Veriopt Theories

July 13, 2022

Contents

1 Runtime Values and Arithmetic **1**
- 1.1 Arithmetic Operators 3
- 1.2 Bitwise Operators and Comparisons 5
- 1.3 Narrowing and Widening Operators 6
- 1.4 Bit-Shifting Operators 9

2 Examples of Narrowing / Widening Functions **11**

3 Nodes **12**
- 3.1 Types of Nodes 12
- 3.2 Hierarchy of Nodes 20

4 Stamp Typing **27**

5 Graph Representation **31**
  - 5.0.1 Example Graphs 35
- 5.1 Control-flow Graph Traversal 36
- 5.2 Structural Graph Comparison 38

1 Runtime Values and Arithmetic

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

lemma $-((x::float)-y) = (y-x)$
  by *simp*

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 sign-extended 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

abbreviation *valid-int-widths* :: *nat set* **where**
  valid-int-widths ≡ {*1*, *8*, *16*, *32*, *64*}

type-synonym *objref* = *nat option*

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

  ObjRef objref |
  ObjStr string

Characterise integer values, covering both 32 and 64 bit. If a node has a stamp smaller than 32 bits (16, 8, or 1 bit), then the value will be sign-extended to 32 bits. This is necessary to match what the stamps specify E.g. an 8-bit stamp has a default range of -128..+127. And a 1-bit stamp has a default range of -1..0, surprisingly.

definition *logic-negate* :: (*'a*::*len*) *word* ⇒ *'a word* **where**
  logic-negate x = (*if x* = *0 then 1 else 0*)

definition *is-IntVal* :: *Value* ⇒ *bool* **where**
  is-IntVal v = (*is-IntVal32 v* ∨ *is-IntVal64 v*)

Extract signed integer values from both 32 and 64 bit.

fun *intval* :: *Value* ⇒ *int* **where**
  intval (*IntVal32 v*) = *sint v* |
  intval (*IntVal64 v*) = *sint v*

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
```

1.1 Arithmetic Operators

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 ⇒ Value ⇒ Value where
  times-Value = intval-mul

instance proof qed
end



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

instantiation Value :: divide
```

begin

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

instance proof qed
end

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

instantiation *Value* :: *modulo*
begin

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

instance proof qed
end

1.2 Bitwise Operators and Comparisons

context
  includes *bit-operations-syntax*
begin

fun *intval-and* :: *Value* ⇒ *Value* ⇒ *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* ⇒ *Value* ⇒ *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* ⇒ *Value* ⇒ *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-short-circuit-or* :: *Value* ⇒ *Value* ⇒ *Value* **where**
intval-short-circuit-or (*IntVal32 v1*) (*IntVal32 v2*) = (*IntVal32* (*v1 OR v2*)) |
intval-short-circuit-or (*IntVal64 v1*) (*IntVal64 v2*) = (*IntVal64* (*v1 OR v2*)) |
intval-short-circuit-or - - = *UndefVal*

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

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

fun *intval-below* :: *Value* ⇒ *Value* ⇒ *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* ⇒ *Value* **where**
intval-not (*IntVal32 v*) = (*IntVal32* (*NOT v*)) |
intval-not (*IntVal64 v*) = (*IntVal64* (*NOT v*)) |
intval-not - = *UndefVal*

fun *intval-negate* :: *Value* ⇒ *Value* **where**
intval-negate (*IntVal32 v*) = *IntVal32* (− *v*) |
intval-negate (*IntVal64 v*) = *IntVal64* (− *v*) |
intval-negate - = *UndefVal*

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

fun *intval-conditional* :: *Value* ⇒ *Value* ⇒ *Value* ⇒ *Value* **where**
intval-conditional cond tv fv = (*if* (*val-to-bool cond*) *then tv else fv*)

fun *intval-logic-negation* :: *Value* ⇒ *Value* **where**
intval-logic-negation (*IntVal32 v*) = (*IntVal32* (*logic-negate v*)) |
intval-logic-negation (*IntVal64 v*) = (*IntVal64* (*logic-negate v*)) |
intval-logic-negation - = *UndefVal*

lemma *intval-eq32*:
assumes *intval-equals* (*IntVal32 v1*) *v2* ≠ *UndefVal*
shows *is-IntVal32 v2*
by (*metis Value.exhaust-disc assms intval-equals.simps*(*10*) *intval-equals.simps*(*12*) *intval-equals.simps*(*15*) *intval-equals.simps*(*16*) *is-IntVal64-def is-ObjRef-def is-ObjStr-def*)

lemma *intval-eq32-simp*:
  assumes *intval-equals* (*IntVal32 v1*) *v2* $\neq$ *UndefVal*
  shows *intval-equals* (*IntVal32 v1*) *v2* = *bool-to-val* (*v1* = *un-IntVal32 v2*)
  by (*metis Value.collapse*(*1*) *assms intval-eq32 intval-equals.simps*(*1*))

1.3 Narrowing and Widening Operators

Note: we allow these operators to have inBits=outBits, because the Graal compiler also seems to allow that case, even though it should rarely / never arise in practice.

