Veriopt Theories

November 24, 2022

Contents

1 Operator Semantics

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, and char is 16-bit unsigned. 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.

During calculations the smaller sizes are sign-extended to 32 bits, but explicit widening nodes will do that, so most binary calculations should see equal input sizes.

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 $\equiv$ { *1, 8, 16, 32, 64* }

experiment begin

Option 2: explicit width stored with each integer value. However, this does not help us to distinguish between short (signed) and char (unsigned).

typedef *IntWidth = { w :: nat . w=1 $\lor$ w=8 $\lor$ w=16 $\lor$ w=32 $\lor$ w=64 }*
  by *blast*

setup-lifting *type-definition-IntWidth*

lift-definition *IntWidthBits :: IntWidth $\Rightarrow$ nat*
  is $\lambda w.\ w$.
end

experiment begin

Option 3: explicit type stored with each integer value.

datatype *IntType = ILong | IInt | IShort | IChar | IByte | IBoolean*

fun *int-bits* :: *IntType* $\Rightarrow$ *nat* **where**
  int-bits ILong = 64 |
  int-bits IInt   = 32 |
  int-bits IShort = 16 |
  int-bits IChar = 16 |
  int-bits IByte = 8 |
  int-bits IBoolean = 1

fun *int-signed* :: *IntType* $\Rightarrow$ *bool* **where**
  int-signed ILong = True |
  int-signed IInt   = True |
  int-signed IShort = True |
  int-signed IChar = False |
  int-signed IByte = True |
  int-signed IBoolean = True
end

Option 4: int64 with the number of significant bits.

type-synonym *iwidth = nat*
type-synonym *objref = nat option*

datatype (*discs-sels*) *Value =*
  UndefVal |

  IntVal iwidth int64 |

  ObjRef objref |
  ObjStr string

fun *intval-bits* :: *Value* $\Rightarrow$ *nat* **where**
  intval-bits (IntVal b v) = b

fun *intval-word* :: *Value* $\Rightarrow$ *int64* **where**
  intval-word (IntVal b v) = v

fun *bit-bounds* :: *nat* $\Rightarrow$ *(int* $\times$ *int)* **where**
  *bit-bounds bits = (((2 ^ bits) div 2) * −1, ((2 ^ bits) div 2) − 1)*

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

fun *int-signed-value* :: *iwidth* $\Rightarrow$ *int64* $\Rightarrow$ *int* **where**
  int-signed-value b v = sint (signed-take-bit (b − 1) v)

fun *int-unsigned-value* :: *iwidth* $\Rightarrow$ *int64* $\Rightarrow$ *int* **where**
  int-unsigned-value b v = uint v

Converts an integer word into a Java value.

fun *new-int* :: *iwidth* $\Rightarrow$ *int64* $\Rightarrow$ *Value* **where**
  new-int b w = IntVal b (take-bit b w)

Converts an integer word into a Java value, iff the two types are equal.

fun *new-int-bin* :: *iwidth* $\Rightarrow$ *iwidth* $\Rightarrow$ *int64* $\Rightarrow$ *Value* **where**
  new-int-bin b1 b2 w = (if b1=b2 then new-int b1 w else UndefVal)

fun *wf-bool* :: *Value* $\Rightarrow$ *bool* **where**
  wf-bool (IntVal b w) = (b = 1) |
  wf-bool - = False

fun *val-to-bool* :: *Value* $\Rightarrow$ *bool* **where**
  val-to-bool (IntVal b val) = (if val = 0 then False else True) |
  val-to-bool val = False

fun *bool-to-val* :: *bool* $\Rightarrow$ *Value* **where**
  bool-to-val True = (IntVal 32 1) |
  bool-to-val False = (IntVal 32 0)

Converts an Isabelle bool into a Java value, iff the two types are equal.

fun *bool-to-val-bin* :: *iwidth* $\Rightarrow$ *iwidth* $\Rightarrow$ *bool* $\Rightarrow$ *Value* **where**
  bool-to-val-bin t1 t2 b = (if t1 = t2 then bool-to-val b else UndefVal)

fun *is-int-val* :: *Value* $\Rightarrow$ *bool* **where**
  is-int-val v = is-IntVal v

A convenience function for directly constructing -1 values of a given bit size.

fun *neg-one* :: *iwidth* $\Rightarrow$ *int64* **where**
  neg-one b = mask b

lemma *neg-one-value[simp]: new-int b (neg-one b) = IntVal b (mask b)*
  by *simp*

lemma *neg-one-signed[simp]:*
  assumes *0 < b*
  shows *int-signed-value b (neg-one b) = −1*
   by *(smt (verit, best) assms diff-le-self diff-less int-signed-value.simps less-one mask-eq-take-bit-minus-one neg-one.simps nle-le signed-minus-1 signed-take-bit-of-minus-1 signed-take-bit-take-bit verit-comp-simplify1(1))*

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 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 (*IntVal b1 v1*) (*IntVal b2 v2*) =
   (*if b1 = b2 then IntVal b1 (take-bit b1 (v1+v2)) else UndefVal*) |
  intval-add - - = UndefVal

fun *intval-sub* :: *Value* ⇒ *Value* ⇒ *Value* **where**
  intval-sub (*IntVal b1 v1*) (*IntVal b2 v2*) = *new-int-bin b1 b2 (v1−v2)* |
  intval-sub - - = UndefVal

fun *intval-mul* :: *Value* ⇒ *Value* ⇒ *Value* **where**
  intval-mul (*IntVal b1 v1*) (*IntVal b2 v2*) = *new-int-bin b1 b2 (v1∗v2)* |
  intval-mul - - = UndefVal

fun *intval-div* :: *Value* ⇒ *Value* ⇒ *Value* **where**
  intval-div (*IntVal b1 v1*) (*IntVal b2 v2*) =
    new-int-bin b1 b2 (word-of-int
     ((*int-signed-value b1 v1*) *sdiv* (*int-signed-value b2 v2*))) |
  intval-div - - = UndefVal

fun *intval-mod* :: *Value* ⇒ *Value* ⇒ *Value* **where**
  intval-mod (*IntVal b1 v1*) (*IntVal b2 v2*) =
    new-int-bin b1 b2 (word-of-int
     ((*int-signed-value b1 v1*) *smod* (*int-signed-value b2 v2*))) |
  intval-mod - - = UndefVal

fun *intval-negate* :: *Value* ⇒ *Value* **where**
  intval-negate (*IntVal t v*) = *new-int t (− v)* |
  intval-negate - = UndefVal

fun *intval-abs* :: *Value* ⇒ *Value* **where**

intval-abs (IntVal t v) = new-int t (if int-signed-value t v < 0 then − v else v) |
intval-abs - = UndefVal

