

# Veriopt Theories

April 17, 2024

## Contents

<b>1</b>	<b>Additional Theorems about Computer Words</b>	<b>1</b>
1.1	Bit-Shifting Operators . . . . .	2
1.2	Fixed-width Word Theories . . . . .	2
1.2.1	Support Lemmas for Upper/Lower Bounds . . . . .	2
1.2.2	Support lemmas for take bit and signed take bit. . . . .	7
1.3	Java min and max operators on 64-bit values . . . . .	8
<b>2</b>	<b>java.lang.Long</b>	<b>8</b>
2.1	Long.highestOneBit . . . . .	9
2.2	Long.lowestOneBit . . . . .	12
2.3	Long.numberOfLeadingZeros . . . . .	13
2.4	Long.numberOfTrailingZeros . . . . .	13
2.5	Long.reverseBytes . . . . .	14
2.6	Long.bitCount . . . . .	14
2.7	Long.zeroCount . . . . .	14
<b>3</b>	<b>Operator Semantics</b>	<b>18</b>
3.1	Arithmetic Operators . . . . .	20
3.2	Bitwise Operators . . . . .	22
3.3	Comparison Operators . . . . .	22
3.4	Narrowing and Widening Operators . . . . .	25
3.5	Bit-Shifting Operators . . . . .	26
3.5.1	Examples of Narrowing / Widening Functions . . . . .	26
3.6	Fixed-width Word Theories . . . . .	28
3.6.1	Support Lemmas for Upper/Lower Bounds . . . . .	28
3.6.2	Support lemmas for take bit and signed take bit. . . . .	32
<b>4</b>	<b>Stamp Typing</b>	<b>33</b>

<b>5</b>	<b>Graph Representation</b>	<b>38</b>
5.1	IR Graph Nodes . . . . .	38
5.2	IR Graph Node Hierarchy . . . . .	47
5.3	IR Graph Type . . . . .	54
5.3.1	Example Graphs . . . . .	59
5.4	Structural Graph Comparison . . . . .	60
5.5	Control-flow Graph Traversal . . . . .	61

## 1 Additional Theorems about Computer Words

**theory** *JavaWords*

**imports**

*HOL-Library.Word*  
*HOL-Library.Signed-Division*  
*HOL-Library.Float*  
*HOL-Library.LaTeXsugar*

**begin**

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.

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

**abbreviation** *valid-int-widths* :: nat set **where**

*valid-int-widths* ≡ {1, 8, 16, 32, 64}

**type-synonym** *iwidth* = nat

**fun** *bit-bounds* :: nat ⇒ (int × int) **where**

*bit-bounds* bits = (((2 ^ bits) div 2) \* -1, ((2 ^ bits) div 2) - 1)

**definition** *logic-negate* :: ('a::len) word ⇒ 'a word **where**

*logic-negate* x = (if x = 0 then 1 else 0)

**fun** *int-signed-value* :: iwidth ⇒ int64 ⇒ int **where**

*int-signed-value* b v = sint (signed-take-bit (b - 1) v)

**fun** *int-unsigned-value* :: iwidth ⇒ int64 ⇒ int **where**

*int-unsigned-value* b v = uint v

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

**fun** *neg-one* :: *iwidth* ⇒ *int64* **where**  
*neg-one* *b* = *mask b*

## 1.1 Bit-Shifting Operators

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

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

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

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

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

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

Unsigned shift right.

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

**corollary** (*255* :: 8 *word*) >>> (*2* :: *nat*) = 63 **by** *code-simp*

Signed shift right.

**definition** *sshiftr* :: *'a* :: *len* *word* ⇒ *nat* ⇒ *'a* :: *len* *word* (**infix** >> 75) **where**  
*sshiftr* *w n* = *word-of-int* ((*sint w*) *div* ( $2^n$ ))

**corollary** (*128* :: 8 *word*) >> 2 = 0xE0 **by** *code-simp*

## 1.2 Fixed-width Word Theories

### 1.2.1 Support Lemmas for Upper/Lower Bounds

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

**lemma** *size64*: *size v* = 64 **for** *v* :: 64 *word*  
**by** (*metis* *numeral-times-numeral semiring-norm*(12) *semiring-norm*(13) *size32*  
*len-of-numeral-defs*(3)  
*size-word.rep-eq*)

**lemma** *lower-bounds-equiv*:  
**assumes**  $0 < N$   
**shows**  $-((2::int) \wedge (N-1)) = (2::int) \wedge N \text{ div } 2 * - 1$   
**by** (*simp add: assms int-power-div-base*)

**lemma** *upper-bounds-equiv*:  
**assumes**  $0 < N$   
**shows**  $(2::int) \wedge (N-1) = (2::int) \wedge N \text{ div } 2$   
**by** (*simp add: assms int-power-div-base*)

Some min/max bounds for 64-bit words

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

**lemma** *bit-bounds-max64*:  $((snd (bit-bounds 64))) \geq (sint (v::int64))$   
**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.

**corollary** *sint(signed-take-bit 7 (128 :: int8)) = -128* **by** *code-simp*

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

**lemma** *signed-take-bit-int-less-exp-word*:  
**fixes** *ival :: 'a :: len word*  
**assumes**  $n < LENGTH('a)$   
**shows**  $sint(signed-take-bit n ival) < (2::int) \wedge n$   
**apply** *transfer* **using** *assms* **apply** *auto*  
**by** (*metis min.commute signed-take-bit-signed-take-bit signed-take-bit-int-less-exp*)

**lemma** *signed-take-bit-int-greater-eq-minus-exp-word*:  
**fixes** *ival :: 'a :: len word*  
**assumes**  $n < LENGTH('a)$   
**shows**  $-(2 \wedge n) \leq sint(signed-take-bit n ival)$   
**apply** *transfer* **using** *assms* **apply** *auto*  
**by** (*metis min.commute signed-take-bit-signed-take-bit signed-take-bit-int-greater-eq-minus-exp*)

```

lemma signed-take-bit-range:
  fixes ival :: 'a :: len word
  assumes  $n < LENGTH('a)$ 
  assumes  $val = sint(signed-take-bit\ n\ ival)$ 
  shows  $-(2 \wedge n) \leq val \wedge val < 2 \wedge n$ 
  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  $fst\ (bit-bounds\ n) \leq val \wedge val \leq snd\ (bit-bounds\ n)$ 
  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 assms apply auto
  apply (smt (verit, ccfv-threshold) sint-greater-eq diff-less len-gt-0 power-strict-increasing
power-less-imp-less-exp signed-take-bit-range len-num1 One-nat-def)
  by (smt (verit, ccfv-threshold) neg-equal-0-iff-equal power-0 signed-minus-1 sint-0
not-gr-zero
word-exp-length-eq-0 diff-less diff-zero len-gt-0 sint-less power-strict-increasing
signed-take-bit-range power-less-imp-less-exp)

```

Some lemmas to relate (int) bit bounds to bit-shifting values.

**lemma** *bit-bounds-lower*:  
**assumes**  $0 < \text{bits}$   
**shows**  $\text{word-of-int } (\text{fst } (\text{bit-bounds } \text{bits})) = ((-1) \ll (\text{bits} - 1))$   
**unfolding** *bit-bounds.simps fst-conv*  
**by** (*metis (mono-tags, opaque-lifting) assms(1) mult-1 mult-minus1-right mult-minus-left of-int-minus of-int-power shiftl-power upper-bounds-equiv word-numeral-alt*)

**lemma** *two-exp-div*:  
**assumes**  $0 < \text{bits}$   
**shows**  $((2::\text{int}) \wedge \text{bits } \text{div } (2::\text{int})) = (2::\text{int}) \wedge (\text{bits} - \text{Suc } 0)$   
**using** *assms* **by** (*auto simp: int-power-div-base*)

**declare** [*show-types*]

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

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

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

A simplification lemma for *new\_int*, showing that upper bits can be ignored.

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

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

**lemma** *take-bit-same-bounds*:  
**fixes** *ival* :: 'a :: len word  
**assumes**  $n = \text{LENGTH}('a)$   
**assumes**  $\text{ival2} = \text{take-bit } n \text{ ival}$   
**shows**  $\text{fst } (\text{bit-bounds } n) \leq \text{sint ival2} \wedge \text{sint ival2} \leq \text{snd } (\text{bit-bounds } n)$   
**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**  $\text{sint } (v :: 'a :: \text{len word}) < M$   
**assumes**  $\text{LENGTH}('a) < \text{LENGTH}('b)$   
**shows**  $\text{sint } ((\text{scast } v) :: 'b :: \text{len word}) < M$   
**using** *assms unfolding Word.scast-eq Word.sint-sbintrunc'* **by** (*simp add: sint-wint*)

**lemma** *scast-min-bound*:  
**assumes**  $M \leq \text{sint } (v :: 'a :: \text{len word})$   
**assumes**  $\text{LENGTH}('a) < \text{LENGTH}('b)$   
**shows**  $M \leq \text{sint } ((\text{scast } v) :: 'b :: \text{len word})$   
**using** *assms unfolding Word.scast-eq Word.sint-sbintrunc'* **by** (*simp add: sint-wint*)

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

**lemma** *scast-bigger-min-bound*:  
**assumes**  $(\text{result} :: 'b :: \text{len word}) = \text{scast } (v :: 'a :: \text{len word})$   
**shows**  $-(2 \wedge \text{LENGTH}('a) \text{ div } 2) \leq \text{sint result}$   
**by** (*metis upper-bounds-equiv assms len-gt-0 nat-less-le not-less scast-max-bound scast-min-bound sint-ge*)

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

## 1.2.2 Support lemmas for take bit and signed take bit.

Lemmas for removing redundant *take\_bit* wrappers.

```

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

```

```

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

```

```

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

```

```

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

```

```

lemma signed-take-bit[simp]:
  fixes  $x :: 'a :: \text{len word}$ 
  assumes  $0 < b$ 
  shows  $\text{signed-take-bit } (b - 1) (\text{take-bit } b x) = \text{signed-take-bit } (b - 1) x$ 
  using assms apply auto
  by (smt (verit, ccfv-threshold) Suc-diff-1 assms lessI linorder-not-less signed-take-bit-take-bit diff-Suc-less Suc-pred One-nat-def)

```

```

lemma mod-larger-ignore:
  fixes  $a :: \text{int}$ 
  fixes  $m n :: \text{nat}$ 
  assumes  $n < m$ 
  shows  $(a \bmod 2^m) \bmod 2^n = a \bmod 2^n$ 
  by (meson assms le-imp-power-dvd less-or-eq-imp-le mod-mod-cancel)

```



```

lemma mod-dist-over-add:
  fixes a b c :: int64
  fixes n :: nat
  assumes 1:  $0 < n$ 
  assumes 2:  $n < 64$ 
  shows  $(a \bmod 2^n + b) \bmod 2^n = (a + b) \bmod 2^n$ 
proof -
  have 3:  $(0 :: \text{int64}) < 2^n$ 
    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

```

### 1.3 Java min and max operators on 64-bit values

Java uses signed comparison, so we define a convenient abbreviation for this to avoid accidental mistakes, because by default the Isabelle min/max will assume unsigned words.

```

abbreviation javaMin64 :: int64  $\Rightarrow$  int64  $\Rightarrow$  int64 where
  javaMin64 a b  $\equiv$  (if a  $\leq$  b then a else b)

```

```

abbreviation javaMax64 :: int64  $\Rightarrow$  int64  $\Rightarrow$  int64 where
  javaMax64 a b  $\equiv$  (if a  $\leq$  b then b else a)

