

Veriopt Theories

March 18, 2022

Contents

1	Runtime Values and Arithmetic	1
2	Nodes	7
2.1	Types of Nodes	7
2.2	Hierarchy of Nodes	15
3	Stamp Typing	22
4	Graph Representation	25
4.0.1	Example Graphs	30
4.1	Control-flow Graph Traversal	31
4.2	Structural Graph Comparison	33

1 Runtime Values and Arithmetic

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

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

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

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

```

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

```

```

type-synonym objref = nat option

```

```

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

  ObjRef objref |
  ObjStr string

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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

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

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

```

fun intval-add :: Value ⇒ Value ⇒ Value where

```

```

    intval-add (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1+v2)) |
    intval-add (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1+v2)) |
    intval-add - - =.UndefVal

instantiation Value :: ab-semigroup-add
begin

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

print-locale! ab-semigroup-add

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

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

instantiation Value :: minus
begin

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

instance proof qed
end

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

instantiation Value :: times
begin

```

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

```
instance proof qed
end
```

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

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

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

```
instance proof qed
end
```

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

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

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

```
instance proof qed
end
```

```
context
  includes bit-operations-syntax
begin
```

```

fun intval-and :: Value  $\Rightarrow$  Value  $\Rightarrow$  Value 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 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 where
  intval-xor (IntVal32 v1) (IntVal32 v2) = (IntVal32 (v1 XOR v2)) |
  intval-xor (IntVal64 v1) (IntVal64 v2) = (IntVal64 (v1 XOR v2)) |
  intval-xor - - = UndefVal

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

fun intval-logic-negation :: Value  $\Rightarrow$  Value where

```

```

    intval-logic-negation (IntVal32 v) = (if v = 0 then (IntVal32 1) else (IntVal32
0)) |
    intval-logic-negation (IntVal64 v) = (if v = 0 then (IntVal64 1) else (IntVal64
0)) |
    intval-logic-negation - =.UndefVal

```

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

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

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

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

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

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

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

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

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

value (128 :: 8 word) >> 2

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

```

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

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

end

```

```

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

```

```

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

```

```

lemma intval-add (IntVal32 ( $2^{31}-1$ )) (IntVal32 ( $2^{31}-1$ )) = IntVal32 ( $-2$ )
  by eval
lemma intval-add (IntVal64 ( $2^{31}-1$ )) (IntVal64 ( $2^{31}-1$ )) = IntVal64 4294967294
  by eval

end

```

2 Nodes

2.1 Types of Nodes

```

theory IRNodes
  imports
    Values
begin

```

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

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

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

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

```

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

```

```

datatype (discs-sels) IRNode =
  AbsNode (ir-value: INPUT)
  | AddNode (ir-x: INPUT) (ir-y: INPUT)
  | AndNode (ir-x: INPUT) (ir-y: INPUT)
  | BeginNode (ir-next: SUCC)
  | BytecodeExceptionNode (ir-arguments: INPUT list) (ir-stateAfter-opt: INPUT-STATE
option) (ir-next: SUCC)
  | ConditionalNode (ir-condition: INPUT-COND) (ir-trueValue: INPUT) (ir-falseValue:
INPUT)
  | ConstantNode (ir-const: Value)
  | DynamicNewArrayNode (ir-elementType: INPUT) (ir-length: INPUT) (ir-voidClass-opt:
INPUT option) (ir-stateBefore-opt: INPUT-STATE option) (ir-next: SUCC)
  | EndNode
  | ExceptionObjectNode (ir-stateAfter-opt: INPUT-STATE option) (ir-next: SUCC)

  | FrameState (ir-monitorIds: INPUT-ASSOC list) (ir-outerFrameState-opt: IN-
PUT-STATE option) (ir-values-opt: INPUT list option) (ir-virtualObjectMappings-opt:
INPUT-STATE list option)
  | IfNode (ir-condition: INPUT-COND) (ir-trueSuccessor: SUCC) (ir-falseSuccessor:
SUCC)
  | IntegerBelowNode (ir-x: INPUT) (ir-y: INPUT)
  | IntegerEqualsNode (ir-x: INPUT) (ir-y: INPUT)
  | IntegerLessThanNode (ir-x: INPUT) (ir-y: INPUT)
  | InvokeNode (ir-nid: ID) (ir-callTarget: INPUT-EXT) (ir-classInit-opt: IN-

```


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