TODO: clarify which widths this should work on: just 1-bit or all?

fun *intval-logic-negation :: Value ⇒ Value* **where**
intval-logic-negation (IntVal b v) = new-int b (logic-negate v) |
intval-logic-negation - = UndefVal

1.2 Bitwise Operators

fun *intval-and :: Value ⇒ Value ⇒ Value* **where**
intval-and (IntVal b1 v1) (IntVal b2 v2) = new-int-bin b1 b2 (and v1 v2) |
intval-and - - = UndefVal

fun *intval-or :: Value ⇒ Value ⇒ Value* **where**
intval-or (IntVal b1 v1) (IntVal b2 v2) = new-int-bin b1 b2 (or v1 v2) |
intval-or - - = UndefVal

fun *intval-xor :: Value ⇒ Value ⇒ Value* **where**
intval-xor (IntVal b1 v1) (IntVal b2 v2) = new-int-bin b1 b2 (xor v1 v2) |
intval-xor - - = UndefVal

fun *intval-not :: Value ⇒ Value* **where**
intval-not (IntVal t v) = new-int t (not v) |
intval-not - = UndefVal

1.3 Comparison Operators

fun *intval-short-circuit-or :: Value ⇒ Value ⇒ Value* **where**
intval-short-circuit-or (IntVal b1 v1) (IntVal b2 v2) = bool-to-val-bin b1 b2 (((v1 ≠ 0) ∨ (v2 ≠ 0))) |
intval-short-circuit-or - - = UndefVal

fun *intval-equals :: Value ⇒ Value ⇒ Value* **where**
intval-equals (IntVal b1 v1) (IntVal b2 v2) = bool-to-val-bin b1 b2 (v1 = v2) |
intval-equals - - = UndefVal

fun *intval-less-than :: Value ⇒ Value ⇒ Value* **where**
intval-less-than (IntVal b1 v1) (IntVal b2 v2) =
  bool-to-val-bin b1 b2 (int-signed-value b1 v1 < int-signed-value b2 v2) |
intval-less-than - - = UndefVal

fun *intval-below :: Value ⇒ Value ⇒ Value* **where**
intval-below (IntVal b1 v1) (IntVal b2 v2) = bool-to-val-bin b1 b2 (v1 < v2) |
intval-below - - = UndefVal

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

1.4 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.

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

fun *intval-narrow* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *Value* $\Rightarrow$ *Value* **where**
  intval-narrow inBits outBits (*IntVal b v*) =
    (*if inBits* = *b* $\wedge$ *0* < *outBits* $\wedge$ *outBits* $\leq$ *inBits* $\wedge$ *inBits* $\leq$ *64*
    then new-int outBits v
    else UndefVal) |
  intval-narrow - - - = *UndefVal*

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

fun *intval-sign-extend* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *Value* $\Rightarrow$ *Value* **where**
  intval-sign-extend inBits outBits (*IntVal b v*) =
    (*if inBits* = *b* $\wedge$ *0* < *inBits* $\wedge$ *inBits* $\leq$ *outBits* $\wedge$ *outBits* $\leq$ *64*
    then new-int outBits (*signed-take-bit* (*inBits* − *1*) *v*)
    else UndefVal) |
  intval-sign-extend - - - = *UndefVal*

fun *intval-zero-extend* :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *Value* $\Rightarrow$ *Value* **where**
  intval-zero-extend inBits outBits (*IntVal b v*) =
    (*if inBits* = *b* $\wedge$ *0* < *inBits* $\wedge$ *inBits* $\leq$ *outBits* $\wedge$ *outBits* $\leq$ *64*
    then new-int outBits (*take-bit inBits v*)
    else UndefVal) |
  intval-zero-extend - - - = *UndefVal*

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

lemma *intval-narrow-ok*:
  assumes *intval-narrow inBits outBits val* $\neq$ *UndefVal*
  shows *0* < *outBits* $\wedge$ *outBits* $\leq$ *inBits* $\wedge$ *inBits* $\leq$ *64* $\wedge$ *outBits* $\leq$ *64* $\wedge$
    is-IntVal val $\wedge$
    intval-bits val = *inBits*
  using *assms intval-narrow.simps neq0-conv intval-bits.simps*
  by (*metis Value.disc*(*2*) *intval-narrow.elims le-trans*)

lemma *intval-sign-extend-ok*:
  assumes *intval-sign-extend inBits outBits val* $\neq$ *UndefVal*
  shows *0* < *inBits* $\wedge$
    inBits $\leq$ *outBits* $\wedge$ *outBits* $\leq$ *64* $\wedge$
    is-IntVal val $\wedge$
    intval-bits val = *inBits*
  using *assms intval-sign-extend.simps neq0-conv*
  by (*metis intval-bits.simps intval-sign-extend.elims is-IntVal-def*)

lemma *intval-zero-extend-ok*:
  assumes *intval-zero-extend inBits outBits val $\neq$ UndefVal*
  shows *0 < inBits $\land$*
      inBits $\leq$ outBits $\land$ outBits $\leq$ 64 $\land$
      is-IntVal val $\land$
      intval-bits val = inBits
  using *assms intval-sign-extend.simps neq0-conv*
  by (*metis intval-bits.simps intval-zero-extend.elims is-IntVal-def*)

1.5 Bit-Shifting Operators

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

lemma *shiftl-power*[*simp*]: *(x::($'$a::len) word) $*$ (2 $\hat{\ }$ 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 $\hat{\ }$ j) + 1) = x $<<$ j + x*
  by (*simp add: distrib-left*)

lemma *(x::($'$a::len) word) $*$ ((2 $\hat{\ }$ j) $-$ 1) = x $<<$ j $-$ x*
  by (*simp add: right-diff-distrib*)

lemma *(x::($'$a::len) word) $*$ ((2$\hat{\ }$j) + (2$\hat{\ }$k)) = x $<<$ j + x $<<$ k*
  by (*simp add: distrib-left*)

lemma *(x::($'$a::len) word) $*$ ((2$\hat{\ }$j) $-$ (2$\hat{\ }$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 *sshiftr* :: $'$a :: len word $\Rightarrow$ nat $\Rightarrow$ $'$a :: len word (**infix** $>>$ *75*) **where**
  sshiftr w n = word-of-int ((sint w) div (2 $\hat{\ }$ 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 *shift-amount* :: *iwidth $\Rightarrow$ int64 $\Rightarrow$ nat* **where**
  shift-amount b val = unat (and val (if b = 64 then 0x3F else 0x1f))

fun *intval-left-shift* :: *Value* ⇒ *Value* ⇒ *Value* **where**
  intval-left-shift (*IntVal b1 v1*) (*IntVal b2 v2*) = *new-int b1* (*v1* << *shift-amount b1 v2*) |
  intval-left-shift - - = *UndefVal*