```

**end**

## 2 java.lang.Long

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

```

theory JavaLong
  imports JavaWords
           HOL-Library.FSet
begin

```

```

lemma negative-all-set-32:
   $n < 32 \implies \text{bit } (-1 :: \text{int32}) n$ 
  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

```

$MinOrHighest\ s\ m = (if\ s = \{\} then\ m\ else\ Min\ s)$

**lemma** *MaxOrNegEmpty*:  
 $MaxOrNeg\ s = -1 \longleftrightarrow s = \{\}$   
**unfolding** *MaxOrNeg-def* **by** *auto*

## 2.1 Long.highestOneBit

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

**lemma** *highestOneBitInvar*:  
 $highestOneBit\ v = j \implies (\forall i::nat. (int\ i > j \longrightarrow \neg (bit\ v\ i)))$   
**apply** (*induction size v; auto*) **unfolding** *highestOneBit-def*  
**by** (*metis linorder-not-less MaxOrNeg-def empty-iff finite-bit-word mem-Collect-eq of-nat-mono Max-ge*)

**lemma** *highestOneBitNeg*:  
 $highestOneBit\ v = -1 \longleftrightarrow v = 0$   
**unfolding** *highestOneBit-def MaxOrNeg-def*  
**by** (*metis Collect-empty-eq-bot bit-0-eq bit-word-eqI int-ops(2) negative-eq-positive one-neq-zero*)

**lemma** *higherBitsFalse*:  
**fixes**  $v :: 'a :: len\ word$   
**shows**  $i > size\ v \implies \neg (bit\ v\ i)$   
**by** (*simp add: bit-word.rep-eq size-word.rep-eq*)

**lemma** *highestOneBitN*:  
**assumes**  $bit\ v\ n$   
**assumes**  $\forall i::nat. (int\ i > n \longrightarrow \neg (bit\ v\ i))$   
**shows**  $highestOneBit\ v = n$   
**unfolding** *highestOneBit-def MaxOrNeg-def*  
**by** (*metis Max-ge Max-in all-not-in-conv assms(1) assms(2) finite-bit-word mem-Collect-eq of-nat-less-iff order-less-le*)

**lemma** *highestOneBitSize*:  
**assumes**  $bit\ v\ n$   
**assumes**  $n = size\ v$   
**shows**  $highestOneBit\ v = n$   
**by** (*metis assms(1) assms(2) not-bit-length wsst-TYs(3)*)

**lemma** *highestOneBitMax*:  
 $highestOneBit\ v < size\ v$   
**unfolding** *highestOneBit-def MaxOrNeg-def*  
**using** *higherBitsFalse*  
**by** (*simp add: bit-imp-le-length size-word.rep-eq*)

```

lemma highestOneBitAtLeast:
  assumes bit v n
  shows highestOneBit v ≥ n
proof (induction size v)
  case 0
  then show ?case by simp
next
  case (Suc x)
  then have  $\forall i. \text{bit } v \ i \longrightarrow i < \text{Suc } x$ 
    by (simp add: bit-imp-le-length wsst-TYS(3))
  then show ?case
    unfolding highestOneBit-def MaxOrNeg-def
    using assms by auto
qed

```

```

lemma highestOneBitElim:
  highestOneBit v = n
   $\implies ((n = -1 \wedge v = 0) \vee (n \geq 0 \wedge \text{bit } v \ n))$ 
  unfolding highestOneBit-def MaxOrNeg-def
  by (metis Max-in finite-bit-word le0 le-minus-one-simps(3) mem-Collect-eq of-nat-0-le-iff of-nat-eq-iff)

```

A recursive implementation of `highestOneBit` that is suitable for code generation.

```

fun highestOneBitRec :: nat  $\Rightarrow$  ('a::len) word  $\Rightarrow$  int where
  highestOneBitRec n v =
    (if bit v n then n
     else if n = 0 then -1
     else highestOneBitRec (n - 1) v)

```

```

lemma highestOneBitRecTrue:
  highestOneBitRec n v = j  $\implies j \geq 0 \implies \text{bit } v \ j$ 
proof (induction n)
  case 0
  then show ?case
    by (metis diff-0 highestOneBitRec.simps leD of-nat-0-eq-iff of-nat-0-le-iff zle-diff1-eq)

```

```

next
  case (Suc n)
  then show ?case
    by (metis diff-Suc-1 highestOneBitRec.elims nat.discI nat-int)
qed

```

```

lemma highestOneBitRecN:
  assumes bit v n
  shows highestOneBitRec n v = n
  by (simp add: assms)

```

```

lemma highestOneBitRecMax:

```

*highestOneBitRec*  $n v \leq n$   
**by** (*induction*  $n$ ; *simp*)

**lemma** *highestOneBitRecElim*:  
**assumes** *highestOneBitRec*  $n v = j$   
**shows**  $((j = -1 \wedge v = 0) \vee (j \geq 0 \wedge \text{bit } v j))$   
**using** *assms highestOneBitRecTrue* **by** *blast*

**lemma** *highestOneBitRecZero*:  
 $v = 0 \implies \text{highestOneBitRec } (\text{size } v) v = -1$   
**by** (*induction rule: highestOneBitRec.induct*; *simp*)

**lemma** *highestOneBitRecLess*:  
**assumes**  $\neg \text{bit } v n$   
**shows** *highestOneBitRec*  $n v = \text{highestOneBitRec } (n - 1) v$   
**using** *assms* **by** *force*

Some lemmas that use masks to restrict *highestOneBit* and relate it to *highestOneBitRec*.

**lemma** *highestOneBitMask*:  
**assumes**  $\text{size } v = n$   
**shows** *highestOneBit*  $v = \text{highestOneBit } (\text{and } v (\text{mask } n))$   
**by** (*metis assms dual-order.refl lt2p-lem mask-eq-iff size-word.rep-eq*)

**lemma** *maskSmaller*:  
**fixes**  $v :: 'a :: \text{len word}$   
**assumes**  $\neg \text{bit } v n$   
**shows**  $\text{and } v (\text{mask } (\text{Suc } n)) = \text{and } v (\text{mask } n)$   
**unfolding** *bit-eq-iff*  
**by** (*metis assms bit-and-iff bit-mask-iff less-Suc-eq*)

**lemma** *highestOneBitSmaller*:  
**assumes**  $\text{size } v = \text{Suc } n$   
**assumes**  $\neg \text{bit } v n$   
**shows** *highestOneBit*  $v = \text{highestOneBit } (\text{and } v (\text{mask } n))$   
**by** (*metis assms highestOneBitMask maskSmaller*)

**lemma** *highestOneBitRecMask*:  
**shows** *highestOneBit*  $(\text{and } v (\text{mask } (\text{Suc } n))) = \text{highestOneBitRec } n v$   
**proof** (*induction*  $n$ )  
**case**  $0$   
**then have** *highestOneBit*  $(\text{and } v (\text{mask } (\text{Suc } 0))) = \text{highestOneBitRec } 0 v$   
**apply** *auto*  
**apply** (*smt (verit, ccfv-threshold) neg-equal-zero negative-eq-positive bit-1-iff*  
*bit-and-iff*  
*highestOneBitN*)  
**by** (*simp add: bit-iff-and-push-bit-not-eq-0 highestOneBitNeg*)  
**then show** *?case*  
**by** *presburger*

```

next
  case (Suc n)
  then show ?case
  proof (cases bit v (Suc n))
    case True
    have 1: highestOneBitRec (Suc n) v = Suc n
      by (simp add: True)
    have  $\forall i::nat. (int i > (Suc n) \longrightarrow \neg (bit (and v (mask (Suc (Suc n)))) i))$ 
      by (simp add: bit-and-iff bit-mask-iff)
    then have 2: highestOneBit (and v (mask (Suc (Suc n)))) = Suc n
      using True highestOneBitN
      by (metis bit-take-bit-iff lessI take-bit-eq-mask)
    then show ?thesis
      using 1 2 by auto
  next
  case False
  then show ?thesis
    by (simp add: Suc maskSmaller)
qed

```

Finally - we can use the mask lemmas to relate highestOneBitRec to its spec.

```

lemma highestOneBitImpl[code]:
  highestOneBit v = highestOneBitRec (size v) v
  by (metis highestOneBitMask highestOneBitRecMask maskSmaller not-bit-length
  wsst-TYs(3))

```

```

lemma highestOneBit (0x5 :: int8) = 2 by code-simp

```

## 2.2 Long.lowestOneBit

```

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

```

## 2.3 Long.numberOfLeadingZeros

```

definition numberOfLeadingZeros :: ('a::len) word  $\Rightarrow$  nat where
  numberOfLeadingZeros v = nat (Nat.size v - highestOneBit v - 1)

```

```

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

```

**lemma** *MaxOrNeg-max*:  $s \neq \{\}$   $\implies$   $\text{MaxOrNeg } s = \text{Max } s$   
**by** (*simp add: MaxOrNeg-def*)

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

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

**lemma** *numberOfLeadingZeros* ( $0::64$  word) =  $64$   
**unfolding** *numberOfLeadingZeros-def* **by** (*simp add: highestOneBitImpl size64*)

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

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

**lemma** *numberOfLeadingZeros-range*:  $0 \leq \text{numberOfLeadingZeros } a \wedge \text{numberOfLeadingZeros } a \leq \text{Nat.size } a$   
**unfolding** *numberOfLeadingZeros-def* **apply** *auto*  
**apply** (*induction highestOneBit a*) **apply** (*simp add: numberOfLeadingZeros-def*)  
**by** (*metis (mono-tags, opaque-lifting) leD negative-zless int-eq-iff diff-right-commute diff-self diff-zero nat-le-iff le-iff-diff-le-0 minus-diff-eq nat-0-le nat-le-linear of-nat-0-le-iff MaxOrNeg-def highestOneBit-def*)

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

## 2.4 Long.numberOfTrailingZeros

**definition** *numberOfTrailingZeros* ::  $(\text{'a}::\text{len}) \text{ word} \Rightarrow \text{nat}$  **where**  
 $\text{numberOfTrailingZeros } v = \text{lowestOneBit } v$

**lemma** *lowestOneBit-bot*:  $\text{lowestOneBit } (0::64 \text{ word}) = 64$   
**unfolding** *lowestOneBit-def MinOrHighest-def*  
**by** (*simp add: size64*)

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

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

**lemma** *numberOfTrailingZeros*  $(0::64 \text{ word}) = 64$   
**unfolding** *numberOfTrailingZeros-def*  
**using** *lowestOneBit-bot* **by** *simp*

## 2.5 Long.reverseBytes

**fun** *reverseBytes-fun* ::  $('a::len) \text{ word} \Rightarrow nat \Rightarrow ('a::len) \text{ word} \Rightarrow ('a::len) \text{ word}$   
**where**  
*reverseBytes-fun*  $v$   $b$   $flip = (if (b = 0) then (flip) else$   
 $(reverseBytes-fun (v >> 8) (b - 8) (or (flip << 8) (take-bit 8$   
 $v))))$

## 2.6 Long.bitCount

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

**fun** *bitCount-fun* ::  $('a::len) \text{ word} \Rightarrow nat \Rightarrow nat$  **where**  
*bitCount-fun*  $v$   $n = (if (n = 0) then$   
 $(if (\text{bit } v \ n) then 1 else 0) else$   
 $if (\text{bit } v \ n) then (1 + \text{bitCount-fun } (v) (n - 1))$   
 $else (0 + \text{bitCount-fun } (v) (n - 1)))$