```

| XorNode (ir-x: INPUT) (ir-y: INPUT)
| ZeroExtendNode (ir-inputBits: nat) (ir-resultBits: nat) (ir-value: INPUT)
| NoNode

```

```

| RefNode (ir-ref:ID)

```

```

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

```

```

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

```

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

```

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

```

virtualObjectMappings) |
inputs-of-IfNode:
inputs-of (IfNode condition trueSuccessor falseSuccessor) = [condition] |
inputs-of-IntegerBelowNode:
inputs-of (IntegerBelowNode x y) = [x, y] |
inputs-of-IntegerEqualsNode:
inputs-of (IntegerEqualsNode x y) = [x, y] |
inputs-of-IntegerLessThanNode:
inputs-of (IntegerLessThanNode x y) = [x, y] |
inputs-of-InvokeNode:
inputs-of (InvokeNode nid0 callTarget classInit stateDuring stateAfter next)
= callTarget # (opt-to-list classInit) @ (opt-to-list stateDuring) @ (opt-to-list
stateAfter) |
inputs-of-InvokeWithExceptionNode:
inputs-of (InvokeWithExceptionNode nid0 callTarget classInit stateDuring stateAfter
next exceptionEdge) = callTarget # (opt-to-list classInit) @ (opt-to-list stateDur-
ing) @ (opt-to-list stateAfter) |
inputs-of-IsNullNode:
inputs-of (IsNullNode value) = [value] |
inputs-of-KillingBeginNode:
inputs-of (KillingBeginNode next) = [] |
inputs-of-LeftShiftNode:
inputs-of (LeftShiftNode x y) = [x, y] |
inputs-of-LoadFieldNode:
inputs-of (LoadFieldNode nid0 field object next) = (opt-to-list object) |
inputs-of-LogicNegationNode:
inputs-of (LogicNegationNode value) = [value] |
inputs-of-LoopBeginNode:
inputs-of (LoopBeginNode ends overflowGuard stateAfter next) = ends @ (opt-to-list
overflowGuard) @ (opt-to-list stateAfter) |
inputs-of-LoopEndNode:
inputs-of (LoopEndNode loopBegin) = [loopBegin] |
inputs-of-LoopExitNode:
inputs-of (LoopExitNode loopBegin stateAfter next) = loopBegin # (opt-to-list
stateAfter) |
inputs-of-MergeNode:
inputs-of (MergeNode ends stateAfter next) = ends @ (opt-to-list stateAfter) |
inputs-of-MethodCallTargetNode:
inputs-of (MethodCallTargetNode targetMethod arguments) = arguments |
inputs-of-MulNode:
inputs-of (MulNode x y) = [x, y] |
inputs-of-NarrowNode:
inputs-of (NarrowNode inputBits resultBits value) = [value] |
inputs-of-NegateNode:
inputs-of (NegateNode value) = [value] |
inputs-of-NewArrayNode:
inputs-of (NewArrayNode length0 stateBefore next) = length0 # (opt-to-list state-
Before) |
inputs-of-NewInstanceNode:

inputs-of (*NewInstanceNode* *nid0* *instanceClass* *stateBefore* *next*) = (*opt-to-list* *stateBefore*) |
inputs-of-NotNode:
inputs-of (*NotNode* *value*) = [*value*] |
inputs-of-OrNode:
inputs-of (*OrNode* *x* *y*) = [*x*, *y*] |
inputs-of-ParameterNode:
inputs-of (*ParameterNode* *index*) = [] |
inputs-of-PiNode:
inputs-of (*PiNode* *object* *guard*) = *object* # (*opt-to-list* *guard*) |
inputs-of-ReturnNode:
inputs-of (*ReturnNode* *result* *memoryMap*) = (*opt-to-list* *result*) @ (*opt-to-list* *memoryMap*) |
inputs-of-RightShiftNode:
inputs-of (*RightShiftNode* *x* *y*) = [*x*, *y*] |
inputs-of-ShortCircuitOrNode:
inputs-of (*ShortCircuitOrNode* *x* *y*) = [*x*, *y*] |
inputs-of-SignExtendNode:
inputs-of (*SignExtendNode* *inputBits* *resultBits* *value*) = [*value*] |
inputs-of-SignedDivNode:
inputs-of (*SignedDivNode* *nid0* *x* *y* *zeroCheck* *stateBefore* *next*) = [*x*, *y*] @ (*opt-to-list* *zeroCheck*) @ (*opt-to-list* *stateBefore*) |
inputs-of-SignedRemNode:
inputs-of (*SignedRemNode* *nid0* *x* *y* *zeroCheck* *stateBefore* *next*) = [*x*, *y*] @ (*opt-to-list* *zeroCheck*) @ (*opt-to-list* *stateBefore*) |
inputs-of-StartNode:
inputs-of (*StartNode* *stateAfter* *next*) = (*opt-to-list* *stateAfter*) |
inputs-of-StoreFieldNode:
inputs-of (*StoreFieldNode* *nid0* *field* *value* *stateAfter* *object* *next*) = *value* # (*opt-to-list* *stateAfter*) @ (*opt-to-list* *object*) |
inputs-of-SubNode:
inputs-of (*SubNode* *x* *y*) = [*x*, *y*] |
inputs-of-UnsignedRightShiftNode:
inputs-of (*UnsignedRightShiftNode* *x* *y*) = [*x*, *y*] |
inputs-of-UnwindNode:
inputs-of (*UnwindNode* *exception*) = [*exception*] |
inputs-of-ValuePhiNode:
inputs-of (*ValuePhiNode* *nid0* *values* *merge*) = *merge* # *values* |
inputs-of-ValueProxyNode:
inputs-of (*ValueProxyNode* *value* *loopExit*) = [*value*, *loopExit*] |
inputs-of-XorNode:
inputs-of (*XorNode* *x* *y*) = [*x*, *y*] |
inputs-of-ZeroExtendNode:
inputs-of (*ZeroExtendNode* *inputBits* *resultBits* *value*) = [*value*] |
inputs-of-NoNode: *inputs-of* (*NoNode*) = [] |

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

