Veriopt

April 17, 2024

Abstract

The Veriopt project aims to prove the optimization pass of the GraalVM compiler. The GraalVM compiler includes a sophisticated Intermediate Representation (IR) in the form of a sea-of-nodes based graph structure. We first define the IR graph structure in the Isabelle/HOL interactive theorem prover. We subsequently give the evaluation of the structure a semantics based on the current understanding of the purpose of each IR graph node. Optimization phases are then encoded including the static analysis passes required for an optimization. Each optimization phase is proved to be correct by proving that a bisimulation exists between the unoptimized and optimized graphs. The following document has been automatically generated from the Isabelle/HOL source to provide a very comprehensive definition of the semantics and optimizations introduced by the Veriopt project.

Contents

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

lemma *shiftl-power*[*simp*]: (*x*::(′*a*::*len*) *word*) ∗ (*2* ^ *j*) = *x* << *j*
  ⟨*proof*⟩

lemma $(x::('a::len)\ word) * ((2\ \hat{}\ j) + 1) = x << j + x$
⟨*proof*⟩

lemma $(x::('a::len)\ word) * ((2\ \hat{}\ j) - 1) = x << j - x$
⟨*proof*⟩

lemma $(x::('a::len)\ word) * ((2\hat{}j) + (2\hat{}k)) = x << j + x << k$
⟨*proof*⟩

lemma $(x::('a::len)\ word) * ((2\hat{}j) - (2\hat{}k)) = x << j - x << k$
⟨*proof*⟩

Unsigned shift right.

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

corollary $(255 :: 8\ word) >>> (2 :: nat) = 63$ ⟨*proof*⟩

Signed shift right.

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

corollary $(128 :: 8\ word) >> 2 = 0xE0$ ⟨*proof*⟩

1.2 Fixed-width Word Theories

1.2.1 Support Lemmas for Upper/Lower Bounds

lemma *size32*: *size v = 32* **for** *v :: 32 word*
⟨*proof*⟩

lemma *size64*: *size v = 64* **for** *v :: 64 word*
⟨*proof*⟩

lemma *lower-bounds-equiv*:
assumes $0 < N$
shows $-(((2::int)\ \hat{}\ (N-1))) = (2::int)\ \hat{}\ N\ div\ 2 * - 1$
⟨*proof*⟩

lemma *upper-bounds-equiv*:
assumes $0 < N$
shows $(2::int)\ \hat{}\ (N-1) = (2::int)\ \hat{}\ N\ div\ 2$
⟨*proof*⟩

Some min/max bounds for 64-bit words

lemma *bit-bounds-min64*: $((fst\ (bit\text{-}bounds\ 64))) \leq (sint\ (v::int64))$

⟨proof⟩

lemma *bit-bounds-max64*: $((snd\ (bit\text{-}bounds\ 64))) \geq (sint\ (v{::}int64))$
⟨proof⟩

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 $sint(signed\text{-}take\text{-}bit\ 7\ (128 :: int8)) = -128$ ⟨proof⟩

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 $sint(signed\text{-}take\text{-}bit\ n\ ival) < (2{::}int)\ \hat{}\ n$
⟨proof⟩

lemma *signed-take-bit-int-greater-eq-minus-exp-word*:
fixes $ival :: 'a :: len\ word$
assumes $n < LENGTH('a)$
shows $-\ (2\ \hat{}\ n) \leq sint(signed\text{-}take\text{-}bit\ n\ ival)$
⟨proof⟩

lemma *signed-take-bit-range*:
fixes $ival :: 'a :: len\ word$
assumes $n < LENGTH('a)$
assumes $val = sint(signed\text{-}take\text{-}bit\ n\ ival)$
shows $-\ (2\ \hat{}\ n) \leq val \wedge val < 2\ \hat{}\ n$
⟨proof⟩

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 $val = sint(signed\text{-}take\text{-}bit\ (n - 1)\ ival)$
shows $fst\ (bit\text{-}bounds\ n) \leq val \wedge val \leq snd\ (bit\text{-}bounds\ n)$
⟨proof⟩

lemma *signed-take-bit-bounds64*:
fixes $ival :: int64$

assumes $n \leq 64$
assumes $0 < n$
assumes $val = sint(signed\text{-}take\text{-}bit\ (n - 1)\ ival)$
shows $fst\ (bit\text{-}bounds\ n) \leq val \wedge val \leq snd\ (bit\text{-}bounds\ n)$
⟨*proof*⟩

lemma *int-signed-value-bounds*:
assumes $b1 \leq 64$
assumes $0 < b1$
shows $fst\ (bit\text{-}bounds\ b1) \leq int\text{-}signed\text{-}value\ b1\ v2\ \wedge$
$int\text{-}signed\text{-}value\ b1\ v2 \leq snd\ (bit\text{-}bounds\ b1)$
⟨*proof*⟩

lemma *int-signed-value-range*:
fixes $ival :: int64$
assumes $val = int\text{-}signed\text{-}value\ n\ ival$
shows $-\ (2\ \hat{}\ (n - 1)) \leq val \wedge val < 2\ \hat{}\ (n - 1)$
⟨*proof*⟩

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

lemma *bit-bounds-lower*:
assumes $0 < bits$
shows $word\text{-}of\text{-}int\ (fst\ (bit\text{-}bounds\ bits)) = ((-1) << (bits - 1))$
⟨*proof*⟩

lemma *two-exp-div*:
assumes $0 < bits$
shows $((2::int)\ \hat{}\ bits\ div\ (2::int)) = (2::int)\ \hat{}\ (bits - Suc\ 0)$
⟨*proof*⟩

declare [[*show-types*]]

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

lemma *take-bit-smaller-range*:
fixes $ival :: 'a :: len\ word$
assumes $n < LENGTH('a)$
assumes $val = sint(take\text{-}bit\ n\ ival)$
shows $0 \leq val \wedge val < (2::int)\ \hat{}\ n$
⟨*proof*⟩

lemma *take-bit-same-size-nochange*:
fixes $ival :: 'a :: len\ word$
assumes $n = LENGTH('a)$
shows $ival = take\text{-}bit\ n\ ival$
⟨*proof*⟩

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

lemma *take-bit-redundant*[*simp*]:

fixes *ival* :: *'a* :: *len word*
assumes *0* < *n*
assumes *n* < *LENGTH('a)*
shows *signed-take-bit* (*n* − *1*) (*take-bit n ival*) = *signed-take-bit* (*n* − *1*) *ival*
⟨*proof*⟩

lemma *take-bit-same-size-range*:
fixes *ival* :: *'a* :: *len word*
assumes *n* = *LENGTH('a)*
assumes *ival2* = *take-bit n ival*
shows − (*2* ^ *n div 2*) ≤ *sint ival2* ∧ *sint ival2* < *2* ^ *n div 2*
⟨*proof*⟩

lemma *take-bit-same-bounds*:
fixes *ival* :: *'a* :: *len word*
assumes *n* = *LENGTH('a)*
assumes *ival2* = *take-bit n ival*
shows *fst* (*bit-bounds n*) ≤ *sint ival2* ∧ *sint ival2* ≤ *snd* (*bit-bounds n*)
⟨*proof*⟩

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

lemma *scast-max-bound*:
assumes *sint* (*v* :: *'a* :: *len word*) < *M*
assumes *LENGTH('a)* < *LENGTH('b)*
shows *sint* ((*scast v*) :: *'b* :: *len word*) < *M*
⟨*proof*⟩

lemma *scast-min-bound*:
assumes *M* ≤ *sint* (*v* :: *'a* :: *len word*)
assumes *LENGTH('a)* < *LENGTH('b)*
shows *M* ≤ *sint* ((*scast v*) :: *'b* :: *len word*)
⟨*proof*⟩

lemma *scast-bigger-max-bound*:
assumes (*result* :: *'b* :: *len word*) = *scast* (*v* :: *'a* :: *len word*)
shows *sint result* < *2* ^ *LENGTH('a) div 2*
⟨*proof*⟩

lemma *scast-bigger-min-bound*:
assumes (*result* :: *'b* :: *len word*) = *scast* (*v* :: *'a* :: *len word*)
shows − (*2* ^ *LENGTH('a) div 2*) ≤ *sint result*
⟨*proof*⟩

lemma *scast-bigger-bit-bounds*:
assumes (*result* :: *'b* :: *len word*) = *scast* (*v* :: *'a* :: *len word*)
shows *fst* (*bit-bounds* (*LENGTH('a)*)) ≤ *sint result* ∧ *sint result* ≤ *snd* (*bit-bounds* (*LENGTH('a)*))

⟨proof⟩

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* :: *len word*
shows *take-bit b* (*take-bit b x* + *y*) = *take-bit b* (*x* + *y*)
⟨proof⟩

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*)
⟨proof⟩

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*)
⟨proof⟩

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*)
⟨proof⟩

lemma *take-bit-dist-neg*[*simp*]:
fixes *ix* :: *'a* :: *len word*
shows *take-bit b* (− *take-bit b* (*ix*)) = *take-bit b* (− *ix*)
⟨proof⟩

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*
⟨proof⟩

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*
⟨proof⟩

lemma *mod-dist-over-add*:
fixes *a b c* :: *int64*
fixes *n* :: *nat*
assumes *1*: *0* < *n*
assumes *2*: *n* < *64*

shows $(a \; mod \; 2\hat{~}n + b) \; mod \; 2\hat{~}n = (a + b) \; mod \; 2\hat{~}n$
⟨*proof*⟩

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* ⇒ *int64* ⇒ *int64* **where**
javaMin64 a b ≡ (*if a* ≤*s b then a else b*)

abbreviation *javaMax64* :: *int64* ⇒ *int64* ⇒ *int64* **where**
javaMax64 a b ≡ (*if a* ≤*s 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*:
$n < 32 \Longrightarrow bit \; (-1{::}int32) \; n$
⟨*proof*⟩

definition *MaxOrNeg* :: *nat set* ⇒ *int* **where**
MaxOrNeg s = (*if s* = {} *then* −*1 else Max s*)

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

lemma *MaxOrNegEmpty*:
MaxOrNeg s = −*1* ⟷ *s* = {}
⟨*proof*⟩

2.1 Long.highestOneBit

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

lemma *highestOneBitInvar*:
$highestOneBit \; v = j \Longrightarrow (\forall \, i{::}nat. \; (int \; i > j \longrightarrow \neg \; (bit \; v \; i)))$

⟨*proof*⟩

lemma *highestOneBitNeg*:
highestOneBit v = −*1* ⟷ *v* = *0*
⟨*proof*⟩

lemma *higherBitsFalse*:
fixes *v* :: *'a* :: *len word*
shows *i* > *size v* ⟹ ¬ (*bit v i*)
⟨*proof*⟩

lemma *highestOneBitN*:
assumes *bit v n*
assumes ∀ *i*::*nat*. (*int i* > *n* ⟶ ¬ (*bit v i*))
shows *highestOneBit v* = *n*
⟨*proof*⟩

lemma *highestOneBitSize*:
assumes *bit v n*
assumes *n* = *size v*
shows *highestOneBit v* = *n*
⟨*proof*⟩

lemma *highestOneBitMax*:
highestOneBit v < *size v*
⟨*proof*⟩

lemma *highestOneBitAtLeast*:
assumes *bit v n*
shows *highestOneBit v* ≥ *n*
⟨*proof*⟩

lemma *highestOneBitElim*:
highestOneBit v = *n*
⟹ ((*n* = −*1* ∧ *v* = *0*) ∨ (*n* ≥ *0* ∧ *bit v n*))
⟨*proof*⟩

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

fun *highestOneBitRec* :: *nat* ⇒ (*'a*::*len*) *word* ⇒ *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* ⟹ *j* ≥ *0* ⟹ *bit v j*
⟨*proof*⟩

lemma *highestOneBitRecN*:
  assumes *bit v n*
  shows *highestOneBitRec n v = n*
  ⟨*proof*⟩

lemma *highestOneBitRecMax*:
  highestOneBitRec n v ≤ n
  ⟨*proof*⟩

lemma *highestOneBitRecElim*:
  assumes *highestOneBitRec n v = j*
  shows ((*j* = −*1* ∧ *v* = *0*) ∨ (*j* ≥ *0* ∧ *bit v j*))
  ⟨*proof*⟩

lemma *highestOneBitRecZero*:
  v = *0* ⟹ *highestOneBitRec* (*size v*) *v* = −*1*
  ⟨*proof*⟩

lemma *highestOneBitRecLess*:
  assumes ¬ *bit v n*
  shows *highestOneBitRec n v* = *highestOneBitRec* (*n* − *1*) *v*
  ⟨*proof*⟩

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

lemma *highestOneBitMask*:
  assumes *size v* = *n*
  shows *highestOneBit v* = *highestOneBit* (*and v* (*mask n*))
  ⟨*proof*⟩

lemma *maskSmaller*:
  fixes *v* :: ′*a* :: *len word*
  assumes ¬ *bit v n*
  shows *and v* (*mask* (*Suc n*)) = *and v* (*mask n*)
  ⟨*proof*⟩

lemma *highestOneBitSmaller*:
  assumes *size v* = *Suc n*
  assumes ¬ *bit v n*
  shows *highestOneBit v* = *highestOneBit* (*and v* (*mask n*))
  ⟨*proof*⟩

lemma *highestOneBitRecMask*:
  shows *highestOneBit* (*and v* (*mask* (*Suc n*))) = *highestOneBitRec n v*
⟨*proof*⟩

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

lemma *highestOneBitImpl*[*code*]:

$highestOneBit\ v = highestOneBitRec\ (size\ v)\ v$
⟨*proof*⟩

lemma $highestOneBit\ (0x5 :: int8) = 2$ ⟨*proof*⟩

2.2 Long.lowestOneBit

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

lemma *max-bit*: $bit\ (v::('a::len)\ word)\ n \Longrightarrow n < size\ v$
⟨*proof*⟩

lemma *max-set-bit*: $MaxOrNeg\ \{n\ .\ bit\ (v::('a::len)\ word)\ n\} < Nat.size\ v$
⟨*proof*⟩

2.3 Long.numberOfLeadingZeros

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

lemma *MaxOrNeg-neg*: $MaxOrNeg\ \{\} = -1$
⟨*proof*⟩

lemma *MaxOrNeg-max*: $s \neq \{\} \Longrightarrow MaxOrNeg\ s = Max\ s$
⟨*proof*⟩

lemma *zero-no-bits*:
$\{n\ .\ bit\ 0\ n\} = \{\}$
⟨*proof*⟩

lemma $highestOneBit\ (0::64\ word) = -1$
⟨*proof*⟩

lemma $numberOfLeadingZeros\ (0::64\ word) = 64$
⟨*proof*⟩

lemma *highestOneBit-top*: $Max\ \{highestOneBit\ (v::64\ word)\} < 64$
⟨*proof*⟩

lemma *numberOfLeadingZeros-top*: $Max\ \{numberOfLeadingZeros\ (v::64\ word)\} \leq 64$
⟨*proof*⟩

lemma *numberOfLeadingZeros-range*: $0 \leq numberOfLeadingZeros\ a \wedge numberOfLeadingZeros\ a \leq Nat.size\ a$
⟨*proof*⟩

lemma *leadingZerosAddHighestOne*: $numberOfLeadingZeros\ v + highestOneBit\ v = Nat.size\ v - 1$

⟨*proof*⟩

2.4 Long.numberOfTrailingZeros

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

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

lemma *bit-zero-set-in-top*: *bit* (−*1*::*'a*::*len word*) *0*
⟨*proof*⟩

lemma *nat-bot-set*: (*0*::*nat*) ∈ *xs* ⟶ (∀ *x* ∈ *xs* . *0* ≤ *x*)
⟨*proof*⟩

lemma *numberOfTrailingZeros* (*0*::*64 word*) = *64*
⟨*proof*⟩

2.5 Long.reverseBytes

fun *reverseBytes-fun* :: (*'a*::*len*) *word* ⇒ *nat* ⇒ (*'a*::*len*) *word* ⇒ (*'a*::*len*) *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*) *word* ⇒ *nat* **where**
bitCount v = *card* {*n* . *bit v n*}

fun *bitCount-fun* :: (*'a*::*len*) *word* ⇒ *nat* ⇒ *nat* **where**
bitCount-fun v n = (*if* (*n* = *0*) *then*
(*if* (*bit v n*) *then 1 else 0*) *else*
if (*bit v n*) *then* (*1* + *bitCount-fun* (*v*) (*n* − *1*))
else (*0* + *bitCount-fun* (*v*) (*n* − *1*)))

lemma *bitCount 0* = *0*
⟨*proof*⟩

2.7 Long.zeroCount

definition *zeroCount* :: (*'a*::*len*) *word* ⇒ *nat* **where**
zeroCount v = *card* {*n*. *n* < *Nat.size v* ∧ ¬(*bit v n*)}

lemma *zeroCount-finite*: *finite* {*n*. *n* < *Nat.size v* ∧ ¬(*bit v n*)}
⟨*proof*⟩