When narrowing to less than 32 bits, we sign extend back to 32 bits, because we always represent integer values as either 32 or 64 bits.

```
fun narrow-helper :: nat ⇒ nat ⇒ int32 ⇒ Value where
  narrow-helper inBits outBits val =
    (if outBits ≤ inBits ∧ outBits ≤ 32 ∧
        outBits ∈ valid-int-widths ∧
        inBits ∈ valid-int-widths
     then IntVal32 (signed-take-bit (outBits − 1) val)
     else UndefVal)

value sint(signed-take-bit 0 (1 :: int32))

fun intval-narrow :: nat ⇒ nat ⇒ Value ⇒ Value where
  intval-narrow inBits outBits (IntVal32 v) =
     (if inBits = 64
      then UndefVal
      else narrow-helper inBits outBits v) |
  intval-narrow inBits outBits (IntVal64 v) =
     (if inBits = 64
      then (if outBits = 64
            then IntVal64 v
            else narrow-helper inBits outBits (scast v))
      else UndefVal) |
  intval-narrow - - - = UndefVal

value intval(intval-narrow 16 8 (IntVal32 (512 − 2)))


fun choose-32-64 :: nat ⇒ int64 ⇒ Value where
  choose-32-64 outBits v = (if outBits = 64 then (IntVal64 v) else (IntVal32 (scast
v)))

value sint (signed-take-bit 7 ((256 + 128) :: int64))

fun sign-extend-helper :: nat ⇒ nat ⇒ int32 ⇒ Value where
  sign-extend-helper inBits outBits val =
    (if inBits ≤ outBits ∧ inBits ≤ 32 ∧
        outBits ∈ valid-int-widths ∧
        inBits ∈ valid-int-widths
     then
        (if outBits = 64
        then IntVal64 (scast (signed-take-bit (inBits − 1) val))
        else IntVal32 (signed-take-bit (inBits − 1) val))
     else UndefVal)
```

```
fun intval-sign-extend :: nat ⇒ nat ⇒ Value ⇒ Value where
  intval-sign-extend inBits outBits (IntVal32 v) =
    sign-extend-helper inBits outBits v |
  intval-sign-extend inBits outBits (IntVal64 v) =
    (if inBits=64 ∧ outBits=64 then IntVal64 v else UndefVal) |
  intval-sign-extend - - - = UndefVal
```

```
fun zero-extend-helper :: nat ⇒ nat ⇒ int32 ⇒ Value where
  zero-extend-helper inBits outBits val =
    (if inBits ≤ outBits ∧ inBits ≤ 32 ∧
      outBits ∈ valid-int-widths ∧
      inBits ∈ valid-int-widths
    then
      (if outBits = 64
      then IntVal64 (ucast (take-bit inBits val))
      else IntVal32 (take-bit inBits val))
    else UndefVal)
```

```
fun intval-zero-extend :: nat ⇒ nat ⇒ Value ⇒ Value where
  intval-zero-extend inBits outBits (IntVal32 v) =
    zero-extend-helper inBits outBits v |
  intval-zero-extend inBits outBits (IntVal64 v) =
    (if inBits=64 ∧ outBits=64 then IntVal64 v else UndefVal) |
  intval-zero-extend - - - = UndefVal
```

Some well-formedness results to help reasoning about narrowing and widening operators

```
lemma narrow-helper-ok:
  assumes narrow-helper inBits outBits val ≠ UndefVal
  shows 0 < outBits ∧ outBits ≤ 32 ∧
      outBits ≤ inBits ∧
      outBits ∈ valid-int-widths ∧
      inBits ∈ valid-int-widths
  using assms narrow-helper.simps neq0-conv by fastforce
```

```
lemma intval-narrow-ok:
  assumes intval-narrow inBits outBits val ≠ UndefVal
  shows 0 < outBits ∧
      outBits ≤ inBits ∧
      outBits ∈ valid-int-widths ∧
      inBits ∈ valid-int-widths
  using assms narrow-helper-ok intval-narrow.simps neq0-conv
  by (smt (verit, best) insertCI intval-sign-extend.elims order-le-less zero-neq-numeral)


lemma narrow-takes-64:
  assumes result = intval-narrow inBits outBits value
  assumes result ≠ UndefVal
  shows is-IntVal64 value = (inBits = 64)
  using assms by (cases value; simp; presburger)

lemma narrow-gives-64:
  assumes result = intval-narrow inBits outBits value
  assumes result ≠ UndefVal
  shows is-IntVal64 result = (outBits = 64)
  using assms
  by (smt (verit, best) Value.case-eq-if Value.discI(1) Value.discI(2) Value.disc-eq-case(3)
add-diff-cancel-left' diff-is-0-eq intval-narrow.elims narrow-helper.simps numeral-Bit0
zero-neq-numeral)


lemma sign-extend-helper-ok:
  assumes sign-extend-helper inBits outBits val ≠ UndefVal
  shows 0 < inBits ∧ inBits ≤ 32 ∧
      inBits ≤ outBits ∧
      outBits ∈ valid-int-widths ∧
      inBits ∈ valid-int-widths
  using assms sign-extend-helper.simps neq0-conv by fastforce

lemma intval-sign-extend-ok:
  assumes intval-sign-extend inBits outBits val ≠ UndefVal
  shows 0 < inBits ∧
      inBits ≤ outBits ∧
      outBits ∈ valid-int-widths ∧
      inBits ∈ valid-int-widths
  using assms sign-extend-helper-ok intval-sign-extend.simps neq0-conv
  by (smt (verit, best) insertCI intval-sign-extend.elims order-le-less zero-neq-numeral)


lemma zero-extend-helper-ok:
  assumes zero-extend-helper inBits outBits val ≠ UndefVal
  shows 0 < inBits ∧ inBits ≤ 32 ∧
      inBits ≤ outBits ∧
      outBits ∈ valid-int-widths ∧
      inBits ∈ valid-int-widths
  using assms zero-extend-helper.simps neq0-conv by fastforce

lemma intval-zero-extend-ok:
  assumes intval-zero-extend inBits outBits val ≠ UndefVal
  shows 0 < inBits ∧
        inBits ≤ outBits ∧
        outBits ∈ valid-int-widths ∧
        inBits ∈ valid-int-widths
  using assms zero-extend-helper-ok intval-zero-extend.simps neq0-conv
  by (smt (verit, best) insertCI intval-zero-extend.elims order-le-less zero-neq-numeral)
```

1.4 Bit-Shifting Operators

```
definition shiftl (infix << 75) where
  shiftl w n = (push-bit n) w

lemma shiftl-power[simp]: (x::('a::len) word) * (2 ^ j) = x << j
  unfolding shiftl-def apply (induction j)
   apply simp unfolding funpow-Suc-right
  by (metis (no-types, 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 ⇒ nat ⇒ 'a :: len word (infix >> 75)
where
  signed-shiftr w n = word-of-int ((sint w) div (2 ^ n))

value (128 :: 8 word) >> 2
```