```

fun successors-of :: IRNode ⇒ ID list where
  successors-of-AbsNode:
    successors-of (AbsNode value) = [] |
  successors-of-AddNode:
    successors-of (AddNode x y) = [] |
  successors-of-AndNode:
    successors-of (AndNode x y) = [] |
  successors-of-BeginNode:
    successors-of (BeginNode next) = [next] |
  successors-of-BytecodeExceptionNode:
    successors-of (BytecodeExceptionNode arguments stateAfter next) = [next] |
  successors-of-ConditionalNode:
    successors-of (ConditionalNode condition trueValue falseValue) = [] |
  successors-of-ConstantNode:
    successors-of (ConstantNode const) = [] |
  successors-of-DynamicNewArrayNode:
    successors-of (DynamicNewArrayNode elementType length0 voidClass stateBefore
next) = [next] |
  successors-of-EndNode:
    successors-of (EndNode) = [] |
  successors-of-ExceptionObjectNode:
    successors-of (ExceptionObjectNode stateAfter next) = [next] |
  successors-of-FrameState:
    successors-of (FrameState monitorIds outerFrameState values virtualObjectMap-
pings) = [] |
  successors-of-IfNode:
    successors-of (IfNode condition trueSuccessor falseSuccessor) = [trueSuccessor,
falseSuccessor] |
  successors-of-IntegerBelowNode:
    successors-of (IntegerBelowNode x y) = [] |
  successors-of-IntegerEqualsNode:
    successors-of (IntegerEqualsNode x y) = [] |
  successors-of-IntegerLessThanNode:
    successors-of (IntegerLessThanNode x y) = [] |
  successors-of-InvokeNode:
    successors-of (InvokeNode nid0 callTarget classInit stateDuring stateAfter next)
= [next] |
  successors-of-InvokeWithExceptionNode:
    successors-of (InvokeWithExceptionNode nid0 callTarget classInit stateDuring
stateAfter next exceptionEdge) = [next, exceptionEdge] |
  successors-of-IsNullNode:
    successors-of (IsNullNode value) = [] |
  successors-of-KillingBeginNode:
    successors-of (KillingBeginNode next) = [next] |
  successors-of-LeftShiftNode:
    successors-of (LeftShiftNode x y) = [] |
  successors-of-LoadFieldNode:
    successors-of (LoadFieldNode nid0 field object next) = [next] |

```

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