**lemma** *bitCount 0 = 0*  
**unfolding** *bitCount-def*  
**by**  $(metis \text{card.empty zero-no-bits})$

## 2.7 Long.zeroCount

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

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

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

**lemma** *negone-all-bits*:  
 $\{n . \text{bit } (-1::('a::len) \text{ word}) \ n\} = \{n . 0 \leq n \wedge n < \text{LENGTH}('a)\}$   
**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 ≤ n ∧ n < x}
by simp

lemma range-of-nat:
{(n::nat) . 0 ≤ n ∧ 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 ∧ n ≤ x}
proof -
have finite: finite {n . y ≤ n ∧ n < x}
by auto
have nonempty: {n . y ≤ n ∧ n < x} ≠ {}
using assms by blast
have simprep: {n . y < n ∧ n ≤ 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 ≤ bitCount n ∧ bitCount n ≤ 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*:  
 $n > \text{highestOneBit } a \implies \neg(\text{bit } a \ n)$   
**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**  $n < \text{lowestOneBit } a$   
**shows**  $\neg(\text{bit } a \ n)$   
**proof** (*cases {i. bit a i} = {}*)  
**case** *True*  
**then show** *?thesis* **by** *simp*  
**next**  
**case** *False*  
**have**  $n < \text{Min } (\text{Collect } (\text{bit } a)) \implies \neg \text{bit } a \ n$   
**using** *False by auto*  
**then show** *?thesis*  
**by** (*metis False MinOrHighest-def assms lowestOneBit-def*)  
**qed**

**lemma** *union-bit-sets*:  
**fixes**  $a :: ('a::\text{len}) \text{ word}$   
**shows**  $\{n . n < \text{Nat.size } a \wedge \text{bit } a \ n\} \cup \{n . n < \text{Nat.size } a \wedge \neg(\text{bit } a \ n)\} = \{n . n < \text{Nat.size } a\}$   
**by** *fastforce*

**lemma** *disjoint-bit-sets*:  
**fixes**  $a :: ('a::\text{len}) \text{ word}$   
**shows**  $\{n . n < \text{Nat.size } a \wedge \text{bit } a \ n\} \cap \{n . n < \text{Nat.size } a \wedge \neg(\text{bit } a \ n)\} = \{\}$   
**by** *blast*

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

**lemma** *card-eq*:  
**assumes**  $\text{finite } x \wedge \text{finite } y \wedge \text{finite } z$   
**assumes**  $x \cup y = z$   
**assumes**  $y \cap x = \{\}$   
**shows**  $\text{card } z - \text{card } y = \text{card } x$   
**using** *assms add-diff-cancel-right' card-Un-disjoint*  
**by** (*metis inf commute*)

**lemma** *card-add*:  
**assumes**  $\text{finite } x \wedge \text{finite } y \wedge \text{finite } z$   
**assumes**  $x \cup y = z$   
**assumes**  $y \cap x = \{\}$   
**shows**  $\text{card } x + \text{card } y = \text{card } z$   
**using** *assms card-Un-disjoint*

by (*metis inf.commute*)

**lemma** *card-add-inverses*:

**assumes** *finite*  $\{n. Q\ n \wedge \neg(P\ n)\} \wedge$  *finite*  $\{n. Q\ n \wedge P\ n\} \wedge$  *finite*  $\{n. Q\ n\}$   
**shows**  $\text{card } \{n. Q\ n \wedge P\ n\} + \text{card } \{n. Q\ n \wedge \neg(P\ n)\} = \text{card } \{n. Q\ n\}$   
**apply** (*rule card-add*)  
**using** *assms apply simp*  
**apply** *auto[1]*  
**by** *auto*

**lemma** *ones-zero-sum-to-width*:

$\text{bitCount } a + \text{zeroCount } a = \text{Nat.size } a$

**proof** –

**have** *add-cards*:  $\text{card } \{n. (\lambda n. n < \text{size } a) n \wedge (\text{bit } a\ n)\} + \text{card } \{n. (\lambda n. n < \text{size } a) n \wedge \neg(\text{bit } a\ n)\} = \text{card } \{n. (\lambda n. n < \text{size } a) n\}$   
**apply** (*rule card-add-inverses*) **by** *simp*  
**then have**  $\dots = \text{Nat.size } a$   
**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*:

$\text{card } \{n . n < \text{Nat.size } a\} - \text{bitCount } a = \text{card } \{n . n < \text{Nat.size } a \wedge \neg(\text{bit } a\ n)\}$

**proof** –

**have** *size-def*:  $\text{Nat.size } a = \text{card } \{n . n < \text{Nat.size } a\}$   
**using** *card-of-range* **by** *simp*  
**have** *bitCount-def*:  $\text{bitCount } a = \text{card } \{n . n < \text{Nat.size } a \wedge \text{bit } a\ n\}$   
**using** *qualified-bitCount* **by** *auto*  
**have** *disjoint*:  $\{n . n < \text{Nat.size } a \wedge \text{bit } a\ n\} \cap \{n . n < \text{Nat.size } a \wedge \neg(\text{bit } a\ n)\} = \{\}$   
**using** *disjoint-bit-sets* **by** *auto*  
**have** *union*:  $\{n . n < \text{Nat.size } a \wedge \text{bit } a\ n\} \cup \{n . n < \text{Nat.size } a \wedge \neg(\text{bit } a\ n)\} = \{n . n < \text{Nat.size } a\}$   
**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*:

$\text{Nat.size } a - \text{bitCount } a = \text{card } \{n . n < \text{Nat.size } a \wedge \neg(\text{bit } a\ n)\}$   
**using** *card-of-range intersect-bitCount-helper* **by** *auto*

**hide-fact** *intersect-bitCount-helper*

**end**

### 3 Operator Semantics

**theory** *Values*

**imports**

*JavaLong*

**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** *objref* = *nat option*

**type-synonym** *length* = *nat*

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

*UndefVal* |

*IntVal iwidth int64* |

*ObjRef objref* |

*ObjStr string* |

*ArrayVal length Value list*

**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*

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** *array-length* :: *Value*  $\Rightarrow$  *Value* **where**  
*array-length* (*ArrayVal* *len list*) = *new-int* 32 (*word-of-nat* *len*)

**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*

**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  
**using** *assms* **apply** *auto*  
**by** (*metis* (*no-types*, *lifting*) *Suc-pred* *diff-Suc-1* *signed-take-take-bit* *assms* *signed-minus-1*  
*int-signed-value.simps* *mask-eq-take-bit-minus-one* *signed-take-bit-of-minus-1*)

**lemma** *word-unsigned*:  
**shows**  $\forall$  *b1 v1*. (*IntVal* *b1* (*word-of-int* (*int-unsigned-value* *b1 v1*))) = *IntVal* *b1*  
*v1*  
**by** *simp*

### 3.1 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) =
    (if v2 = 0 then UndefVal else
     new-int-bin b1 b2 (word-of-int
      ((int-signed-value b1 v1) sdiv (int-signed-value b2 v2)))) |
  intval-div - - = UndefVal
```

```
value intval-div (IntVal 32 5) (IntVal 32 0)
```

```
fun intval-mod :: Value ⇒ Value ⇒ Value where
  intval-mod (IntVal b1 v1) (IntVal b2 v2) =
    (if v2 = 0 then UndefVal else
     new-int-bin b1 b2 (word-of-int
      ((int-signed-value b1 v1) smod (int-signed-value b2 v2)))) |
  intval-mod - - = UndefVal
```

```

fun intval-mul-high :: Value ⇒ Value ⇒ Value where
  intval-mul-high (IntVal b1 v1) (IntVal b2 v2) = (
    if (b1 = b2 ∧ b1 = 64) then (
      if (((int-signed-value b1 v1) < 0) ∨ ((int-signed-value b2 v2) < 0))
      then (

        let x1 = (v1 >> 32)           in
        let x2 = (and v1 4294967295)  in
        let y1 = (v2 >> 32)           in
        let y2 = (and v2 4294967295)  in
        let z2 = (x2 * y2)             in
        let t  = (x1 * y2 + (z2 >>> 32)) in
        let z1 = (and t 4294967295)    in
        let z0 = (t >> 32)             in
        let z1 = (z1 + (x2 * y1))      in

        let result = (x1 * y1 + z0 + (z1 >> 32)) in

        (new-int b1 result)
      ) else (

        let x1 = (v1 >>> 32)           in
        let y1 = (v2 >>> 32)           in
        let x2 = (and v1 4294967295)  in
        let y2 = (and v2 4294967295)  in
        let A  = (x1 * y1)             in
        let B  = (x2 * y2)             in
        let C  = ((x1 + x2) * (y1 + y2)) in
        let K  = (C - A - B)           in

        let result = (((B >>> 32) + K) >>> 32) + A in

        (new-int b1 result)
      )
    ) else (
      if (b1 = b2 ∧ b1 = 32) then (

        let newv1 = (word-of-int (int-signed-value b1 v1)) in
        let newv2 = (word-of-int (int-signed-value b1 v2)) in
        let r = (newv1 * newv2) in

        let result = (r >> 32) in

        (new-int b1 result)
      ) else UndefVal)
    ) |
  intval-mul-high - - = UndefVal

```

```

fun intval-reverse-bytes :: Value ⇒ Value where

```

```

intval-reverse-bytes (IntVal b1 v1) = (new-int b1 (reverseBytes-fun v1 b1 0)) |
intval-reverse-bytes - = UndefVal

```

```

fun intval-bit-count :: Value ⇒ Value where
  intval-bit-count (IntVal b1 v1) = (new-int 32 (word-of-nat (bitCount-fun v1 64)))
  |
  intval-bit-count - = 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

```

### 3.2 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

```

### 3.3 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)

fun intval-is-null :: Value ⇒ Value where
    intval-is-null (ObjRef (v)) = (if (v=(None)) then bool-to-val True else bool-to-val
    False) |
    intval-is-null - = UndefVal

fun intval-test :: Value ⇒ Value ⇒ Value where
    intval-test (IntVal b1 v1) (IntVal b2 v2) = bool-to-val-bin b1 b2 ((and v1 v2) =
    0) |
    intval-test - - = UndefVal

fun intval-normalize-compare :: Value ⇒ Value ⇒ Value where
    intval-normalize-compare (IntVal b1 v1) (IntVal b2 v2) =
        (if (b1 = b2) then new-int 32 (if (v1 < v2) then -1 else (if (v1 = v2) then 0
        else 1))
        else UndefVal) |
    intval-normalize-compare - - = UndefVal