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

lemma *negone-all-bits*:
$\{n\ .\ bit\ (-1::('a::len)\ word)\ n\} = \{n\ .\ 0 \leq n \wedge n < LENGTH('a)\}$
⟨*proof*⟩

lemma *bitCount-finite*:
$finite\ \{n\ .\ bit\ (v::('a::len)\ word)\ n\}$
⟨*proof*⟩

lemma *card-of-range*:
$x = card\ \{n\ .\ 0 \leq n \wedge n < x\}$
⟨*proof*⟩

lemma *range-of-nat*:
$\{(n::nat)\ .\ 0 \leq n \wedge n < x\} = \{n\ .\ n < x\}$
⟨*proof*⟩

lemma *finite-range*:
$finite\ \{n::nat\ .\ n < x\}$
⟨*proof*⟩

lemma *range-eq*:
fixes $x\ y :: nat$
shows $card\ \{y..<x\} = card\ \{y<..x\}$
⟨*proof*⟩

lemma *card-of-range-bound*:
fixes $x\ y :: nat$
assumes $x > y$
shows $x - y = card\ \{n\ .\ y < n \wedge n \leq x\}$
⟨*proof*⟩

lemma $bitCount\ (-1::('a::len)\ word) = LENGTH('a)$
⟨*proof*⟩

lemma *bitCount-range*:
fixes $n :: ('a::len)\ word$
shows $0 \leq bitCount\ n \wedge bitCount\ n \leq Nat.size\ n$
⟨*proof*⟩

lemma *zerosAboveHighestOne*:
$n > highestOneBit\ a \Longrightarrow \neg(bit\ a\ n)$
⟨*proof*⟩

lemma *zerosBelowLowestOne*:

assumes $n < lowestOneBit\ a$
shows $\neg(bit\ a\ n)$
⟨*proof*⟩

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

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

lemma *qualified-bitCount*:
$bitCount\ v = card\ \{n\ .\ n < Nat.size\ v \wedge bit\ v\ n\}$
⟨*proof*⟩

lemma *card-eq*:
assumes $finite\ x \wedge finite\ y \wedge finite\ z$
assumes $x \cup y = z$
assumes $y \cap x = \{\}$
shows $card\ z - card\ y = card\ x$
⟨*proof*⟩

lemma *card-add*:
assumes $finite\ x \wedge finite\ y \wedge finite\ z$
assumes $x \cup y = z$
assumes $y \cap x = \{\}$
shows $card\ x + card\ y = card\ z$
⟨*proof*⟩

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 $card\ \{n.\ Q\ n \wedge P\ n\} + card\ \{n.\ Q\ n \wedge \neg(P\ n)\} = card\ \{n.\ Q\ n\}$
⟨*proof*⟩

lemma *ones-zero-sum-to-width*:
$bitCount\ a + zeroCount\ a = Nat.size\ a$
⟨*proof*⟩

lemma *intersect-bitCount-helper*:
$card\ \{n\ .\ n < Nat.size\ a\} - bitCount\ a = card\ \{n\ .\ n < Nat.size\ a \wedge \neg(bit\ a\ n)\}$
⟨*proof*⟩

lemma *intersect-bitCount*:
$Nat.size\ a - bitCount\ a = card\ \{n\ .\ n < Nat.size\ a \wedge \neg(bit\ a\ n)\}$

⟨*proof*⟩

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* ⇒ *nat* **where**
intval-bits (*IntVal b v*) = *b*

fun *intval-word* :: *Value* ⇒ *int64* **where**
intval-word (*IntVal b v*) = *v*

Converts an integer word into a Java value.

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

fun *array-length* :: *Value* ⇒ *Value* **where**
array-length (*ArrayVal len list*) = *new-int 32* (*word-of-nat len*)

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

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

fun *is-int-val* :: *Value* ⇒ *bool* **where**
is-int-val v = *is-IntVal v*

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

lemma *neg-one-signed*[*simp*]:
assumes *0* < *b*
shows *int-signed-value b* (*neg-one b*) = −*1*
⟨*proof*⟩

lemma *word-unsigned*:
shows ∀ *b1 v1*. (*IntVal b1* (*word-of-int* (*int-unsigned-value b1 v1*))) = *IntVal b1 v1*
⟨*proof*⟩

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*⟩

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* ⟨*proof*⟩
corollary *sint* (*signed-take-bit 7* ((*256* + *128*) :: *int64*)) = −*128* ⟨*proof*⟩
corollary *sint* (*take-bit 7* ((*256* + *128* + *64*) :: *int64*)) = *64* ⟨*proof*⟩
corollary *sint* (*take-bit 8* ((*256* + *128* + *64*) :: *int64*)) = *128* + *64* ⟨*proof*⟩

fun *intval-narrow* :: *nat* ⇒ *nat* ⇒ *Value* ⇒ *Value* **where**

intval-narrow inBits outBits (*IntVal b v*) =
(*if inBits* = *b* ∧ *0* < *outBits* ∧ *outBits* ≤ *inBits* ∧ *inBits* ≤ *64*
then new-int outBits v
else UndefVal) |
intval-narrow - - - = *UndefVal*

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

fun *intval-zero-extend* :: *nat* ⇒ *nat* ⇒ *Value* ⇒ *Value* **where**
intval-zero-extend inBits outBits (*IntVal b v*) =
(*if inBits* = *b* ∧ *0* < *inBits* ∧ *inBits* ≤ *outBits* ∧ *outBits* ≤ *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* ≠ *UndefVal*
shows *0* < *outBits* ∧ *outBits* ≤ *inBits* ∧ *inBits* ≤ *64* ∧ *outBits* ≤ *64* ∧
is-IntVal val ∧
intval-bits val = *inBits*
⟨*proof*⟩

lemma *intval-sign-extend-ok*:
assumes *intval-sign-extend inBits outBits val* ≠ *UndefVal*
shows *0* < *inBits* ∧
inBits ≤ *outBits* ∧ *outBits* ≤ *64* ∧
is-IntVal val ∧
intval-bits val = *inBits*
⟨*proof*⟩

lemma *intval-zero-extend-ok*:
assumes *intval-zero-extend inBits outBits val* ≠ *UndefVal*
shows *0* < *inBits* ∧
inBits ≤ *outBits* ∧ *outBits* ≤ *64* ∧
is-IntVal val ∧
intval-bits val = *inBits*
⟨*proof*⟩

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 ⟨proof⟩
corollary intval-narrow 32 8 (IntVal 32 (−2)) = IntVal 8 254 ⟨proof⟩
corollary intval-narrow 32 1 (IntVal 32 (−2)) = IntVal 1 0    ⟨proof⟩
corollary intval-narrow 32 1 (IntVal 32 (−3)) = IntVal 1 1 ⟨proof⟩


corollary intval-narrow 32 8 (IntVal 64 (−2)) = UndefVal ⟨proof⟩
corollary intval-narrow 64 8 (IntVal 32 (−2)) = UndefVal ⟨proof⟩
corollary intval-narrow 64 8 (IntVal 64 254) = IntVal 8 254 ⟨proof⟩
corollary intval-narrow 64 8 (IntVal 64 (256+127)) = IntVal 8 127 ⟨proof⟩
corollary intval-narrow 64 64 (IntVal 64 (−2)) = IntVal 64 (−2) ⟨proof⟩
end
```