Signed shift is more complex, because we sometimes have to insert 1 bits at the correct point, which is at b1 bits.

fun *intval-right-shift* :: *Value* ⇒ *Value* ⇒ *Value* **where**
  intval-right-shift (*IntVal b1 v1*) (*IntVal b2 v2*) =
    (*let shift* = *shift-amount b1 v2 in*
     let ones = *and* (*mask b1*) (*not* (*mask* (*b1* − *shift*) :: *int64*)) *in*
     (*if int-signed-value b1 v1* < *0*
       then new-int b1 (*or ones* (*v1* >>> *shift*))
       else new-int b1 (*v1* >>> *shift*))) |
  intval-right-shift - - = *UndefVal*

fun *intval-uright-shift* :: *Value* ⇒ *Value* ⇒ *Value* **where**
  intval-uright-shift (*IntVal b1 v1*) (*IntVal b2 v2*) = *new-int b1* (*v1* >>> *shift-amount b1 v2*) |
  intval-uright-shift - - = *UndefVal*

1.5.1 Examples of Narrowing / Widening Functions

experiment begin
corollary *intval-narrow 32 8* (*IntVal 32* (*256* + *128*)) = *IntVal 8 128* **by** *simp*
corollary *intval-narrow 32 8* (*IntVal 32* (*−2*)) = *IntVal 8 254* **by** *simp*
corollary *intval-narrow 32 1* (*IntVal 32* (*−2*)) = *IntVal 1 0*   **by** *simp*
corollary *intval-narrow 32 1* (*IntVal 32* (*−3*)) = *IntVal 1 1* **by** *simp*

corollary *intval-narrow 32 8* (*IntVal 64* (*−2*)) = *UndefVal* **by** *simp*
corollary *intval-narrow 64 8* (*IntVal 32* (*−2*)) = *UndefVal* **by** *simp*
corollary *intval-narrow 64 8* (*IntVal 64 254*) = *IntVal 8 254* **by** *simp*
corollary *intval-narrow 64 8* (*IntVal 64* (*256+127*)) = *IntVal 8 127* **by** *simp*
corollary *intval-narrow 64 64* (*IntVal 64* (*−2*)) = *IntVal 64* (*−2*) **by** *simp*
end

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

corollary *intval-sign-extend 8 32* (*IntVal 64 254*) = *UndefVal* **by** *simp*
corollary *intval-sign-extend 8 64* (*IntVal 32 254*) = *UndefVal* **by** *simp*

corollary *intval-sign-extend 8 64 (IntVal 8 254) = IntVal 64 (−2)* **by** *simp*
corollary *intval-sign-extend 32 64 (IntVal 32 (2^32 − 2)) = IntVal 64 (−2)* **by** *simp*
corollary *intval-sign-extend 64 64 (IntVal 64 (−2)) = IntVal 64 (−2)* **by** *simp*
end

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

corollary *intval-zero-extend 8 32 (IntVal 64 (−2)) = UndefVal* **by** *simp*
corollary *intval-zero-extend 8 64 (IntVal 64 (−2)) = UndefVal* **by** *simp*
corollary *intval-zero-extend 8 64 (IntVal 8 254) = IntVal 64 254* **by** *simp*
corollary *intval-zero-extend 32 64 (IntVal 32 (2^32 − 2)) = IntVal 64 (2^32 − 2)* **by** *simp*
corollary *intval-zero-extend 64 64 (IntVal 64 (−2)) = IntVal 64 (−2)* **by** *simp*
end

experiment begin
corollary *intval-right-shift (IntVal 8 128) (IntVal 8 0) = IntVal 8 128* **by** *eval*
corollary *intval-right-shift (IntVal 8 128) (IntVal 8 1) = IntVal 8 192* **by** *eval*
corollary *intval-right-shift (IntVal 8 128) (IntVal 8 2) = IntVal 8 224* **by** *eval*
corollary *intval-right-shift (IntVal 8 128) (IntVal 8 8) = IntVal 8 255* **by** *eval*
corollary *intval-right-shift (IntVal 8 128) (IntVal 8 31) = IntVal 8 255* **by** *eval*
end

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

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

lemma *intval-add (IntVal 32 (2^31−1)) (IntVal 32 (2^31−1)) = IntVal 32 (2^32 − 2)*
  by *eval*
lemma *intval-add (IntVal 64 (2^31−1)) (IntVal 64 (2^31−1)) = IntVal 64 4294967294*
  by *eval*

10

end

1.6   Fixed-width Word Theories

theory *ValueThms*
  imports *Values*
begin

1.6.1   Support Lemmas for Upper/Lower Bounds

lemma *size32*: *size v = 32* **for** *v :: 32 word*
  using *size-word.rep-eq*
  using *One-nat-def add.right-neutral add-Suc-right len-of-numeral-defs(2) len-of-numeral-defs(3)*
mult.right-neutral mult-Suc-right numeral-2-eq-2 numeral-Bit0
  by (*smt* (*verit*, *del-insts*) *mult.commute*)

lemma *size64*: *size v = 64* **for** *v :: 64 word*
  using *size-word.rep-eq*
  using *One-nat-def add.right-neutral add-Suc-right len-of-numeral-defs(2) len-of-numeral-defs(3)*
mult.right-neutral mult-Suc-right numeral-2-eq-2 numeral-Bit0
  by (*smt* (*verit*, *del-insts*) *mult.commute*)

lemma *lower-bounds-equiv*:
  assumes *0 < N*
  shows $-(((2::int) \char`\^ (N-1))) = (2::int) \char`\^ N \; div \; 2 * - \; 1$
  by (*simp add*: *assms int-power-div-base*)

lemma *upper-bounds-equiv*:
  assumes *0 < N*
  shows $(2::int) \char`\^ (N-1) = (2::int) \char`\^ N \; div \; 2$
  by (*simp add*: *assms int-power-div-base*)

Some min/max bounds for 64-bit words

lemma *bit-bounds-min64*: $((fst \; (bit\text{-}bounds \; 64))) \leq (sint \; (v::int64))$
  unfolding *bit-bounds.simps fst-def*
  using *sint-ge*[*of v*] **by** *simp*

lemma *bit-bounds-max64*: $((snd \; (bit\text{-}bounds \; 64))) \geq (sint \; (v::int64))$
  unfolding *bit-bounds.simps fst-def*
  using *sint-lt*[*of v*] **by** *simp*

Extend these min/max bounds to extracting smaller signed words using
signed_take_bit.