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

definition default-values :: Value list where
    default-values = [new-int 32 0, new-int 64 0, ObjRef None]

definition short-types-32 :: string list where
    short-types-32 = ["Z", "I", "C", "B", "S"]

definition short-types-64 :: string list where
    short-types-64 = ["J"]

fun default-value :: string ⇒ Value where

```



```

default-value n = (if (find-index n short-types-32) < (length short-types-32)
  then (default-values!0) else
  (if (find-index n short-types-64) < (length short-types-64)
    then (default-values!1)
    else (default-values!2)))

fun populate-array :: nat ⇒ Value list ⇒ string ⇒ Value list where
  populate-array len a s = (if (len = 0) then (a)
    else (a @ (populate-array (len-1) [default-value s] s)))

fun intval-new-array :: Value ⇒ string ⇒ Value where
  intval-new-array (IntVal b1 v1) s = (ArrayVal (nat (int-signed-value b1 v1))
    (populate-array (nat (int-signed-value b1 v1)) [] s)) |
  intval-new-array - - = UndefVal

fun intval-load-index :: Value ⇒ Value ⇒ Value where
  intval-load-index (ArrayVal len cons) (IntVal b1 v1) = (if (v1 ≥ (word-of-nat
    len)) then (UndefVal)
    else (cons!(nat (int-signed-value b1
    v1)))) |
  intval-load-index - - = UndefVal

fun intval-store-index :: Value ⇒ Value ⇒ Value ⇒ Value where
  intval-store-index (ArrayVal len cons) (IntVal b1 v1) val =
    (if (v1 ≥ (word-of-nat len)) then (UndefVal)
    else (ArrayVal len (list-update cons (nat (int-signed-value b1
    v1)) (val)))) |
  intval-store-index - - - = UndefVal

lemma intval-equals-result:
  assumes intval-equals v1 v2 = r
  assumes r ≠ UndefVal
  shows r = IntVal 32 0 ∨ r = IntVal 32 1
proof -
  obtain b1 i1 where i1: v1 = IntVal b1 i1
  by (metis assms intval-bits.elims intval-equals.simps(2,3,4,5))
  obtain b2 i2 where i2: v2 = IntVal b2 i2
  by (smt (z3) assms intval-equals.elims)
  then have b1 = b2
  by (metis i1 assms bool-to-val-bin.elims intval-equals.simps(1))
  then show ?thesis
  using assms(1) bool-to-val.elims i1 i2 by auto
qed

```

### 3.4 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.

Some sanity checks that  $take\_bitN$  and  $signed\_take\_bit(N - 1)$  match up as expected.

**corollary**  $sint (signed\_take\_bit\ 0\ (1 :: int32)) = -1$  **by** *code-simp*

**corollary**  $sint (signed\_take\_bit\ 7\ ((256 + 128) :: int64)) = -128$  **by** *code-simp*

**corollary**  $sint (take\_bit\ 7\ ((256 + 128 + 64) :: int64)) = 64$  **by** *code-simp*

**corollary**  $sint (take\_bit\ 8\ ((256 + 128 + 64) :: int64)) = 128 + 64$  **by** *code-simp*

**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*

**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* **apply** (*cases* *val*; *auto*) **apply** (*meson* *le-trans*)**+** **by** *presburger*

**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*

**by** (*metis* *intval-bits.simps* *intval-sign-extend.elims* *is-IntVal-def* *assms*)

**lemma** *intval-zero-extend-ok*:

**assumes** *intval-zero-extend* *inBits* *outBits* *val*  $\neq$  *UndefVal*

**shows**  $0 < inBits \wedge$

$inBits \leq outBits \wedge outBits \leq 64 \wedge$

```

    is-IntVal val ∧
    intval-bits val = inBits
  by (metis intval-bits.simps intval-zero-extend.elims is-IntVal-def assms)

```

### 3.5 Bit-Shifting Operators

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 ⇒ int64 ⇒ 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

```

#### 3.5.1 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*)

```

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

end

```

### 3.6 Fixed-width Word Theories

```

theory ValueThms
imports Values
begin

```

#### 3.6.1 Support Lemmas for Upper/Lower Bounds

```

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

```

```

lemma size64: size  $v = 64$  for  $v :: 64$  word
by (simp add: size64)

```

```

lemma lower-bounds-equiv:
assumes  $0 < N$ 
shows  $-(((2::int) \wedge (N-1))) = (2::int) \wedge N \text{ div } 2 * - 1$ 
by (simp add: assms int-power-div-base)

```

```

lemma upper-bounds-equiv:
assumes  $0 < N$ 
shows  $(2::int) \wedge (N-1) = (2::int) \wedge N \text{ div } 2$ 
by (simp add: assms int-power-div-base)

```

Some min/max bounds for 64-bit words

```

lemma bit-bounds-min64:  $((fst (bit-bounds 64))) \leq (sint (v::int64))$ 
using sint-ge[of v] by simp

```

```

lemma bit-bounds-max64:  $((snd (bit-bounds 64))) \geq (sint (v::int64))$ 
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) 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) ≤ sint(signed-take-bit n ival)
```

```
  using signed-take-bit-int-greater-eq-minus-exp-word assms by blast
```

```
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) ≤ val ∧ val < 2 ^ n
```

```
  by (auto simp add: assms signed-take-bit-int-greater-eq-minus-exp-word
      signed-take-bit-int-less-exp-word)
```

A `bit_bounds` version of the above lemma.

```
lemma signed-take-bit-bounds:
```

```
  fixes ival :: 'a :: len word
```

```
  assumes n ≤ LENGTH('a)
```

```
  assumes 0 < n
```

```
  assumes val = sint(signed-take-bit (n - 1) ival)
```

```
  shows fst (bit-bounds n) ≤ val ∧ val ≤ snd (bit-bounds n)
```

```
  by (metis bit-bounds.simps fst-conv less-imp-diff-less nat-less-le sint-ge sint-lt
      snd-conv
```

```
      zle-diff1-eq upper-bounds-equiv lower-bounds-equiv signed-take-bit-range assms)
```

```
lemma signed-take-bit-bounds64:
```

```
  fixes ival :: int64
```

```
  assumes n ≤ 64
```

```
  assumes 0 < n
```

```
  assumes val = sint(signed-take-bit (n - 1) ival)
```

```
  shows fst (bit-bounds n) ≤ val ∧ val ≤ snd (bit-bounds n)
```

```
  by (metis size64 word-size signed-take-bit-bounds assms)
```

**lemma** *int-signed-value-bounds*:  
**assumes**  $b1 \leq 64$   
**assumes**  $0 < b1$   
**shows**  $\text{fst } (\text{bit-bounds } b1) \leq \text{int-signed-value } b1 \ v2 \wedge$   
 $\text{int-signed-value } b1 \ v2 \leq \text{snd } (\text{bit-bounds } b1)$   
**using** *signed-take-bit-bounds64* **by** (*simp add: assms*)

**lemma** *int-signed-value-range*:  
**fixes**  $\text{ival} :: \text{int64}$   
**assumes**  $\text{val} = \text{int-signed-value } n \ \text{ival}$   
**shows**  $-(2^{n-1}) \leq \text{val} \wedge \text{val} < 2^n$   
**using** *assms int-signed-value-range* **by** *blast*

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

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

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

A simplification lemma for *new\_int*, showing that upper bits can be ignored.

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

**lemma** *take-bit-same-size-range*:  
**fixes**  $\text{ival} :: 'a :: \text{len word}$   
**assumes**  $n = \text{LENGTH('a)}$   
**assumes**  $\text{ival2} = \text{take-bit } n \ \text{ival}$   
**shows**  $-(2^{n \div 2}) \leq \text{sint } \text{ival2} \wedge \text{sint } \text{ival2} < 2^{n \div 2}$

**using** *lower-bounds-equiv sint-ge sint-lt* **by** (*auto simp add: assms*)

**lemma** *take-bit-same-bounds*:

**fixes** *ival* :: 'a :: len word

**assumes**  $n = \text{LENGTH}('a)$

**assumes**  $\text{ival2} = \text{take-bit } n \text{ ival}$

**shows**  $\text{fst} (\text{bit-bounds } n) \leq \text{sint ival2} \wedge \text{sint ival2} \leq \text{snd} (\text{bit-bounds } n)$

**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**  $\text{sint} (v :: 'a :: \text{len word}) < M$

**assumes**  $\text{LENGTH}('a) < \text{LENGTH}('b)$

**shows**  $\text{sint} ((\text{scast } v) :: 'b :: \text{len word}) < M$

**using** *scast-max-bound assms* **by** *fast*

**lemma** *scast-min-bound*:

**assumes**  $M \leq \text{sint} (v :: 'a :: \text{len word})$

**assumes**  $\text{LENGTH}('a) < \text{LENGTH}('b)$

**shows**  $M \leq \text{sint} ((\text{scast } v) :: 'b :: \text{len word})$

**by** (*simp add: scast-min-bound assms*)

**lemma** *scast-bigger-max-bound*:

**assumes**  $(\text{result} :: 'b :: \text{len word}) = \text{scast} (v :: 'a :: \text{len word})$

**shows**  $\text{sint result} < 2 \wedge \text{LENGTH}('a) \text{ div } 2$

**using** *assms scast-bigger-max-bound* **by** *blast*

**lemma** *scast-bigger-min-bound*:

**assumes**  $(\text{result} :: 'b :: \text{len word}) = \text{scast} (v :: 'a :: \text{len word})$

**shows**  $-(2 \wedge \text{LENGTH}('a) \text{ div } 2) \leq \text{sint result}$

**using** *scast-bigger-min-bound assms* **by** *blast*

**lemma** *scast-bigger-bit-bounds*:

**assumes**  $(\text{result} :: 'b :: \text{len word}) = \text{scast} (v :: 'a :: \text{len word})$

**shows**  $\text{fst} (\text{bit-bounds} (\text{LENGTH}('a))) \leq \text{sint result} \wedge \text{sint result} \leq \text{snd} (\text{bit-bounds} (\text{LENGTH}('a)))$

**by** (*auto simp add: scast-bigger-max-bound scast-bigger-min-bound assms*)

Results about *new\_int*.

**lemma** *new-int-take-bits*:

**assumes**  $\text{IntVal } b \text{ val} = \text{new-int } b \text{ ival}$

**shows**  $\text{take-bit } b \text{ val} = \text{val}$

**using** *assms* **by** *simp*



### 3.6.2 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-egs(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)
  by (metis add.commute take-bit-dist-addL)

```

```

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)
  by (metis (no-types) take-bit-dist-subL diff-add-cancel diff-right-commute diff-self)

```

```

lemma take-bit-dist-neg[simp]:
  fixes ix :: 'a :: len word
  shows take-bit b (- take-bit b (ix)) = take-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-bit b x) = signed-take-bit (b - 1) x
  using signed-take-take-bit assms by blast

```

```

lemma mod-larger-ignore:
  fixes a :: int
  fixes m n :: nat
  assumes n < m
  shows (a mod 2 ^ m) mod 2 ^ n = a mod 2 ^ n
  using mod-larger-ignore assms by blast

```

```

lemma mod-dist-over-add:
  fixes a b c :: int64
  fixes n :: nat
  assumes 1:  $0 < n$ 
  assumes 2:  $n < 64$ 
  shows  $(a \bmod 2^n + b) \bmod 2^n = (a + b) \bmod 2^n$ 
proof -
  have 3:  $(0 :: \text{int64}) < 2^n$ 
    by (simp add: size64 word-2p-lem assms)
  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

```

## 4 Stamp Typing

```

theory Stamp
  imports Values
begin

```

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

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

```

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

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

```

To help with supporting masks in future, this constructor allows masks but ignores them.