```
experiment begin
corollary intval-sign-extend 8 32 (IntVal 8 (256 + 128)) = IntVal 32 (2^32 −
128) ⟨proof⟩
corollary intval-sign-extend 8 32 (IntVal 8 (−2)) = IntVal 32 (2^32 − 2) ⟨proof⟩
corollary intval-sign-extend 1 32 (IntVal 1 (−2)) = IntVal 32 0    ⟨proof⟩
corollary intval-sign-extend 1 32 (IntVal 1 (−3)) = IntVal 32 (mask 32) ⟨proof⟩
```

corollary *intval-sign-extend 8 32* (*IntVal 64 254*) = *UndefVal* ⟨*proof*⟩
corollary *intval-sign-extend 8 64* (*IntVal 32 254*) = *UndefVal* ⟨*proof*⟩
corollary *intval-sign-extend 8 64* (*IntVal 8 254*) = *IntVal 64* (−*2*) ⟨*proof*⟩
corollary *intval-sign-extend 32 64* (*IntVal 32* (*2*^*32* − *2*)) = *IntVal 64* (−*2*) ⟨*proof*⟩
corollary *intval-sign-extend 64 64* (*IntVal 64* (−*2*)) = *IntVal 64* (−*2*) ⟨*proof*⟩
end

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

corollary *intval-zero-extend 8 32* (*IntVal 64* (−*2*)) = *UndefVal* ⟨*proof*⟩
corollary *intval-zero-extend 8 64* (*IntVal 64* (−*2*)) = *UndefVal* ⟨*proof*⟩
corollary *intval-zero-extend 8 64* (*IntVal 8 254*) = *IntVal 64 254* ⟨*proof*⟩
corollary *intval-zero-extend 32 64* (*IntVal 32* (*2*^*32* − *2*)) = *IntVal 64* (*2*^*32* − *2*) ⟨*proof*⟩
corollary *intval-zero-extend 64 64* (*IntVal 64* (−*2*)) = *IntVal 64* (−*2*) ⟨*proof*⟩
end

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

lemma *intval-add-sym*:
  shows *intval-add a b* = *intval-add b a*
  ⟨*proof*⟩

lemma *intval-add* (*IntVal 32* (*2*^*31*−*1*)) (*IntVal 32* (*2*^*31*−*1*)) = *IntVal 32* (*2*^*32* − *2*)
  ⟨*proof*⟩
lemma *intval-add* (*IntVal 64* (*2*^*31*−*1*)) (*IntVal 64* (*2*^*31*−*1*)) = *IntVal 64 4294967294*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *size64*: *size v = 64* **for** *v :: 64 word*
  ⟨*proof*⟩

lemma *lower-bounds-equiv*:
  assumes $0 < N$
  shows $-(((2::int)\ \hat{}\ (N-1))) = (2::int)\ \hat{}\ N\ div\ 2 * -\ 1$
  ⟨*proof*⟩

lemma *upper-bounds-equiv*:
  assumes $0 < N$
  shows $(2::int)\ \hat{}\ (N-1) = (2::int)\ \hat{}\ N\ div\ 2$
  ⟨*proof*⟩

Some min/max bounds for 64-bit words

lemma *bit-bounds-min64*: ((*fst* (*bit-bounds 64*))) ≤ (*sint* (*v::int64*))
  ⟨*proof*⟩

lemma *bit-bounds-max64*: ((*snd* (*bit-bounds 64*))) ≥ (*sint* (*v::int64*))
  ⟨*proof*⟩

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 $sint(signed\text{-}take\text{-}bit\ n\ ival) < (2::int)\ \hat{}\ n$
  ⟨*proof*⟩

lemma *signed-take-bit-int-greater-eq-minus-exp-word*:
  fixes *ival* :: *'a* :: *len word*
  assumes $n < LENGTH('a)$
  shows $- (2 \,\hat{}\, n) \leq sint(signed\text{-}take\text{-}bit\ n\ ival)$
  ⟨*proof*⟩

lemma *signed-take-bit-range*:
  fixes *ival* :: *'a* :: *len word*
  assumes $n < LENGTH('a)$
  assumes $val = sint(signed\text{-}take\text{-}bit\ n\ ival)$
  shows $- (2 \,\hat{}\, n) \leq val \wedge val < 2 \,\hat{}\, n$
  ⟨*proof*⟩

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 $val = sint(signed\text{-}take\text{-}bit\ (n - 1)\ ival)$
  shows $fst\ (bit\text{-}bounds\ n) \leq val \wedge val \leq snd\ (bit\text{-}bounds\ n)$
  ⟨*proof*⟩

lemma *signed-take-bit-bounds64*:
  fixes *ival* :: *int64*
  assumes $n \leq 64$
  assumes $0 < n$
  assumes $val = sint(signed\text{-}take\text{-}bit\ (n - 1)\ ival)$
  shows $fst\ (bit\text{-}bounds\ n) \leq val \wedge val \leq snd\ (bit\text{-}bounds\ n)$
  ⟨*proof*⟩

lemma *int-signed-value-bounds*:
  assumes $b1 \leq 64$
  assumes $0 < b1$
  shows $fst\ (bit\text{-}bounds\ b1) \leq int\text{-}signed\text{-}value\ b1\ v2\ \wedge$
        $int\text{-}signed\text{-}value\ b1\ v2 \leq snd\ (bit\text{-}bounds\ b1)$
  ⟨*proof*⟩

lemma *int-signed-value-range*:
  fixes *ival* :: *int64*
  assumes $val = int\text{-}signed\text{-}value\ n\ ival$
  shows $- (2 \,\hat{}\, (n - 1)) \leq val \wedge val < 2 \,\hat{}\, (n - 1)$
  ⟨*proof*⟩

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

lemma *take-bit-smaller-range*:
  fixes *ival* :: *'a* :: *len word*
  assumes $n < LENGTH('a)$

assumes *val* = *sint*(*take-bit n ival*)
shows *0* ≤ *val* ∧ *val* < (*2*::*int*) ^ *n*
⟨*proof*⟩

lemma *take-bit-same-size-nochange*:
fixes *ival* :: ′*a* :: *len word*
assumes *n* = *LENGTH*(′*a*)
shows *ival* = *take-bit n ival*
⟨*proof*⟩

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

lemma *take-bit-redundant*[*simp*]:
fixes *ival* :: ′*a* :: *len word*
assumes *0* < *n*
assumes *n* < *LENGTH*(′*a*)
shows *signed-take-bit* (*n* − *1*) (*take-bit n ival*) = *signed-take-bit* (*n* − *1*) *ival*
⟨*proof*⟩

lemma *take-bit-same-size-range*:
fixes *ival* :: ′*a* :: *len word*
assumes *n* = *LENGTH*(′*a*)
assumes *ival2* = *take-bit n ival*
shows − (*2* ^ *n div 2*) ≤ *sint ival2* ∧ *sint ival2* < *2* ^ *n div 2*
⟨*proof*⟩

lemma *take-bit-same-bounds*:
fixes *ival* :: ′*a* :: *len word*
assumes *n* = *LENGTH*(′*a*)
assumes *ival2* = *take-bit n ival*
shows *fst* (*bit-bounds n*) ≤ *sint ival2* ∧ *sint ival2* ≤ *snd* (*bit-bounds n*)
⟨*proof*⟩

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

lemma *scast-max-bound*:
assumes *sint* (*v* :: ′*a* :: *len word*) < *M*
assumes *LENGTH*(′*a*) < *LENGTH*(′*b*)
shows *sint* ((*scast v*) :: ′*b* :: *len word*) < *M*
⟨*proof*⟩

lemma *scast-min-bound*:
assumes *M* ≤ *sint* (*v* :: ′*a* :: *len word*)
assumes *LENGTH*(′*a*) < *LENGTH*(′*b*)
shows *M* ≤ *sint* ((*scast v*) :: ′*b* :: *len word*)
⟨*proof*⟩

lemma *scast-bigger-max-bound*:

assumes (*result* :: *'b* :: *len word*) = *scast* (*v* :: *'a* :: *len word*)
shows *sint result* < *2* ^ *LENGTH*(*'a*) *div 2*
⟨*proof*⟩

lemma *scast-bigger-min-bound*:
assumes (*result* :: *'b* :: *len word*) = *scast* (*v* :: *'a* :: *len word*)
shows − (*2* ^ *LENGTH*(*'a*) *div 2*) ≤ *sint result*
⟨*proof*⟩

lemma *scast-bigger-bit-bounds*:
assumes (*result* :: *'b* :: *len word*) = *scast* (*v* :: *'a* :: *len word*)
shows *fst* (*bit-bounds* (*LENGTH*(*'a*))) ≤ *sint result* ∧ *sint result* ≤ *snd* (*bit-bounds* (*LENGTH*(*'a*)))
⟨*proof*⟩

Results about *new_int*.

lemma *new-int-take-bits*:
assumes *IntVal b val* = *new-int b ival*
shows *take-bit b val* = *val*
⟨*proof*⟩

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*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *take-bit-dist-neg*[*simp*]:
fixes *ix* :: *'a* :: *len word*
shows *take-bit b* (− *take-bit b* (*ix*)) = *take-bit b* (− *ix*)
⟨*proof*⟩

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*
  ⟨*proof*⟩

lemma *mod-larger-ignore*:
  fixes *a* :: *int*
  fixes *m n* :: *nat*
  assumes $n < m$
  shows $(a \; mod \; 2 \,\hat{}\, m) \; mod \; 2 \,\hat{}\, n = a \; mod \; 2 \,\hat{}\, n$
  ⟨*proof*⟩

lemma *mod-dist-over-add*:
  fixes *a b c* :: *int64*
  fixes *n* :: *nat*
  assumes *1*: $0 < n$
  assumes *2*: $n < 64$
  shows $(a \; mod \; 2\hat{}n + b) \; mod \; 2\hat{}n = (a + b) \; mod \; 2\hat{}n$
⟨*proof*⟩

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* ⇒ *int* ⇒ *int* ⇒ *int64* ⇒ *int64* ⇒ *Stamp*
where
IntegerStampM b lo hi down up ≡ *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*) ⟨*proof*⟩
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*)
  ⟨*proof*⟩

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*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *NotNode* (*ir-value*: *INPUT*)
| *OrNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *ParameterNode* (*ir-index*: *nat*)
| *PiNode* (*ir-object*: *INPUT*) (*ir-guard-opt*: *INPUT-GUARD option*)
| *ReturnNode* (*ir-result-opt*: *INPUT option*) (*ir-memoryMap-opt*: *INPUT-EXT option*)
| *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*: *INPUT-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-list-to-list None = [] |
opt-list-to-list (*Some x*) = *x*

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

fun *inputs-of* :: *IRNode* ⇒ *ID list* **where**
inputs-of-AbsNode:
inputs-of (*AbsNode value*) = [*value*] |
inputs-of-AddNode:
inputs-of (*AddNode x y*) = [*x*, *y*] |
inputs-of-AndNode:
inputs-of (*AndNode x y*) = [*x*, *y*] |
inputs-of-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*, *trueValue*, *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 stateDuring*) @ (*opt-to-list stateAfter*) |
inputs-of-IsNullNode:
inputs-of (*IsNullNode value*) = [*value*] |
inputs-of-KillingBeginNode:
inputs-of (*KillingBeginNode next*) = [] |
inputs-of-LeftShiftNode:
inputs-of (*LeftShiftNode x y*) = [*x*, *y*] |
inputs-of-LoadFieldNode:
inputs-of (*LoadFieldNode nid0 field object next*) = (*opt-to-list object*) |
inputs-of-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*) = *arguments* |
inputs-of-MulNode:
inputs-of (*MulNode x y*) = [*x*, *y*] |
inputs-of-NarrowNode:

inputs-of (*NarrowNode inputBits resultBits value*) = [*value*] |
inputs-of-NegateNode:
inputs-of (*NegateNode value*) = [*value*] |
inputs-of-NewArrayNode:
inputs-of (*NewArrayNode length0 stateBefore next*) = *length0* # (*opt-to-list stateBefore*) |
inputs-of-NewInstanceNode:
inputs-of (*NewInstanceNode nid0 instanceClass stateBefore next*) = (*opt-to-list stateBefore*) |
inputs-of-NotNode:
inputs-of (*NotNode value*) = [*value*] |
inputs-of-OrNode:
inputs-of (*OrNode x y*) = [*x*, *y*] |
inputs-of-ParameterNode:
inputs-of (*ParameterNode index*) = [] |
inputs-of-PiNode:
inputs-of (*PiNode object guard*) = *object* # (*opt-to-list guard*) |
inputs-of-ReturnNode:
inputs-of (*ReturnNode result memoryMap*) = (*opt-to-list result*) @ (*opt-to-list memoryMap*) |
inputs-of-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: *successors-of* (*RefNode ref*) = [*ref*]

lemma *inputs-of* (*FrameState x* (*Some y*) (*Some z*) *None*) = *x* @ [*y*] @ *z*
⟨*proof*⟩

lemma *successors-of* (*FrameState x* (*Some y*) (*Some z*) *None*) = []
⟨*proof*⟩

lemma *inputs-of* (*IfNode c t f*) = [*c*]
⟨*proof*⟩

lemma *successors-of* (*IfNode c t f*) = [*t*, *f*]
⟨*proof*⟩

lemma *inputs-of* (*EndNode*) = [] ∧ *successors-of* (*EndNode*) = []
⟨*proof*⟩

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* ⇒ *bool* **where**
is-EndNode EndNode = *True* |
is-EndNode - = *False*

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

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

is-BinaryArithmeticNode n = ((*is-AddNode n*) ∨ (*is-AndNode n*) ∨ (*is-MulNode n*) ∨ (*is-OrNode n*) ∨ (*is-SubNode n*) ∨ (*is-XorNode n*) ∨ (*is-IntegerNormalizeCompareNode n*) ∨ (*is-IntegerMulHighNode n*))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

is-ProxyNode n = ((*is-ValueProxyNode n*))

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

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

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

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

fun *is-AbstractFixedGuardNode* :: *IRNode* ⇒ *bool* **where**
is-AbstractFixedGuardNode n = (*is-FixedGuardNode n*)

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

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

fun *is-DeoptimizingFixedWithNextNode* :: *IRNode* ⇒ *bool* **where**
is-DeoptimizingFixedWithNextNode n = ((*is-AbstractNewObjectNode n*) ∨ (*is-FixedBinaryNode n*) ∨ (*is-AbstractFixedGuardNode n*))

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

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

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

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

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

fun *is-AccessArrayNode* :: *IRNode* ⇒ *bool* **where**
is-AccessArrayNode n = ((*is-LoadIndexedNode n*) ∨ (*is-StoreIndexedNode n*))

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

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

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

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

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

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

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

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

fun *is-Node* :: *IRNode* ⇒ *bool* **where**
is-Node n = ((*is-ValueNode n*) ∨ (*is-VirtualState n*))

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

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

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

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

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

is-IndirectCanonicalization n = ((*is-LogicNode n*))

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

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

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

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

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

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

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

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

fun *is-SingleMemoryKill* :: *IRNode* ⇒ *bool* **where**
is-SingleMemoryKill n = ((*is-BytecodeExceptionNode n*) ∨ (*is-ExceptionObjectNode n*) ∨ (*is-InvokeNode n*) ∨ (*is-InvokeWithExceptionNode n*) ∨ (*is-KillingBeginNode n*) ∨ (*is-StartNode n*))

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

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

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

(*is-IntegerConvertNode n*) ∨ (*is-NotNode n*) ∨ (*is-ShiftNode n*) ∨ (*is-UnaryArithmeticNode n*))

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

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

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

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

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

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

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

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

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

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

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

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

fun *is-Simplifiable* :: *IRNode* ⇒ *bool* **where**
is-Simplifiable n = ((*is-AbstractMergeNode n*) ∨ (*is-BeginNode n*) ∨ (*is-IfNode n*) ∨ (*is-LoopExitNode n*) ∨ (*is-MethodCallTargetNode n*) ∨ (*is-NewArrayNode n*))

fun *is-StateSplit* :: *IRNode* ⇒ *bool* **where**
is-StateSplit n = ((*is-AbstractStateSplit n*) ∨ (*is-BeginStateSplitNode n*) ∨ (*is-StoreFieldNode n*))

fun *is-ConvertNode* :: *IRNode* ⇒ *bool* **where**
is-ConvertNode n = ((*is-IntegerConvertNode n*))

fun *is-sequential-node* :: *IRNode* ⇒ *bool* **where**
is-sequential-node (*StartNode* - -) = *True* |
is-sequential-node (*BeginNode* -) = *True* |
is-sequential-node (*KillingBeginNode* -) = *True* |
is-sequential-node (*LoopBeginNode* - - - -) = *True* |
is-sequential-node (*LoopExitNode* - - -) = *True* |
is-sequential-node (*MergeNode* - - -) = *True* |
is-sequential-node (*RefNode* -) = *True* |
is-sequential-node (*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*⟩

setup-lifting *type-definition-IRGraph*

lift-definition *ids* :: *IRGraph* ⇒ *ID set*
  is λ*g*. {*nid* ∈ *dom g* . ∄ *s*. *g nid* = (*Some* (*NoNode*, *s*))} ⟨*proof*⟩

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

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

lift-definition *stamp* :: *IRGraph* ⇒ *ID* ⇒ *Stamp*
  is *with-default IllegalStamp snd* ⟨*proof*⟩

lift-definition *add-node* :: *ID* ⇒ (*IRNode* × *Stamp*) ⇒ *IRGraph* ⇒ *IRGraph*
  is λ*nid k g*. *if fst k* = *NoNode then g else g*(*nid* ↦ *k*) ⟨*proof*⟩

lift-definition *remove-node* :: *ID* ⇒ *IRGraph* ⇒ *IRGraph*
  is λ*nid g*. *g*(*nid* := *None*) ⟨*proof*⟩

lift-definition *replace-node* :: *ID* ⇒ (*IRNode* × *Stamp*) ⇒ *IRGraph* ⇒ *IRGraph*
  is λ*nid k g*. *if fst k* = *NoNode then g else g*(*nid* ↦ *k*) ⟨*proof*⟩

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

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

lift-definition *irgraph* :: (*ID* × (*IRNode* × *Stamp*)) *list* ⇒ *IRGraph*
  is *map-of* ∘ *no-node*
  ⟨*proof*⟩

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

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