Note: we could use signed to convert between bit-widths, instead of signed_take_bit.
But that would have to be done separately for each bit-width type.

value *sint*(*signed-take-bit* *7* (*128* :: *int8*))

ML-val ‹@{*thm signed-take-bit-decr-length-iff*}›
declare [[*show-types=true*]]
ML-val ‹@{*thm signed-take-bit-int-less-exp*}›

lemma *signed-take-bit-int-less-exp-word*:
  fixes *ival* :: *'a* :: *len word*
  assumes $n < LENGTH('a)$
  shows *sint*(*signed-take-bit n ival*) $< (2::int)$ ̂ *n*
  apply *transfer*
  by (*smt* (*verit, best*) *not-take-bit-negative signed-take-bit-eq-take-bit-shift*
    signed-take-bit-int-less-exp take-bit-int-greater-self-iff)

lemma *signed-take-bit-int-greater-eq-minus-exp-word*:
  fixes *ival* :: *'a* :: *len word*
  assumes $n < LENGTH('a)$
  shows $-$ (*2* ̂ *n*) $\leq$ *sint*(*signed-take-bit n ival*)
  apply *transfer*
  by (*smt* (*verit, best*) *signed-take-bit-int-greater-eq-minus-exp*
    signed-take-bit-int-greater-eq-self-iff signed-take-bit-int-less-exp)

lemma *signed-take-bit-range*:
  fixes *ival* :: *'a* :: *len word*
  assumes $n < LENGTH('a)$
  assumes *val* = *sint*(*signed-take-bit n ival*)
  shows $-$ (*2* ̂ *n*) $\leq$ *val* $\land$ *val* $<$ *2* ̂ *n*
  using *signed-take-bit-int-greater-eq-minus-exp-word signed-take-bit-int-less-exp-word*
  using *assms* **by** *blast*

A *bit_bounds* version of the above lemma.

lemma *signed-take-bit-bounds*:
  fixes *ival* :: *'a* :: *len word*
  assumes $n \leq LENGTH('a)$
  assumes $0 < n$
  assumes *val* = *sint*(*signed-take-bit* (*n* $-$ *1*) *ival*)
  shows *fst* (*bit-bounds n*) $\leq$ *val* $\land$ *val* $\leq$ *snd* (*bit-bounds n*)
  using *assms signed-take-bit-range lower-bounds-equiv upper-bounds-equiv*
   by (*metis bit-bounds.simps fst-conv less-imp-diff-less nat-less-le sint-ge sint-lt*
snd-conv zle-diff1-eq)

lemma *signed-take-bit-bounds64*:
  fixes *ival* :: *int64*
  assumes $n \leq 64$
  assumes $0 < n$
  assumes *val* = *sint*(*signed-take-bit* (*n* $-$ *1*) *ival*)

shows *fst* (*bit-bounds n*) $\leq$ *val* $\wedge$ *val* $\leq$ *snd* (*bit-bounds n*)
  using *assms signed-take-bit-bounds*
  by (*metis size64 word-size*)

lemma *int-signed-value-bounds*:
  assumes *b1* $\leq$ *64*
  assumes *0* $<$ *b1*
  shows *fst* (*bit-bounds b1*) $\leq$ *int-signed-value b1 v2* $\wedge$
        int-signed-value b1 v2 $\leq$ *snd* (*bit-bounds b1*)
  using *assms int-signed-value.simps signed-take-bit-bounds64* **by** *blast*

lemma *int-signed-value-range*:
  fixes *ival* :: *int64*
  assumes *val = int-signed-value n ival*
  shows $-$ (*2* $^\wedge$ (*n* $-$ *1*)) $\leq$ *val* $\wedge$ *val* $<$ *2* $^\wedge$ (*n* $-$ *1*)
  using *signed-take-bit-range assms*
   by (*smt* (*verit, ccfv-SIG*) *One-nat-def diff-less int-signed-value.elims len-gt-0*
len-num1 power-less-imp-less-exp power-strict-increasing sint-greater-eq sint-less)

Some lemmas about unsigned words smaller than 64-bit, for zero-extend operators.

lemma *take-bit-smaller-range*:
  fixes *ival* :: $'a$ :: *len word*
  assumes *n* $<$ *LENGTH*($'a$)
  assumes *val = sint*(*take-bit n ival*)
  shows *0* $\leq$ *val* $\wedge$ *val* $<$ (*2*::*int*) $^\wedge$ *n*
  by (*simp add*: *assms signed-take-bit-eq*)

lemma *take-bit-same-size-nochange*:
  fixes *ival* :: $'a$ :: *len word*
  assumes *n = LENGTH*($'a$)
  shows *ival = take-bit n ival*
  by (*simp add*: *assms*)

A simplification lemma for *new_int*, showing that upper bits can be ignored.

lemma *take-bit-redundant*[*simp*]:
  fixes *ival* :: $'a$ :: *len word*
  assumes *0* $<$ *n*
  assumes *n* $<$ *LENGTH*($'a$)
  shows *signed-take-bit* (*n* $-$ *1*) (*take-bit n ival*) = *signed-take-bit* (*n* $-$ *1*) *ival*
proof $-$
  have $\neg$ (*n* $\leq$ *n* $-$ *1*) **using** *assms* **by** *arith*
  then have $\bigwedge i$. *signed-take-bit* (*n* $-$ *1*) (*take-bit n i*) = *signed-take-bit* (*n*$-$*1*) *i*
    using *signed-take-bit-take-bit* **by** (*metis* (*mono-tags*))
  then show *?thesis*
    by *blast*
qed

lemma *take-bit-same-size-range*:

fixes *ival* :: *'a* :: *len word*
assumes $n = LENGTH('a)$
assumes *ival2* $=$ *take-bit n ival*
shows $- (2 \;\hat{}\; n \; div \; 2) \le sint \; ival2 \wedge sint \; ival2 < 2 \;\hat{}\; n \; div \; 2$
using *assms lower-bounds-equiv sint-ge sint-lt* **by** *auto*

lemma *take-bit-same-bounds*:
  fixes *ival* :: *'a* :: *len word*
  assumes $n = LENGTH('a)$
  assumes *ival2* $=$ *take-bit n ival*
  shows *fst* (*bit-bounds n*) $\le$ *sint ival2* $\wedge$ *sint ival2* $\le$ *snd* (*bit-bounds n*)
  unfolding *bit-bounds.simps*
  using *assms take-bit-same-size-range*
  by *force*