Note that Java shift operators use unary numeric promotion, unlike other binary operators, which use binary numeric promotion (see the Java language reference manual). This means that the left-hand input determines the output size, while the right-hand input can be any size.

```
fun intval-left-shift :: Value ⇒ Value ⇒ Value where
  intval-left-shift (IntVal32 v1) (IntVal32 v2) = IntVal32 (v1 << unat (v2 AND
```

0x1f)) |
  intval-left-shift (*IntVal32 v1*) (*IntVal64 v2*) = *IntVal32* (*v1* << *unat* (*v2 AND 0x1f*)) |
  intval-left-shift (*IntVal64 v1*) (*IntVal32 v2*) = *IntVal64* (*v1* << *unat* (*v2 AND 0x3f*)) |
  intval-left-shift (*IntVal64 v1*) (*IntVal64 v2*) = *IntVal64* (*v1* << *unat* (*v2 AND 0x3f*)) |
  intval-left-shift - - = *UndefVal*

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

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

end

2 Examples of Narrowing / Widening Functions

experiment begin
corollary *intval-narrow 32 8* (*IntVal32* (*256* + *128*)) = *IntVal32* (−*128*) **by** *simp*
corollary *intval-narrow 32 8* (*IntVal32* (−*2*)) = *IntVal32* (−*2*) **by** *simp*
corollary *intval-narrow 32 1* (*IntVal32* (−*2*)) = *IntVal32 0* **by** *simp*
corollary *intval-narrow 32 1* (*IntVal32* (−*3*)) = *IntVal32* (−*1*) **by** *simp*

corollary *intval-narrow 32 8* (*IntVal64* (−*2*)) = *UndefVal* **by** *simp*
corollary *intval-narrow 64 8* (*IntVal32* (−*2*)) = *UndefVal* **by** *simp*
corollary *intval-narrow 64 8* (*IntVal64* (−*2*)) = *IntVal32* (−*2*) **by** *simp*
corollary *intval-narrow 64 8* (*IntVal64* (*256*+*127*)) = *IntVal32 127* **by** *simp*
corollary *intval-narrow 64 32* (*IntVal64* (−*2*)) = *IntVal32* (−*2*) **by** *simp*
corollary *intval-narrow 64 64* (*IntVal64* (−*2*)) = *IntVal64* (−*2*) **by** *simp*
end

experiment begin
corollary *intval-sign-extend 8 32* (*IntVal32* (*256* + *128*)) = *IntVal32* (−*128*) **by** *simp*
corollary *intval-sign-extend 8 32* (*IntVal32* (−*2*)) = *IntVal32* (−*2*) **by** *simp*
corollary *intval-sign-extend 1 32* (*IntVal32* (−*2*)) = *IntVal32 0* **by** *simp*
corollary *intval-sign-extend 1 32* (*IntVal32* (−*3*)) = *IntVal32* (−*1*) **by** *simp*

corollary *intval-sign-extend 8 32* (*IntVal64* (−*2*)) = *UndefVal* **by** *simp*
corollary *intval-sign-extend 8 64* (*IntVal64* (−*2*)) = *UndefVal* **by** *simp*
corollary *intval-sign-extend 8 64* (*IntVal32* (−*2*)) = *IntVal64* (−*2*) **by** *simp*
corollary *intval-sign-extend 32 64* (*IntVal32* (−*2*)) = *IntVal64* (−*2*) **by** *simp*
corollary *intval-sign-extend 64 64* (*IntVal64* (−*2*)) = *IntVal64* (−*2*) **by** *simp*
end

experiment begin
corollary *intval-zero-extend 8 32* (*IntVal32* (*256* + *128*)) = *IntVal32 128* **by** *simp*
corollary *intval-zero-extend 8 32* (*IntVal32* (−*2*)) = *IntVal32 254* **by** *simp*
corollary *intval-zero-extend 1 32* (*IntVal32* (−*1*)) = *IntVal32 1* **by** *simp*
corollary *intval-zero-extend 1 32* (*IntVal32* (−*2*)) = *IntVal32 0* **by** *simp*

corollary *intval-zero-extend 8 32* (*IntVal64* (−*2*)) = *UndefVal* **by** *simp*
corollary *intval-zero-extend 8 64* (*IntVal64* (−*2*)) = *UndefVal* **by** *simp*
corollary *intval-zero-extend 8 64* (*IntVal32* (−*2*)) = *IntVal64 254* **by** *simp*
corollary *intval-zero-extend 32 64* (*IntVal32* (−*2*)) = *IntVal64 4294967294* **by** *simp*
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