```

abbreviation IntegerStampM :: nat  $\Rightarrow$  int  $\Rightarrow$  int  $\Rightarrow$  int64  $\Rightarrow$  int64  $\Rightarrow$  Stamp
where
  IntegerStampM b lo hi down up  $\equiv$  IntegerStamp b lo hi

```

```

fun is-stamp-empty :: Stamp ⇒ 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 ⇒ bool where
  valid-stamp (IntegerStamp bits lo hi) =
    (0 < bits ∧ bits ≤ 64 ∧
     fst (bit-bounds bits) ≤ lo ∧ lo ≤ snd (bit-bounds bits) ∧
     fst (bit-bounds bits) ≤ hi ∧ hi ≤ 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*)  
) |  
*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 new-int 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 (ObjRef (None)) = ObjectStamp "" False False True |
  constantAsStamp (ObjRef (Some n)) = ObjectStamp "" False True False |

  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)  
**by** (simp add: *wf-value-def*)

**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) = (hi1 < lo2)  
|  
*stamp-under* - - = False

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

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

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

## 5 Graph Representation

### 5.1 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.

```
datatype IRInvokeKind =  
  Interface | Special | Static | Virtual
```

```
fun isDirect :: IRInvokeKind ⇒ bool where  
  isDirect Interface = False |  
  isDirect Special = True |  
  isDirect Static = True |  
  isDirect Virtual = False
```

```
fun hasReceiver :: IRInvokeKind ⇒ bool where  
  hasReceiver Static = False |  
  hasReceiver - = True
```

```
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)  
  | ArrayLengthNode (ir-value: INPUT) (ir-next: SUCC)  
  | BeginNode (ir-next: SUCC)  
  | BitCountNode (ir-value: INPUT)
```

- | *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)
- | *ControlFlowAnchorNode* (*ir-next*: SUCC)
- | *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)
  
- | *FixedGuardNode* (*ir-condition*: INPUT-COND) (*ir-stateBefore-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)
- | *IntegerMulHighNode* (*ir-x*: INPUT) (*ir-y*: INPUT)
- | *IntegerNormalizeCompareNode* (*ir-x*: INPUT) (*ir-y*: INPUT)
- | *IntegerTestNode* (*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)
- | *LoadIndexedNode* (*ir-index*: INPUT) (*ir-guard-opt*: INPUT-GUARD option) (*ir-value*: INPUT) (*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) (*ir-invoke-kind*: IRInvokeKind)
- | *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: IN-
PUT-STATE option) (ir-next: SUCC)
| NotNode (ir-value: INPUT)
| OrNode (ir-x: INPUT) (ir-y: INPUT)
| ParameterNode (ir-index: nat)
| PiNode (ir-object: INPUT) (ir-guard-opt: INPUT-GUARD option)
| ReturnNode (ir-result-opt: INPUT option) (ir-memoryMap-opt: INPUT-EXT
option)
| ReverseBytesNode (ir-value: INPUT)
| 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)

| SignedFloatingIntegerDivNode (ir-x: INPUT) (ir-y: INPUT)
| SignedFloatingIntegerRemNode (ir-x: INPUT) (ir-y: INPUT)
| 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)
| StoreIndexedNode (ir-storeCheck: INPUT-GUARD option) (ir-value: ID) (ir-stateAfter-opt:
INPUT-STATE option) (ir-index: INPUT) (ir-guard-opt: INPUT-GUARD option)
(ir-array: INPUT) (ir-next: SUCC)
| SubNode (ir-x: INPUT) (ir-y: INPUT)
| UnsignedRightShiftNode (ir-x: INPUT) (ir-y: INPUT)
| UnwindNode (ir-exception: INPUT)
| ValuePhiNode (ir-nid: ID) (ir-values: INPUT list) (ir-merge: INPUT-ASSOC)

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

| RefNode (ir-ref:ID)

```

```

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

```

```

fun opt-list-to-list :: 'a list option ⇒ 'a list where

```

$opt\text{-}list\text{-}to\text{-}list\ None = [] \mid$   
 $opt\text{-}list\text{-}to\text{-}list\ (Some\ x) = x$

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

**fun** `inputs-of` :: *IRNode* ⇒ *ID list* **where**

`inputs-of-AbsNode`:  
`inputs-of (AbsNode value) = [value] |`  
`inputs-of-AddNode`:  
`inputs-of (AddNode x y) = [x, y] |`  
`inputs-of-AndNode`:  
`inputs-of (AndNode x y) = [x, y] |`  
`inputs-of-ArrayLengthNode`:  
`inputs-of (ArrayLengthNode x next) = [x] |`  
`inputs-of-BeginNode`:  
`inputs-of (BeginNode next) = [] |`  
`inputs-of-BitCountNode`:  
`inputs-of (BitCountNode value) = [value] |`  
`inputs-of-BytecodeExceptionNode`:  
`inputs-of (BytecodeExceptionNode arguments stateAfter next) = arguments @`  
`(opt-to-list stateAfter) |`  
`inputs-of-ConditionalNode`:  
`inputs-of (ConditionalNode condition trueValue falseValue) = [condition, true-`  
`Value, falseValue] |`  
`inputs-of-ConstantNode`:  
`inputs-of (ConstantNode const) = [] |`  
`inputs-of-ControlFlowAnchorNode`:  
`inputs-of (ControlFlowAnchorNode n) = [] |`  
`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-FixedGuardNode`:  
`inputs-of (FixedGuardNode condition stateBefore next) = [condition] |`  
`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-IntegerMulHighNode:*  
*inputs-of (IntegerMulHighNode x y) = [x, y] |*  
*inputs-of-IntegerNormalizeCompareNode:*  
*inputs-of (IntegerNormalizeCompareNode x y) = [x, y] |*  
*inputs-of-IntegerTestNode:*  
*inputs-of (IntegerTestNode x y) = [x, y] |*  
*inputs-of-InvokeNode:*  
*inputs-of (InvokeNode nid0 callTarget classInit stateDuring stateAfter next) =*  
*callTarget # (opt-to-list classInit) @ (opt-to-list stateDuring) @ (opt-to-list stateAfter)*  
*|*  
*inputs-of-InvokeWithExceptionNode:*  
*inputs-of (InvokeWithExceptionNode nid0 callTarget classInit stateDuring stateAfter*  
*next exceptionEdge) = callTarget # (opt-to-list classInit) @ (opt-to-list stateDur-*  
*ing) @ (opt-to-list stateAfter) |*  
*inputs-of-IsNullNode:*  
*inputs-of (IsNullNode value) = [value] |*  
*inputs-of-KillingBeginNode:*  
*inputs-of (KillingBeginNode next) = [] |*  
*inputs-of-LeftShiftNode:*  
*inputs-of (LeftShiftNode x y) = [x, y] |*  
*inputs-of-LoadFieldNode:*  
*inputs-of (LoadFieldNode nid0 field object next) = (opt-to-list object) |*  
*inputs-of-LoadIndexedNode:*  
*inputs-of (LoadIndexedNode index guard x next) = [x] |*  
*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 invoke-kind) = argu-*  
*ments |*  
*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-ReverseBytesNode*:  
*inputs-of (ReverseBytesNode value) = [value] |*  
*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-SignedFloatingIntegerDivNode*:  
*inputs-of (SignedFloatingIntegerDivNode x y) = [x, y] |*  
*inputs-of-SignedFloatingIntegerRemNode*:  
*inputs-of (SignedFloatingIntegerRemNode x y) = [x, y] |*  
*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-StoreIndexedNode*:  
*inputs-of (StoreIndexedNode check val st index guard array nid') = [val, array] |*  
*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-ArrayLengthNode*:  
*successors-of* (*ArrayLengthNode x next*) = [*next*] |  
*successors-of-BeginNode*:  
*successors-of* (*BeginNode next*) = [*next*] |  
*successors-of-BitCountNode*:  
*successors-of* (*BitCountNode value*) = [] |  
*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-ControlFlowAnchorNode*:  
*successors-of* (*ControlFlowAnchorNode next*) = [*next*] |  
*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-FixedGuardNode*:  
*successors-of* (*FixedGuardNode condition stateBefore next*) = [*next*] |  
*successors-of-FrameState*:  
*successors-of* (*FrameState monitorIds outerFrameState values virtualObjectMappings*) = [] |  
*successors-of-IfNode*:  
*successors-of* (*IfNode condition trueSuccessor falseSuccessor*) = [*trueSuccessor, falseSuccessor*] |  
*successors-of-IntegerBelowNode*:  
*successors-of* (*IntegerBelowNode x y*) = [] |

*successors-of-IntegerEqualsNode:*  
*successors-of (IntegerEqualsNode x y) = [] |*  
*successors-of-IntegerLessThanNode:*  
*successors-of (IntegerLessThanNode x y) = [] |*  
*successors-of-IntegerMulHighNode:*  
*successors-of (IntegerMulHighNode x y) = [] |*  
*successors-of-IntegerNormalizeCompareNode:*  
*successors-of (IntegerNormalizeCompareNode x y) = [] |*  
*successors-of-IntegerTestNode:*  
*successors-of (IntegerTestNode 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-LoadIndexedNode:*  
*successors-of (LoadIndexedNode index guard x 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 invoke-kind) = []*  
*|*  
*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-ReverseBytesNode:*  
*successors-of (ReverseBytesNode value) = [] |*  
*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-SignedFloatingIntegerDivNode:*  
*successors-of (SignedFloatingIntegerDivNode x y) = [] |*  
*successors-of-SignedFloatingIntegerRemNode:*  
*successors-of (SignedFloatingIntegerRemNode x y) = [] |*  
*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-StoreIndexedNode:*  
*successors-of (StoreIndexedNode check val st index guard array 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*:  $\text{successors-of } (\text{RefNode } \text{ref}) = [\text{ref}]$

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

**lemma** *successors-of* (*FrameState*  $x$  (*Some*  $y$ ) (*Some*  $z$ ) *None*) = []  
**by** *simp*

**lemma** *inputs-of* (*IfNode*  $c$   $t$   $f$ ) = [ $c$ ]  
**by** *simp*

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

**lemma** *inputs-of* (*EndNode*) = []  $\wedge$  *successors-of* (*EndNode*) = []  
**by** *simp*

**end**

## 5.2 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*  $\Rightarrow$  *bool* **where**  
  *is-EndNode* *EndNode* = *True* |  
  *is-EndNode* - = *False*

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

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

**fun** *is-ShiftNode* :: *IRNode*  $\Rightarrow$  *bool* **where**



*is-ShiftNode*  $n = ((\text{is-LeftShiftNode } n) \vee (\text{is-RightShiftNode } n) \vee (\text{is-UnsignedRightShiftNode } n))$

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

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

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

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

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

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

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

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

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

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

**fun** *is-BinaryOpLogicNode* :: *IRNode*  $\Rightarrow$  *bool* **where**  
*is-BinaryOpLogicNode*  $n = ((\text{is-CompareNode } n) \vee (\text{is-IntegerTestNode } n))$

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

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

**fun** *is-FloatingNode* :: *IRNode*  $\Rightarrow$  *bool* **where**  
*is-FloatingNode*  $n = ((\text{is-AbstractLocalNode } n) \vee (\text{is-BinaryNode } n) \vee (\text{is-ConditionalNode } n) \vee (\text{is-ConstantNode } n) \vee (\text{is-FloatingGuardedNode } n) \vee (\text{is-LogicNode } n) \vee$

$(is-PhiNode\ n) \vee (is-ProxyNode\ n) \vee (is-UnaryNode\ n)$

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

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

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

**fun** *is-AbstractFixedGuardNode* :: *IRNode*  $\Rightarrow$  *bool* **where**  
*is-AbstractFixedGuardNode* *n* =  $(is-FixedGuardNode\ n)$

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

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

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

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

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

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

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

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

**fun** *is-AccessArrayNode* :: *IRNode*  $\Rightarrow$  *bool* **where**  
*is-AccessArrayNode* *n* =  $((is-LoadIndexedNode\ n) \vee (is-StoreIndexedNode\ n))$

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

$\vee$  (*is-AccessFieldNode* *n*)  $\vee$  (*is-DeoptimizingFixedWithNextNode* *n*)  $\vee$  (*is-ControlFlowAnchorNode* *n*)  $\vee$  (*is-ArrayLengthNode* *n*)  $\vee$  (*is-AccessArrayNode* *n*)

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

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

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

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

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

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

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

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

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

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

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

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

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

**fun** *is-IterableNodeType* :: *IRNode*  $\Rightarrow$  *bool* **where**  
*is-IterableNodeType* *n* = ((*is-AbstractBeginNode* *n*)  $\vee$  (*is-AbstractMergeNode* *n*)  $\vee$  (*is-FrameState* *n*)  $\vee$  (*is-IfNode* *n*)  $\vee$  (*is-IntegerDivRemNode* *n*)  $\vee$  (*is-InvokeWithExceptionNode* *n*))

$n) \vee (is-LoopBeginNode\ n) \vee (is-LoopExitNode\ n) \vee (is-MethodCallTargetNode\ n) \vee (is-ParameterNode\ n) \vee (is-ReturnNode\ n) \vee (is-ShortCircuitOrNode\ n))$

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

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

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

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

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

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

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

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

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

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

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

**fun** *is-SwitchFoldable* :: *IRNode*  $\Rightarrow$  *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 (ControlFlowAnchorNode -) = True |
  is-sequential-node - = False