Next we show that casting a word to a wider word preserves any upper/lower
bounds. (These lemmas may not be needed any more, since we are not using
scast now?)

lemma *scast-max-bound*:
  assumes *sint* (*v* :: *'a* :: *len word*) $< M$
  assumes $LENGTH('a) < LENGTH('b)$
  shows *sint* ((*scast v*) :: *'b* :: *len word*) $< M$
  unfolding *Word.scast-eq Word.sint-sbintrunc'*
  using *Bit-Operations.signed-take-bit-int-eq-self-iff*
  by (*smt* (*verit, best*) *One-nat-def assms*(*1*) *assms*(*2*) *decr-length-less-iff linorder-not-le power-strict-increasing-iff signed-take-bit-int-less-self-iff sint-greater-eq*)

lemma *scast-min-bound*:
  assumes $M \le sint$ (*v* :: *'a* :: *len word*)
  assumes $LENGTH('a) < LENGTH('b)$
  shows $M \le sint$ ((*scast v*) :: *'b* :: *len word*)
  unfolding *Word.scast-eq Word.sint-sbintrunc'*
  using *Bit-Operations.signed-take-bit-int-eq-self-iff*
  by (*smt* (*verit*) *One-nat-def Suc-pred assms*(*1*) *assms*(*2*) *len-gt-0 less-Suc-eq order-less-le order-less-le-trans power-le-imp-le-exp signed-take-bit-int-greater-eq-self-iff sint-lt*)

lemma *scast-bigger-max-bound*:
  assumes (*result* :: *'b* :: *len word*) $=$ *scast* (*v* :: *'a* :: *len word*)
  shows *sint result* $< 2 \;\hat{}\; LENGTH('a) \; div \; 2$
  using *sint-lt upper-bounds-equiv scast-max-bound*
  by (*smt* (*verit, best*) *assms*(*1*) *len-gt-0 signed-scast-eq signed-take-bit-int-greater-self-iff sint-ge sint-less upper-bounds-equiv*)

lemma *scast-bigger-min-bound*:
  assumes (*result* :: *'b* :: *len word*) $=$ *scast* (*v* :: *'a* :: *len word*)
  shows $- (2 \;\hat{}\; LENGTH('a) \; div \; 2) \le sint \; result$
  using *sint-ge lower-bounds-equiv scast-min-bound*

by (*smt* (*verit*) *assms len-gt-0 nat-less-le not-less scast-max-bound*)

lemma *scast-bigger-bit-bounds*:
  assumes (*result* :: $'b$ :: *len word*) = *scast* (*v* :: $'a$ :: *len word*)
  shows *fst* (*bit-bounds* (*LENGTH*($'a$))) $\leq$ *sint result* $\land$ *sint result* $\leq$ *snd* (*bit-bounds*
(*LENGTH*($'a$)))
  using *assms scast-bigger-min-bound scast-bigger-max-bound*
  by *auto*

Results about *new_int*.

lemma *new-int-take-bits*:
  assumes *IntVal b val* = *new-int b ival*
  shows *take-bit b val* = *val*
  using *assms* **by** *force*

1.6.2 Support lemmas for take bit and signed take bit.

Lemmas for removing redundant take_bit wrappers.

lemma *take-bit-dist-addL*[*simp*]:
  fixes x :: $'a$ :: *len word*
  shows *take-bit b* (*take-bit b x* + *y*) = *take-bit b* (*x* + *y*)
proof (*induction b*)
  case *0*
  then show *?case*
    by *simp*
next
  case (*Suc b*)
  then show *?case*
    by (*simp add*: *add.commute mask-eqs*(*2*) *take-bit-eq-mask*)
qed

lemma *take-bit-dist-addR*[*simp*]:
  fixes x :: $'a$ :: *len word*
  shows *take-bit b* (*x* + *take-bit b y*) = *take-bit b* (*x* + *y*)
  using *take-bit-dist-addL* **by** (*metis add.commute*)

lemma *take-bit-dist-subL*[*simp*]:
  fixes x :: $'a$ :: *len word*
  shows *take-bit b* (*take-bit b x* − *y*) = *take-bit b* (*x* − *y*)
  by (*metis take-bit-dist-addR uminus-add-conv-diff*)

lemma *take-bit-dist-subR*[*simp*]:
  fixes x :: $'a$ :: *len word*
  shows *take-bit b* (*x* − *take-bit b y*) = *take-bit b* (*x* − *y*)
  using *take-bit-dist-subL*
  by (*metis* (*no-types, opaque-lifting*) *diff-add-cancel diff-right-commute diff-self*)

lemma *take-bit-dist-neg*[*simp*]:
  fixes $ix :: {}'a :: len\ word$
  shows *take-bit* b $(-\ take\text{-}bit\ b\ (ix)) = take\text{-}bit\ b\ (-\ ix)$
  by (*metis diff-0 take-bit-dist-subR*)

lemma *signed-take-take-bit*[*simp*]:
  fixes $x :: {}'a :: len\ word$
  assumes $0 < b$
  shows *signed-take-bit* $(b - 1)$ $(take\text{-}bit\ b\ x) = signed\text{-}take\text{-}bit\ (b - 1)\ x$
  by (*smt* (*verit, best*) *Suc-diff-1 assms lessI linorder-not-less signed-take-bit-take-bit*)

lemma *mod-larger-ignore*:
  fixes $a :: int$
  fixes $m\ n :: nat$
  assumes $n < m$
  shows $(a\ mod\ 2\ \hat{}\ m)\ mod\ 2\ \hat{}\ n = a\ mod\ 2\ \hat{}\ n$
 by (*smt* (*verit, del-insts*) *assms exp-mod-exp linorder-not-le mod-0-imp-dvd mod-mod-cancel mod-self order-less-imp-le*)