3 Nodes

3.1 Types of Nodes

theory *IRNodes*
  imports
    Values
begin

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

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

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

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

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

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

| *DynamicNewArrayNode* (*ir-elementType*: *INPUT*) (*ir-length*: *INPUT*) (*ir-voidClass-opt*: *INPUT option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *EndNode*
| *ExceptionObjectNode* (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *FrameState* (*ir-monitorIds*: *INPUT-ASSOC list*) (*ir-outerFrameState-opt*: *INPUT-STATE option*) (*ir-values-opt*: *INPUT list option*) (*ir-virtualObjectMappings-opt*: *INPUT-STATE list option*)
| *IfNode* (*ir-condition*: *INPUT-COND*) (*ir-trueSuccessor*: *SUCC*) (*ir-falseSuccessor*: *SUCC*)
| *IntegerBelowNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *IntegerEqualsNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *IntegerLessThanNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *InvokeNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *InvokeWithExceptionNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*) (*ir-exceptionEdge*: *SUCC*)
| *IsNullNode* (*ir-value*: *INPUT*)
| *KillingBeginNode* (*ir-next*: *SUCC*)
| *LeftShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *LoadFieldNode* (*ir-nid*: *ID*) (*ir-field*: *string*) (*ir-object-opt*: *INPUT option*) (*ir-next*: *SUCC*)
| *LogicNegationNode* (*ir-value*: *INPUT-COND*)
| *LoopBeginNode* (*ir-ends*: *INPUT-ASSOC list*) (*ir-overflowGuard-opt*: *INPUT-GUARD option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *LoopEndNode* (*ir-loopBegin*: *INPUT-ASSOC*)
| *LoopExitNode* (*ir-loopBegin*: *INPUT-ASSOC*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *MergeNode* (*ir-ends*: *INPUT-ASSOC list*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *MethodCallTargetNode* (*ir-targetMethod*: *string*) (*ir-arguments*: *INPUT list*)
| *MulNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *NarrowNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)
| *NegateNode* (*ir-value*: *INPUT*)
| *NewArrayNode* (*ir-length*: *INPUT*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *NewInstanceNode* (*ir-nid*: *ID*) (*ir-instanceClass*: *string*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *NotNode* (*ir-value*: *INPUT*)
| *OrNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *ParameterNode* (*ir-index*: *nat*)
| *PiNode* (*ir-object*: *INPUT*) (*ir-guard-opt*: *INPUT-GUARD option*)
| *ReturnNode* (*ir-result-opt*: *INPUT option*) (*ir-memoryMap-opt*: *INPUT-EXT option*)
| *RightShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *ShortCircuitOrNode* (*ir-x*: *INPUT-COND*) (*ir-y*: *INPUT-COND*)
| *SignExtendNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)

| *SignedDivNode* (*ir-nid*: *ID*) (*ir-x*: *INPUT*) (*ir-y*: *INPUT*) (*ir-zeroCheck-opt*: *INPUT-GUARD option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *SignedRemNode* (*ir-nid*: *ID*) (*ir-x*: *INPUT*) (*ir-y*: *INPUT*) (*ir-zeroCheck-opt*: *INPUT-GUARD option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *StartNode* (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *StoreFieldNode* (*ir-nid*: *ID*) (*ir-field*: *string*) (*ir-value*: *INPUT*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-object-opt*: *INPUT option*) (*ir-next*: *SUCC*)
| *SubNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *UnsignedRightShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *UnwindNode* (*ir-exception*: *INPUT*)
| *ValuePhiNode* (*ir-nid*: *ID*) (*ir-values*: *INPUT list*) (*ir-merge*: *INPUT-ASSOC*)

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

| *RefNode* (*ir-ref*:*ID*)

fun *opt-to-list* :: *'a option* ⇒ *'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*, *trueValue*, *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 stateDuring*) @ (*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 stateBefore*) |
inputs-of-NewInstanceNode:
inputs-of (*NewInstanceNode nid0 instanceClass stateBefore next*) = (*opt-to-list stateBefore*) |
inputs-of-NotNode:
inputs-of (*NotNode value*) = [*value*] |
inputs-of-OrNode:
inputs-of (*OrNode x y*) = [*x*, *y*] |
inputs-of-ParameterNode:
inputs-of (*ParameterNode index*) = [] |
inputs-of-PiNode:
inputs-of (*PiNode object guard*) = *object* # (*opt-to-list guard*) |
inputs-of-ReturnNode:
inputs-of (*ReturnNode result memoryMap*) = (*opt-to-list result*) @ (*opt-to-list memoryMap*) |
inputs-of-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 virtualObjectMappings*) = [] |
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

3.2 Hierarchy of Nodes

theory *IRNodeHierarchy*
imports *IRNodes*
begin

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

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

These functions have been automatically generated from the compiler.

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

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

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

fun *is-ShiftNode* :: *IRNode* ⇒ *bool* **where**
  is-ShiftNode n = ((*is-LeftShiftNode n*) ∨ (*is-RightShiftNode n*) ∨ (*is-UnsignedRightShiftNode n*))

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

fun *is-AbstractLocalNode* :: *IRNode* ⇒ *bool* **where**
  is-AbstractLocalNode n = ((*is-ParameterNode n*))

fun *is-IntegerConvertNode* :: *IRNode* ⇒ *bool* **where**
  is-IntegerConvertNode n = ((*is-NarrowNode n*) ∨ (*is-SignExtendNode n*) ∨ (*is-ZeroExtendNode n*))

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

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

fun *is-PhiNode* :: *IRNode* ⇒ *bool* **where**
is-PhiNode n = ((*is-ValuePhiNode n*))

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

fun *is-UnaryOpLogicNode* :: *IRNode* ⇒ *bool* **where**
is-UnaryOpLogicNode n = ((*is-IsNullNode n*))

fun *is-IntegerLowerThanNode* :: *IRNode* ⇒ *bool* **where**
is-IntegerLowerThanNode n = ((*is-IntegerBelowNode n*) ∨ (*is-IntegerLessThanNode n*))

fun *is-CompareNode* :: *IRNode* ⇒ *bool* **where**
is-CompareNode n = ((*is-IntegerEqualsNode n*) ∨ (*is-IntegerLowerThanNode n*))

fun *is-BinaryOpLogicNode* :: *IRNode* ⇒ *bool* **where**
is-BinaryOpLogicNode n = ((*is-CompareNode n*))

fun *is-LogicNode* :: *IRNode* ⇒ *bool* **where**
is-LogicNode n = ((*is-BinaryOpLogicNode n*) ∨ (*is-LogicNegationNode n*) ∨ (*is-ShortCircuitOrNode n*) ∨ (*is-UnaryOpLogicNode n*))