```

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

```

fun is-same-ir-node-type :: IRNode ⇒ IRNode ⇒ bool where
is-same-ir-node-type n1 n2 = (
  ((is-AbsNode n1) ∧ (is-AbsNode n2)) ∨
  ((is-AddNode n1) ∧ (is-AddNode n2)) ∨
  ((is-AndNode n1) ∧ (is-AndNode n2)) ∨
  ((is-BeginNode n1) ∧ (is-BeginNode n2)) ∨
  ((is-BytecodeExceptionNode n1) ∧ (is-BytecodeExceptionNode n2)) ∨
  ((is-ConditionalNode n1) ∧ (is-ConditionalNode n2)) ∨
  ((is-ConstantNode n1) ∧ (is-ConstantNode n2)) ∨
  ((is-DynamicNewArrayNode n1) ∧ (is-DynamicNewArrayNode n2)) ∨
  ((is-EndNode n1) ∧ (is-EndNode n2)) ∨
  ((is-ExceptionObjectNode n1) ∧ (is-ExceptionObjectNode n2)) ∨
  ((is-FrameState n1) ∧ (is-FrameState n2)) ∨
  ((is-IfNode n1) ∧ (is-IfNode n2)) ∨
  ((is-IntegerBelowNode n1) ∧ (is-IntegerBelowNode n2)) ∨
  ((is-IntegerEqualsNode n1) ∧ (is-IntegerEqualsNode n2)) ∨
  ((is-IntegerLessThanNode n1) ∧ (is-IntegerLessThanNode n2)) ∨
  ((is-InvokeNode n1) ∧ (is-InvokeNode n2)) ∨
  ((is-InvokeWithExceptionNode n1) ∧ (is-InvokeWithExceptionNode n2)) ∨
  ((is-IsNullNode n1) ∧ (is-IsNullNode n2)) ∨
  ((is-KillingBeginNode n1) ∧ (is-KillingBeginNode n2)) ∨
  ((is-LeftShiftNode n1) ∧ (is-LeftShiftNode n2)) ∨
  ((is-LoadFieldNode n1) ∧ (is-LoadFieldNode n2)) ∨
  ((is-LogicNegationNode n1) ∧ (is-LogicNegationNode n2)) ∨
  ((is-LoopBeginNode n1) ∧ (is-LoopBeginNode n2)) ∨
  ((is-LoopEndNode n1) ∧ (is-LoopEndNode n2)) ∨
  ((is-LoopExitNode n1) ∧ (is-LoopExitNode n2)) ∨
  ((is-MergeNode n1) ∧ (is-MergeNode n2)) ∨

```

```

((is-MethodCallTargetNode n1) ∧ (is-MethodCallTargetNode n2)) ∨
((is-MulNode n1) ∧ (is-MulNode n2)) ∨
((is-NarrowNode n1) ∧ (is-NarrowNode n2)) ∨
((is-NegateNode n1) ∧ (is-NegateNode n2)) ∨
((is-NewArrayNode n1) ∧ (is-NewArrayNode n2)) ∨
((is-NewInstanceNode n1) ∧ (is-NewInstanceNode n2)) ∨
((is-NotNode n1) ∧ (is-NotNode n2)) ∨
((is-OrNode n1) ∧ (is-OrNode n2)) ∨
((is-ParameterNode n1) ∧ (is-ParameterNode n2)) ∨
((is-PiNode n1) ∧ (is-PiNode n2)) ∨
((is-ReturnNode n1) ∧ (is-ReturnNode n2)) ∨
((is-RightShiftNode n1) ∧ (is-RightShiftNode n2)) ∨
((is-ShortCircuitOrNode n1) ∧ (is-ShortCircuitOrNode n2)) ∨
((is-SignedDivNode n1) ∧ (is-SignedDivNode n2)) ∨
((is-SignedFloatingIntegerDivNode n1) ∧ (is-SignedFloatingIntegerDivNode n2))
∨
((is-SignedFloatingIntegerRemNode n1) ∧ (is-SignedFloatingIntegerRemNode n2))
∨
((is-SignedRemNode n1) ∧ (is-SignedRemNode n2)) ∨
((is-SignExtendNode n1) ∧ (is-SignExtendNode n2)) ∨
((is-StartNode n1) ∧ (is-StartNode n2)) ∨
((is-StoreFieldNode n1) ∧ (is-StoreFieldNode n2)) ∨
((is-SubNode n1) ∧ (is-SubNode n2)) ∨
((is-UnsignedRightShiftNode n1) ∧ (is-UnsignedRightShiftNode n2)) ∨
((is-UnwindNode n1) ∧ (is-UnwindNode n2)) ∨
((is-ValuePhiNode n1) ∧ (is-ValuePhiNode n2)) ∨
((is-ValueProxyNode n1) ∧ (is-ValueProxyNode n2)) ∨
((is-XorNode n1) ∧ (is-XorNode n2)) ∨
((is-ZeroExtendNode n1) ∧ (is-ZeroExtendNode n2)))

```

**end**

### 5.3 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*  $\Rightarrow$  *ID set*  
**is**  $\lambda g. \{nid \in dom\ g \cdot \nexists s. g\ nid = (Some\ (NoNode,\ s))\}$  .

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

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

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

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

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

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

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

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

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

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

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

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



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

**code-datatype** *irgraph*

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

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

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

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

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

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

— Get the inputs set of a given node ID

**fun** *inputs* ::  $IRGraph \Rightarrow ID \Rightarrow ID\ set$  **where**  
*inputs*  $g\ nid = set\ (inputs-of\ (kind\ g\ nid))$

— Get the successor set of a given node ID

**fun** *succ* ::  $IRGraph \Rightarrow ID \Rightarrow ID\ set$  **where**  
*succ*  $g\ nid = set\ (successors-of\ (kind\ g\ nid))$

— Gives a relation between node IDs - between a node and its input nodes

**fun** *input-edges* ::  $IRGraph \Rightarrow ID\ rel$  **where**  
*input-edges*  $g = (\bigcup i \in ids\ g. \{(i,j) \mid j \in (inputs\ g\ i)\})$

— Find all the nodes in the graph that have nid as an input - the usages of nid

**fun** *usages* ::  $IRGraph \Rightarrow ID \Rightarrow ID\ set$  **where**  
*usages*  $g\ nid = \{i. i \in ids\ g \wedge nid \in inputs\ g\ i\}$

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

**fun** *predecessors* ::  $IRGraph \Rightarrow ID \Rightarrow ID\ set$  **where**  
*predecessors*  $g\ nid = \{i. i \in ids\ g \wedge nid \in succ\ g\ i\}$

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

**fun** *edge* ::  $(IRNode \Rightarrow 'a) \Rightarrow ID \Rightarrow IRGraph \Rightarrow 'a$  **where**

```

edge sel nid g = sel (kind g nid)

fun filtered-inputs :: IRGraph ⇒ 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
    by (auto simp add: kind.rep-eq ids.rep-eq)
  have kind g x ≠ NoNode ⟶ x ∈ ids g
    by (cases Rep-IRGraph g x = None; auto simp add: ids-def kind-def)
  from this that show ?thesis
    by auto
qed

lemma not-in-g:
  assumes nid ∉ ids g
  shows kind g nid = NoNode
  using assms by simp

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

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

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

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

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

lemma [simp]: finite (dom g) ⟶ stamp (Abs-IRGraph g) = (λx . (case g x of

```

$None \Rightarrow IllegalStamp \mid Some\ n \Rightarrow snd\ n))$   
**by** (*simp add: stamp.rep-eq*)

**lemma** [*simp*]:  $ids\ (irgraph\ g) = set\ (map\ fst\ (no-node\ g))$   
**by** (*simp add: irgraph*)

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

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

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

**lemma** [*code*]:  $replace-node\ nid\ k\ (irgraph\ g) = (irgraph\ ((nid, k) \# g))$   
**proof** (*cases\ fst\ k = NoNode*)  
**case** *True*  
**then show** *?thesis*  
**by** (*metis\ (mono-tags, lifting)\ Rep-IRGraph-inject\ filter.simps(2)\ irgraph.abs-eq\ no-node.simps\ replace-node.rep-eq\ snd-conv*)  
**next**  
**case** *False*  
**then show** *?thesis*  
**by** (*smt\ (verit, cfv-SIG)\ irgraph-def\ 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\ replace-node-def\ replace-node.abs-eq\ snd-eqD*)  
**qed**

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

**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)

```

### 5.3.1 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

```

## 5.4 Structural Graph Comparison

**theory**

*Comparison*

```
imports  
  IRGraph  
begin
```

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

```
fun find-ref-nodes :: IRGraph ⇒ (ID → ID) where  
find-ref-nodes g = map-of  
  (map (λn. (n, ir-ref (kind g n))) (filter (λid. is-RefNode (kind g id)) (sorted-list-of-set  
  (ids g))))
```

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

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

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

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

reachables .

```
inductive nodeEq :: (ID → ID) ⇒ IRGraph ⇒ ID ⇒ IRGraph ⇒ ID ⇒ bool  
where  
[[ kind g1 n1 = RefNode ref; nodeEq m g1 ref g2 n2 ]] ⇒ nodeEq m g1 n1 g2 n2  
|  
[[ x = kind g1 n1;  
  y = kind g2 n2;  
  is-same-ir-node-type x y;  
  replace-ref-nodes g1 m (successors-of x) = successors-of y;  
  replace-ref-nodes g1 m (inputs-of x) = inputs-of y  
⇒ nodeEq m g1 n1 g2 n2
```

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

```
fun diffNodesGraph :: IRGraph ⇒ IRGraph ⇒ ID set where  
diffNodesGraph g1 g2 = (let refNodes = find-ref-nodes g1 in
```

```

    { n . n ∈ Predicate.the (reachables-i-i-i-o g1 [0] { }) ∧ (case refNodes n of Some
- ⇒ False | - ⇒ True) ∧ ¬(nodeEq refNodes g1 n g2 n)}

```

```

fun diffNodesInfo :: IRGraph ⇒ IRGraph ⇒ (ID × IRNode × IRNode) set (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