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

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

code-datatype *irgraph*

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

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

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

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

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

lemma [*code*]: $Rep\text{-}IRGraph\ (irgraph\ m) = map\text{-}of\ (no\text{-}node\ m)$
⟨*proof*⟩
fun *inputs* :: $IRGraph \Rightarrow ID \Rightarrow ID\ set$ **where**
$inputs\ g\ nid = set\ (inputs\text{-}of\ (kind\ g\ nid))$
— Get the successor set of a given node ID
fun *succ* :: $IRGraph \Rightarrow ID \Rightarrow ID\ set$ **where**
$succ\ g\ nid = set\ (successors\text{-}of\ (kind\ g\ nid))$
— Gives a relation between node IDs - between a node and its input nodes
fun *input-edges* :: $IRGraph \Rightarrow ID\ rel$ **where**
$input\text{-}edges\ g = (\bigcup\ i \in ids\ g.\ \{(i,j)|j.\ j \in (inputs\ g\ i)\})$
— 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**
$successor\text{-}edges\ g = (\bigcup\ i \in ids\ g.\ \{(i,j)|j\ .\ j \in (succ\ \ g\ i)\})$
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**
$nodes\text{-}of\ g\ sel = \{nid \in ids\ g\ .\ sel\ (kind\ g\ nid)\}$
fun *edge* :: $(IRNode \Rightarrow 'a) \Rightarrow ID \Rightarrow IRGraph \Rightarrow 'a$ **where**
$edge\ sel\ nid\ g = sel\ (kind\ g\ nid)$

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

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*⟩

lemma *not-in-g*:
assumes *nid* ∉ *ids g*
shows *kind g nid* = *NoNode*
⟨*proof*⟩

lemma *valid-creation*[*simp*]:
finite (*dom g*) ⟷ *Rep-IRGraph* (*Abs-IRGraph g*) = *g*
⟨*proof*⟩

lemma [*simp*]: *finite* (*ids g*)
⟨*proof*⟩

lemma [*simp*]: *finite* (*ids* (*irgraph g*))
⟨*proof*⟩

lemma [*simp*]: *finite* (*dom g*) ⟶ *ids* (*Abs-IRGraph g*) = {*nid* ∈ *dom g* . ∄ *s*. *g nid* = *Some* (*NoNode*, *s*)}
⟨*proof*⟩

lemma [*simp*]: *finite* (*dom g*) ⟶ *kind* (*Abs-IRGraph g*) = (λ*x* . (*case g x of None* ⇒ *NoNode* | *Some n* ⇒ *fst n*))
⟨*proof*⟩

lemma [*simp*]: *finite* (*dom g*) ⟶ *stamp* (*Abs-IRGraph g*) = (λ*x* . (*case g x of None* ⇒ *IllegalStamp* | *Some n* ⇒ *snd n*))
⟨*proof*⟩

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

lemma [*simp*]: *kind* (*irgraph g*) = (λ*nid*. (*case* (*map-of* (*no-node g*)) *nid of None* ⇒ *NoNode* | *Some n* ⇒ *fst n*))
⟨*proof*⟩

lemma [*simp*]: *stamp* (*irgraph g*) = (λ*nid*. (*case* (*map-of* (*no-node g*)) *nid of None*

⇒ *IllegalStamp* | *Some n* ⇒ *snd n*))
  ⟨*proof*⟩

lemma *map-of-upd*: (*map-of g*)(*k* ↦ *v*) = (*map-of* ((*k*, *v*) # *g*))
  ⟨*proof*⟩

lemma [*code*]: *replace-node nid k* (*irgraph g*) = (*irgraph* (((*nid*, *k*) # *g*)))
⟨*proof*⟩

lemma [*code*]: *add-node nid k* (*irgraph g*) = (*irgraph* (((*nid*, *k*) # *g*)))
  ⟨*proof*⟩

lemma *add-node-lookup*:
  gup = *add-node nid* (*k*, *s*) *g* ⟶
    (*if k* ≠ *NoNode then kind gup nid* = *k* ∧ *stamp gup nid* = *s else kind gup nid* = *kind g nid*)
⟨*proof*⟩

lemma *remove-node-lookup*:
  gup = *remove-node nid g* ⟶ *kind gup nid* = *NoNode* ∧ *stamp gup nid* = *IllegalStamp*
  ⟨*proof*⟩

lemma *replace-node-lookup*[*simp*]:
  gup = *replace-node nid* (*k*, *s*) *g* ∧ *k* ≠ *NoNode* ⟶ *kind gup nid* = *k* ∧ *stamp gup nid* = *s*
  ⟨*proof*⟩

lemma *replace-node-unchanged*:
  gup = *replace-node nid* (*k*, *s*) *g* ⟶ (∀ *n* ∈ (*ids g* − {*nid*}) . *n* ∈ *ids g* ∧ *n* ∈ *ids gup* ∧ *kind g n* = *kind gup n*)
  ⟨*proof*⟩

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* (λ*n*. (*n*, *ir-ref* (*kind g n*))) (*filter* (λ*id*. *is-RefNode* (*kind g id*)) (*sorted-list-of-set* (*ids g*))))

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

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

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

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

reachables ⟨*proof*⟩

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* ⟨*proof*⟩

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** $\cap_s$ *20*)
  where
diffNodesInfo g1 g2 = {(*nid*, *kind g1 nid*, *kind g2 nid*) | *nid* . *nid* ∈ *diffNodesGraph g1 g2*}

fun *eqGraph* :: *IRGraph* ⇒ *IRGraph* ⇒ *bool* (**infix** $\approx_s$ *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* (λ*nid'*. *nid'* ∉ *seen*) (*successors-of* (*kind g nid*))) *in*
    (*if length nids* > *0 then Some* (*hd nids*) *else None*))

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

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

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

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

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

type-synonym *'a TraversalState* = (*ID* × *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*;

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

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

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

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

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

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

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

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

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

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

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

code-pred $(modes\colon i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool)\ Step\ \langle proof \rangle$

end

6 Data-flow Semantics

theory *IRTreeEval*
imports
Graph.Stamp
begin

We define a tree representation of data-flow nodes, as an abstraction of the graph view.

Data-flow trees are evaluated in the context of a method state (currently called MapState in the theories for historical reasons).

The method state consists of the values for each method parameter, references to method parameters use an index of the parameter within the parameter list, as such we store a list of parameter values which are looked up at parameter references.

The method state also stores a mapping of node ids to values. The contents of this mapping is calculates during the traversal of the control flow graph.

As a concrete example, as the *SignedDivNode*::$'a$ can have side-effects (during division by zero), it is treated as part of the control-flow, since the data-flow phase is specified to be side-effect free. As a result, the control-flow semantics for *SignedDivNode*::$'a$ calculates the value of a node and maps the node identifier to the value within the method state. The data-flow semantics then just reads the value stored in the method state for the node.

type-synonym *ID* = *nat*
type-synonym *MapState* = *ID* ⇒ *Value*
type-synonym *Params* = *Value list*

definition *new-map-state* :: *MapState* **where**
new-map-state = (λ*x*. *UndefVal*)

6.1 Data-flow Tree Representation

datatype *IRUnaryOp* =
UnaryAbs
| *UnaryNeg*
| *UnaryNot*
| *UnaryLogicNegation*
| *UnaryNarrow* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*)
| *UnarySignExtend* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*)
| *UnaryZeroExtend* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*)
| *UnaryIsNull*
| *UnaryReverseBytes*
| *UnaryBitCount*

datatype *IRBinaryOp* =
BinAdd
| *BinSub*
| *BinMul*
| *BinDiv*
| *BinMod*
| *BinAnd*
| *BinOr*
| *BinXor*
| *BinShortCircuitOr*
| *BinLeftShift*
| *BinRightShift*
| *BinURightShift*
| *BinIntegerEquals*
| *BinIntegerLessThan*

| *BinIntegerBelow*
| *BinIntegerTest*
| *BinIntegerNormalizeCompare*
| *BinIntegerMulHigh*

datatype (*discs-sels*) *IRExpr* =
UnaryExpr (*ir-uop*: *IRUnaryOp*) (*ir-value*: *IRExpr*)
| *BinaryExpr* (*ir-op*: *IRBinaryOp*) (*ir-x*: *IRExpr*) (*ir-y*: *IRExpr*)
| *ConditionalExpr* (*ir-condition*: *IRExpr*) (*ir-trueValue*: *IRExpr*) (*ir-falseValue*: *IRExpr*)

| *ParameterExpr* (*ir-index*: *nat*) (*ir-stamp*: *Stamp*)

| *LeafExpr* (*ir-nid*: *ID*) (*ir-stamp*: *Stamp*)

| *ConstantExpr* (*ir-const*: *Value*)
| *ConstantVar* (*ir-name*: *String.literal*)
| *VariableExpr* (*ir-name*: *String.literal*) (*ir-stamp*: *Stamp*)

fun *is-ground* :: *IRExpr* ⇒ *bool* **where**
is-ground (*UnaryExpr op e*) = *is-ground e* |
is-ground (*BinaryExpr op e1 e2*) = (*is-ground e1* ∧ *is-ground e2*) |
is-ground (*ConditionalExpr b e1 e2*) = (*is-ground b* ∧ *is-ground e1* ∧ *is-ground e2*) |
is-ground (*ParameterExpr i s*) = *True* |
is-ground (*LeafExpr n s*) = *True* |
is-ground (*ConstantExpr v*) = *True* |
is-ground (*ConstantVar name*) = *False* |
is-ground (*VariableExpr name s*) = *False*

typedef *GroundExpr* = { *e* :: *IRExpr* . *is-ground e* }
⟨*proof*⟩

6.2 Functions for re-calculating stamps

Note: in Java all integer calculations are done as 32 or 64 bit calculations. However, here we generalise the operators to allow any size calculations. Many operators have the same output bits as their inputs. However, the unary integer operators that are not *normal_unary* are narrowing or widening operators, so the result bits is specified by the operator. The binary integer operators are divided into three groups: (1) *binary_fixed_*32 operators always output 32 bits, (2) *binary_shift_ops* operators output size is determined by their left argument, and (3) other operators output the same number of bits as both their inputs.

abbreviation *binary-normal* :: *IRBinaryOp set* **where**
binary-normal ≡ {*BinAdd*, *BinMul*, *BinDiv*, *BinMod*, *BinSub*, *BinAnd*, *BinOr*, *BinXor*}

abbreviation *binary-fixed-32-ops* :: *IRBinaryOp set* **where**
binary-fixed-32-ops ≡ {*BinShortCircuitOr*, *BinIntegerEquals*, *BinIntegerLessThan*, *BinIntegerBelow*, *BinIntegerTest*, *BinIntegerNormalizeCompare*}

abbreviation *binary-shift-ops* :: *IRBinaryOp set* **where**
binary-shift-ops ≡ {*BinLeftShift*, *BinRightShift*, *BinURightShift*}

abbreviation *binary-fixed-ops* :: *IRBinaryOp set* **where**
binary-fixed-ops ≡ {*BinIntegerMulHigh*}

abbreviation *normal-unary* :: *IRUnaryOp set* **where**
normal-unary ≡ {*UnaryAbs*, *UnaryNeg*, *UnaryNot*, *UnaryLogicNegation*, *UnaryReverseBytes*}

abbreviation *unary-fixed-32-ops* :: *IRUnaryOp set* **where**
unary-fixed-32-ops ≡ {*UnaryBitCount*}

abbreviation *boolean-unary* :: *IRUnaryOp set* **where**
boolean-unary ≡ {*UnaryIsNull*}

lemma *binary-ops-all*:
shows *op* ∈ *binary-normal* ∨ *op* ∈ *binary-fixed-32-ops* ∨ *op* ∈ *binary-fixed-ops* ∨ *op* ∈ *binary-shift-ops*
⟨*proof*⟩

lemma *binary-ops-distinct-normal*:
shows *op* ∈ *binary-normal* ⟹ *op* ∉ *binary-fixed-32-ops* ∧ *op* ∉ *binary-fixed-ops* ∧ *op* ∉ *binary-shift-ops*
⟨*proof*⟩

lemma *binary-ops-distinct-fixed-32*:
shows *op* ∈ *binary-fixed-32-ops* ⟹ *op* ∉ *binary-normal* ∧ *op* ∉ *binary-fixed-ops* ∧ *op* ∉ *binary-shift-ops*
⟨*proof*⟩

lemma *binary-ops-distinct-fixed*:
shows *op* ∈ *binary-fixed-ops* ⟹ *op* ∉ *binary-fixed-32-ops* ∧ *op* ∉ *binary-normal* ∧ *op* ∉ *binary-shift-ops*
⟨*proof*⟩

lemma *binary-ops-distinct-shift*:
shows *op* ∈ *binary-shift-ops* ⟹ *op* ∉ *binary-fixed-32-ops* ∧ *op* ∉ *binary-fixed-ops* ∧ *op* ∉ *binary-normal*

⟨*proof*⟩

lemma *unary-ops-distinct*:
shows $op \in normal\text{-}unary \Longrightarrow op \notin boolean\text{-}unary \wedge op \notin unary\text{-}fixed\text{-}32\text{-}ops$
and $op \in boolean\text{-}unary \Longrightarrow op \notin normal\text{-}unary \wedge op \notin unary\text{-}fixed\text{-}32\text{-}ops$
and $op \in unary\text{-}fixed\text{-}32\text{-}ops \Longrightarrow op \notin boolean\text{-}unary \wedge op \notin normal\text{-}unary$
⟨*proof*⟩

fun *stamp-unary* :: *IRUnaryOp* ⇒ *Stamp* ⇒ *Stamp* **where**

stamp-unary UnaryIsNull - = (IntegerStamp 32 0 1) |
stamp-unary op (IntegerStamp b lo hi) =
unrestricted-stamp (IntegerStamp
(if op ∈ normal-unary then b else
if op ∈ boolean-unary then 32 else
if op ∈ unary-fixed-32-ops then 32 else
(ir-resultBits op)) lo hi) |

stamp-unary op - = IllegalStamp

fun *stamp-binary* :: *IRBinaryOp* ⇒ *Stamp* ⇒ *Stamp* ⇒ *Stamp* **where**
stamp-binary op (IntegerStamp b1 lo1 hi1) (IntegerStamp b2 lo2 hi2) =
(if op ∈ binary-shift-ops then unrestricted-stamp (IntegerStamp b1 lo1 hi1)
else if b1 ≠ b2 then IllegalStamp else
(if op ∈ binary-fixed-32-ops
then unrestricted-stamp (IntegerStamp 32 lo1 hi1)
else unrestricted-stamp (IntegerStamp b1 lo1 hi1))) |

stamp-binary op - - = IllegalStamp

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

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

6.3 Data-flow Tree Evaluation

fun *unary-eval* :: *IRUnaryOp* ⇒ *Value* ⇒ *Value* **where**
unary-eval UnaryAbs v = intval-abs v |
unary-eval UnaryNeg v = intval-negate v |
unary-eval UnaryNot v = intval-not v |
unary-eval UnaryLogicNegation v = intval-logic-negation v |

unary-eval (*UnaryNarrow inBits outBits*) *v* = *intval-narrow inBits outBits v* |
unary-eval (*UnarySignExtend inBits outBits*) *v* = *intval-sign-extend inBits outBits v* |
unary-eval (*UnaryZeroExtend inBits outBits*) *v* = *intval-zero-extend inBits outBits v* |
unary-eval UnaryIsNull v = *intval-is-null v* |
unary-eval UnaryReverseBytes v = *intval-reverse-bytes v* |
unary-eval UnaryBitCount v = *intval-bit-count v*

fun *bin-eval* :: *IRBinaryOp* ⇒ *Value* ⇒ *Value* ⇒ *Value* **where**
bin-eval BinAdd v1 v2 = *intval-add v1 v2* |
bin-eval BinSub v1 v2 = *intval-sub v1 v2* |
bin-eval BinMul v1 v2 = *intval-mul v1 v2* |
bin-eval BinDiv v1 v2 = *intval-div v1 v2* |
bin-eval BinMod v1 v2 = *intval-mod v1 v2* |
bin-eval BinAnd v1 v2 = *intval-and v1 v2* |
bin-eval BinOr v1 v2 = *intval-or v1 v2* |
bin-eval BinXor v1 v2 = *intval-xor v1 v2* |
bin-eval BinShortCircuitOr v1 v2 = *intval-short-circuit-or v1 v2* |
bin-eval BinLeftShift v1 v2 = *intval-left-shift v1 v2* |
bin-eval BinRightShift v1 v2 = *intval-right-shift v1 v2* |
bin-eval BinURightShift v1 v2 = *intval-uright-shift v1 v2* |
bin-eval BinIntegerEquals v1 v2 = *intval-equals v1 v2* |
bin-eval BinIntegerLessThan v1 v2 = *intval-less-than v1 v2* |
bin-eval BinIntegerBelow v1 v2 = *intval-below v1 v2* |
bin-eval BinIntegerTest v1 v2 = *intval-test v1 v2* |
bin-eval BinIntegerNormalizeCompare v1 v2 = *intval-normalize-compare v1 v2* |
bin-eval BinIntegerMulHigh v1 v2 = *intval-mul-high v1 v2*

lemma *defined-eval-is-intval*:
shows *bin-eval op x y* ≠ *UndefVal* ⟹ (*is-IntVal x* ∧ *is-IntVal y*)
⟨*proof*⟩

lemmas *eval-thms* =
intval-abs.simps intval-negate.simps intval-not.simps
intval-logic-negation.simps intval-narrow.simps
intval-sign-extend.simps intval-zero-extend.simps
intval-add.simps intval-mul.simps intval-sub.simps
intval-and.simps intval-or.simps intval-xor.simps
intval-left-shift.simps intval-right-shift.simps
intval-uright-shift.simps intval-equals.simps
intval-less-than.simps intval-below.simps

inductive *not-undef-or-fail* :: *Value* ⇒ *Value* ⇒ *bool* **where**
⟦*value* ≠ *UndefVal*⟧ ⟹ *not-undef-or-fail value value*

notation (*latex* **output**)

not-undef-or-fail (- = -)

inductive
evaltree :: *MapState* ⇒ *Params* ⇒ *IRExpr* ⇒ *Value* ⇒ *bool* ([-,-] ⊢ - ↦ - *55*)
for *m p* **where**

ConstantExpr:
⟦*wf-value c*⟧
⟹ [*m*,*p*] ⊢ (*ConstantExpr c*) ↦ *c* |

ParameterExpr:
⟦*i* < *length p*; *valid-value* (*p*!*i*) *s*⟧
⟹ [*m*,*p*] ⊢ (*ParameterExpr i s*) ↦ *p*!*i* |

ConditionalExpr:
⟦[*m*,*p*] ⊢ *ce* ↦ *cond*;
cond ≠ *UndefVal*;
branch = (*if val-to-bool cond then te else fe*);
[*m*,*p*] ⊢ *branch* ↦ *result*;
result ≠ *UndefVal*;

[*m*,*p*] ⊢ *te* ↦ *true*; *true* ≠ *UndefVal*;
[*m*,*p*] ⊢ *fe* ↦ *false*; *false* ≠ *UndefVal*⟧
⟹ [*m*,*p*] ⊢ (*ConditionalExpr ce te fe*) ↦ *result* |

UnaryExpr:
⟦[*m*,*p*] ⊢ *xe* ↦ *x*;
result = (*unary-eval op x*);
result ≠ *UndefVal*⟧
⟹ [*m*,*p*] ⊢ (*UnaryExpr op xe*) ↦ *result* |

BinaryExpr:
⟦[*m*,*p*] ⊢ *xe* ↦ *x*;
[*m*,*p*] ⊢ *ye* ↦ *y*;
result = (*bin-eval op x y*);
result ≠ *UndefVal*⟧
⟹ [*m*,*p*] ⊢ (*BinaryExpr op xe ye*) ↦ *result* |

LeafExpr:
⟦*val* = *m n*;
valid-value val s⟧
⟹ [*m*,*p*] ⊢ *LeafExpr n s* ↦ *val*

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

inductive