lemma *mod-dist-over-add*:
  fixes $a\ b\ c :: int64$
  fixes $n :: nat$
  assumes *1*: $0 < n$
  assumes *2*: $n < 64$
  shows $(a\ mod\ 2\hat{}n + b)\ mod\ 2\hat{}n = (a + b)\ mod\ 2\hat{}n$
proof −
  have *3*: $(0 :: int64) < 2\ \hat{}\ n$
    using *assms* **by** (*simp add*: *size64 word-2p-lem*)
  then show *?thesis*
    unfolding *word-mod-2p-is-mask*[*OF 3*]
    apply *transfer*
    by (*metis* (*no-types, opaque-lifting*) *and.right-idem take-bit-add take-bit-eq-mask*)
qed

end

2 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 *is-stamp-empty* :: *Stamp* $\Rightarrow$ *bool* **where**
  is-stamp-empty (*IntegerStamp b lower upper*) = (*upper* < *lower*) |

  is-stamp-empty x = *False*

Just like the IntegerStamp class, we need to know that our lo/hi bounds fit into the given number of bits (either signed or unsigned). Our integer stamps have infinite lo/hi bounds, so if the lower bound is non-negative, we can assume that all values are positive, and the integer bits of a related value can be interpreted as unsigned. This is similar (but slightly more general) to what IntegerStamp.java does with its test: if (sameSignBounds()) in the unsignedUpperBound() method.

Note that this is a bit different and more accurate than what StampFactory.forUnsignedInteger does (it widens large unsigned ranges to the max signed range to allow all bit patterns) because its lo/hi values are only 64-bit.

fun *valid-stamp* :: *Stamp* $\Rightarrow$ *bool* **where**
  valid-stamp (*IntegerStamp bits lo hi*) =
    (*0* < *bits* $\wedge$ *bits* $\leq$ *64* $\wedge$
    fst (*bit-bounds bits*) $\leq$ *lo* $\wedge$ *lo* $\leq$ *snd* (*bit-bounds bits*) $\wedge$
    fst (*bit-bounds bits*) $\leq$ *hi* $\wedge$ *hi* $\leq$ *snd* (*bit-bounds bits*)) |
  valid-stamp s = *True*

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

— 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*)