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

fun *is-FloatingNode* :: *IRNode* ⇒ *bool* **where**
is-FloatingNode n = ((*is-AbstractLocalNode n*) ∨ (*is-BinaryNode n*) ∨ (*is-ConditionalNode n*) ∨ (*is-ConstantNode n*) ∨ (*is-FloatingGuardedNode n*) ∨ (*is-LogicNode n*) ∨ (*is-PhiNode n*) ∨ (*is-ProxyNode n*) ∨ (*is-UnaryNode n*))

fun *is-AccessFieldNode* :: *IRNode* ⇒ *bool* **where**
is-AccessFieldNode n = ((*is-LoadFieldNode n*) ∨ (*is-StoreFieldNode n*))

fun *is-AbstractNewArrayNode* :: *IRNode* ⇒ *bool* **where**
is-AbstractNewArrayNode n = ((*is-DynamicNewArrayNode n*) ∨ (*is-NewArrayNode n*))

fun *is-AbstractNewObjectNode* :: *IRNode* ⇒ *bool* **where**
is-AbstractNewObjectNode n = ((*is-AbstractNewArrayNode n*) ∨ (*is-NewInstanceNode n*))

fun *is-IntegerDivRemNode* :: *IRNode* ⇒ *bool* **where**
is-IntegerDivRemNode n = ((*is-SignedDivNode n*) ∨ (*is-SignedRemNode n*))

fun *is-FixedBinaryNode* :: *IRNode* ⇒ *bool* **where**
is-FixedBinaryNode n = ((*is-IntegerDivRemNode n*))

fun *is-DeoptimizingFixedWithNextNode* :: *IRNode* ⇒ *bool* **where**

is-DeoptimizingFixedWithNextNode n = ((*is-AbstractNewObjectNode n*) ∨ (*is-FixedBinaryNode n*))

fun *is-AbstractMemoryCheckpoint* :: *IRNode* ⇒ *bool* **where**
is-AbstractMemoryCheckpoint n = ((*is-BytecodeExceptionNode n*) ∨ (*is-InvokeNode n*))

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

fun *is-AbstractMergeNode* :: *IRNode* ⇒ *bool* **where**
is-AbstractMergeNode n = ((*is-LoopBeginNode n*) ∨ (*is-MergeNode n*))

fun *is-BeginStateSplitNode* :: *IRNode* ⇒ *bool* **where**
is-BeginStateSplitNode n = ((*is-AbstractMergeNode n*) ∨ (*is-ExceptionObjectNode n*) ∨ (*is-LoopExitNode n*) ∨ (*is-StartNode n*))

fun *is-AbstractBeginNode* :: *IRNode* ⇒ *bool* **where**
is-AbstractBeginNode n = ((*is-BeginNode n*) ∨ (*is-BeginStateSplitNode n*) ∨ (*is-KillingBeginNode n*))

fun *is-FixedWithNextNode* :: *IRNode* ⇒ *bool* **where**
is-FixedWithNextNode n = ((*is-AbstractBeginNode n*) ∨ (*is-AbstractStateSplit n*) ∨ (*is-AccessFieldNode n*) ∨ (*is-DeoptimizingFixedWithNextNode n*))

fun *is-WithExceptionNode* :: *IRNode* ⇒ *bool* **where**
is-WithExceptionNode n = ((*is-InvokeWithExceptionNode n*))

fun *is-ControlSplitNode* :: *IRNode* ⇒ *bool* **where**
is-ControlSplitNode n = ((*is-IfNode n*) ∨ (*is-WithExceptionNode n*))

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

fun *is-AbstractEndNode* :: *IRNode* ⇒ *bool* **where**
is-AbstractEndNode n = ((*is-EndNode n*) ∨ (*is-LoopEndNode n*))

fun *is-FixedNode* :: *IRNode* ⇒ *bool* **where**
is-FixedNode n = ((*is-AbstractEndNode n*) ∨ (*is-ControlSinkNode n*) ∨ (*is-ControlSplitNode n*) ∨ (*is-FixedWithNextNode n*))

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

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

fun *is-Node* :: *IRNode* ⇒ *bool* **where**

is-Node n = ((*is-ValueNode n*) ∨ (*is-VirtualState n*))

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

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

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

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

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

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

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

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

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

fun *is-ValueNodeInterface* :: *IRNode* ⇒ *bool* **where**
is-ValueNodeInterface n = ((*is-ValueNode n*))

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

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

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

fun *is-SingleMemoryKill* :: *IRNode* ⇒ *bool* **where**

is-SingleMemoryKill n = ((*is-BytecodeExceptionNode n*) ∨ (*is-ExceptionObjectNode n*) ∨ (*is-InvokeNode n*) ∨ (*is-InvokeWithExceptionNode n*) ∨ (*is-KillingBeginNode n*) ∨ (*is-StartNode n*))

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

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

fun *is-ArithmeticLIRLowerable* :: *IRNode* ⇒ *bool* **where**
is-ArithmeticLIRLowerable n = ((*is-AbsNode n*) ∨ (*is-BinaryArithmeticNode n*) ∨ (*is-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* ⇒ *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-ConvertNode* :: *IRNode* ⇒ *bool* **where**
is-ConvertNode n = ((*is-IntegerConvertNode 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*)) ∨