```

successors-of (UnsignedRightShiftNode x y) = [] |
successors-of UnwindNode:
successors-of (UnwindNode exception) = [] |
successors-of ValuePhiNode:
successors-of (ValuePhiNode nid0 values merge) = [] |
successors-of ValueProxyNode:
successors-of (ValueProxyNode value loopExit) = [] |
successors-of XorNode:
successors-of (XorNode x y) = [] |
successors-of ZeroExtendNode:
successors-of (ZeroExtendNode inputBits resultBits value) = [] |
successors-of NoNode: successors-of (NoNode) = [] |

```

```

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

```

```

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

```

```

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

```

```

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

```

end

2.2 Hierarchy of Nodes

```

theory IRNodeHierarchy
imports IRNodes
begin

```

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

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

These functions have been automatically generated from the compiler.

```

fun is-EndNode :: IRNode  $\Rightarrow$  bool where
  is-EndNode EndNode = True |
  is-EndNode - = False

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

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

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

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

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

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

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

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

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

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

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

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

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

```



```

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

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

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

fun is-FloatingNode :: IRNode  $\Rightarrow$  bool where
  is-FloatingNode n = ((is-AbstractLocalNode n)  $\vee$  (is-BinaryNode n)  $\vee$  (is-ConditionalNode
    n)  $\vee$  (is-ConstantNode n)  $\vee$  (is-FloatingGuardedNode n)  $\vee$  (is-LogicNode n)  $\vee$ 
    (is-PhiNode n)  $\vee$  (is-ProxyNode n)  $\vee$  (is-UnaryNode n))

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

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

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

```

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

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

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

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

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

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

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

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

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

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

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

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

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

```

is-ArithmeticLIRLowerable *n* = ((*is-AbsNode* *n*) ∨ (*is-BinaryArithmeticNode* *n*) ∨ (*is-IntegerConvertNode* *n*) ∨ (*is-NotNode* *n*) ∨ (*is-ShiftNode* *n*) ∨ (*is-UnaryArithmeticNode* *n*))

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

```

```

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

```

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

```

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

```

```

((is-LoopBeginNode n1) ∧ (is-LoopBeginNode n2)) ∨
((is-LoopEndNode n1) ∧ (is-LoopEndNode n2)) ∨
((is-LoopExitNode n1) ∧ (is-LoopExitNode n2)) ∨
((is-MergeNode n1) ∧ (is-MergeNode n2)) ∨
((is-MethodCallTargetNode n1) ∧ (is-MethodCallTargetNode n2)) ∨
((is-MulNode n1) ∧ (is-MulNode n2)) ∨
((is-NegateNode n1) ∧ (is-NegateNode n2)) ∨
((is-NewArrayNode n1) ∧ (is-NewArrayNode n2)) ∨
((is-NewInstanceNode n1) ∧ (is-NewInstanceNode n2)) ∨
((is-NotNode n1) ∧ (is-NotNode n2)) ∨
((is-OrNode n1) ∧ (is-OrNode n2)) ∨
((is-ParameterNode n1) ∧ (is-ParameterNode n2)) ∨
((is-PiNode n1) ∧ (is-PiNode n2)) ∨
((is-ReturnNode n1) ∧ (is-ReturnNode n2)) ∨
((is-ShortCircuitOrNode n1) ∧ (is-ShortCircuitOrNode n2)) ∨
((is-SignedDivNode n1) ∧ (is-SignedDivNode n2)) ∨
((is-StartNode n1) ∧ (is-StartNode n2)) ∨
((is-StoreFieldNode n1) ∧ (is-StoreFieldNode n2)) ∨
((is-SubNode n1) ∧ (is-SubNode n2)) ∨
((is-UnwindNode n1) ∧ (is-UnwindNode n2)) ∨
((is-ValuePhiNode n1) ∧ (is-ValuePhiNode n2)) ∨
((is-ValueProxyNode n1) ∧ (is-ValueProxyNode n2)) ∨
((is-XorNode n1) ∧ (is-XorNode n2))

```

end

3 Stamp Typing

