
evaltree <proof>

inductive

evaltrees :: MapState ⇒ Params ⇒ IRExpr list ⇒ Value list ⇒ bool ([-,] ⊢ - ⊢_L
- 55)

for *m p* **where**

EvalNil:

[m,p] ⊢ [] ⊢_L [] |

EvalCons:

[[m,p] ⊢ x ⊢ xval;

[m,p] ⊢ yy ⊢_L yyval]

⇒ [m,p] ⊢ (x#yy) ⊢_L (xval#yyval)

code-pred (*modes: i ⇒ i ⇒ i ⇒ o ⇒ bool as evalTs*)
evaltrees <proof>

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

declare *evaltree.intros [intro]*

declare *evaltrees.intros [intro]*

5.3 Data-flow Tree Refinement

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

definition *equiv-exprs :: IRExpr ⇒ IRExpr ⇒ bool (- ≐ - 55) where*
(e1 ≐ e2) = (∀ m p v. (([m,p] ⊢ e1 ⊢ v) ⇔ ([m,p] ⊢ e2 ⊢ v)))

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

lemma *equivp equiv-exprs*
 ⟨proof⟩

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-expr-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-expr-def [simp]: $(e1 < e2) \longleftrightarrow (e1 \leq e2 \wedge \neg (e1 \doteq e2))$

instance ⟨proof⟩

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*:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = ConstantNode\ c \implies$
 $e = ConstantExpr\ c$
 ⟨proof⟩

lemma *rep-parameter*:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = ParameterNode\ i \implies$
 $(\exists s. e = ParameterExpr\ i\ s)$
 ⟨proof⟩

lemma *rep-conditional*:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = ConditionalNode\ c\ t\ f \implies$

$(\exists ce\ te\ fe. e = ConditionalExpr\ ce\ te\ fe)$
 $\langle proof \rangle$

lemma *rep-abs*:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = AbsNode\ x \implies$
 $(\exists xe. e = UnaryExpr\ UnaryAbs\ xe)$
 $\langle proof \rangle$

lemma *rep-not*:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = NotNode\ x \implies$
 $(\exists xe. e = UnaryExpr\ UnaryNot\ xe)$
 $\langle proof \rangle$

lemma *rep-negate*:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = NegateNode\ x \implies$
 $(\exists xe. e = UnaryExpr\ UnaryNeg\ xe)$
 $\langle proof \rangle$

lemma *rep-logicnegation*:

$g \vdash n \triangleright e \implies$
 $kind\ g\ n = LogicNegationNode\ x \implies$
 $(\exists xe. e = UnaryExpr\ UnaryLogicNegation\ xe)$
 $\langle proof \rangle$

lemma *rep-add*:

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

lemma *rep-sub*:

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

lemma *rep-mul*:

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

lemma *rep-and*:

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

$\langle \text{proof} \rangle$

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-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*:

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

lemma *rep-load-field*:

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

lemma *repDet*:

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

lemma *evalDet*:

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

lemma *evalAllDet*:

$[m, p] \vdash e \mapsto_L v1 \implies$
 $[m, p] \vdash e \mapsto_L v2 \implies$

$v1 = v2$
 $\langle proof \rangle$

A valid value cannot be *UndefVal*.

lemma *valid-not-undef*:
assumes *a1*: *valid-value s val*
assumes *a2*: $s \neq VoidStamp$
shows $val \neq UndefVal$
 $\langle proof \rangle$

lemma *valid-VoidStamp[elim]*:
shows *valid-value VoidStamp val* \implies
 $val = UndefVal$
 $\langle proof \rangle$

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

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

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

TODO: could we prove that expression evaluation never returns *UndefVal*?
 But this might require restricting unary and binary operators to be total...