$((\textit{is-ConditionalNode n1}) \wedge (\textit{is-ConditionalNode n2})) \vee$
$((\textit{is-ConstantNode n1}) \wedge (\textit{is-ConstantNode n2})) \vee$
$((\textit{is-DynamicNewArrayNode n1}) \wedge (\textit{is-DynamicNewArrayNode n2})) \vee$
$((\textit{is-EndNode n1}) \wedge (\textit{is-EndNode n2})) \vee$
$((\textit{is-ExceptionObjectNode n1}) \wedge (\textit{is-ExceptionObjectNode n2})) \vee$
$((\textit{is-FrameState n1}) \wedge (\textit{is-FrameState n2})) \vee$
$((\textit{is-IfNode n1}) \wedge (\textit{is-IfNode n2})) \vee$
$((\textit{is-IntegerBelowNode n1}) \wedge (\textit{is-IntegerBelowNode n2})) \vee$
$((\textit{is-IntegerEqualsNode n1}) \wedge (\textit{is-IntegerEqualsNode n2})) \vee$
$((\textit{is-IntegerLessThanNode n1}) \wedge (\textit{is-IntegerLessThanNode n2})) \vee$
$((\textit{is-InvokeNode n1}) \wedge (\textit{is-InvokeNode n2})) \vee$
$((\textit{is-InvokeWithExceptionNode n1}) \wedge (\textit{is-InvokeWithExceptionNode n2})) \vee$
$((\textit{is-IsNullNode n1}) \wedge (\textit{is-IsNullNode n2})) \vee$
$((\textit{is-KillingBeginNode n1}) \wedge (\textit{is-KillingBeginNode n2})) \vee$
$((\textit{is-LoadFieldNode n1}) \wedge (\textit{is-LoadFieldNode n2})) \vee$
$((\textit{is-LogicNegationNode n1}) \wedge (\textit{is-LogicNegationNode n2})) \vee$
$((\textit{is-LoopBeginNode n1}) \wedge (\textit{is-LoopBeginNode n2})) \vee$
$((\textit{is-LoopEndNode n1}) \wedge (\textit{is-LoopEndNode n2})) \vee$
$((\textit{is-LoopExitNode n1}) \wedge (\textit{is-LoopExitNode n2})) \vee$
$((\textit{is-MergeNode n1}) \wedge (\textit{is-MergeNode n2})) \vee$
$((\textit{is-MethodCallTargetNode n1}) \wedge (\textit{is-MethodCallTargetNode n2})) \vee$
$((\textit{is-MulNode n1}) \wedge (\textit{is-MulNode n2})) \vee$
$((\textit{is-NegateNode n1}) \wedge (\textit{is-NegateNode n2})) \vee$
$((\textit{is-NewArrayNode n1}) \wedge (\textit{is-NewArrayNode n2})) \vee$
$((\textit{is-NewInstanceNode n1}) \wedge (\textit{is-NewInstanceNode n2})) \vee$
$((\textit{is-NotNode n1}) \wedge (\textit{is-NotNode n2})) \vee$
$((\textit{is-OrNode n1}) \wedge (\textit{is-OrNode n2})) \vee$
$((\textit{is-ParameterNode n1}) \wedge (\textit{is-ParameterNode n2})) \vee$
$((\textit{is-PiNode n1}) \wedge (\textit{is-PiNode n2})) \vee$
$((\textit{is-ReturnNode n1}) \wedge (\textit{is-ReturnNode n2})) \vee$
$((\textit{is-ShortCircuitOrNode n1}) \wedge (\textit{is-ShortCircuitOrNode n2})) \vee$
$((\textit{is-SignedDivNode n1}) \wedge (\textit{is-SignedDivNode n2})) \vee$
$((\textit{is-StartNode n1}) \wedge (\textit{is-StartNode n2})) \vee$
$((\textit{is-StoreFieldNode n1}) \wedge (\textit{is-StoreFieldNode n2})) \vee$
$((\textit{is-SubNode n1}) \wedge (\textit{is-SubNode n2})) \vee$
$((\textit{is-UnwindNode n1}) \wedge (\textit{is-UnwindNode n2})) \vee$
$((\textit{is-ValuePhiNode n1}) \wedge (\textit{is-ValuePhiNode n2})) \vee$
$((\textit{is-ValueProxyNode n1}) \wedge (\textit{is-ValueProxyNode n2})) \vee$
$((\textit{is-XorNode n1}) \wedge (\textit{is-XorNode n2})))$

end

4 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*)

experiment begin
corollary *bit-bounds 1* = (−*1*, *0*) **by** *simp*
end

— A stamp which includes the full range of the type
fun *unrestricted-stamp* :: *Stamp* ⇒ *Stamp* **where**
  unrestricted-stamp VoidStamp = *VoidStamp* |
  unrestricted-stamp (*IntegerStamp bits lower upper*) = (*IntegerStamp bits* (*fst* (*bit-bounds bits*)) (*snd* (*bit-bounds bits*))) |

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

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

```
— A stamp which provides type information but has an empty range of values
fun empty-stamp :: Stamp ⇒ Stamp where
  empty-stamp VoidStamp = VoidStamp |
  empty-stamp (IntegerStamp bits lower upper) = (IntegerStamp bits (snd (bit-bounds
bits)) (fst (bit-bounds bits))) |

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

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

  is-stamp-empty x = False

— Calculate the meet stamp of two stamps
fun meet :: Stamp ⇒ Stamp ⇒ Stamp where
  meet VoidStamp VoidStamp = VoidStamp |
  meet (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1 ≠ b2 then IllegalStamp else
    (IntegerStamp b1 (min l1 l2) (max u1 u2))
  ) |

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

— Calculate the join stamp of two stamps
fun join :: Stamp ⇒ Stamp ⇒ Stamp where
  join VoidStamp VoidStamp = VoidStamp |
  join (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1 ≠ b2 then IllegalStamp else
    (IntegerStamp b1 (max l1 l2) (min u1 u2))
  ) |
```

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

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

fun *asConstant* :: *Stamp* ⇒ *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* :: *Value* ⇒ *Stamp* ⇒ *bool* **where**
valid-value (*IntVal32 v*) (*IntegerStamp b l h*) = ((*b*=*32* ∨ *b*=*16* ∨ *b*=*8* ∨ *b*=*1*) ∧ (*sint v* ≥ *l*) ∧ (*sint v* ≤ *h*)) |
valid-value (*IntVal64 v*) (*IntegerStamp b l h*) = (*b*=*64* ∧ (*sint v* ≥ *l*) ∧ (*sint v* ≤ *h*)) |

```
  valid-value (ObjRef ref) (ObjectStamp klass exact nonNull alwaysNull) =
    ((alwaysNull ⟶ ref = None) ∧ (ref=None ⟶ ¬ nonNull)) |
  valid-value stamp val = False
```

```
fun compatible :: Stamp ⇒ Stamp ⇒ bool where
  compatible (IntegerStamp b1 - -) (IntegerStamp b2 - -) = (b1 = b2) |
  compatible (VoidStamp) (VoidStamp) = True |
  compatible - - = False

fun stamp-under :: Stamp ⇒ Stamp ⇒ 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
```