```

theory Stamp
  imports Values
begin

```

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

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

```

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

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

```

```
| RawPointerStamp (stp-nonNull: bool) (stp-alwaysNull: bool)
| IllegalStamp
```

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

— A stamp which includes the full range of the type

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

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

```
fun is-stamp-unrestricted :: Stamp ⇒ bool where
  is-stamp-unrestricted s = (s = unrestricted-stamp s)
```

— A stamp which provides type information but has an empty range of values

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

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

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

  is-stamp-empty x = False
```

— Calculate the meet stamp of two stamps

```
fun meet :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  Stamp where
  meet VoidStamp VoidStamp = VoidStamp |
  meet (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1  $\neq$  b2 then IllegalStamp else
    (IntegerStamp b1 (min l1 l2) (max u1 u2))
  ) |

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

— Calculate the join stamp of two stamps

```
fun join :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  Stamp where
  join VoidStamp VoidStamp = VoidStamp |
  join (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1  $\neq$  b2 then IllegalStamp else
    (IntegerStamp b1 (max l1 l2) (min u1 u2))
  ) |

  join (KlassPointerStamp nn1 an1) (KlassPointerStamp nn2 an2) = (
    if ((nn1  $\vee$  nn2)  $\wedge$  (an1  $\vee$  an2))
    then (empty-stamp (KlassPointerStamp nn1 an1))
    else (KlassPointerStamp (nn1  $\vee$  nn2) (an1  $\vee$  an2))
  ) |
  join (MethodCountersPointerStamp nn1 an1) (MethodCountersPointerStamp nn2
an2) = (
    if ((nn1  $\vee$  nn2)  $\wedge$  (an1  $\vee$  an2))
    then (empty-stamp (MethodCountersPointerStamp nn1 an1))
    else (MethodCountersPointerStamp (nn1  $\vee$  nn2) (an1  $\vee$  an2))
  ) |
  join (MethodPointersStamp nn1 an1) (MethodPointersStamp nn2 an2) = (
    if ((nn1  $\vee$  nn2)  $\wedge$  (an1  $\vee$  an2))
    then (empty-stamp (MethodPointersStamp nn1 an1))
    else (MethodPointersStamp (nn1  $\vee$  nn2) (an1  $\vee$  an2))
  ) |
  join s1 s2 = IllegalStamp
```

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


```

fun asConstant :: Stamp  $\Rightarrow$  Value where
  asConstant (IntegerStamp b l h) = (if l = h then IntVal64 (word-of-int l) else
```

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

```

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

```

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

```

fun neverDistinct :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  bool where
  neverDistinct stamp1 stamp2 = (asConstant stamp1 = asConstant stamp2  $\wedge$ 
asConstant stamp1  $\neq$  UndefVal)

```

```

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

  constantAsStamp - = IllegalStamp

```

— Define when a runtime value is valid for a stamp

```

fun valid-value :: Value  $\Rightarrow$  Stamp  $\Rightarrow$  bool where
  valid-value (IntVal32 v) (IntegerStamp b l h) = (b=32  $\wedge$  (sint v  $\geq$  l)  $\wedge$  (sint v  $\leq$ 
h)) |
  valid-value (IntVal64 v) (IntegerStamp b l h) = (b=64  $\wedge$  (sint v  $\geq$  l)  $\wedge$  (sint v  $\leq$ 
h)) |

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

```

```

fun compatible :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  bool where
  compatible (IntegerStamp b1 - -) (IntegerStamp b2 - -) = (b1 = b2) |
  compatible (VoidStamp) (VoidStamp) = True |
  compatible - - = False

```

```

fun stamp-under :: Stamp  $\Rightarrow$  Stamp  $\Rightarrow$  bool where
  stamp-under x y = ((stpi-upper x) < (stpi-lower y))

```

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

```

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

```

```

end

```

4 Graph Representation

```

theory IRGraph
  imports
    IRNodeHierarchy
    Stamp
    HOL-Library.FSet
    HOL.Relation
begin

```

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

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