```
evaltrees :: MapState ⇒ Params ⇒ IRExpr list ⇒ Value list ⇒ bool ([-,-] ⊢ - [↦]
- 55)
  for m p where

  EvalNil:
  [m,p] ⊢ [] [↦] [] |

  EvalCons:
  ⟦[m,p] ⊢ x ↦ xval;
    [m,p] ⊢ yy [↦] yyval⟧
    ⟹ [m,p] ⊢ (x#yy) [↦] (xval#yyval)

code-pred (modes: i ⇒ i ⇒ i ⇒ o ⇒ bool as evalTs)
  evaltrees ⟨proof⟩

definition sq-param0 :: IRExpr where
  sq-param0 = BinaryExpr BinMul
    (ParameterExpr 0 (IntegerStamp 32 (− 2147483648) 2147483647))
    (ParameterExpr 0 (IntegerStamp 32 (− 2147483648) 2147483647))

values {v. evaltree new-map-state [IntVal 32 5] sq-param0 v}


declare evaltree.intros [intro]
declare evaltrees.intros [intro]
```

6.4 Data-flow Tree Refinement

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

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

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

```
lemma equivp equiv-exprs
  ⟨proof⟩
```

We define a refinement ordering over IRExpr and show that it is a preorder. Note that it is asymmetric because e2 may refer to fewer variables than e1.

```
instantiation IRExpr :: preorder begin

notation less-eq (infix ⊑ 65)