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

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

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

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

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

lemma *int-stamp-implies-valid-value*:
 $[m,p] \vdash expr \mapsto val \implies$
valid-value (*stamp-expr* *expr*) *val*
 $\langle proof \rangle$

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

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

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*
 $\langle proof \rangle$

lemma *a0a*: (*BinaryExpr* *BinAdd* (*LeafExpr* 1 *default-stamp*) (*ConstantExpr* (*IntVal32* 0)))
 \leq (*LeafExpr* 1 *default-stamp*) (**is** ?*L* \leq ?*R*)
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

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) ≤ (UnaryExpr op e')
⟨proof⟩

```

lemma *mono-binary*:

```

assumes  $x \leq x'$ 
assumes  $y \leq y'$ 
shows (BinaryExpr op x y) ≤ (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) ≤ (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 $('a, 'b) \text{Heap} = 'a \Rightarrow 'b \Rightarrow \text{Value}$

type-synonym $\text{Free} = \text{nat}$

type-synonym $('a, 'b) \text{DynamicHeap} = ('a, 'b) \text{Heap} \times \text{Free}$

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

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

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

type-synonym $\text{FieldRefHeap} = (\text{string}, \text{objref}) \text{DynamicHeap}$

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

7.2 Intraprocedural Semantics

Intraprocedural semantics are given as a small-step semantics.

Within the context of a graph, the configuration triple, $(ID, \text{MethodState}, \text{Heap})$, is related to the subsequent configuration.

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

SequentialNode:

$\llbracket \text{is-sequential-node } (\text{kind } g \ nid);$
 $\quad \text{nid}' = (\text{successors-of } (\text{kind } g \ nid))!0 \rrbracket$
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m, h) \mid$

IfNode:

$\llbracket \text{kind } g \ nid = (\text{IfNode } \text{cond } tb \ fb);$
 $\quad g \vdash \text{cond} \triangleright \text{condE};$
 $\quad [m, p] \vdash \text{condE} \mapsto \text{val};$
 $\quad \text{nid}' = (\text{if } \text{val-to-bool } \text{val} \text{ then } tb \text{ else } fb) \rrbracket$
 $\implies g, p \vdash (nid, m, h) \rightarrow (nid', m, h) \mid$

EndNodes:

$\llbracket is-AbstractEndNode \ (kind \ g \ nid);$
 $\quad merge = any-usage \ g \ nid;$
 $\quad is-AbstractMergeNode \ (kind \ g \ merge);$

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

 $m' = set-phis \ phis \ vs \ m \rrbracket$
 $\implies g, p \vdash (nid, m, h) \rightarrow (merge, m', h) \mid$

NewInstanceNode:

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

LoadFieldNode:

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

SignedDivNode:

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

SignedRemNode:

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

StaticLoadFieldNode:

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

StoreFieldNode:

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

StaticStoreFieldNode:

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

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

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:

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

InvokeNodeStep:

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

$$\begin{aligned} & \text{callTarget} = \text{ir-callTarget } (\text{kind } g \text{ nid}); \\ & \text{kind } g \text{ callTarget} = (\text{MethodCallTargetNode } \text{targetMethod } \text{arguments}); \\ & \text{Some } \text{targetGraph} = P \text{ targetMethod}; \end{aligned}$$

$m' = \text{new-map-state};$
 $g \vdash \text{arguments} \triangleright_L \text{argsE};$
 $[m, p] \vdash \text{argsE} \mapsto_L p \llbracket$
 $\implies P \vdash ((g, \text{nid}, m, p) \# \text{stk}, h) \longrightarrow ((\text{targetGraph}, 0, m', p') \# (g, \text{nid}, m, p) \# \text{stk}, h)$

ReturnNode:

$\llbracket \text{kind } g \text{ nid} = (\text{ReturnNode } (\text{Some } \text{expr}) -);$
 $g \vdash \text{expr} \triangleright e;$
 $[m, p] \vdash e \mapsto v;$

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

ReturnNodeVoid:

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

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

UnwindNode:

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

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

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

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

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 - | - \longrightarrow^* - | -)

for P
where
 $\llbracket P \vdash (((g, \text{nid}, m, p) \# xs), h) \longrightarrow (((g', \text{nid}', m', p') \# ys), h'); \neg(\text{has-return } m') \rrbracket$
 $l' = (l @ [(g, \text{nid}, m, p)]);$
 $\text{exec } P (((g', \text{nid}', m', p') \# ys), h') \text{ } l' \text{ next-state } l'' \rrbracket$
 $\implies \text{exec } P (((g, \text{nid}, m, p) \# xs), h) \text{ } l \text{ next-state } l''$
 $|$
 $\llbracket P \vdash (((g, \text{nid}, m, p) \# xs), h) \longrightarrow (((g', \text{nid}', m', p') \# ys), h'); \text{has-return } m';$
 $l' = (l @ [(g, \text{nid}, m, p)]) \rrbracket$
 $\implies \text{exec } P (((g, \text{nid}, m, p) \# xs), h) \text{ } l \text{ } (((g', \text{nid}', m', p') \# ys), h') \text{ } l'$
code-pred ($\text{modes: } i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow \text{bool as } \text{Exec}$) $\text{exec } \langle \text{proof} \rangle$

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

7.4.1 Heap Testing

definition $p3 :: \text{Params}$ **where**

$p3 = [\text{IntVal32 } 3]$

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

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

$\text{field-sq} = \text{"sq"}$

definition $\text{eg3-sq} :: \text{IRGraph}$ **where**

```

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

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

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

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

end

```

8 Canonicalization Phase

```

theory CanonicalizationTree
imports
  Semantics.IRTreeEval
begin

```

```

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

```

```

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

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

```

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

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

is-neutral - - = False

fun *is-zero* :: *IRBinaryOp* \Rightarrow *Value* \Rightarrow *bool* **where**
is-zero BinMul (IntVal32 x) = (sint (x) = 0) |
is-zero BinMul (IntVal64 x) = (sint (x) = 0) |
is-zero - - = False

fun *int-to-value* :: *Value* \Rightarrow *int* \Rightarrow *Value* **where**
int-to-value (IntVal32 -) y = (IntVal32 (word-of-int y)) |
int-to-value (IntVal64 -) y = (IntVal64 (word-of-int y)) |
int-to-value - - = UndefVal

inductive *CanonicalizeBinaryOp* :: *IRExpr* \Rightarrow *IRExpr* \Rightarrow *bool* **where**
binary-const-fold:
 $\llbracket x = (\text{ConstantExpr } \text{val1});$
 $y = (\text{ConstantExpr } \text{val2});$
 $\text{val} = \text{bin-eval op val1 val2} \rrbracket$
 $\Rightarrow \text{CanonicalizeBinaryOp } (\text{BinaryExpr op } x y) (\text{ConstantExpr val}) \mid$

binary-fold-yneutral:
 $\llbracket y = (\text{ConstantExpr } c);$
 $\text{is-neutral op } c \rrbracket$
 $\Rightarrow \text{CanonicalizeBinaryOp } (\text{BinaryExpr op } x y) x \mid$

binary-fold-yzero:
 $\llbracket y = \text{ConstantExpr } c;$
 $\text{is-zero op } c;$
 $\text{zero} = (\text{int-to-value } c (\text{int } 0)) \rrbracket$
 $\Rightarrow \text{CanonicalizeBinaryOp } (\text{BinaryExpr op } x y) (\text{ConstantExpr zero})$

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

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

mul-negate32:
 $\llbracket y = \text{ConstantExpr } (\text{IntVal32 } (-1));$
 $\text{stamp-expr } x = \text{IntegerStamp } 32 \text{ lo hi} \rrbracket$

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

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

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

add-y-sub:

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

add-xnegate:

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

add-ynegate:

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

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

sub-same32:

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

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

sub-left-add1:

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

sub-left-add2:

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

sub-left-sub:

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

sub-right-add1:

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

sub-right-add2:

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

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

sub-right-sub:

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

sub-xzero32:

$\llbracket stamp_x = stamp_expr\ x;$
 $stamp_x = IntegerStamp\ 32\ lo\ hi$
 $\implies CanonicalizeSub\ (BinaryExpr\ BinSub\ (ConstantExpr\ (IntVal32\ 0))\ x)$
 $(UnaryExpr\ UnaryNeg\ x)\ |$

sub-xzero64:

$\llbracket stamp_x = stamp_expr\ x;$
 $stamp_x = IntegerStamp\ 64\ lo\ hi$
 $\implies CanonicalizeSub\ (BinaryExpr\ BinSub\ (ConstantExpr\ (IntVal64\ 0))\ x)$
 $(UnaryExpr\ UnaryNeg\ x)\ |$

sub-y-negate:

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

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

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

negate-sub:

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

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

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

abs-neg:

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

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

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

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

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

and-demorgans:

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

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:

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

$$\text{zero} = (\text{ConstantExpr (IntVal32 0)}) \rrbracket$$

$$\implies \text{CanonicalizeIntegerEquals } (\text{BinaryExpr BinIntegerEquals left } x) \ (\text{BinaryExpr BinIntegerEquals } y \ \text{zero}) \mid$$

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:

$$\llbracket c = (\text{BinaryExpr BinIntegerEquals } x \ y);$$

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

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

$$\text{is-IntegerStamp stamp } x \wedge \text{is-IntegerStamp stamp } y;$$

$$\text{stp-bits stamp } x = \text{stp-bits stamp } y \rrbracket$$

$$\implies \text{CanonicalizeConditional } (\text{ConditionalExpr } c \ x \ y) \ y \mid$$

condition-bounds-x:

$$\llbracket c = (\text{BinaryExpr BinIntegerLessThan } x \ y);$$

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

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

$$\text{stpi-upper stamp-}x \leq \text{stpi-lower stamp-}y \rrbracket$$

$$\implies \text{CanonicalizeConditional } (\text{ConditionalExpr } c \ x \ y) \ x \mid$$

condition-bounds-y:

$$\llbracket c = (\text{BinaryExpr BinIntegerLessThan } x \ y);$$

$stamp-x = stamp-expr\ x;$
 $stamp-y = stamp-expr\ y;$
 $stpi-upper\ stamp-x \leq stpi-lower\ stamp-y$
 $\implies CanonicalizeConditional\ (ConditionalExpr\ c\ y\ x)\ y\ |$

negate-condition:

$\llbracket nc = (UnaryExpr\ UnaryLogicNegation\ c);$
 $stampc = stamp-expr\ c;$
 $stampc = IntegerStamp\ 32\ lo\ hi$
 $\implies CanonicalizeConditional\ (ConditionalExpr\ nc\ x\ y)\ (ConditionalExpr\ c\ y\ x)$
 $|$

const-true:

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

const-false:

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

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

BinaryNode:
 $\llbracket CanonicalizeBinaryOp\ expr\ expr' \rrbracket$
 $\implies CanonicalizationStep\ expr\ expr'\ |$

UnaryNode:
 $\llbracket CanonicalizeUnaryOp\ expr\ expr' \rrbracket$
 $\implies CanonicalizationStep\ expr\ expr'\ |$

NegateNode:
 $\llbracket CanonicalizeNegate\ expr\ expr' \rrbracket$
 $\implies CanonicalizationStep\ expr\ expr'\ |$

NotNode:
 $\llbracket CanonicalizeNegate\ expr\ expr' \rrbracket$
 $\implies CanonicalizationStep\ expr\ expr'\ |$

```

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

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

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

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

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

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

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

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

code-pred (modes: i ⇒ o ⇒ bool) CanonicalizationStep ⟨proof⟩

end