```

typedef IRGraph = {g :: ID  $\rightarrow$  (IRNode  $\times$  Stamp) . finite (dom g)}
proof -
  have finite(dom(Map.empty))  $\wedge$  ran Map.empty = {} by auto
  then show ?thesis
    by fastforce
qed

```

setup-lifting *type-definition-IRGraph*

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

fun no-node :: (ID  $\times$  (IRNode  $\times$  Stamp)) list  $\Rightarrow$  (ID  $\times$  (IRNode  $\times$  Stamp)) list
where

```

$no-node\ g = filter\ (\lambda n. fst\ (snd\ n) \neq NoNode)\ g$

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

definition $as-set :: IRGraph \Rightarrow (ID \times (IRNode \times Stamp))\ set$ **where**
 $as-set\ g = \{(n, kind\ g\ n, stamp\ g\ n) \mid n . n \in ids\ g\}$

definition $true-ids :: IRGraph \Rightarrow ID\ set$ **where**
 $true-ids\ g = ids\ g - \{n \in ids\ g. \exists n'. kind\ g\ n = RefNode\ n'\}$

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

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

code-datatype $irgraph$

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

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

lemma $dom-eq$:
assumes $\forall x \in set\ xs. fst\ (snd\ x) \neq NoNode$
shows $filter-none\ (map-of\ xs) = dom\ (map-of\ xs)$
unfolding $filter-none.simps$ **using** $assms\ map-of-SomeD$
by $fastforce$

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

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

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

— Get the inputs set of a given node ID
fun *inputs* :: *IRGraph* \Rightarrow *ID* \Rightarrow *ID set* **where**
inputs g nid = *set (inputs-of (kind g nid))*

— Get the successor set of a given node ID
fun *succ* :: *IRGraph* \Rightarrow *ID* \Rightarrow *ID set* **where**
succ g nid = *set (successors-of (kind g nid))*

— Gives a relation between node IDs - between a node and its input nodes
fun *input-edges* :: *IRGraph* \Rightarrow *ID rel* **where**
input-edges g = $(\bigcup i \in \text{ids } g. \{(i,j) | j \in (\text{inputs } g \ i)\})$

— Find all the nodes in the graph that have nid as an input - the usages of nid
fun *usages* :: *IRGraph* \Rightarrow *ID* \Rightarrow *ID set* **where**
usages g nid = $\{j. j \in \text{ids } g \wedge (j, \text{nid}) \in \text{input-edges } g\}$

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

fun *predecessors* :: *IRGraph* \Rightarrow *ID* \Rightarrow *ID set* **where**
predecessors g nid = $\{j. j \in \text{ids } g \wedge (j, \text{nid}) \in \text{successor-edges } g\}$

fun *nodes-of* :: *IRGraph* \Rightarrow (*IRNode* \Rightarrow *bool*) \Rightarrow *ID set* **where**
nodes-of g sel = $\{\text{nid} \in \text{ids } g. \text{sel } (\text{kind } g \ \text{nid})\}$

fun *edge* :: (*IRNode* \Rightarrow 'a) \Rightarrow *ID* \Rightarrow *IRGraph* \Rightarrow 'a **where**
edge sel nid g = *sel (kind g nid)*

fun *filtered-inputs* :: *IRGraph* \Rightarrow *ID* \Rightarrow (*IRNode* \Rightarrow *bool*) \Rightarrow *ID list* **where**
filtered-inputs g nid f = *filter (f \circ (kind g)) (inputs-of (kind g nid))*

fun *filtered-successors* :: *IRGraph* \Rightarrow *ID* \Rightarrow (*IRNode* \Rightarrow *bool*) \Rightarrow *ID list* **where**
filtered-successors g nid f = *filter (f \circ (kind g)) (successors-of (kind g nid))*

fun *filtered-usages* :: *IRGraph* \Rightarrow *ID* \Rightarrow (*IRNode* \Rightarrow *bool*) \Rightarrow *ID set* **where**
filtered-usages g nid f = $\{n \in (\text{usages } g \ \text{nid}). f (\text{kind } g \ n)\}$