5 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 ⇀ (IRNode × Stamp) . finite (dom g)}
proof −
  have finite(dom(Map.empty)) ∧ ran Map.empty = {} by auto
  then show ?thesis
    by fastforce
qed

setup-lifting type-definition-IRGraph

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

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

lift-definition *kind* :: *IRGraph* ⇒ (*ID* ⇒ *IRNode*)
  is *with-default NoNode fst* **.**

lift-definition *stamp* :: *IRGraph* ⇒ *ID* ⇒ *Stamp*
  is *with-default IllegalStamp snd* **.**

lift-definition *add-node* :: *ID* ⇒ (*IRNode* × *Stamp*) ⇒ *IRGraph* ⇒ *IRGraph*
  is *λnid k g. if fst k* = *NoNode then g else g*(*nid* ↦ *k*) **by** *simp*

lift-definition *remove-node* :: *ID* ⇒ *IRGraph* ⇒ *IRGraph*
  is *λnid g. g*(*nid* := *None*) **by** *simp*

lift-definition *replace-node* :: *ID* ⇒ (*IRNode* × *Stamp*) ⇒ *IRGraph* ⇒ *IRGraph*
  is *λnid k g. if fst k* = *NoNode then g else g*(*nid* ↦ *k*) **by** *simp*

lift-definition *as-list* :: *IRGraph* ⇒ (*ID* × *IRNode* × *Stamp*) *list*
  is *λg. map* (*λk.* (*k*, *the* (*g k*))) (*sorted-list-of-set* (*dom g*)) **.**

fun *no-node* :: (*ID* × (*IRNode* × *Stamp*)) *list* ⇒ (*ID* × (*IRNode* × *Stamp*)) *list*
where
  no-node g = *filter* (*λn. fst* (*snd n*) ≠ *NoNode*) *g*

lift-definition *irgraph* :: (*ID* × (*IRNode* × *Stamp*)) *list* ⇒ *IRGraph*
  is *map-of* ∘ *no-node*
  by (*simp add*: *finite-dom-map-of*)

definition *as-set* :: *IRGraph* ⇒ (*ID* × (*IRNode* × *Stamp*)) *set* **where**
  as-set g = {(*n*, *kind g n*, *stamp g n*) | *n* . *n* ∈ *ids g*}

definition *true-ids* :: *IRGraph* ⇒ *ID set* **where**
  true-ids g = *ids g* − {*n* ∈ *ids g*. ∃ *n'* . *kind g n* = *RefNode n'*}

definition *domain-subtraction* :: *'a set* ⇒ (*'a* × *'b*) *set* ⇒ (*'a* × *'b*) *set*
  (**infix** ⊴ *30*) **where**
  domain-subtraction s r = {(*x*, *y*) . (*x*, *y*) ∈ *r* ∧ *x* ∉ *s*}

notation (*latex*)
  domain-subtraction (- ⩤ -)

code-datatype *irgraph*

fun *filter-none* **where**
  filter-none g = {*nid* ∈ *dom g* . ∄ *s. g nid* = (*Some* (*NoNode*, *s*))}

lemma *no-node-clears*:
  res = *no-node xs* ⟶ (∀ *x* ∈ *set res*. *fst* (*snd x*) ≠ *NoNode*)
  by *simp*

lemma *dom-eq*:
  assumes ∀ *x* ∈ *set xs*. *fst* (*snd x*) ≠ *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* ⇒ *ID* ⇒ *ID set* **where**
  inputs g nid = *set* (*inputs-of* (*kind g nid*))
— Get the successor set of a given node ID
fun *succ* :: *IRGraph* ⇒ *ID* ⇒ *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* ⇒ *ID rel* **where**
  input-edges g = (⋃ *i* ∈ *ids g*. {(*i*,*j*)|*j*. *j* ∈ (*inputs g i*)})
— Find all the nodes in the graph that have nid as an input - the usages of nid
fun *usages* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
  usages g nid = {*i*. *i* ∈ *ids g* ∧ *nid* ∈ *inputs g i*}
fun *successor-edges* :: *IRGraph* ⇒ *ID rel* **where**
  successor-edges g = (⋃ *i* ∈ *ids g*. {(*i*,*j*)|*j* . *j* ∈ (*succ g i*)})
fun *predecessors* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
  predecessors g nid = {*i*. *i* ∈ *ids g* ∧ *nid* ∈ *succ g i*}
fun *nodes-of* :: *IRGraph* ⇒ (*IRNode* ⇒ *bool*) ⇒ *ID set* **where**
  nodes-of g sel = {*nid* ∈ *ids g* . *sel* (*kind g nid*)}
fun *edge* :: (*IRNode* ⇒ *'a*) ⇒ *ID* ⇒ *IRGraph* ⇒ *'a* **where**
  edge sel nid g = *sel* (*kind g nid*)

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

```
fun filtered-successors :: IRGraph ⇒ ID ⇒ (IRNode ⇒ bool) ⇒ ID list where
  filtered-successors g nid f = filter (f ∘ (kind g)) (successors-of (kind g nid))
fun filtered-usages :: IRGraph ⇒ ID ⇒ (IRNode ⇒ bool) ⇒ ID set where
  filtered-usages g nid f = {n ∈ (usages g nid). f (kind g n)}

fun is-empty :: IRGraph ⇒ bool where
  is-empty g = (ids g = {})

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

lemma ids-some[simp]: x ∈ ids g ⟷ kind g x ≠ NoNode
proof −
  have that: x ∈ ids g ⟶ kind g x ≠ NoNode
    using ids.rep-eq kind.rep-eq by force
  have kind g x ≠ NoNode ⟶ x ∈ 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 nid ∉ ids g
  shows kind g nid = NoNode
  using assms ids-some by blast

lemma valid-creation[simp]:
  finite (dom g) ⟷ 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) ⟶ ids (Abs-IRGraph g) = {nid ∈ dom g . ∄ s. g
nid = Some (NoNode, s)}
  using ids.rep-eq by simp

lemma [simp]: finite (dom g) ⟶ kind (Abs-IRGraph g) = (λx . (case g x of None
⇒ NoNode | Some n ⇒ fst n))
  by (simp add: kind.rep-eq)

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

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

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


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

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

lemma add-node-lookup:
  gup = add-node nid (k, s) g ⟶
    (if k ≠ NoNode then kind gup nid = k ∧ stamp gup nid = s else kind gup nid
= kind g nid)
proof (cases k = NoNode)
  case True
  then show ?thesis
    by (simp add: add-node.rep-eq kind.rep-eq)
next
  case False
  then show ?thesis
    by (simp add: kind.rep-eq add-node.rep-eq stamp.rep-eq)
qed

lemma remove-node-lookup:
  gup = remove-node nid g ⟶ kind gup nid = NoNode ∧ stamp gup nid = Ille-
galStamp
```

by (*simp add*: *kind.rep-eq remove-node.rep-eq stamp.rep-eq*)