```

9 Canonicalization Phase

```

theory CanonicalizationTreeProofs
  imports
    CanonicalizationTree

```

Semantics.IRTreeEvalThms
begin

lemma *mul-rewrite-helper*:
shows *valid-value* (*IntegerStamp* 32 lo hi) $x \implies \text{intval-mul } x \text{ (IntVal32 } (-1)) = \text{intval-negate } x$
and *valid-value* (*IntegerStamp* 64 lo hi) $x \implies \text{intval-mul } x \text{ (IntVal64 } (-1)) = \text{intval-negate } x$
 ⟨*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 $\text{intval-add } (\text{intval-sub } x \ y) \ y = x$
and $\text{intval-add } x \ (\text{intval-sub } y \ x) = y$
and $\text{intval-add } (\text{intval-negate } x) \ y = \text{intval-sub } y \ x$
and $\text{intval-add } x \ (\text{intval-negate } y) = \text{intval-sub } x \ y$
 ⟨*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 $\text{intval-sub } (\text{intval-add } x \ y) \ y = x$
and $\text{intval-sub } (\text{intval-add } x \ y) \ x = y$
and $\text{intval-sub } (\text{intval-sub } x \ y) \ x = \text{intval-negate } y$
and $\text{intval-sub } x \ (\text{intval-add } x \ y) = \text{intval-negate } y$
and $\text{intval-sub } y \ (\text{intval-add } x \ y) = \text{intval-negate } x$
and $\text{intval-sub } x \ (\text{intval-sub } x \ y) = y$
and $\text{intval-sub } x \ (\text{intval-negate } y) = \text{intval-add } x \ y$
 $\langle \text{proof} \rangle$

lemma *sub-single-rewrites-helper*:
assumes $\text{valid-value } (\text{IntegerStamp } b \ \text{lox } hix) \ x$
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 $\text{CanonicalizeSub } \text{before } \text{after}$
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 $\text{CanonicalizeNegate } \text{before } \text{after}$
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 *abs-helper*:
assumes $\exists \ v1. \ x = \text{IntVal32 } (v1)$
shows $v1 < s \ 0 \implies \text{intval-abs } x = \text{IntVal32 } (-v1)$

and $\neg(v1 <_s 0) \implies \text{intval-abs } x = \text{IntVal32 } (v1)$
 $\langle \text{proof} \rangle$

lemma *abs-helper2*:

assumes $\exists v1. x = \text{IntVal64 } (v1)$
 shows $v1 <_s 0 \implies \text{intval-abs } x = \text{IntVal64 } (-v1)$
 and $\neg(v1 <_s 0) \implies \text{intval-abs } x = \text{IntVal64 } (v1)$
 $\langle \text{proof} \rangle$

lemma *abs-rewrite-helper*:

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

lemma *CanonicalizeAbsProof*:

assumes *CanonicalizeAbs* *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 *not-rewrite-helper*:

assumes *valid-value* (*IntegerStamp* *b lox hix*) *x*
 shows $\text{intval-not } (\text{intval-not } x) = x$
 $\langle \text{proof} \rangle$

lemma *CanonicalizeNotProof*:

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

lemma *demorgans-rewrites-helper*:

assumes *valid-value* (*IntegerStamp* *b lox hix*) *x*
 and *valid-value* (*IntegerStamp* *b loy hiy*) *y*

 shows $\text{intval-and } (\text{intval-not } x) (\text{intval-not } y) = \text{intval-not } (\text{intval-or } x y)$
 and $\text{intval-or } (\text{intval-not } x) (\text{intval-not } y) = \text{intval-not } (\text{intval-and } x y)$
 and $x = y \implies \text{intval-and } x y = x$
 and $x = y \implies \text{intval-or } x y = x$
 $\langle \text{proof} \rangle$

lemma *CanonicalizeAndProof*:

assumes *CanonicalizeAnd* *before after*
 assumes $[m, p] \vdash \text{before} \mapsto \text{res}$

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

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

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