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

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 –
have *that*: $x \in \text{ids } g \longrightarrow \text{kind } g \ x \neq \text{NoNode}$
using *ids.rep-eq kind.rep-eq* **by** *force*
have $\text{kind } g \ x \neq \text{NoNode} \longrightarrow x \in \text{ids } g$
unfolding *with-default.simps kind-def ids-def*
by (*cases Rep-IRGraph g x = None; auto*)
from this that show *?thesis* **by** *auto*
qed

lemma *not-in-g*:
assumes $\text{nid} \notin \text{ids } g$
shows $\text{kind } g \ \text{nid} = \text{NoNode}$
using *assms ids-some* **by** *blast*

```

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

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

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

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

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

lemma [simp]: finite (dom g)  $\longrightarrow$  stamp (Abs-IRGraph g) = ( $\lambda x$  . (case g x of
None  $\Rightarrow$  IllegalStamp | Some n  $\Rightarrow$  snd n))
  using stamp.abs-eq stamp.rep-eq by auto

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

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

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

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

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

```

place-node.abs-eq replace-node-def snd-eqD)
qed

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

lemma *add-node-lookup*:
 $gup = \text{add-node } nid \ (k, s) \ g \longrightarrow$
 (if $k \neq \text{NoNode}$ then $\text{kind } gup \ nid = k \wedge \text{stamp } gup \ nid = s$ else $\text{kind } gup \ nid = \text{kind } g \ nid$)
proof (*cases k = NoNode*)
case *True*
then show *?thesis*
by (*simp add: add-node.rep-eq kind.rep-eq*)
next
case *False*
then show *?thesis*
by (*simp add: kind.rep-eq add-node.rep-eq stamp.rep-eq*)
qed

lemma *remove-node-lookup*:
 $gup = \text{remove-node } nid \ g \longrightarrow \text{kind } gup \ nid = \text{NoNode} \wedge \text{stamp } gup \ nid = \text{IllegalStamp}$
by (*simp add: kind.rep-eq remove-node.rep-eq stamp.rep-eq*)

lemma *replace-node-lookup[simp]*:
 $gup = \text{replace-node } nid \ (k, s) \ g \wedge k \neq \text{NoNode} \longrightarrow \text{kind } gup \ nid = k \wedge \text{stamp } gup \ nid = s$
by (*simp add: replace-node.rep-eq kind.rep-eq stamp.rep-eq*)

lemma *replace-node-unchanged*:
 $gup = \text{replace-node } nid \ (k, s) \ g \longrightarrow (\forall \ n \in (\text{ids } g - \{nid\}) . n \in \text{ids } g \wedge n \in \text{ids } gup \wedge \text{kind } g \ n = \text{kind } gup \ n)$
by (*simp add: kind.rep-eq replace-node.rep-eq*)

4.0.1 Example Graphs

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

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

Example 2: public static int sq(int x) return x * x;
 [1 P(0)] / [0 Start] [4 *] | / V / [5 Return]

definition *eg2-sq* :: *IRGraph* **where**
eg2-sq = *irgraph* [
 (0, *StartNode* *None* 5, *VoidStamp*),

```

    (1, ParameterNode 0, default-stamp),
    (4, MulNode 1 1, default-stamp),
    (5, ReturnNode (Some 4) None, default-stamp)
  ]

```

```

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

```

```

end

```

4.1 Control-flow Graph Traversal

```

theory
  Traversal
imports
  IRGraph
begin

```

```

type-synonym Seen = ID set

```

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

```

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

```

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

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

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

```

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

```

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

type-synonym $'a \text{ TraversalState} = (ID \times Seen \times 'a)$

inductive Step

$:: ('a \text{ TraversalState} \Rightarrow 'a) \Rightarrow IRGraph \Rightarrow 'a \text{ TraversalState} \Rightarrow 'a \text{ TraversalState} \text{ option} \Rightarrow bool$

for $sa \ g$ **where**

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

$\llbracket kind \ g \ nid = \text{BeginNode } nid';$

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

$Some \ ifcond = pred \ g \ nid;$
 $kind \ g \ ifcond = \text{IfNode } cond \ t \ f;$

$analysis' = sa \ (nid, seen, analysis) \rrbracket$
 $\implies Step \ sa \ g \ (nid, seen, analysis) \ (Some \ (nid', seen', analysis')) \mid$

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

$\llbracket kind \ g \ nid = \text{EndNode};$

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

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

$analysis' = sa \ (nid, seen, analysis) \rrbracket$
 $\implies Step \ sa \ g \ (nid, seen, analysis) \ (Some \ (nid', seen', analysis')) \mid$