```

## 5.5 Control-flow Graph Traversal

```

theory

```

```

Traversal

```

```

imports

```

```

IRGraph

```

```

begin

```

```

type-synonym Seen = ID set

```

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

```

fun nextEdge :: Seen ⇒ ID ⇒ IRGraph ⇒ ID option where

```

```

nextEdge seen nid g =

```

```

  (let nids = (filter (λnid'. nid' ∉ seen) (successors-of (kind g nid))) in
   (if length nids > 0 then Some (hd nids) else None))

```

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

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

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

```

fun pred :: IRGraph ⇒ ID ⇒ ID option where

```

```

pred g nid = (case kind g nid of

```

```

  (MergeNode ends -) ⇒ Some (hd ends) |

```

```

- ⇒

```

```

  (if IRGraph.predecessors g nid = { })

```

```

    then None else
    Some (hd (sorted-list-of-set (IRGraph.predecessors g nid)))
  )
)

```

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

**type-synonym** 'a TraversalState = (ID × Seen × 'a)

**inductive** Step

```

:: ('a TraversalState ⇒ 'a) ⇒ IRGraph ⇒ 'a TraversalState ⇒ 'a TraversalState
option ⇒ bool

```

**for** sa g **where**

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

```

[[kind g nid = BeginNode nid;

```

```

  nid ∉ seen;
  seen' = {nid} ∪ seen;

```

```

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

```

```

  analysis' = sa (nid, seen, analysis)]
⇒ 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 ∉ seen;
  seen' = {nid} ∪ seen;

```

```

  nid' = any-usage g nid;

```

```

  analysis' = sa (nid, seen, analysis)]
⇒ Step sa g (nid, seen, analysis) (Some (nid', seen', analysis')) |

```

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

```

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

```

```

  nid ∉ seen;
  seen' = {nid} ∪ seen;

```

```

    Some nid' = nextEdge seen' nid g;

    analysis' = sa (nid, seen, analysis)]
    ==> Step sa g (nid, seen, analysis) (Some (nid', seen', analysis')) |

— We can not 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
theory Class
  imports Main
begin

```

Representation of a standard class containing fields, methods and constructors

— Representation of Fields and Parameters —

```

type-synonym FieldName = string
type-synonym FieldType = string
type-synonym ParameterType = string

```

```

datatype JVMField =
  NewField (field-name: FieldName)
    (field-type: FieldType) |
  NewParameter (parameter-type: ParameterType)

```

— Representation of a Method —

```

type-synonym MethodName = string
type-synonym Return_Type = string
type-synonym MethodParameters = JVMField list
type-synonym Method_UniqueName = string

```

```

datatype JVMMethod =
  NewMethod (method-name: MethodName)
    (method-return_Type: Return_Type)

```



(*method-parameters*: *MethodParameters*)  
(*method-unique-name*: *MethodUniqueName*)

— Representation of a Constructor —

**type-synonym** *ConstructorParameters* = *JVMField list*

**datatype** *JVMConstructor* =  
*NewConstructor* (*constructor-params*: *ConstructorParameters*)

— Representation of a standard class —

**type-synonym** *Fields* = *JVMField list*

**type-synonym** *Methods* = *JVMMethod list*

**type-synonym** *Constructors* = *JVMConstructor list*

**type-synonym** *ClassName* = *string*

**type-synonym** *ParentClass* = *string*

**datatype** *JVMClass* =  
*NewClass* (*class-name*: *ClassName*)  
    (*class-fields*: *Fields*)  
    (*class-methods*: *Methods*)  
    (*class-constructors*: *Constructors*)  
    (*class-parents*: *ParentClass list*)  
    (*class-parent*: *ParentClass*)

**definition** *emptyClass* :: *JVMClass* **where**  
*emptyClass* = *NewClass* "name-empty" [] [] [] "parent-empty"

**definition** *jlObject* :: *JVMClass* **where**  
*jlObject* = *NewClass*  
    "java.lang.Object"  
    []  
    [*NewMethod* "finalize" "V" [] "java.lang.Object.finalize() V",  
      *NewMethod* "wait" "V" [*NewParameter* "J", *NewParameter* "I"] "java.lang.Object.wait(JI) V",  
  
      *NewMethod* "wait" "V" [] "java.lang.Object.wait() V",  
      *NewMethod* "wait" "V" [*NewParameter* "J"] "java.lang.Object.wait(J) V",  
      *NewMethod* "equals" "Z" [*NewParameter* "java.lang.Object"] "java.lang.Object.equals(java.lang.Object) Z",  
  
      *NewMethod* "toString" "java.lang.String" [] "java.lang.Object.toString() java.lang.String",  
  
      *NewMethod* "hashCode" "I" [] "java.lang.Object.hashCode() I",  
      *NewMethod* "getClass" "java.lang.Class" [] "java.lang.Object.getClass() java.lang.Class",  
  
      *NewMethod* "clone" "java.lang.Object" [] "java.lang.Object.clone() java.lang.Object",  
  
      *NewMethod* "notify" "V" [] "java.lang.Object.notify() V",

```

    NewMethod "notifyAll" "V" [] "java.lang.Object.notifyAll() V"]
  [NewConstructor []
   ["None"]
   "None"

```

— General Functions —

```

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

```

— Functions to interact with JVMClass lists —

```

fun find-class-index :: string ⇒ JVMClass list ⇒ nat where
  find-class-index name cl = find-index name (map class-name cl)

```

```

fun get-JVMClass :: string ⇒ JVMClass list ⇒ JVMClass where
  get-JVMClass cName cList =
    (if ((find-class-index cName cList) = (length cList))
     then (emptyClass)
     else (nth cList (find-class-index cName cList)))

```

```

fun get-Methods :: string ⇒ JVMClass list ⇒ JVMMethod list where
  get-Methods cname clist = class-methods (get-JVMClass cname clist)

```

```

fun get-simple-signature :: string ⇒ string where
  get-simple-signature fqN = rev (take (find-index (CHR ".") (rev fqN)) (rev fqN))

```

```

fun simple-signatures :: string ⇒ JVMClass list ⇒ string list where
  simple-signatures cname clist =
    (map get-simple-signature (map method-unique-name (get-Methods cname clist)))

```

```

fun classNames :: JVMClass list ⇒ string set where
  classNames cl = set (map class-name cl)

```

```

fun parentRel :: JVMClass list ⇒ string rel where
  parentRel cl = (set (map (λc. (class-name c, class-parent c)) cl))

```

```

fun parentRel2 :: JVMClass list ⇒ (string × string) list where
  parentRel2 cl = (map (λc. (class-name c, class-parent c)) cl)

```

```

fun parentOf :: JVMClass list ⇒ (string → string) where
  parentOf [] = Map.empty |
  parentOf (c#cl) = (parentOf cl)((class-name c) ↦ (class-parent c))

```

```

fun superclassOf :: JVMClass list ⇒ (string rel) where
  superclassOf cl = trancl (parentRel cl)

```

```

lemma finite (set (l::('a list)))
  by simp

lemma domainUnion:
  dom (m(a→b)) = dom (m) ∪ {a}
  by simp

lemma finite (dom (parentOf m))
  proof (induction m)
  case Nil
  then show ?case
    by simp
next
  case (Cons a m)
  then show ?case unfolding parentOf.simps
    by (metis insert-def singleton-conv sup-commute finite.simps domainUnion)
qed

lemma wellFoundedParent:
  assumes acyclic (parentRel cl)
  shows wf (parentRel cl)
  using assms unfolding parentRel.simps by (metis (no-types, lifting) wf-set)

lemma transSuperClassOf[simp]:
  trans (superclassOf cl)
  by simp

definition bestClassEver :: JVMClass where
  bestClassEver =
    NewClass "bestClassEver"
      [NewField "x" "I", NewField "y" "float"]
      [NewMethod "getX" "I" [NewParameter "null"] "bestClassEver-getXI(n)",
       NewMethod "setY" "null" [NewParameter "float"] "bestClassEver-SetYn(f)"]
      [NewConstructor [NewParameter "I"], NewConstructor [NewParameter
"float"]]
      ["Object"]
      "Object"

value class-name bestClassEver
value class-parent bestClassEver
value class-fields bestClassEver

```

```

value class-methods bestClassEver
value class-constructors bestClassEver

value field-name (hd (class-fields bestClassEver))
value field-type (hd (class-fields bestClassEver))

value field-name (hd (tl (class-fields bestClassEver)))
value field-type (hd (tl (class-fields bestClassEver)))

value method-name (hd (class-methods bestClassEver))
value method-returnType (hd (class-methods bestClassEver))
value method-parameters (hd (class-methods bestClassEver))
value method-unique-name (hd (class-methods bestClassEver))

value method-name (hd (tl (class-methods bestClassEver)))
value method-returnType (hd (tl (class-methods bestClassEver)))
value method-parameters (hd (tl (class-methods bestClassEver)))
value method-unique-name (hd (tl (class-methods bestClassEver)))

value constructor-params (hd (class-constructors bestClassEver))

value constructor-params (hd (tl (class-constructors bestClassEver)))

value parameter-type (hd (method-parameters (hd (class-methods bestClassEver))))
value parameter-type (hd (method-parameters (hd (tl (class-methods bestClassEver))))))

value parameter-type (hd (constructor-params (hd (class-constructors bestClassEver))))
value parameter-type (hd (constructor-params (hd (tl (class-constructors bestClassEver))))))

definition unit-InstanceOfTest-instanceOfSnippet4-mapping :: JVMClass list where
unit-InstanceOfTest-instanceOfSnippet4-mapping = [
NewClass "org.graalvm.compiler.core.test.InstanceOfTest$B"
  []
  []
  [NewConstructor []]
  ["org.graalvm.compiler.core.test.InstanceOfTest$A", "java.lang.Object", "None"]
  ["org.graalvm.compiler.core.test.InstanceOfTest$A",

NewClass "org.graalvm.compiler.core.test.InstanceOfTest$A"
  []
  []