18

```
) |
  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 IntVal b (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 (IntVal b v) = (IntegerStamp b (int-signed-value b v) (int-signed-value b v)) |

 constantAsStamp - = IllegalStamp

— Define when a runtime value is valid for a stamp. The stamp bounds must be valid, and val must be zero-extended.
fun *valid-value* :: *Value* ⇒ *Stamp* ⇒ *bool* **where**
 valid-value (IntVal b1 val) (IntegerStamp b l h) =
   (if b1 = b then
     valid-stamp (IntegerStamp b l h) ∧
     take-bit b val = val ∧
     l ≤ int-signed-value b val ∧ int-signed-value b val ≤ h
     else False) |

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

definition *wf-value* :: *Value* ⇒ *bool* **where**
 wf-value v = valid-value v (constantAsStamp v)

lemma *unfold-wf-value*[*simp*]:
 wf-value v ⟹ valid-value v (constantAsStamp v)
 using *wf-value-def* **by** *auto*

fun *compatible* :: *Stamp* ⇒ *Stamp* ⇒ *bool* **where**
 compatible (IntegerStamp b1 lo1 hi1) (IntegerStamp b2 lo2 hi2) =
   (b1 = b2 ∧ valid-stamp (IntegerStamp b1 lo1 hi1) ∧ valid-stamp (IntegerStamp b2 lo2 hi2)) |
 compatible (VoidStamp) (VoidStamp) = True |
 compatible - - = False

fun *stamp-under* :: *Stamp* ⇒ *Stamp* ⇒ *bool* **where**
 stamp-under (IntegerStamp b1 lo1 hi1) (IntegerStamp b2 lo2 hi2) = (b1 = b2 ∧ hi1 < lo2) |
 stamp-under - - = False

— The most common type of stamp within the compiler (apart from the Void-

Stamp) is a 32 bit integer stamp with an unrestricted range. We use *default-stamp* as it is a frequently used stamp.

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

value *valid-value* (*IntVal 8* (*255*)) (*IntegerStamp 8* (−*128*) *127*)
end

3 Graph Representation

3.1 IR Graph 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*: *IN-PUT-GUARD option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

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

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

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

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

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

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

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

fun *inputs-of* :: *IRNode* $\Rightarrow$ *ID list* **where**
  inputs-of-AbsNode:
  inputs-of (*AbsNode value*) = [*value*] |
  inputs-of-AddNode:
  inputs-of (*AddNode x y*) = [*x*, *y*] |
  inputs-of-AndNode:
  inputs-of (*AndNode x y*) = [*x*, *y*] |
  inputs-of-BeginNode:
  inputs-of (*BeginNode next*) = [] |
  inputs-of-BytecodeExceptionNode:

inputs-of (*BytecodeExceptionNode arguments stateAfter next*) = *arguments* @ (*opt-to-list stateAfter*) |

  inputs-of-ConditionalNode:
  inputs-of (*ConditionalNode condition trueValue falseValue*) = [*condition, 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 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

3.2 IR Graph Node Hierarchy

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))* ∨

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

end

3.3 IR Graph Type

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 = ($\bigcup$ *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 = ($\bigcup$ *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*:

38

assumes *nid* $\notin$ *ids g*
shows *kind g nid = 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*) = (λx . (*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*) = (λx . (*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*) = (λnid. (*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*) = (λnid. (*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*

39

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* $\longrightarrow$
    (*if k $\neq$ NoNode then kind gup nid = k $\wedge$ 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 $\longrightarrow$ *kind gup nid = NoNode $\wedge$ stamp gup nid = IllegalStamp*
  by (*simp add: kind.rep-eq remove-node.rep-eq stamp.rep-eq*)

lemma *replace-node-lookup*[*simp*]:
  gup = replace-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-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*)

3.3.1 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   java.lang.Long

Utility functions from the Long class that Graal occasionally makes use of.

theory *Long*
  imports *ValueThms*
begin

lemma *negative-all-set-32*:
  $n < 32 \implies bit\ (-1::int32)\ n$
  apply *transfer* **by** *auto*

definition *MaxOrNeg* :: *nat set* $\Rightarrow$ *int* **where**
  MaxOrNeg s = (*if s* = {} *then* -1 *else Max s*)

definition *MinOrHighest* :: *nat set* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
  MinOrHighest s m = (*if s* = {} *then m else Min s*)

definition *highestOneBit* :: ($'a$::*len*) *word* $\Rightarrow$ *int* **where**
  highestOneBit v = *MaxOrNeg* {*n . bit v n*}

definition *lowestOneBit* :: ($'a$::*len*) *word* $\Rightarrow$ *nat* **where**
  lowestOneBit v = *MinOrHighest* {*n . bit v n*} (*size v*)

lemma *max-bit*: *bit* (*v*::($'a$::*len*) *word*) $n \implies n < size\ v$
  by (*simp add*: *bit-imp-le-length size-word.rep-eq*)

lemma *max-set-bit*: *MaxOrNeg* {*n . bit* (*v*::($'a$::*len*) *word*) *n*} $< Nat.size\ v$
  using *max-bit* **unfolding** *MaxOrNeg-def*
  by *force*

4.1 Long.numberOfLeadingZeros

definition *numberOfLeadingZeros* :: (′*a::len*) *word* ⇒ *nat* **where**
  numberOfLeadingZeros v = nat (Nat.size v − highestOneBit v − 1)

lemma *MaxOrNeg-neg*: *MaxOrNeg {} = −1*
  by (*simp add*: *MaxOrNeg-def*)

lemma *MaxOrNeg-max*: *s ≠ {} ⟹ MaxOrNeg s = Max s*
  by (*simp add*: *MaxOrNeg-def*)

lemma *zero-no-bits*:
  {n . bit 0 n} = {}
  by *simp*

lemma *highestOneBit (0::64 word) = −1*
  by (*simp add*: *MaxOrNeg-neg highestOneBit-def*)

lemma *numberOfLeadingZeros (0::64 word) = 64*
  unfolding *numberOfLeadingZeros-def* **using** *MaxOrNeg-neg highestOneBit-def size64*
  by (*smt (verit) nat-int zero-no-bits*)

lemma *highestOneBit-top*: *Max {highestOneBit (v::64 word)} < 64*
  unfolding *highestOneBit-def*
  by (*metis Max-singleton int-eq-iff-numeral max-set-bit size64*)

lemma *numberOfLeadingZeros-top*: *Max {numberOfLeadingZeros (v::64 word)} ≤ 64*
  unfolding *numberOfLeadingZeros-def*
  using *size64*
  by (*simp add*: *MaxOrNeg-def highestOneBit-def nat-le-iff*)

lemma *numberOfLeadingZeros-range*: *0 ≤ numberOfLeadingZeros a ∧ numberOfLeadingZeros a ≤ Nat.size a*
  unfolding *numberOfLeadingZeros-def*
  using *MaxOrNeg-def highestOneBit-def nat-le-iff*
  by (*smt (verit) bot-nat-0.extremum int-eq-iff*)

lemma *leadingZerosAddHighestOne*: *numberOfLeadingZeros v + highestOneBit v = Nat.size v − 1*
  unfolding *numberOfLeadingZeros-def highestOneBit-def*
  using *MaxOrNeg-def int-nat-eq int-ops(6) max-bit order-less-irrefl* **by** *fastforce*

4.2 Long.numberOfTrailingZeros

definition *numberOfTrailingZeros* :: (′*a::len*) *word* ⇒ *nat* **where**
  numberOfTrailingZeros v = lowestOneBit v

lemma *lowestOneBit-bot*: *lowestOneBit (0::64 word) = 64*

unfolding *lowestOneBit-def MinOrHighest-def*
by (*simp add: size64*)

lemma *bit-zero-set-in-top*: *bit* (-1::$'a$::*len word*) *0*
  by *auto*

lemma *nat-bot-set*: (*0*::*nat*) $\in$ *xs* $\longrightarrow$ ($\forall\, x \in xs$. *0* $\leq$ *x*)
  by *fastforce*

lemma *numberOfTrailingZeros* (*0*::*64 word*) = *64*
  unfolding *numberOfTrailingZeros-def*
  using *lowestOneBit-bot* **by** *simp*