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

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

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

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

$analysis' = sa \ (nid, seen, analysis) \rrbracket$
 $\implies Step \ sa \ g \ (nid, seen, analysis) \ (Some \ (nid', seen', analysis')) \mid$

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

```

[[¬(is-EndNode (kind g nid));
  ¬(is-BEGINNode (kind g nid));

  nid ∉ seen;
  seen' = {nid} ∪ seen;

  None = nextEdge seen' nid g]]
⇒ Step sa g (nid, seen, analysis) None |

— We've already seen this node, give back None
[[nid ∈ seen]] ⇒ Step sa g (nid, seen, analysis) None

```

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

end

4.2 Structural Graph Comparison

theory

Comparison

imports

IRGraph

begin

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

fun *find-ref-nodes* :: *IRGraph* \Rightarrow (*ID* \rightarrow *ID*) **where**
find-ref-nodes *g* = *map-of*
 (*map* ($\lambda n. (n, \text{ir-ref } (\text{kind } g \ n))$) (*filter* ($\lambda id. \text{is-RefNode } (\text{kind } g \ id)$) (*sorted-list-of-set*
 (*ids* *g*))))

fun *replace-ref-nodes* :: *IRGraph* \Rightarrow (*ID* \rightarrow *ID*) \Rightarrow *ID list* \Rightarrow *ID list* **where**
replace-ref-nodes *g m xs* = *map* ($\lambda id. (\text{case } (m \ id) \text{ of } \text{Some } other \Rightarrow other \mid \text{None} \Rightarrow id)$) *xs*

fun *find-next* :: *ID list* \Rightarrow *ID set* \Rightarrow *ID option* **where**
find-next to-see seen = (*let* *l* = (*filter* ($\lambda nid. nid \notin \text{seen}$) *to-see*)
 in (*case* *l* of [] \Rightarrow None | *xs* \Rightarrow Some (*hd* *xs*)))

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

seen'

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

inductive *nodeEq* :: (*ID* \rightarrow *ID*) \Rightarrow *IRGraph* \Rightarrow *ID* \Rightarrow *IRGraph* \Rightarrow *ID* \Rightarrow *bool*
where
 $\llbracket \text{kind } g1 \text{ } n1 = \text{RefNode } ref; \text{nodeEq } m \text{ } g1 \text{ } ref \text{ } g2 \text{ } n2 \rrbracket \implies \text{nodeEq } m \text{ } g1 \text{ } n1 \text{ } g2 \text{ } n2 \mid$
 $\llbracket x = \text{kind } g1 \text{ } n1;$
 $y = \text{kind } g2 \text{ } n2;$
 $\text{is-same-ir-node-type } x \text{ } y;$
 $\text{replace-ref-nodes } g1 \text{ } m \text{ } (\text{successors-of } x) = \text{successors-of } y;$
 $\text{replace-ref-nodes } g1 \text{ } m \text{ } (\text{inputs-of } x) = \text{inputs-of } y \rrbracket$
 $\implies \text{nodeEq } m \text{ } g1 \text{ } n1 \text{ } g2 \text{ } n2$

code-pred [*show-modes*] *nodeEq* .

fun *diffNodesGraph* :: *IRGraph* \Rightarrow *IRGraph* \Rightarrow *ID* *set* **where**
diffNodesGraph *g1* *g2* = (let *refNodes* = *find-ref-nodes* *g1* in
 $\{ n . n \in \text{Predicate.the } (\text{reachables-i-i-i-o } g1 \text{ } [0] \text{ } \{\}) \} \wedge (\text{case } refNodes \text{ } n \text{ of Some } - \Rightarrow \text{False} \mid - \Rightarrow \text{True}) \wedge \neg(\text{nodeEq } refNodes \text{ } g1 \text{ } n \text{ } g2 \text{ } n) \}$)

fun *diffNodesInfo* :: *IRGraph* \Rightarrow *IRGraph* \Rightarrow (*ID* \times *IRNode* \times *IRNode*) *set* **where**
diffNodesInfo *g1* *g2* = $\{(nid, \text{kind } g1 \text{ } nid, \text{kind } g2 \text{ } nid) \mid nid . nid \in \text{diffNodesGraph } g1 \text{ } g2\}$

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

end