```

```

[NewConstructor []
["java.lang.Object", "None"]
"java.lang.Object",

NewClass "java.lang.Object"
[]
[NewMethod "finalize" "V" [] "java.lang.Object.finalize() V", NewMethod "wait"
"V" [NewParameter "J", NewParameter "I"] "java.lang.Object.wait(JI) V", NewMethod
"wait" "V" [] "java.lang.Object.wait() V", NewMethod "wait" "V" [NewParameter
"J"] "java.lang.Object.wait(J) V", NewMethod "equals" "Z" [NewParameter "java.lang.Object"
"java.lang.Object.equals(java.lang.Object) Z", NewMethod "toString" "java.lang.String"
[] "java.lang.Object.toString() java.lang.String", NewMethod "hashCode" "I" [] "java.lang.Object.hashCode() I"
NewMethod "getClass" "java.lang.Class" [] "java.lang.Object.getClass() java.lang.Class",
NewMethod "clone" "java.lang.Object" [] "java.lang.Object.clone() java.lang.Object",
NewMethod "notify" "V" [] "java.lang.Object.notify() V", NewMethod "notifyAll"
"V" [] "java.lang.Object.notifyAll() V"]
[NewConstructor []
["None"]
"None"]

definition unit-InvokeVirtual-01-test-mapping :: JVMClass list where
unit-InvokeVirtual-01-test-mapping = [
NewClass "org.graalvm.compiler.jtt.micro.InvokeVirtual-01$B"
[]
[NewMethod "plus" "I" [NewParameter "I"] "org.graalvm.compiler.jtt.micro.InvokeVirtual-01$B.plus(I) I"
[NewConstructor []
["org.graalvm.compiler.jtt.micro.InvokeVirtual-01$A", "java.lang.Object", "None"]
"org.graalvm.compiler.jtt.micro.InvokeVirtual-01$A",

NewClass "org.graalvm.compiler.jtt.micro.InvokeVirtual-01$C"
[]
[NewMethod "plus" "I" [NewParameter "I"] "org.graalvm.compiler.jtt.micro.InvokeVirtual-01$C.plus(I) I"
[NewConstructor []
["org.graalvm.compiler.jtt.micro.InvokeVirtual-01$A", "java.lang.Object", "None"]
"org.graalvm.compiler.jtt.micro.InvokeVirtual-01$A",

NewClass "java.lang.Object"
[]
[NewMethod "finalize" "V" [] "java.lang.Object.finalize() V", NewMethod "wait"
"V" [NewParameter "J", NewParameter "I"] "java.lang.Object.wait(JI) V", NewMethod
"wait" "V" [] "java.lang.Object.wait() V", NewMethod "wait" "V" [NewParameter
"J"] "java.lang.Object.wait(J) V", NewMethod "equals" "Z" [NewParameter "java.lang.Object"
"java.lang.Object.equals(java.lang.Object) Z", NewMethod "toString" "java.lang.String"
[] "java.lang.Object.toString() java.lang.String", NewMethod "hashCode" "I" [] "java.lang.Object.hashCode() I"
NewMethod "getClass" "java.lang.Class" [] "java.lang.Object.getClass() java.lang.Class",
NewMethod "clone" "java.lang.Object" [] "java.lang.Object.clone() java.lang.Object",
NewMethod "notify" "V" [] "java.lang.Object.notify() V", NewMethod "notifyAll"
"V" [] "java.lang.Object.notifyAll() V"]
[NewConstructor []

```

```

["None"]
"None",

NewClass "org.graalvm.compiler.jtt.micro.InvokeVirtual-01$A"
[]
[NewMethod "plus" "I" [NewParameter "I"] "org.graalvm.compiler.jtt.micro.InvokeVirtual-01$A.plus(I)I"]
[NewConstructor []]
["java.lang.Object", "None"]
"java.lang.Object"]

value parentRel unit-InvokeVirtual-01-test-mapping
value superclassOf unit-InvokeVirtual-01-test-mapping
value classNames unit-InvokeVirtual-01-test-mapping
value the (parentOf unit-InvokeVirtual-01-test-mapping "org.graalvm.compiler.jtt.micro.InvokeVirtual-01$A")
value set (simple-signatures "org.graalvm.compiler.jtt.micro.InvokeVirtual-01$A"
unit-InvokeVirtual-01-test-mapping)

value find-class-index "org.graalvm.compiler.jtt.micro.InvokeVirtual-01$A" unit-InvokeVirtual-01-test-mapping
value get-JVMClass "org.graalvm.compiler.jtt.micro.InvokeVirtual-01$B" unit-InvokeVirtual-01-test-mapping
value get-simple-signature "org.graalvm.compiler.jtt.micro.InvokeVirtual-01$A.plus(I)I"

definition inheritsFromObject :: JVMClass list  $\Rightarrow$  bool where
  inheritsFromObject cl = ((remdups (map List.last (map class-parents cl))) =
["None"])

lemma containsObjImplies[simp]:
  shows List.member cl jlObject  $\longrightarrow$ 
    ("java.lang.Object", "None")  $\in$  parentRel cl  $\longrightarrow$ 
    List.member (parentRel2 cl) ("java.lang.Object", "None")
  using List.member-def by fastforce

lemma containsObjImpliesNonEmpty:
  shows List.member cl jlObject  $\longrightarrow$  cl  $\neq$  []
  using List.member-def by force

lemma acyclic-jlObj:
  shows acyclic (parentRel [jlObject])
  by (simp add: jlObject-def wf-acyclic)

lemma inheritsFromObj-jlObj:
  shows inheritsFromObject [jlObject]
  unfolding inheritsFromObject-def jlObject-def by simp

lemma acyclicDef:
  fixes cl :: JVMClass list
  shows acyclic (parentRel cl)  $\implies$  ( $\forall j. j \in$  (set cl)  $\longrightarrow$  (class-name j  $\neq$  class-parent
j))

```

**unfolding** *acyclic-def* **by** *auto*

**lemma** *acyclicParent-super*:

**shows** (*acyclic* (*parentRel* *cl*))  $\implies$  (*acyclic* (*superclassOf* *cl*))

**unfolding** *parentRel.simps* *superclassOf.simps* *acyclic-def* **by** *simp*

**lemma** *remdupsInherit*:

**shows** *inheritsFromObject* *cl*  $\implies$  *inheritsFromObject* (*remdups* *cl*)

**using** *inheritsFromObject-def* **by** (*simp* *add*: *remdups-map-remdups*)

**typedef** *Classes* = {*cl* :: *JVMClass* *list* .  
    *List.member* *cl* *jlObject*  $\wedge$   
    *cl*  $\neq$  []  $\wedge$   
    *acyclic* (*parentRel* *cl*)  $\wedge$   
    *inheritsFromObject* *cl*  $\wedge$   
    *distinct* *cl*}

**morphisms** *classToJVMList* *Abs-Classes*

**proof** –

**obtain** *cl* **where** *cl* = [*jlObject*]

**by** *simp*

**then have** *a*: *cl*  $\neq$  []

**by** *simp*

**have** *b*: *List.member* *cl* *jlObject*

**by** (*simp* *add*: *member-rec(1)* *cl*)

**have** *c*: *acyclic* (*parentRel* *cl*)

**using** *acyclic-jlObj* **by** (*simp* *add*: *cl*)

**have** *d*: *inheritsFromObject* *cl*

**by** (*simp* *add*: *cl inheritsFromObj-jlObj*)

**have** *e*: *distinct* *cl*

**by** (*simp* *add*: *cl*)

**then show** *?thesis*

**using** *cl b c d* **by** *blast*

**qed**

**lemma** *classes-eq-iff*:

*cl1* = *cl2*  $\iff$  *classToJVMList* *cl1* = *classToJVMList* *cl2*

**by** (*simp* *add*: *classToJVMList-inject*)

**lemma** *classes-eqI*:

*classToJVMList* *cl1* = *classToJVMList* *cl2*  $\implies$  *cl1* = *cl2*

**by** (*simp* *add*: *classToJVMList-inject*)

**setup-lifting** *type-definition-Classes*

**lift-definition** *JVMClasses* :: *JVMClass* *list*  $\Rightarrow$  *Classes* **is**

$\lambda j. (if (List.member j jlObject \wedge acyclic (parentRel j) \wedge inheritsFromObject j)$   
    *then* (*if* (*distinct* *j*) *then* *j* *else* (*remdups* *j*))

```

    else [jlObject])
  using List.member-def acyclic-jlObj containsObjImpliesNonEmpty inheritsFromObj-jlObj
    remdupsInherit
  by fastforce

```

```

lemma nonempty-cl [simp, intro]:
  (classToJVMList cl) ≠ []
  using classToJVMList [of cl] by simp

```

```

lemma containsjlobj-cl [simp, intro]:
  List.member (classToJVMList cl) jlObject
  using classToJVMList [of cl] by simp

```

```

lemma acyclic-cl [simp, intro]:
  acyclic (parentRel (classToJVMList cl))
  using classToJVMList [of cl] by simp

```

```

lemma inheritsFromObj-cl [simp, intro]:
  inheritsFromObject (classToJVMList cl)
  using classToJVMList [of cl] by simp

```

```

lemma distinct-cl [simp, intro]:
  distinct (classToJVMList cl)
  using classToJVMList [of cl] by simp

```

```

lemma original-jvm [simp]:
  classToJVMList (JVMClasses cl) =
    (if (List.member cl jlObject ∧ acyclic (parentRel cl) ∧ inheritsFromObject cl)
      then (if (distinct cl) then cl else (remdups cl))
      else [jlObject])
  using JVMClasses.rep-eq by auto

```

```

lemma classesToClasses [simp, code abstype]:
  JVMClasses (classToJVMList cl) = cl
  using acyclic-cl classes-eqI by auto

```

```

context
begin

```

```

qualified definition empty :: Classes where
  empty = JVMClasses []

```

```

qualified definition mapJVMSFunc :: (JVMClass ⇒ 'b) ⇒ Classes ⇒ 'b list
where

```



```

    mapJVMFunc cf cl = List.map cf (classToJVMList cl)

qualified definition member :: Classes ⇒ JVMClass ⇒ bool where
    member cl c = List.member (classToJVMList cl) c

qualified definition length :: Classes ⇒ nat where
    length cl = List.length (classToJVMList cl)

qualified definition nth :: Classes ⇒ nat ⇒ JVMClass where
    nth cl n = List.nth (classToJVMList cl) n

end

lemma classToJVM-empty [simp, code abstract]:
    classToJVMList Class.empty = [jLObject]
by (metis JVMClasses.rep-eq containsObjImpliesNonEmpty Class.empty-def)

lemma classToJVM-map [simp, code]:
    (Class.mapJVMFunc f cl) = List.map f (classToJVMList cl)
by (simp add: Class.mapJVMFunc-def)

code-datatype JVMClasses

lemma [code]:
    classToJVMList (JVMClasses cl) =
      (if (List.member cl jLObject ∧ acyclic (parentRel cl) ∧ inheritsFromObject cl)
        then (if (distinct cl) then cl else (remdups cl))
        else [jLObject])
by (simp add: JVMClasses.rep-eq)

definition newclass :: Classes where
    newclass = JVMClasses [NewClass "name" [] [] ["parent", "None"] "parent",
    jLObject, jLObject]

definition cyclicClass :: JVMClass list where
    cyclicClass = [NewClass "name" [] [] ["name"] "name"]

value newclass
value classToJVMList newclass
value Class.mapJVMFunc class-name newclass
value Class.mapJVMFunc class-parent newclass

value classToJVMList (JVMClasses [])
value classToJVMList (JVMClasses cyclicClass)
value acyclic (parentRel cyclicClass)

```

```
value acyclic (parentRel (classToJVMList (JVMClasses cyclicClass)))
```

```
fun CLfind-class-index :: string  $\Rightarrow$  Classes  $\Rightarrow$  nat where  
  CLfind-class-index name cl = find-index name (Class.mapJVMMFunc class-name  
  cl)
```

```
fun CLget-JVMClass :: string  $\Rightarrow$  Classes  $\Rightarrow$  JVMClass where  
  CLget-JVMClass cName cList =  
    (if ((CLfind-class-index cName cList) = (Class.length cList))  
      then (emptyClass)  
      else (Class.nth cList (CLfind-class-index cName cList)))
```

```
fun CLget-Methods :: string  $\Rightarrow$  Classes  $\Rightarrow$  JVMMethod list where  
  CLget-Methods cname clist = class-methods (CLget-JVMClass cname clist)
```

```
fun CLsimple-signatures :: string  $\Rightarrow$  Classes  $\Rightarrow$  string list where  
  CLsimple-signatures cname clist =  
    (map get-simple-signature (map method-unique-name (CLget-Methods cname  
  clist)))
```

```
lemma finiteSuper:  
  fixes cl :: Classes  
  shows finite (superclassOf (classToJVMList cl))  
  by simp
```

```
lemma finiteClasses:  
  fixes cl :: Classes  
  shows finite (set (classToJVMList cl))  
  by simp
```

```
end
```