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

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

5.0.1 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

5.1 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 TraversalState = (ID × Seen × 'a)

inductive Step
  :: ('a TraversalState ⇒ 'a) ⇒ IRGraph ⇒ 'a TraversalState ⇒ 'a TraversalState
option ⇒ 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

```
  ⟦kind g nid = BeginNode nid';

    nid ∉ seen;
    seen' = {nid} ∪ seen;

    Some ifcond = pred g nid;
    kind g ifcond = IfNode cond t f;
```

$\quad analysis' = sa\ (nid,\ seen,\ analysis)]\!]$
$\Longrightarrow Step\ sa\ g\ (nid,\ seen,\ analysis)\ (Some\ (nid',\ seen',\ analysis'))\ |$

— Hit an EndNode 1. nid' will be the usage of EndNode 2. pop the conditions and stamp stack
$[\![kind\ g\ nid = EndNode;$

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

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

$\quad analysis' = sa\ (nid,\ seen,\ analysis)]\!]$
$\Longrightarrow Step\ sa\ g\ (nid,\ seen,\ analysis)\ (Some\ (nid',\ seen',\ analysis'))\ |$

— We can find a successor edge that is not in seen, go there
$[\![\neg(is\text{-}EndNode\ (kind\ g\ nid));$
$\quad \neg(is\text{-}BeginNode\ (kind\ g\ nid));$

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

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

$\quad analysis' = sa\ (nid,\ seen,\ analysis)]\!]$
$\Longrightarrow Step\ sa\ g\ (nid,\ seen,\ analysis)\ (Some\ (nid',\ seen',\ analysis'))\ |$

— We can cannot find a successor edge that is not in seen, give back None
$[\![\neg(is\text{-}EndNode\ (kind\ g\ nid));$
$\quad \neg(is\text{-}BeginNode\ (kind\ g\ nid));$

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

$\quad None = nextEdge\ seen'\ nid\ g]\!]$
$\quad \Longrightarrow Step\ sa\ g\ (nid,\ seen,\ analysis)\ None\ |$

— We've already seen this node, give back None
$[\![nid \in seen]\!] \Longrightarrow Step\ sa\ g\ (nid,\ seen,\ analysis)\ None$

code-pred $(modes\colon i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool)\ Step$ **.**

end

5.2 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* ⇒ (*ID* ⇀ *ID*) **where**
find-ref-nodes g = *map-of*
  (*map* (*λn.* (*n*, *ir-ref* (*kind g n*))) (*filter* (*λid. is-RefNode* (*kind g id*)) (*sorted-list-of-set* (*ids g*))))

fun *replace-ref-nodes* :: *IRGraph* ⇒ (*ID* ⇀ *ID*) ⇒ *ID list* ⇒ *ID list* **where**
replace-ref-nodes g m xs = *map* (*λid.* (*case* (*m id*) *of Some other* ⇒ *other* | *None* ⇒ *id*)) *xs*

fun *find-next* :: *ID list* ⇒ *ID set* ⇒ *ID option* **where**
  find-next to-see seen = (*let l* = (*filter* (*λnid. nid* ∉ *seen*) *to-see*)
    in (*case l of* [] ⇒ *None* | *xs* ⇒ *Some* (*hd xs*)))

inductive *reachables* :: *IRGraph* ⇒ *ID list* ⇒ *ID set* ⇒ *ID set* ⇒ *bool* **where**
reachables g [] {} {} |
⟦*None* = *find-next to-see seen*⟧ ⟹ *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′* ⟧ ⟹ *reachables g to-see seen seen′*

code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*) [*show-steps*,*show-mode-inference*,*show-intermediate-results*]

reachables **.**

inductive *nodeEq* :: (*ID* ⇀ *ID*) ⇒ *IRGraph* ⇒ *ID* ⇒ *IRGraph* ⇒ *ID* ⇒ *bool*
where
⟦ *kind g1 n1* = *RefNode ref*; *nodeEq m g1 ref g2 n2* ⟧ ⟹ *nodeEq m g1 n1 g2 n2*
|
⟦ *x* = *kind g1 n1*;
  y = *kind g2 n2*;
  is-same-ir-node-type x y;
  replace-ref-nodes g1 m (*successors-of x*) = *successors-of y*;
  replace-ref-nodes g1 m (*inputs-of x*) = *inputs-of y* ⟧
  ⟹ *nodeEq m g1 n1 g2 n2*

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

fun *diffNodesGraph* :: *IRGraph* ⇒ *IRGraph* ⇒ *ID set* **where**
diffNodesGraph g1 g2 = (*let refNodes* = *find-ref-nodes g1 in*
    { *n* . *n* ∈ *Predicate.the* (*reachables-i-i-i-o g1* [*0*] {}) ∧ (*case refNodes n of Some*

- ⇒ *False* | - ⇒ *True*) ∧ ¬(*nodeEq refNodes g1 n g2 n*)})

fun *diffNodesInfo* :: *IRGraph* ⇒ *IRGraph* ⇒ (*ID* × *IRNode* × *IRNode*) *set* **where**
diffNodesInfo g1 g2 = {(*nid*, *kind g1 nid*, *kind g2 nid*) | *nid* . *nid* ∈ *diffNodesGraph g1 g2*}

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

end