definition
```

le-expr-def [*simp*]:
$(e_2 \leq e_1) \longleftrightarrow (\forall\ m\ p\ v.\ (([m,p] \vdash e_1 \mapsto v) \longrightarrow ([m,p] \vdash e_2 \mapsto v)))$

definition
lt-expr-def [*simp*]:
$(e_1 < e_2) \longleftrightarrow (e_1 \leq e_2 \wedge \neg\ (e_1 \doteq e_2))$

instance ⟨*proof*⟩

end

abbreviation (**output**) *Refines* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* (**infix** ⊒ *64*)
where $e_1 \sqsupseteq e_2 \equiv (e_2 \leq e_1)$

6.5 Stamp Masks

A stamp can contain additional range information in the form of masks. A stamp has an up mask and a down mask, corresponding to a the bits that may be set and the bits that must be set.

Examples: A stamp where no range information is known will have; an up mask of -1 as all bits may be set, and a down mask of 0 as no bits must be set.

A stamp known to be one should have; an up mask of 1 as only the first bit may be set, no others, and a down mask of 1 as the first bit must be set and no others.

We currently don't carry mask information in stamps, and instead assume correct masks to prove optimizations.

locale *stamp-mask* =
fixes *up* :: *IRExpr* ⇒ *int64* (↑)
fixes *down* :: *IRExpr* ⇒ *int64* (↓)
assumes *up-spec*: $[m,\ p] \vdash e \mapsto IntVal\ b\ v \Longrightarrow (and\ v\ (not\ ((ucast\ (\uparrow e))))) = 0$
and *down-spec*: $[m,\ p] \vdash e \mapsto IntVal\ b\ v \Longrightarrow (and\ (not\ v)\ (ucast\ (\downarrow e))) = 0$
begin

lemma *may-implies-either*:
$[m,\ p] \vdash e \mapsto IntVal\ b\ v \Longrightarrow bit\ (\uparrow e)\ n \Longrightarrow bit\ v\ n = False \vee bit\ v\ n = True$
⟨*proof*⟩

lemma *not-may-implies-false*:
$[m,\ p] \vdash e \mapsto IntVal\ b\ v \Longrightarrow \neg(bit\ (\uparrow e)\ n) \Longrightarrow bit\ v\ n = False$
⟨*proof*⟩

lemma *must-implies-true*:
$[m,\ p] \vdash e \mapsto IntVal\ b\ v \Longrightarrow bit\ (\downarrow e)\ n \Longrightarrow bit\ v\ n = True$
⟨*proof*⟩

lemma *not-must-implies-either*:

$[m, p] \vdash e \mapsto IntVal\ b\ v \Longrightarrow \neg(bit\ (\downarrow e)\ n) \Longrightarrow bit\ v\ n = False \vee bit\ v\ n = True$
⟨*proof*⟩

lemma *must-implies-may*:
$[m, p] \vdash e \mapsto IntVal\ b\ v \Longrightarrow n < 32 \Longrightarrow bit\ (\downarrow e)\ n \Longrightarrow bit\ (\uparrow e)\ n$
⟨*proof*⟩

lemma *up-mask-and-zero-implies-zero*:
assumes $and\ (\uparrow x)\ (\uparrow y) = 0$
assumes $[m, p] \vdash x \mapsto IntVal\ b\ xv$
assumes $[m, p] \vdash y \mapsto IntVal\ b\ yv$
shows $and\ xv\ yv = 0$
⟨*proof*⟩

lemma *not-down-up-mask-and-zero-implies-zero*:
assumes $and\ (not\ (\downarrow x))\ (\uparrow y) = 0$
assumes $[m, p] \vdash x \mapsto IntVal\ b\ xv$
assumes $[m, p] \vdash y \mapsto IntVal\ b\ yv$
shows $and\ xv\ yv = yv$
⟨*proof*⟩

end

definition *IRExpr-up* :: *IRExpr* ⇒ *int64* **where**
IRExpr-up e = *not 0*

definition *IRExpr-down* :: *IRExpr* ⇒ *int64* **where**
IRExpr-down e = *0*

lemma *ucast-zero*: (*ucast* (*0*::*int64*)::*int32*) = *0*
⟨*proof*⟩

lemma *ucast-minus-one*: (*ucast* (−*1*::*int64*)::*int32*) = −*1*
⟨*proof*⟩

interpretation *simple-mask*: *stamp-mask*
IRExpr-up :: *IRExpr* ⇒ *int64*
IRExpr-down :: *IRExpr* ⇒ *int64*
⟨*proof*⟩

end

6.6 Data-flow Tree Theorems

theory *IRTreeEvalThms*
imports
Graph.ValueThms
IRTreeEval
begin

6.6.1 Deterministic Data-flow Evaluation

lemma *evalDet*:
$[m,p] \vdash e \mapsto v_1 \Longrightarrow$
$[m,p] \vdash e \mapsto v_2 \Longrightarrow$
$v_1 = v_2$
⟨*proof*⟩

lemma *evalAllDet*:
$[m,p] \vdash e\ [\mapsto]\ v1 \Longrightarrow$
$[m,p] \vdash e\ [\mapsto]\ v2 \Longrightarrow$
$v1 = v2$
⟨*proof*⟩

6.6.2 Typing Properties for Integer Evaluation Functions

We use three simple typing properties on integer values: $is_IntVal32, is_IntVal64$ and the more general is_IntVal.

lemma *unary-eval-not-obj-ref*:
shows *unary-eval op x* ≠ *ObjRef v*
⟨*proof*⟩

lemma *unary-eval-not-obj-str*:
shows *unary-eval op x* ≠ *ObjStr v*
⟨*proof*⟩

lemma *unary-eval-not-array*:
shows *unary-eval op x* ≠ *ArrayVal len v*
⟨*proof*⟩

lemma *unary-eval-int*:
assumes *unary-eval op x* ≠ *UndefVal*
shows *is-IntVal* (*unary-eval op x*)
⟨*proof*⟩

lemma *bin-eval-int*:
assumes *bin-eval op x y* ≠ *UndefVal*
shows *is-IntVal* (*bin-eval op x y*)
⟨*proof*⟩

lemma *IntVal0*:
(*IntVal 32 0*) = (*new-int 32 0*)
⟨*proof*⟩

lemma *IntVal1*:
(*IntVal 32 1*) = (*new-int 32 1*)
⟨*proof*⟩

lemma *bin-eval-new-int*:
assumes *bin-eval op x y* ≠ *UndefVal*
shows ∃ *b v*. (*bin-eval op x y*) = *new-int b v* ∧
b = (*if op* ∈ *binary-fixed-32-ops then 32 else intval-bits x*)
⟨*proof*⟩

lemma *int-stamp*:
assumes *is-IntVal v*
shows *is-IntegerStamp* (*constantAsStamp v*)
⟨*proof*⟩

lemma *validStampIntConst*:
assumes *v* = *IntVal b ival*
assumes *0* < *b* ∧ *b* ≤ *64*
shows *valid-stamp* (*constantAsStamp v*)
⟨*proof*⟩

lemma *validDefIntConst*:
assumes *v*: *v* = *IntVal b ival*
assumes *0* < *b* ∧ *b* ≤ *64*
assumes *take-bit b ival* = *ival*
shows *valid-value v* (*constantAsStamp v*)
⟨*proof*⟩

6.6.3 Evaluation Results are Valid

A valid value cannot be $UndefVal$.

lemma *valid-not-undef*:
assumes *valid-value val s*
assumes *s* ≠ *VoidStamp*
shows *val* ≠ *UndefVal*
⟨*proof*⟩

lemma *valid-VoidStamp*[*elim*]:
shows *valid-value val VoidStamp* ⟹ *val* = *UndefVal*
⟨*proof*⟩

lemma *valid-ObjStamp*[*elim*]:
shows *valid-value val* (*ObjectStamp klass exact nonNull alwaysNull*) ⟹ (∃ *v*. *val* = *ObjRef v*)
⟨*proof*⟩

lemma *valid-int*[*elim*]:
  shows *valid-value val* (*IntegerStamp b lo hi*) $\Longrightarrow$ ($\exists$ *v*. *val* = *IntVal b v*)
  ⟨*proof*⟩

lemmas *valid-value-elims* =
  valid-VoidStamp
  valid-ObjStamp
  valid-int

lemma *evaltree-not-undef*:
  fixes *m p e v*
  shows ([*m*,*p*] ⊢ *e* ↦ *v*) $\Longrightarrow$ *v* ≠ *UndefVal*
  ⟨*proof*⟩

lemma *leafint*:
  assumes [*m*,*p*] ⊢ *LeafExpr i* (*IntegerStamp b lo hi*) ↦ *val*
  shows ∃ *b v*. *val* = (*IntVal b v*)

⟨*proof*⟩

lemma *default-stamp* [*simp*]: *default-stamp* = *IntegerStamp 32* (−*2147483648*) *2147483647*
  ⟨*proof*⟩

lemma *valid-value-signed-int-range* [*simp*]:
  assumes *valid-value val* (*IntegerStamp b lo hi*)
  assumes *lo* < *0*
  shows ∃ *v*. (*val* = *IntVal b v* ∧
      lo ≤ *int-signed-value b v* ∧
      int-signed-value b v ≤ *hi*)
  ⟨*proof*⟩

6.6.4 Example Data-flow Optimisations

6.6.5 Monotonicity of Expression Refinement

We prove that each subexpression position is monotonic. That is, optimizing a subexpression anywhere deep inside a top-level expression also optimizes that top-level expression.

Note that we might also be able to do this via reusing Isabelle's *mono* operator (HOL.Orderings theory), proving instantiations like $mono(UnaryExprop)$, but it is not obvious how to do this for both arguments of the binary expressions.

lemma *mono-unary*:
  assumes $x \geq x'$
  shows (*UnaryExpr op x*) ≥ (*UnaryExpr op x*′)
  ⟨*proof*⟩

lemma *mono-binary*:
  assumes $x \geq x'$
  assumes $y \geq y'$
  shows $(BinaryExpr\ op\ x\ y) \geq (BinaryExpr\ op\ x'\ y')$
  ⟨*proof*⟩

lemma *never-void*:
  assumes $[m, p] \vdash x \mapsto xv$
  assumes *valid-value xv* (*stamp-expr xe*)
  shows *stamp-expr xe* $\neq$ *VoidStamp*
  ⟨*proof*⟩

lemma *compatible-trans*:
  compatible x y $\wedge$ *compatible y z* $\Longrightarrow$ *compatible x z*
  ⟨*proof*⟩

lemma *compatible-refl*:
  compatible x y $\Longrightarrow$ *compatible y x*
  ⟨*proof*⟩

lemma *mono-conditional*:
  assumes $c \geq c'$
  assumes $t \geq t'$
  assumes $f \geq f'$
  shows $(ConditionalExpr\ c\ t\ f) \geq (ConditionalExpr\ c'\ t'\ f')$
⟨*proof*⟩

6.7 Unfolding rules for evaltree quadruples down to bin-eval level

These rewrite rules can be useful when proving optimizations. They support top-down rewriting of each level of the tree into the lower-level $bin_e val$ / $unary_e val$ level, simply by saying $unfoldingunfold_e valtree$.

lemma *unfold-const*:
  $([m,p] \vdash ConstantExpr\ c \mapsto v) = (wf\text{-}value\ v \wedge v = c)$
  ⟨*proof*⟩

lemma *unfold-binary*:
  shows $([m,p] \vdash BinaryExpr\ op\ xe\ ye \mapsto val) = (\exists\ x\ y.$
    $(([m,p] \vdash xe \mapsto x) \wedge$

$([m,p] \vdash ye \mapsto y) \wedge$
$(val = bin\text{-}eval\ op\ x\ y) \wedge$
$(val \neq UndefVal)$
$))$ (**is** $?L = ?R$)
⟨*proof*⟩

lemma *unfold-unary*:
shows $([m,p] \vdash UnaryExpr\ op\ xe \mapsto val)$
$= (\exists\ x.$
$(([m,p] \vdash xe \mapsto x) \wedge$
$(val = unary\text{-}eval\ op\ x) \wedge$
$(val \neq UndefVal)$
$))$ (**is** $?L = ?R$)
⟨*proof*⟩

lemmas *unfold-evaltree* =
unfold-binary
unfold-unary

6.8 Lemmas about *new_int* and integer eval results.

lemma *unary-eval-new-int*:
assumes *def*: $unary\text{-}eval\ op\ x \neq UndefVal$
shows $\exists\, b\ v.\ (unary\text{-}eval\ op\ x = new\text{-}int\ b\ v \wedge$

$b = (if\ op \in normal\text{-}unary\quad then\ intval\text{-}bits\ x\quad else$
$if\ op \in boolean\text{-}unary\quad then\ 32\quad else$
$if\ op \in unary\text{-}fixed\text{-}32\text{-}ops\ then\ 32\quad else$
$ir\text{-}resultBits\ op))$
⟨*proof*⟩

lemma *new-int-unused-bits-zero*:
assumes $IntVal\ b\ ival = new\text{-}int\ b\ ival0$
shows $take\text{-}bit\ b\ ival = ival$
⟨*proof*⟩

lemma *unary-eval-unused-bits-zero*:
assumes $unary\text{-}eval\ op\ x = IntVal\ b\ ival$
shows $take\text{-}bit\ b\ ival = ival$
⟨*proof*⟩

lemma *bin-eval-unused-bits-zero*:
assumes $bin\text{-}eval\ op\ x\ y = (IntVal\ b\ ival)$
shows $take\text{-}bit\ b\ ival = ival$
⟨*proof*⟩

lemma *eval-unused-bits-zero*:

$[m,p] \vdash xe \mapsto (IntVal\ b\ ix) \Longrightarrow take\text{-}bit\ b\ ix = ix$
⟨*proof*⟩

lemma *unary-normal-bitsize*:
assumes $unary\text{-}eval\ op\ x = IntVal\ b\ ival$
assumes $op \in normal\text{-}unary$
shows $\exists\ ix.\ x = IntVal\ b\ ix$
⟨*proof*⟩

lemma *unary-not-normal-bitsize*:
assumes $unary\text{-}eval\ op\ x = IntVal\ b\ ival$
assumes $op \notin normal\text{-}unary \land op \notin boolean\text{-}unary \land op \notin unary\text{-}fixed\text{-}32\text{-}ops$
shows $b = ir\text{-}resultBits\ op \land 0 < b \land b \leq 64$
⟨*proof*⟩

lemma *unary-eval-bitsize*:
assumes $unary\text{-}eval\ op\ x = IntVal\ b\ ival$
assumes *2*: $x = IntVal\ bx\ ix$
assumes $0 < bx \land bx \leq 64$
shows $0 < b \land b \leq 64$
⟨*proof*⟩

lemma *bin-eval-inputs-are-ints*:
assumes $bin\text{-}eval\ op\ x\ y = IntVal\ b\ ix$
obtains *xb yb xi yi* **where** $x = IntVal\ xb\ xi \land y = IntVal\ yb\ yi$
⟨*proof*⟩

lemma *eval-bits-1-64*:
$[m,p] \vdash xe \mapsto (IntVal\ b\ ix) \Longrightarrow 0 < b \land b \leq 64$
⟨*proof*⟩

lemma *bin-eval-normal-bits*:
assumes $op \in binary\text{-}normal$
assumes $bin\text{-}eval\ op\ x\ y = xy$
assumes $xy \neq UndefVal$
shows $\exists xv\ yv\ xyv\ b.\ (x = IntVal\ b\ xv \land y = IntVal\ b\ yv \land xy = IntVal\ b\ xyv)$
⟨*proof*⟩

lemma *unfold-binary-width-bin-normal*:
assumes $op \in binary\text{-}normal$
shows $\bigwedge xv\ yv.$
$IntVal\ b\ val = bin\text{-}eval\ op\ xv\ yv \Longrightarrow$
$[m,p] \vdash xe \mapsto xv \Longrightarrow$
$[m,p] \vdash ye \mapsto yv \Longrightarrow$
$bin\text{-}eval\ op\ xv\ yv \neq UndefVal \Longrightarrow$
$\exists xa.$

$(([m,p] \vdash xe \mapsto IntVal\ b\ xa) \wedge$
$(\exists\, ya.\ (([m,p] \vdash ye \mapsto IntVal\ b\ ya) \wedge$
$bin\text{-}eval\ op\ xv\ yv = bin\text{-}eval\ op\ (IntVal\ b\ xa)\ (IntVal\ b\ ya))))$
⟨*proof*⟩

lemma *unfold-binary-width*:
assumes $op \in binary\text{-}normal$
shows $([m,p] \vdash BinaryExpr\ op\ xe\ ye \mapsto IntVal\ b\ val) = (\exists\ x\ y.$
$(([m,p] \vdash xe \mapsto IntVal\ b\ x) \wedge$
$([m,p] \vdash ye \mapsto IntVal\ b\ y) \wedge$
$(IntVal\ b\ val = bin\text{-}eval\ op\ (IntVal\ b\ x)\ (IntVal\ b\ y)) \wedge$
$(IntVal\ b\ val \neq UndefVal)$
)) (**is** *?L = ?R*)
⟨*proof*⟩

end

7 Tree to Graph

theory *TreeToGraph*
imports
Semantics.IRTreeEval
Graph.IRGraph
Snippets.Snipping
begin

7.1 Subgraph to Data-flow Tree

fun *find-node-and-stamp* :: *IRGraph* ⇒ (*IRNode* × *Stamp*) ⇒ *ID option* **where**
find-node-and-stamp g (*n*,*s*) =
find (λ*i*. *kind g i* = *n* ∧ *stamp g i* = *s*) (*sorted-list-of-set*(*ids g*))

export-code *find-node-and-stamp*

fun *is-preevaluated* :: *IRNode* ⇒ *bool* **where**
is-preevaluated (*InvokeNode n - - - - -*) = *True* |
is-preevaluated (*InvokeWithExceptionNode n - - - - - -*) = *True* |
is-preevaluated (*NewInstanceNode n - - -*) = *True* |
is-preevaluated (*LoadFieldNode n - - -*) = *True* |
is-preevaluated (*SignedDivNode n - - - - -*) = *True* |
is-preevaluated (*SignedRemNode n - - - - -*) = *True* |
is-preevaluated (*ValuePhiNode n - -*) = *True* |
is-preevaluated (*BytecodeExceptionNode n - -*) = *True* |
is-preevaluated (*NewArrayNode n - -*) = *True* |
is-preevaluated (*ArrayLengthNode n -*) = *True* |
is-preevaluated (*LoadIndexedNode n - - -*) = *True* |
is-preevaluated (*StoreIndexedNode n - - - - - -*) = *True* |
is-preevaluated - = *False*

inductive
rep :: *IRGraph* ⇒ *ID* ⇒ *IRExpr* ⇒ *bool* (- ⊢ - ≃ - 55)
for *g* **where**

ConstantNode:
$\llbracket$*kind g n* = *ConstantNode c*$\rrbracket$
$\Longrightarrow$ *g* ⊢ *n* ≃ (*ConstantExpr c*) |

ParameterNode:
$\llbracket$*kind g n* = *ParameterNode i*;
stamp g n = *s*$\rrbracket$
$\Longrightarrow$ *g* ⊢ *n* ≃ (*ParameterExpr i s*) |

ConditionalNode:
$\llbracket$*kind g n* = *ConditionalNode c t f*;
g ⊢ *c* ≃ *ce*;
g ⊢ *t* ≃ *te*;
g ⊢ *f* ≃ *fe*$\rrbracket$
$\Longrightarrow$ *g* ⊢ *n* ≃ (*ConditionalExpr ce te fe*) |

AbsNode:
$\llbracket$*kind g n* = *AbsNode x*;
g ⊢ *x* ≃ *xe*$\rrbracket$
$\Longrightarrow$ *g* ⊢ *n* ≃ (*UnaryExpr UnaryAbs xe*) |

ReverseBytesNode:
$\llbracket$*kind g n* = *ReverseBytesNode x*;
g ⊢ *x* ≃ *xe*$\rrbracket$
$\Longrightarrow$ *g* ⊢ *n* ≃ (*UnaryExpr UnaryReverseBytes xe*) |

BitCountNode:
$\llbracket$*kind g n* = *BitCountNode x*;
g ⊢ *x* ≃ *xe*$\rrbracket$
$\Longrightarrow$ *g* ⊢ *n* ≃ (*UnaryExpr UnaryBitCount xe*) |

NotNode:
$\llbracket$*kind g n* = *NotNode x*;
g ⊢ *x* ≃ *xe*$\rrbracket$
$\Longrightarrow$ *g* ⊢ *n* ≃ (*UnaryExpr UnaryNot xe*) |

NegateNode:
$\llbracket$*kind g n* = *NegateNode x*;
g ⊢ *x* ≃ *xe*$\rrbracket$
$\Longrightarrow$ *g* ⊢ *n* ≃ (*UnaryExpr UnaryNeg xe*) |

LogicNegationNode:
$\llbracket$*kind g n* = *LogicNegationNode x*;

$g \vdash x \simeq xe\rrbracket$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryLogicNegation\ xe)$ |

AddNode:
$\llbracket kind\ g\ n = AddNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinAdd\ xe\ ye)$ |

MulNode:
$\llbracket kind\ g\ n = MulNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinMul\ xe\ ye)$ |

DivNode:
$\llbracket kind\ g\ n = SignedFloatingIntegerDivNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinDiv\ xe\ ye)$ |

ModNode:
$\llbracket kind\ g\ n = SignedFloatingIntegerRemNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinMod\ xe\ ye)$ |

SubNode:
$\llbracket kind\ g\ n = SubNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinSub\ xe\ ye)$ |

AndNode:
$\llbracket kind\ g\ n = AndNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinAnd\ xe\ ye)$ |

OrNode:
$\llbracket kind\ g\ n = OrNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinOr\ xe\ ye)$ |

XorNode:
$\llbracket kind\ g\ n = XorNode\ x\ y;$
$g \vdash x \simeq xe;$

$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinXor\ xe\ ye)$ |

ShortCircuitOrNode:
$\llbracket kind\ g\ n = ShortCircuitOrNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinShortCircuitOr\ xe\ ye)$ |

LeftShiftNode:
$\llbracket kind\ g\ n = LeftShiftNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinLeftShift\ xe\ ye)$ |

RightShiftNode:
$\llbracket kind\ g\ n = RightShiftNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinRightShift\ xe\ ye)$ |

UnsignedRightShiftNode:
$\llbracket kind\ g\ n = UnsignedRightShiftNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinURightShift\ xe\ ye)$ |

IntegerBelowNode:
$\llbracket kind\ g\ n = IntegerBelowNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerBelow\ xe\ ye)$ |

IntegerEqualsNode:
$\llbracket kind\ g\ n = IntegerEqualsNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerEquals\ xe\ ye)$ |

IntegerLessThanNode:
$\llbracket kind\ g\ n = IntegerLessThanNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerLessThan\ xe\ ye)$ |

IntegerTestNode:
$\llbracket kind\ g\ n = IntegerTestNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye\rrbracket$

$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerTest\ xe\ ye)$ |

IntegerNormalizeCompareNode:
$[\![kind\ g\ n = IntegerNormalizeCompareNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerNormalizeCompare\ xe\ ye)$ |

IntegerMulHighNode:
$[\![kind\ g\ n = IntegerMulHighNode\ x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinIntegerMulHigh\ xe\ ye)$ |

NarrowNode:
$[\![kind\ g\ n = NarrowNode\ inputBits\ resultBits\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ (UnaryNarrow\ inputBits\ resultBits)\ xe)$ |

SignExtendNode:
$[\![kind\ g\ n = SignExtendNode\ inputBits\ resultBits\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ (UnarySignExtend\ inputBits\ resultBits)\ xe)$ |

ZeroExtendNode:
$[\![kind\ g\ n = ZeroExtendNode\ inputBits\ resultBits\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ (UnaryZeroExtend\ inputBits\ resultBits)\ xe)$ |

LeafNode:
$[\![is\text{-}preevaluated\ (kind\ g\ n);$
$stamp\ g\ n = s]\!]$
$\Longrightarrow g \vdash n \simeq (LeafExpr\ n\ s)$ |

PiNode:
$[\![kind\ g\ n = PiNode\ n'\ guard;$
$g \vdash n' \simeq e]\!]$
$\Longrightarrow g \vdash n \simeq e$ |

RefNode:
$[\![kind\ g\ n = RefNode\ n';$
$g \vdash n' \simeq e]\!]$
$\Longrightarrow g \vdash n \simeq e$ |

IsNullNode:
$[\![kind\ g\ n = IsNullNode\ v;$
$g \vdash v \simeq lfn]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryIsNull\ lfn)$

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$ *as exprE*) *rep* ⟨*proof*⟩

inductive
replist :: *IRGraph* ⇒ *ID list* ⇒ *IRExpr list* ⇒ *bool* (- ⊢ - [≃] - *55*)
for *g* **where**

RepNil:
$g \vdash [] [\simeq] []$ |

RepCons:
$[\![g \vdash x \simeq xe;$
$g \vdash xs\ [\simeq]\ xse]\!]$
$\Longrightarrow g \vdash x\#xs\ [\simeq]\ xe\#xse$

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

definition *wf-term-graph* :: *MapState* ⇒ *Params* ⇒ *IRGraph* ⇒ *ID* ⇒ *bool* **where**
wf-term-graph m p g n = $(\exists\ e.\ (g \vdash n \simeq e) \wedge (\exists\ v.\ ([m, p] \vdash e \mapsto v)))$

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

7.2 Data-flow Tree to Subgraph

fun *unary-node* :: *IRUnaryOp* ⇒ *ID* ⇒ *IRNode* **where**
unary-node UnaryAbs v = AbsNode v |
unary-node UnaryNot v = NotNode v |
unary-node UnaryNeg v = NegateNode v |
unary-node UnaryLogicNegation v = LogicNegationNode v |
unary-node (*UnaryNarrow ib rb*) *v = NarrowNode ib rb v* |
unary-node (*UnarySignExtend ib rb*) *v = SignExtendNode ib rb v* |
unary-node (*UnaryZeroExtend ib rb*) *v = ZeroExtendNode ib rb v* |
unary-node UnaryIsNull v = IsNullNode v |
unary-node UnaryReverseBytes v = ReverseBytesNode v |
unary-node UnaryBitCount v = BitCountNode v

fun *bin-node* :: *IRBinaryOp* ⇒ *ID* ⇒ *ID* ⇒ *IRNode* **where**
bin-node BinAdd x y = AddNode x y |
bin-node BinMul x y = MulNode x y |
bin-node BinDiv x y = SignedFloatingIntegerDivNode x y |
bin-node BinMod x y = SignedFloatingIntegerRemNode x y |
bin-node BinSub x y = SubNode x y |
bin-node BinAnd x y = AndNode x y |

bin-node BinOr x y = OrNode x y |
bin-node BinXor x y = XorNode x y |
bin-node BinShortCircuitOr x y = ShortCircuitOrNode x y |
bin-node BinLeftShift x y = LeftShiftNode x y |
bin-node BinRightShift x y = RightShiftNode x y |
bin-node BinURightShift x y = UnsignedRightShiftNode x y |
bin-node BinIntegerEquals x y = IntegerEqualsNode x y |
bin-node BinIntegerLessThan x y = IntegerLessThanNode x y |
bin-node BinIntegerBelow x y = IntegerBelowNode x y |
bin-node BinIntegerTest x y = IntegerTestNode x y |
bin-node BinIntegerNormalizeCompare x y = IntegerNormalizeCompareNode x y
|
bin-node BinIntegerMulHigh x y = IntegerMulHighNode x y

inductive *fresh-id* :: *IRGraph* ⇒ *ID* ⇒ *bool* **where**
n ∉ *ids g* ⟹ *fresh-id g n*

code-pred *fresh-id* ⟨*proof*⟩

fun *get-fresh-id* :: *IRGraph* ⇒ *ID* **where**

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

export-code *get-fresh-id*

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

inductive *unique* :: *IRGraph* ⇒ (*IRNode* × *Stamp*) ⇒ (*IRGraph* × *ID*) ⇒ *bool*
where
Exists:
⟦*find-node-and-stamp g node = Some n*⟧
⟹ *unique g node* (*g*, *n*) |
New:
⟦*find-node-and-stamp g node = None*;
n = get-fresh-id g;
g′ = add-node n node g⟧
⟹ *unique g node* (*g′*, *n*)

code-pred (*modes*: *i* ⇒ *i* ⇒ *o* ⇒ *bool as uniqueE*) *unique* ⟨*proof*⟩

inductive
unrep :: *IRGraph* ⇒ *IRExpr* ⇒ (*IRGraph* × *ID*) ⇒ *bool* (- ⊕ - ⇝ - *55*)
where

UnrepConstantNode:
⟦*unique g* (*ConstantNode c*, *constantAsStamp c*) (g_1, *n*)⟧

$\Longrightarrow g \oplus (ConstantExpr\ c) \rightsquigarrow (g_1,\ n)$ |

UnrepParameterNode:
$[\![unique\ g\ (ParameterNode\ i,\ s)\ (g_1,\ n)]\!]$
$\Longrightarrow g \oplus (ParameterExpr\ i\ s) \rightsquigarrow (g_1,\ n)$ |

UnrepConditionalNode:
$[\![g \oplus ce \rightsquigarrow (g_1,\ c);$
$g_1 \oplus te \rightsquigarrow (g_2,\ t);$
$g_2 \oplus fe \rightsquigarrow (g_3,\ f);$
$s' = meet\ (stamp\ g_3\ t)\ (stamp\ g_3\ f);$
$unique\ g_3\ (ConditionalNode\ c\ t\ f,\ s')\ (g_4,\ n)]\!]$
$\Longrightarrow g \oplus (ConditionalExpr\ ce\ te\ fe) \rightsquigarrow (g_4,\ n)$ |

UnrepUnaryNode:
$[\![g \oplus xe \rightsquigarrow (g_1,\ x);$
$s' = stamp\text{-}unary\ op\ (stamp\ g_1\ x);$
$unique\ g_1\ (unary\text{-}node\ op\ x,\ s')\ (g_2,\ n)]\!]$
$\Longrightarrow g \oplus (UnaryExpr\ op\ xe) \rightsquigarrow (g_2,\ n)$ |

UnrepBinaryNode:
$[\![g \oplus xe \rightsquigarrow (g_1,\ x);$
$g_1 \oplus ye \rightsquigarrow (g_2,\ y);$
$s' = stamp\text{-}binary\ op\ (stamp\ g_2\ x)\ (stamp\ g_2\ y);$
$unique\ g_2\ (bin\text{-}node\ op\ x\ y,\ s')\ (g_3,\ n)]\!]$
$\Longrightarrow g \oplus (BinaryExpr\ op\ xe\ ye) \rightsquigarrow (g_3,\ n)$ |

AllLeafNodes:
$[\![stamp\ g\ n = s;$
$is\text{-}preevaluated\ (kind\ g\ n)]\!]$
$\Longrightarrow g \oplus (LeafExpr\ n\ s) \rightsquigarrow (g,\ n)$

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$ *as unrepE*)
unrep ⟨*proof*⟩

uniqueRules

$$\frac{find\text{-}node\text{-}and\text{-}stamp\ (g{::}IRGraph)\ (node{::}IRNode \times Stamp) = Some\ (n{::}nat)}{unique\ g\ node\ (g,\ n)}$$

$$\frac{find\text{-}node\text{-}and\text{-}stamp\ (g{::}IRGraph)\ (node{::}IRNode \times Stamp) = None \quad (n{::}nat) = get\text{-}fresh\text{-}id\ g \quad (g'{::}IRGraph) = add\text{-}node\ n\ node\ g}{unique\ g\ node\ (g',\ n)}$$

unrepRules

$$\frac{\textit{unique}\ (g{::}\textit{IRGraph})\ (\textit{ConstantNode}\ (c{::}\textit{Value}),\ \textit{constantAsStamp}\ c)\ (g_1{::}\textit{IRGraph},\ n{::}\textit{nat})}{g \oplus \textit{ConstantExpr}\ c \rightsquigarrow (g_1,\ n)}$$

$$\frac{\textit{unique}\ (g{::}\textit{IRGraph})\ (\textit{ParameterNode}\ (i{::}\textit{nat}),\ s{::}\textit{Stamp})\ (g_1{::}\textit{IRGraph},\ n{::}\textit{nat})}{g \oplus \textit{ParameterExpr}\ i\ s \rightsquigarrow (g_1,\ n)}$$

$$\frac{\begin{array}{c} g{::}\textit{IRGraph} \oplus ce{::}\textit{IRExpr} \rightsquigarrow (g_1{::}\textit{IRGraph},\ c{::}\textit{nat}) \\ g_1 \oplus te{::}\textit{IRExpr} \rightsquigarrow (g_2{::}\textit{IRGraph},\ t{::}\textit{nat}) \\ g_2 \oplus fe{::}\textit{IRExpr} \rightsquigarrow (g_3{::}\textit{IRGraph},\ f{::}\textit{nat}) \\ (s'{::}\textit{Stamp}) = \textit{meet}\ (\textit{stamp}\ g_3\ t)\ (\textit{stamp}\ g_3\ f) \\ \textit{unique}\ g_3\ (\textit{ConditionalNode}\ c\ t\ f,\ s')\ (g_4{::}\textit{IRGraph},\ n{::}\textit{nat}) \end{array}}{g \oplus \textit{ConditionalExpr}\ ce\ te\ fe \rightsquigarrow (g_4,\ n)}$$

$$\frac{\begin{array}{c} g{::}\textit{IRGraph} \oplus xe{::}\textit{IRExpr} \rightsquigarrow (g_1{::}\textit{IRGraph},\ x{::}\textit{nat}) \\ g_1 \oplus ye{::}\textit{IRExpr} \rightsquigarrow (g_2{::}\textit{IRGraph},\ y{::}\textit{nat}) \\ (s'{::}\textit{Stamp}) = \textit{stamp-binary}\ (op{::}\textit{IRBinaryOp})\ (\textit{stamp}\ g_2\ x)\ (\textit{stamp}\ g_2\ y) \\ \textit{unique}\ g_2\ (\textit{bin-node}\ op\ x\ y,\ s')\ (g_3{::}\textit{IRGraph},\ n{::}\textit{nat}) \end{array}}{g \oplus \textit{BinaryExpr}\ op\ xe\ ye \rightsquigarrow (g_3,\ n)}$$

$$\frac{\begin{array}{c} g{::}\textit{IRGraph} \oplus xe{::}\textit{IRExpr} \rightsquigarrow (g_1{::}\textit{IRGraph},\ x{::}\textit{nat}) \\ (s'{::}\textit{Stamp}) = \textit{stamp-unary}\ (op{::}\textit{IRUnaryOp})\ (\textit{stamp}\ g_1\ x) \\ \textit{unique}\ g_1\ (\textit{unary-node}\ op\ x,\ s')\ (g_2{::}\textit{IRGraph},\ n{::}\textit{nat}) \end{array}}{g \oplus \textit{UnaryExpr}\ op\ xe \rightsquigarrow (g_2,\ n)}$$

$$\frac{\begin{array}{c} \textit{stamp}\ (g{::}\textit{IRGraph})\ (n{::}\textit{nat}) = (s{::}\textit{Stamp}) \\ \textit{is-preevaluated}\ (\textit{kind}\ g\ n) \end{array}}{g \oplus \textit{LeafExpr}\ n\ s \rightsquigarrow (g,\ n)}$$

7.3 Lift Data-flow Tree Semantics

inductive *encodeeval* :: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *ID* ⇒ *Value* ⇒ *bool*
([-,-,-] ⊢ - ↦ - *50*)
where
$(g \vdash n \simeq e) \land ([m,p] \vdash e \mapsto v) \Longrightarrow [g,\ m,\ p] \vdash n \mapsto v$

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

inductive *encodeEvalAll* :: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *ID list* ⇒ *Value list* ⇒ *bool*
([-,-,-] ⊢ - [↦] - *60*) **where**
$(g \vdash nids\ [\simeq]\ es) \land ([m,\ p] \vdash es\ [\mapsto]\ vs) \Longrightarrow ([g,\ m,\ p] \vdash nids\ [\mapsto]\ vs)$

code-pred ($modes$: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) $encodeEvalAll$ ⟨$proof$⟩

7.4 Graph Refinement

definition $graph\text{-}represents\text{-}expression$ :: $IRGraph \Rightarrow ID \Rightarrow IRExpr \Rightarrow bool$
  ($\text{-} \vdash \text{-} \trianglelefteq \text{-}$ 50)
  where
  $(g \vdash n \trianglelefteq e) = (\exists e' . (g \vdash n \simeq e') \wedge (e' \leq e))$

definition $graph\text{-}refinement$ :: $IRGraph \Rightarrow IRGraph \Rightarrow bool$ **where**
  $graph\text{-}refinement\ g_1\ g_2 =$
    $((ids\ g_1 \subseteq ids\ g_2) \wedge$
    $(\forall\ n . n \in ids\ g_1 \longrightarrow (\forall e. (g_1 \vdash n \simeq e) \longrightarrow (g_2 \vdash n \trianglelefteq e))))$

lemma $graph\text{-}refinement$:
  $graph\text{-}refinement\ g1\ g2 \Longrightarrow$
  $(\forall n\ m\ p\ v.\ n \in ids\ g1 \longrightarrow ([g1, m, p] \vdash n \mapsto v) \longrightarrow ([g2, m, p] \vdash n \mapsto v))$
  ⟨$proof$⟩

7.5 Maximal Sharing

definition $maximal\text{-}sharing$:
  $maximal\text{-}sharing\ g = (\forall\ n_1\ n_2 . n_1 \in true\text{-}ids\ g \wedge n_2 \in true\text{-}ids\ g \longrightarrow$
    $(\forall\ e. (g \vdash n_1 \simeq e) \wedge (g \vdash n_2 \simeq e) \wedge (stamp\ g\ n_1 = stamp\ g\ n_2) \longrightarrow n_1 = n_2))$

end

7.6 Formedness Properties

theory $Form$
imports
  $Semantics.TreeToGraph$
begin

definition $wf\text{-}start$ **where**
  $wf\text{-}start\ g = (0 \in ids\ g \wedge$
    $is\text{-}StartNode\ (kind\ g\ 0))$

definition $wf\text{-}closed$ **where**
  $wf\text{-}closed\ g =$
    $(\forall\ n \in ids\ g .$
      $inputs\ g\ n \subseteq ids\ g \wedge$
      $succ\ g\ n \subseteq ids\ g \wedge$
      $kind\ g\ n \neq NoNode)$

definition $wf\text{-}phis$ **where**
  $wf\text{-}phis\ g =$
    $(\forall\ n \in ids\ g.$

is-PhiNode (*kind g n*) ⟶
length (*ir-values* (*kind g n*))
= *length* (*ir-ends*
(*kind g* (*ir-merge* (*kind g n*)))))

definition *wf-ends* **where**
wf-ends g =
(∀ *n* ∈ *ids g* .
is-AbstractEndNode (*kind g n*) ⟶
card (*usages g n*) > *0*)

fun *wf-graph* :: *IRGraph* ⇒ *bool* **where**
wf-graph g = (*wf-start g* ∧ *wf-closed g* ∧ *wf-phis g* ∧ *wf-ends g*)

lemmas *wf-folds* =
wf-graph.simps
wf-start-def
wf-closed-def
wf-phis-def
wf-ends-def

fun *wf-stamps* :: *IRGraph* ⇒ *bool* **where**
wf-stamps g = (∀ *n* ∈ *ids g* .
(∀ *v m p e* . (*g* ⊢ *n* ≃ *e*) ∧ ([*m*, *p*] ⊢ *e* ↦ *v*) ⟶ *valid-value v* (*stamp-expr e*)))

fun *wf-stamp* :: *IRGraph* ⇒ (*ID* ⇒ *Stamp*) ⇒ *bool* **where**
wf-stamp g s = (∀ *n* ∈ *ids g* .
(∀ *v m p e* . (*g* ⊢ *n* ≃ *e*) ∧ ([*m*, *p*] ⊢ *e* ↦ *v*) ⟶ *valid-value v* (*s n*)))

lemma *wf-empty*: *wf-graph start-end-graph*
⟨*proof*⟩

lemma *wf-eg2-sq*: *wf-graph eg2-sq*
⟨*proof*⟩

fun *wf-logic-node-inputs* :: *IRGraph* ⇒ *ID* ⇒ *bool* **where**
wf-logic-node-inputs g n =
(∀ *inp* ∈ *set* (*inputs-of* (*kind g n*)) . (∀ *v m p* . ([*g*, *m*, *p*] ⊢ *inp* ↦ *v*) ⟶ *wf-bool v*))

fun *wf-values* :: *IRGraph* ⇒ *bool* **where**
wf-values g = (∀ *n* ∈ *ids g* .
(∀ *v m p* . ([*g*, *m*, *p*] ⊢ *n* ↦ *v*) ⟶
(*is-LogicNode* (*kind g n*) ⟶
wf-bool v ∧ *wf-logic-node-inputs g n*)))

end

7.7 Dynamic Frames

This theory defines two operators, 'unchanged' and 'changeonly', that are useful for specifying which nodes in an IRGraph can change. The dynamic framing idea originates from 'Dynamic Frames' in software verification, started by Ioannis T. Kassios in "Dynamic frames: Support for framing, dependencies and sharing without restrictions", In FM 2006.

theory *IRGraphFrames*
  imports
    Form
begin

fun *unchanged* :: *ID set* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
  unchanged ns g1 g2 = (∀ *n* . *n* ∈ *ns* ⟶
    (*n* ∈ *ids g1* ∧ *n* ∈ *ids g2* ∧ *kind g1 n* = *kind g2 n* ∧ *stamp g1 n* = *stamp g2 n*))

fun *changeonly* :: *ID set* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
  changeonly ns g1 g2 = (∀ *n* . *n* ∈ *ids g1* ∧ *n* ∉ *ns* ⟶
    (*n* ∈ *ids g1* ∧ *n* ∈ *ids g2* ∧ *kind g1 n* = *kind g2 n* ∧ *stamp g1 n* = *stamp g2 n*))

lemma *node-unchanged*:
  assumes *unchanged ns g1 g2*
  assumes *nid* ∈ *ns*
  shows *kind g1 nid* = *kind g2 nid*
  ⟨*proof*⟩

lemma *other-node-unchanged*:
  assumes *changeonly ns g1 g2*
  assumes *nid* ∈ *ids g1*
  assumes *nid* ∉ *ns*
  shows *kind g1 nid* = *kind g2 nid*
  ⟨*proof*⟩

Some notation for input nodes used

inductive *eval-uses*:: *IRGraph* ⇒ *ID* ⇒ *ID* ⇒ *bool*
  for *g* **where**

  use0: *nid* ∈ *ids g*
    ⟹ *eval-uses g nid nid* |

  use-inp: *nid'* ∈ *inputs g n*
    ⟹ *eval-uses g nid nid'* |

  use-trans: ⟦*eval-uses g nid nid'*;
    eval-uses g nid' nid''⟧
    ⟹ *eval-uses g nid nid''*

fun *eval-usages* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
eval-usages g nid = {*n* ∈ *ids g* . *eval-uses g nid n*}

lemma *eval-usages-self*:
assumes *nid* ∈ *ids g*
shows *nid* ∈ *eval-usages g nid*
⟨*proof*⟩

lemma *not-in-g-inputs*:
assumes *nid* ∉ *ids g*
shows *inputs g nid* = {}
⟨*proof*⟩

lemma *child-member*:
assumes *n* = *kind g nid*
assumes *n* ≠ *NoNode*
assumes *List.member* (*inputs-of n*) *child*
shows *child* ∈ *inputs g nid*
⟨*proof*⟩

lemma *child-member-in*:
assumes *nid* ∈ *ids g*
assumes *List.member* (*inputs-of* (*kind g nid*)) *child*
shows *child* ∈ *inputs g nid*
⟨*proof*⟩

lemma *inp-in-g*:
assumes *n* ∈ *inputs g nid*
shows *nid* ∈ *ids g*
⟨*proof*⟩

lemma *inp-in-g-wf*:
assumes *wf-graph g*
assumes *n* ∈ *inputs g nid*
shows *n* ∈ *ids g*
⟨*proof*⟩

lemma *kind-unchanged*:
assumes *nid* ∈ *ids g1*
assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
shows *kind g1 nid* = *kind g2 nid*
⟨*proof*⟩

lemma *stamp-unchanged*:
assumes *nid* ∈ *ids g1*
assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
shows *stamp g1 nid* = *stamp g2 nid*
⟨*proof*⟩

lemma *child-unchanged*:
assumes *child* ∈ *inputs g1 nid*
assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
shows *unchanged* (*eval-usages g1 child*) *g1 g2*
⟨*proof*⟩

lemma *eval-usages*:
assumes *us* = *eval-usages g nid*
assumes *nid′* ∈ *ids g*
shows *eval-uses g nid nid′* ⟷ *nid′* ∈ *us* (**is** *?P* ⟷ *?Q*)
⟨*proof*⟩

lemma *inputs-are-uses*:
assumes *nid′* ∈ *inputs g nid*
shows *eval-uses g nid nid′*
⟨*proof*⟩

lemma *inputs-are-usages*:
assumes *nid′* ∈ *inputs g nid*
assumes *nid′* ∈ *ids g*
shows *nid′* ∈ *eval-usages g nid*
⟨*proof*⟩

lemma *inputs-of-are-usages*:
assumes *List.member* (*inputs-of* (*kind g nid*)) *nid′*
assumes *nid′* ∈ *ids g*
shows *nid′* ∈ *eval-usages g nid*
⟨*proof*⟩

lemma *usage-includes-inputs*:
assumes *us* = *eval-usages g nid*
assumes *ls* = *inputs g nid*
assumes *ls* ⊆ *ids g*
shows *ls* ⊆ *us*
⟨*proof*⟩

lemma *elim-inp-set*:
assumes *k* = *kind g nid*
assumes *k* ≠ *NoNode*
assumes *child* ∈ *set* (*inputs-of k*)
shows *child* ∈ *inputs g nid*
⟨*proof*⟩

lemma *encode-in-ids*:
assumes *g* ⊢ *nid* ≃ *e*
shows *nid* ∈ *ids g*
⟨*proof*⟩

lemma *eval-in-ids*:
assumes $[g, m, p] \vdash nid \mapsto v$
shows $nid \in ids\ g$
⟨*proof*⟩

lemma *transitive-kind-same*:
assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
shows $\forall\ nid' \in (eval\text{-}usages\ g1\ nid)\ .\ kind\ g1\ nid' = kind\ g2\ nid'$
⟨*proof*⟩

theorem *stay-same-encoding*:
assumes *nc*: *unchanged* (*eval-usages g1 nid*) *g1 g2*
assumes *g1*: $g1 \vdash nid \simeq e$
assumes *wf*: *wf-graph g1*
shows $g2 \vdash nid \simeq e$
⟨*proof*⟩

theorem *stay-same*:
assumes *nc*: *unchanged* (*eval-usages g1 nid*) *g1 g2*
assumes *g1*: $[g1, m, p] \vdash nid \mapsto v1$
assumes *wf*: *wf-graph g1*
shows $[g2, m, p] \vdash nid \mapsto v1$
⟨*proof*⟩

lemma *add-changed*:
assumes *gup* = *add-node new k g*
shows *changeonly* {*new*} *g gup*
⟨*proof*⟩

lemma *disjoint-change*:
assumes *changeonly change g gup*
assumes *nochange* = *ids g* − *change*
shows *unchanged nochange g gup*
⟨*proof*⟩

lemma *add-node-unchanged*:
assumes $new \notin ids\ g$
assumes $nid \in ids\ g$
assumes *gup* = *add-node new k g*
assumes *wf-graph g*
shows *unchanged* (*eval-usages g nid*) *g gup*
⟨*proof*⟩

lemma *eval-uses-imp*:
$((nid' \in ids\ g \land nid = nid')$
$\lor\ nid' \in inputs\ g\ nid$
$\lor\ (\exists\, nid''\ .\ eval\text{-}uses\ g\ nid\ nid'' \land eval\text{-}uses\ g\ nid''\ nid'))$
$\longleftrightarrow eval\text{-}uses\ g\ nid\ nid'$

⟨*proof*⟩

lemma *wf-use-ids*:
assumes *wf-graph g*
assumes $nid \in ids\ g$
assumes *eval-uses g nid nid′*
shows $nid' \in ids\ g$
⟨*proof*⟩

lemma *no-external-use*:
assumes *wf-graph g*
assumes $nid' \notin ids\ g$
assumes $nid \in ids\ g$
shows $\neg(eval\text{-}uses\ g\ nid\ nid')$
⟨*proof*⟩

end

7.8 Tree to Graph Theorems