4.3   Long.bitCount

definition *bitCount* :: ($'a$::*len*) *word* $\Rightarrow$ *nat* **where**
  bitCount v = *card* {*n . bit v n*}

lemma *bitCount 0* = *0*
  unfolding *bitCount-def*
  by (*metis card.empty zero-no-bits*)

4.4   Long.zeroCount

definition *zeroCount* :: ($'a$::*len*) *word* $\Rightarrow$ *nat* **where**
  zeroCount v = *card* {*n. n* < *Nat.size v* $\land$ $\neg$(*bit v n*)}

lemma *zeroCount-finite*: *finite* {*n. n* < *Nat.size v* $\land$ $\neg$(*bit v n*)}
  using *finite-nat-set-iff-bounded* **by** *blast*

lemma *negone-set*:
  bit (-1::($'a$::*len*) *word*) *n* $\longleftrightarrow$ *n* < *LENGTH*($'a$)
  by *simp*

lemma *negone-all-bits*:
  {*n . bit* (-1::($'a$::*len*) *word*) *n*} = {*n . 0* $\leq$ *n* $\land$ *n* < *LENGTH*($'a$)}
  using *negone-set*
  by *auto*

lemma *bitCount-finite*:
  finite {*n . bit* (*v*::($'a$::*len*) *word*) *n*}
  by *simp*

lemma *card-of-range*:
  x = *card* {*n . 0* $\leq$ *n* $\land$ *n* < *x*}
  by *simp*

lemma *range-of-nat*:
  {(*n*::*nat*) . *0* $\leq$ *n* $\land$ *n* < *x*} = {*n . n* < *x*}
  by *simp*

lemma *finite-range*:
  finite $\{n::nat \; . \; n < x\}$
  by *simp*

lemma *range-eq*:
  fixes $x \; y :: nat$
  shows *card* $\{y..<x\}$ = *card* $\{y<..x\}$
  using *card-atLeastLessThan card-greaterThanAtMost* **by** *presburger*

lemma *card-of-range-bound*:
  fixes $x \; y :: nat$
  assumes $x > y$
  shows $x - y$ = *card* $\{n \; . \; y < n \land n \leq x\}$
proof $-$
  have *finite*: *finite* $\{n \; . \; y \leq n \land n < x\}$
    by *auto*
  have *nonempty*: $\{n \; . \; y \leq n \land n < x\} \neq \{\}$
    using *assms* **by** *blast*
  have *simprep*: $\{n \; . \; y < n \land n \leq x\}$ = $\{y<..x\}$
    by *auto*
  have $x - y$ = *card* $\{y<..x\}$
    by *auto*
  then show *?thesis*
    unfolding *simprep* **by** *blast*
qed

lemma *bitCount* $(-1::('a::len) \; word)$ = $LENGTH('a)$
  unfolding *bitCount-def* **using** *card-of-range*
  by (*metis* (*no-types, lifting*) *Collect-cong negone-all-bits*)

lemma *bitCount-range*:
  fixes $n :: ('a::len) \; word$
  shows $0 \leq bitCount \; n \land bitCount \; n \leq Nat.size \; n$
  unfolding *bitCount-def*
 by (*metis atLeastLessThan-iff bot-nat-0.extremum max-bit mem-Collect-eq subsetI*
subset-eq-atLeast0-lessThan-card)

lemma *zerosAboveHighestOne*:
  $n > highestOneBit \; a \implies \neg(bit \; a \; n)$
  unfolding *highestOneBit-def MaxOrNeg-def*
   by (*metis* (*mono-tags, opaque-lifting*) *Collect-empty-eq Max-ge finite-bit-word*
less-le-not-le mem-Collect-eq of-nat-le-iff)

lemma *zerosBelowLowestOne*:
  assumes $n < lowestOneBit \; a$
  shows $\neg(bit \; a \; n)$
proof (*cases* $\{i. \; bit \; a \; i\}$ = $\{\}$)

case *True*
  then show *?thesis* **by** *simp*
next
  case *False*
  have *n < Min (Collect (bit a))* $\Longrightarrow$ ¬ *bit a n*
    using *False* **by** *auto*
  then show *?thesis*
    by (*metis False MinOrHighest-def assms lowestOneBit-def*)
qed

lemma *union-bit-sets*:
  fixes *a* :: (*'a::len*) *word*
  shows {*n . n < Nat.size a* ∧ *bit a n*} ∪ {*n . n < Nat.size a* ∧ ¬(*bit a n*)} = {*n . n < Nat.size a*}
  by *fastforce*

lemma *disjoint-bit-sets*:
  fixes *a* :: (*'a::len*) *word*
  shows {*n . n < Nat.size a* ∧ *bit a n*} ∩ {*n . n < Nat.size a* ∧ ¬(*bit a n*)} = {}
  by *blast*

lemma *qualified-bitCount*:
  bitCount v = card {*n . n < Nat.size v* ∧ *bit v n*}
  by (*metis* (*no-types, lifting*) *Collect-cong bitCount-def max-bit*)

lemma *card-eq*:
  assumes *finite x* ∧ *finite y* ∧ *finite z*
  assumes *x* ∪ *y = z*
  assumes *y* ∩ *x* = {}
  shows *card z* − *card y = card x*
  using *assms add-diff-cancel-right' card-Un-disjoint*
  by (*metis inf.commute*)

lemma *card-add*:
  assumes *finite x* ∧ *finite y* ∧ *finite z*
  assumes *x* ∪ *y = z*
  assumes *y* ∩ *x* = {}
  shows *card x + card y = card z*
  using *assms card-Un-disjoint*
  by (*metis inf.commute*)

lemma *card-add-inverses*:
  assumes *finite* {*n. Q n* ∧ ¬(*P n*)} ∧ *finite* {*n. Q n* ∧ *P n*} ∧ *finite* {*n. Q n*}
  shows *card* {*n. Q n* ∧ *P n*} + *card* {*n. Q n* ∧ ¬(*P n*)} = *card* {*n. Q n*}
  apply (*rule card-add*)
  using *assms* **apply** *simp*
  apply *auto*[*1*]
  by *auto*

lemma *ones-zero-sum-to-width*:
  bitCount a + zeroCount a = Nat.size a
proof −
  have *add-cards*: *card {n. (λn. n < size a) n ∧ (bit a n)} + card {n. (λn. n <*
size a) n ∧ ¬(bit a n)} = card {n. (λn. n < size a) n}
    apply (*rule card-add-inverses*) **by** *simp*
  then have *... = Nat.size a*
    by *auto*
 then show *?thesis*
    unfolding *bitCount-def zeroCount-def* **using** *max-bit*
    by (*metis (mono-tags, lifting) Collect-cong add-cards*)
qed

lemma *intersect-bitCount-helper*:
  card {n . n < Nat.size a} − bitCount a = card {n . n < Nat.size a ∧ ¬(bit a n)}
proof −
  have *size-def*: *Nat.size a = card {n . n < Nat.size a}*
    using *card-of-range* **by** *simp*
  have *bitCount-def*: *bitCount a = card {n . n < Nat.size a ∧ bit a n}*
    using *qualified-bitCount* **by** *auto*
  have *disjoint*: *{n . n < Nat.size a ∧ bit a n} ∩ {n . n < Nat.size a ∧ ¬(bit a*
n)} = {}
    using *disjoint-bit-sets* **by** *auto*
  have *union*: *{n . n < Nat.size a ∧ bit a n} ∪ {n . n < Nat.size a ∧ ¬(bit a n)}*
= {n . n < Nat.size a}
    using *union-bit-sets* **by** *auto*
  show *?thesis*
    unfolding *bitCount-def*
    apply (*rule card-eq*)
    using *finite-range* **apply** *simp*
    using *union* **apply** *blast*
    using *disjoint* **by** *simp*
qed

lemma *intersect-bitCount*:
  Nat.size a − bitCount a = card {n . n < Nat.size a ∧ ¬(bit a n)}
  using *card-of-range intersect-bitCount-helper* **by** *auto*

hide-fact *intersect-bitCount-helper*

end

4.5   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* (λ*n*. (*n*, *ir-ref* (*kind g n*))) (*filter* (λ*id*. *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 (λ*id*. (*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* (λ*nid*. *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* (**infix** ∩$_s$ *20*)
  where
diffNodesInfo g1 g2 = {(*nid, kind g1 nid, kind g2 nid*) | *nid . nid* ∈ *diffNodesGraph g1 g2*}

fun *eqGraph* :: *IRGraph* ⇒ *IRGraph* ⇒ *bool* (**infix** ≈$_s$ *20*)
  where
eqGraph isabelle-graph graal-graph = ((*diffNodesGraph isabelle-graph graal-graph*) = {})

end

4.6   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* (λ*nid′. 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* $= (ID \times Seen \times 'a)$

inductive *Step*
  $:: ('a\ TraversalState \Rightarrow 'a) \Rightarrow IRGraph \Rightarrow 'a\ TraversalState \Rightarrow 'a\ TraversalState$
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
  $[\![kind\ g\ nid = BeginNode\ nid';$

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

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

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

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

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

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

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

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

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

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

49

analysis′ = sa (*nid, seen, analysis*)⟧
⟹ *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
⟦¬(*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 ⇒ i ⇒ i ⇒ o ⇒ bool*) *Step* **.**

end