theory *TreeToGraphThms*
imports
IRTreeEvalThms
IRGraphFrames
HOL−Eisbach.Eisbach
HOL−Eisbach.Eisbach-Tools
begin

7.8.1 Extraction and Evaluation of Expression Trees is Deterministic.

First, we prove some extra rules that relate each type of IRNode to the corresponding IRExpr type that 'rep' will produce. These are very helpful for proving that 'rep' is deterministic.

named-theorems *rep*

lemma *rep-constant* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = ConstantNode\ c \Longrightarrow$
$e = ConstantExpr\ c$
⟨*proof*⟩

lemma *rep-parameter* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = ParameterNode\ i \Longrightarrow$
$(\exists\, s.\ e = ParameterExpr\ i\ s)$
⟨*proof*⟩

lemma *rep-conditional* [*rep*]:

$g \vdash n \simeq e \Longrightarrow$
kind g n = *ConditionalNode c t f* $\Longrightarrow$
($\exists$ *ce te fe*. *e* = *ConditionalExpr ce te fe*)
⟨*proof*⟩

lemma *rep-abs* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *AbsNode x* $\Longrightarrow$
($\exists$ *xe*. *e* = *UnaryExpr UnaryAbs xe*)
⟨*proof*⟩

lemma *rep-reverse-bytes* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *ReverseBytesNode x* $\Longrightarrow$
($\exists$ *xe*. *e* = *UnaryExpr UnaryReverseBytes xe*)
⟨*proof*⟩

lemma *rep-bit-count* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *BitCountNode x* $\Longrightarrow$
($\exists$ *xe*. *e* = *UnaryExpr UnaryBitCount xe*)
⟨*proof*⟩

lemma *rep-not* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *NotNode x* $\Longrightarrow$
($\exists$ *xe*. *e* = *UnaryExpr UnaryNot xe*)
⟨*proof*⟩

lemma *rep-negate* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *NegateNode x* $\Longrightarrow$
($\exists$ *xe*. *e* = *UnaryExpr UnaryNeg xe*)
⟨*proof*⟩

lemma *rep-logicnegation* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *LogicNegationNode x* $\Longrightarrow$
($\exists$ *xe*. *e* = *UnaryExpr UnaryLogicNegation xe*)
⟨*proof*⟩

lemma *rep-add* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *AddNode x y* $\Longrightarrow$
($\exists$ *xe ye*. *e* = *BinaryExpr BinAdd xe ye*)
⟨*proof*⟩

lemma *rep-sub* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$

kind g n = SubNode x y $\Longrightarrow$
($\exists$ *xe ye. e = BinaryExpr BinSub xe ye*)
⟨*proof*⟩

lemma *rep-mul* [*rep*]:
g ⊢ *n* ≃ *e* $\Longrightarrow$
kind g n = MulNode x y $\Longrightarrow$
($\exists$ *xe ye. e = BinaryExpr BinMul xe ye*)
⟨*proof*⟩

lemma *rep-div* [*rep*]:
g ⊢ *n* ≃ *e* $\Longrightarrow$
kind g n = SignedFloatingIntegerDivNode x y $\Longrightarrow$
($\exists$ *xe ye. e = BinaryExpr BinDiv xe ye*)
⟨*proof*⟩

lemma *rep-mod* [*rep*]:
g ⊢ *n* ≃ *e* $\Longrightarrow$
kind g n = SignedFloatingIntegerRemNode x y $\Longrightarrow$
($\exists$ *xe ye. e = BinaryExpr BinMod xe ye*)
⟨*proof*⟩

lemma *rep-and* [*rep*]:
g ⊢ *n* ≃ *e* $\Longrightarrow$
kind g n = AndNode x y $\Longrightarrow$
($\exists$ *xe ye. e = BinaryExpr BinAnd xe ye*)
⟨*proof*⟩

lemma *rep-or* [*rep*]:
g ⊢ *n* ≃ *e* $\Longrightarrow$
kind g n = OrNode x y $\Longrightarrow$
($\exists$ *xe ye. e = BinaryExpr BinOr xe ye*)
⟨*proof*⟩

lemma *rep-xor* [*rep*]:
g ⊢ *n* ≃ *e* $\Longrightarrow$
kind g n = XorNode x y $\Longrightarrow$
($\exists$ *xe ye. e = BinaryExpr BinXor xe ye*)
⟨*proof*⟩

lemma *rep-short-circuit-or* [*rep*]:
g ⊢ *n* ≃ *e* $\Longrightarrow$
kind g n = ShortCircuitOrNode x y $\Longrightarrow$
($\exists$ *xe ye. e = BinaryExpr BinShortCircuitOr xe ye*)
⟨*proof*⟩

lemma *rep-left-shift* [*rep*]:
g ⊢ *n* ≃ *e* $\Longrightarrow$
kind g n = LeftShiftNode x y $\Longrightarrow$

$(\exists\, xe\ ye.\ e = BinaryExpr\ BinLeftShift\ xe\ ye)$
⟨proof⟩

lemma *rep-right-shift* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = RightShiftNode\ x\ y \Longrightarrow$
$(\exists\, xe\ ye.\ e = BinaryExpr\ BinRightShift\ xe\ ye)$
⟨proof⟩

lemma *rep-unsigned-right-shift* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = UnsignedRightShiftNode\ x\ y \Longrightarrow$
$(\exists\, xe\ ye.\ e = BinaryExpr\ BinURightShift\ xe\ ye)$
⟨proof⟩

lemma *rep-integer-below* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = IntegerBelowNode\ x\ y \Longrightarrow$
$(\exists\, xe\ ye.\ e = BinaryExpr\ BinIntegerBelow\ xe\ ye)$
⟨proof⟩

lemma *rep-integer-equals* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = IntegerEqualsNode\ x\ y \Longrightarrow$
$(\exists\, xe\ ye.\ e = BinaryExpr\ BinIntegerEquals\ xe\ ye)$
⟨proof⟩

lemma *rep-integer-less-than* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = IntegerLessThanNode\ x\ y \Longrightarrow$
$(\exists\, xe\ ye.\ e = BinaryExpr\ BinIntegerLessThan\ xe\ ye)$
⟨proof⟩

lemma *rep-integer-mul-high* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = IntegerMulHighNode\ x\ y \Longrightarrow$
$(\exists\, xe\ ye.\ e = BinaryExpr\ BinIntegerMulHigh\ xe\ ye)$
⟨proof⟩

lemma *rep-integer-test* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = IntegerTestNode\ x\ y \Longrightarrow$
$(\exists\, xe\ ye.\ e = BinaryExpr\ BinIntegerTest\ xe\ ye)$
⟨proof⟩

lemma *rep-integer-normalize-compare* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = IntegerNormalizeCompareNode\ x\ y \Longrightarrow$
$(\exists\, xe\ ye.\ e = BinaryExpr\ BinIntegerNormalizeCompare\ xe\ ye)$

$\langle proof \rangle$

lemma *rep-narrow* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = NarrowNode\ ib\ rb\ x \Longrightarrow$
$(\exists x.\ e = UnaryExpr\ (UnaryNarrow\ ib\ rb)\ x)$
$\langle proof \rangle$

lemma *rep-sign-extend* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = SignExtendNode\ ib\ rb\ x \Longrightarrow$
$(\exists x.\ e = UnaryExpr\ (UnarySignExtend\ ib\ rb)\ x)$
$\langle proof \rangle$

lemma *rep-zero-extend* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = ZeroExtendNode\ ib\ rb\ x \Longrightarrow$
$(\exists x.\ e = UnaryExpr\ (UnaryZeroExtend\ ib\ rb)\ x)$
$\langle proof \rangle$

lemma *rep-load-field* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$is\text{-}preevaluated\ (kind\ g\ n) \Longrightarrow$
$(\exists s.\ e = LeafExpr\ n\ s)$
$\langle proof \rangle$

lemma *rep-bytecode-exception* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$(kind\ g\ n) = BytecodeExceptionNode\ gu\ st\ n' \Longrightarrow$
$(\exists s.\ e = LeafExpr\ n\ s)$
$\langle proof \rangle$

lemma *rep-new-array* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$(kind\ g\ n) = NewArrayNode\ len\ st\ n' \Longrightarrow$
$(\exists s.\ e = LeafExpr\ n\ s)$
$\langle proof \rangle$

lemma *rep-array-length* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$(kind\ g\ n) = ArrayLengthNode\ x\ n' \Longrightarrow$
$(\exists s.\ e = LeafExpr\ n\ s)$
$\langle proof \rangle$

lemma *rep-load-index* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$(kind\ g\ n) = LoadIndexedNode\ index\ guard\ x\ n' \Longrightarrow$
$(\exists s.\ e = LeafExpr\ n\ s)$
$\langle proof \rangle$

lemma *rep-store-index* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$(kind\ g\ n) = StoreIndexedNode\ check\ val\ st\ index\ guard\ x\ n' \Longrightarrow$
$(\exists\, s.\ e = LeafExpr\ n\ s)$
⟨*proof*⟩

lemma *rep-ref* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = RefNode\ n' \Longrightarrow$
$g \vdash n' \simeq e$
⟨*proof*⟩

lemma *rep-pi* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = PiNode\ n'\ gu \Longrightarrow$
$g \vdash n' \simeq e$
⟨*proof*⟩

lemma *rep-is-null* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = IsNullNode\ x \Longrightarrow$
$(\exists\, xe.\ e = (UnaryExpr\ UnaryIsNull\ xe))$
⟨*proof*⟩

method *solve-det* **uses** *node* =
(*match node* **in** *kind - - = node -* **for** *node* ⇒
‹*match rep in r*: - ⟹ - = *node* - ⟹ - ⇒
‹*match IRNode.inject in i*: (*node* - = *node* -) = - ⇒
‹*match RepE in e*: - ⟹ (⋀*x*. - = *node x* ⟹ -) ⟹ - ⇒
‹*match IRNode.distinct in d*: *node* - ≠ *RefNode* - ⇒
‹*match IRNode.distinct in f*: *node* - ≠ *PiNode* - - ⇒
‹*metis i e r d f*›››››› |
match node **in** *kind - - = node - -* **for** *node* ⇒
‹*match rep in r*: - ⟹ - = *node* - - ⟹ - ⇒
‹*match IRNode.inject in i*: (*node* - - = *node* - -) = - ⇒
‹*match RepE in e*: - ⟹ (⋀*x y*. - = *node x y* ⟹ -) ⟹ - ⇒
‹*match IRNode.distinct in d*: *node* - - ≠ *RefNode* - ⇒
‹*match IRNode.distinct in f*: *node* - - ≠ *PiNode* - - ⇒
‹*metis i e r d f*›››››› |
match node **in** *kind - - = node - - -* **for** *node* ⇒
‹*match rep in r*: - ⟹ - = *node* - - - ⟹ - ⇒
‹*match IRNode.inject in i*: (*node* - - - = *node* - - -) = - ⇒
‹*match RepE in e*: - ⟹ (⋀*x y z*. - = *node x y z* ⟹ -) ⟹ - ⇒
‹*match IRNode.distinct in d*: *node* - - - ≠ *RefNode* - ⇒
‹*match IRNode.distinct in f*: *node* - - - ≠ *PiNode* - - ⇒
‹*metis i e r d f*›››››› |
match node **in** *kind - - = node - - -* **for** *node* ⇒
‹*match rep in r*: - ⟹ - = *node* - - - ⟹ - ⇒

‹*match IRNode.inject in i*: (*node* - - - = *node* - - -) = - ⇒
‹*match RepE in e*: - ⟹ (⋀*x*. - = *node* - - *x* ⟹ -) ⟹ - ⇒
‹*match IRNode.distinct in d*: *node* - - - ≠ *RefNode* - ⇒
‹*match IRNode.distinct in f*: *node* - - - ≠ *PiNode* - - ⇒
‹*metis i e r d f*›››››)

Now we can prove that 'rep' and 'eval', and their list versions, are deterministic.

lemma *repDet*:
shows $(g \vdash n \simeq e_1) \Longrightarrow (g \vdash n \simeq e_2) \Longrightarrow e_1 = e_2$
⟨*proof*⟩

lemma *repAllDet*:
$g \vdash xs\ [\simeq]\ e1 \Longrightarrow$
$g \vdash xs\ [\simeq]\ e2 \Longrightarrow$
$e1 = e2$
⟨*proof*⟩

lemma *encodeEvalDet*:
$[g,m,p] \vdash e \mapsto v1 \Longrightarrow$
$[g,m,p] \vdash e \mapsto v2 \Longrightarrow$
$v1 = v2$
⟨*proof*⟩

lemma *graphDet*: $([g,m,p] \vdash n \mapsto v_1) \land ([g,m,p] \vdash n \mapsto v_2) \Longrightarrow v_1 = v_2$
⟨*proof*⟩

lemma *encodeEvalAllDet*:
$[g,\ m,\ p] \vdash nids\ [\mapsto]\ vs \Longrightarrow [g,\ m,\ p] \vdash nids\ [\mapsto]\ vs' \Longrightarrow vs = vs'$
⟨*proof*⟩

7.8.2 Monotonicity of Graph Refinement

Lift refinement monotonicity to graph level. Hopefully these shouldn't really be required.

lemma *mono-abs*:
assumes $kind\ g1\ n = AbsNode\ x \land kind\ g2\ n = AbsNode\ x$
assumes $(g1 \vdash x \simeq xe1) \land (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \land (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-not*:
assumes $kind\ g1\ n = NotNode\ x \land kind\ g2\ n = NotNode\ x$
assumes $(g1 \vdash x \simeq xe1) \land (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \land (g2 \vdash n \simeq e2)$

shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-negate*:
assumes $kind\ g1\ n = NegateNode\ x \wedge kind\ g2\ n = NegateNode\ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-logic-negation*:
assumes $kind\ g1\ n = LogicNegationNode\ x \wedge kind\ g2\ n = LogicNegationNode\ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-narrow*:
assumes $kind\ g1\ n = NarrowNode\ ib\ rb\ x \wedge kind\ g2\ n = NarrowNode\ ib\ rb\ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-sign-extend*:
assumes $kind\ g1\ n = SignExtendNode\ ib\ rb\ x \wedge kind\ g2\ n = SignExtendNode\ ib\ rb\ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-zero-extend*:
assumes $kind\ g1\ n = ZeroExtendNode\ ib\ rb\ x \wedge kind\ g2\ n = ZeroExtendNode\ ib\ rb\ x$
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $xe1 \geq xe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-conditional-graph*:
assumes $kind\ g1\ n = ConditionalNode\ c\ t\ f \wedge kind\ g2\ n = ConditionalNode\ c\ t\ f$
assumes $(g1 \vdash c \simeq ce1) \wedge (g2 \vdash c \simeq ce2)$
assumes $(g1 \vdash t \simeq te1) \wedge (g2 \vdash t \simeq te2)$

assumes $(g1 \vdash f \simeq fe1) \wedge (g2 \vdash f \simeq fe2)$
assumes $ce1 \geq ce2 \wedge te1 \geq te2 \wedge fe1 \geq fe2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-add*:
assumes *kind g1 n = AddNode x y* ∧ *kind g2 n = AddNode x y*
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $(g1 \vdash y \simeq ye1) \wedge (g2 \vdash y \simeq ye2)$
assumes $xe1 \geq xe2 \wedge ye1 \geq ye2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-mul*:
assumes *kind g1 n = MulNode x y* ∧ *kind g2 n = MulNode x y*
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $(g1 \vdash y \simeq ye1) \wedge (g2 \vdash y \simeq ye2)$
assumes $xe1 \geq xe2 \wedge ye1 \geq ye2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-div*:
assumes *kind g1 n = SignedFloatingIntegerDivNode x y* ∧ *kind g2 n = SignedFloatingIntegerDivNode x y*
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $(g1 \vdash y \simeq ye1) \wedge (g2 \vdash y \simeq ye2)$
assumes $xe1 \geq xe2 \wedge ye1 \geq ye2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *mono-mod*:
assumes *kind g1 n = SignedFloatingIntegerRemNode x y* ∧ *kind g2 n = SignedFloatingIntegerRemNode x y*
assumes $(g1 \vdash x \simeq xe1) \wedge (g2 \vdash x \simeq xe2)$
assumes $(g1 \vdash y \simeq ye1) \wedge (g2 \vdash y \simeq ye2)$
assumes $xe1 \geq xe2 \wedge ye1 \geq ye2$
assumes $(g1 \vdash n \simeq e1) \wedge (g2 \vdash n \simeq e2)$
shows $e1 \geq e2$
⟨*proof*⟩

lemma *term-graph-evaluation*:
$(g \vdash n \trianglelefteq e) \Longrightarrow (\forall\ m\ p\ v\ .\ ([m,p] \vdash e \mapsto v) \longrightarrow ([g,m,p] \vdash n \mapsto v))$
⟨*proof*⟩

lemma *encodes-contains*:

$g \vdash n \simeq e \Longrightarrow$
kind g n $\neq$ *NoNode*
⟨*proof*⟩

lemma *no-encoding*:
assumes $n \notin$ *ids g*
shows $\neg(g \vdash n \simeq e)$
⟨*proof*⟩

lemma *not-excluded-keep-type*:
assumes $n \in$ *ids g1*
assumes $n \notin$ *excluded*
assumes (*excluded* $\trianglelefteq$ *as-set g1*) $\subseteq$ *as-set g2*
shows *kind g1 n* = *kind g2 n* $\wedge$ *stamp g1 n* = *stamp g2 n*
⟨*proof*⟩

method *metis-node-eq-unary* **for** *node* :: $'a \Rightarrow$ *IRNode* =
(*match IRNode.inject* **in** *i*: (*node* - = *node* -) = - $\Rightarrow$
‹*metis i*›)
method *metis-node-eq-binary* **for** *node* :: $'a \Rightarrow 'a \Rightarrow$ *IRNode* =
(*match IRNode.inject* **in** *i*: (*node* - - = *node* - -) = - $\Rightarrow$
‹*metis i*›)
method *metis-node-eq-ternary* **for** *node* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow$ *IRNode* =
(*match IRNode.inject* **in** *i*: (*node* - - - = *node* - - -) = - $\Rightarrow$
‹*metis i*›)

7.8.3 Lift Data-flow Tree Refinement to Graph Refinement

theorem *graph-semantics-preservation*:
assumes *a*: $e1' \geq e2'$
assumes *b*: $(\{n'\} \trianglelefteq$ *as-set g1*) $\subseteq$ *as-set g2*
assumes *c*: $g1 \vdash n' \simeq e1'$
assumes *d*: $g2 \vdash n' \simeq e2'$
shows *graph-refinement g1 g2*
⟨*proof*⟩

lemma *graph-semantics-preservation-subscript*:
assumes *a*: ${e_1}' \geq {e_2}'$
assumes *b*: $(\{n\} \trianglelefteq$ *as-set* $g_1) \subseteq$ *as-set* g_2
assumes *c*: $g_1 \vdash n \simeq {e_1}'$
assumes *d*: $g_2 \vdash n \simeq {e_2}'$
shows *graph-refinement* g_1 g_2
⟨*proof*⟩

lemma *tree-to-graph-rewriting*:
$e_1 \geq e_2$
$\wedge$ $(g_1 \vdash n \simeq e_1)$ $\wedge$ *maximal-sharing* g_1
$\wedge$ $(\{n\} \trianglelefteq$ *as-set* $g_1) \subseteq$ *as-set* g_2
$\wedge$ $(g_2 \vdash n \simeq e_2)$ $\wedge$ *maximal-sharing* g_2

$\Longrightarrow$ *graph-refinement* g_1 g_2
⟨*proof*⟩

declare [[*simp-trace*]]
lemma *equal-refines*:
fixes *e1 e2* :: *IRExpr*
assumes *e1* = *e2*
shows *e1* ≥ *e2*
⟨*proof*⟩
declare [[*simp-trace=false*]]

lemma *eval-contains-id*[*simp*]: *g1* ⊢ *n* ≃ *e* ⟹ *n* ∈ *ids g1*
⟨*proof*⟩

lemma *subset-kind*[*simp*]: *as-set g1* ⊆ *as-set g2* ⟹ *g1* ⊢ *n* ≃ *e* ⟹ *kind g1 n* = *kind g2 n*
⟨*proof*⟩

lemma *subset-stamp*[*simp*]: *as-set g1* ⊆ *as-set g2* ⟹ *g1* ⊢ *n* ≃ *e* ⟹ *stamp g1 n* = *stamp g2 n*
⟨*proof*⟩

method *solve-subset-eval* **uses** *as-set eval* =
(*metis eval as-set subset-kind subset-stamp* |
metis eval as-set subset-kind)

lemma *subset-implies-evals*:
assumes *as-set g1* ⊆ *as-set g2*
assumes (*g1* ⊢ *n* ≃ *e*)
shows (*g2* ⊢ *n* ≃ *e*)
⟨*proof*⟩

lemma *subset-refines*:
assumes *as-set g1* ⊆ *as-set g2*
shows *graph-refinement g1 g2*
⟨*proof*⟩

lemma *graph-construction*:
$e_1 \geq e_2$
$\wedge$ *as-set* $g_1 \subseteq$ *as-set* g_2
$\wedge$ ($g_2 \vdash n \simeq e_2$)
$\Longrightarrow$ ($g_2 \vdash n \trianglelefteq e_1$) $\wedge$ *graph-refinement* g_1 g_2
⟨*proof*⟩

7.8.4 Term Graph Reconstruction

lemma *find-exists-kind*:

assumes *find-node-and-stamp g* (*node*, *s*) = *Some nid*
shows *kind g nid* = *node*
⟨*proof*⟩

lemma *find-exists-stamp*:
assumes *find-node-and-stamp g* (*node*, *s*) = *Some nid*
shows *stamp g nid* = *s*
⟨*proof*⟩

lemma *find-new-kind*:
assumes g' = *add-node nid* (*node*, *s*) *g*
assumes *node* ≠ *NoNode*
shows *kind* g' *nid* = *node*
⟨*proof*⟩

lemma *find-new-stamp*:
assumes g' = *add-node nid* (*node*, *s*) *g*
assumes *node* ≠ *NoNode*
shows *stamp* g' *nid* = *s*
⟨*proof*⟩

lemma *sorted-bottom*:
assumes *finite xs*
assumes $x \in xs$
shows $x \leq$ *last*(*sorted-list-of-set*(*xs*::*nat set*))
⟨*proof*⟩

lemma *fresh*: *finite xs* ⟹ *last*(*sorted-list-of-set*(*xs*::*nat set*)) + *1* ∉ *xs*
⟨*proof*⟩

lemma *fresh-ids*:
assumes *n* = *get-fresh-id g*
shows *n* ∉ *ids g*
⟨*proof*⟩

lemma *graph-unchanged-rep-unchanged*:
assumes ∀ *n* ∈ *ids g*. *kind g n* = *kind* g' *n*
assumes ∀ *n* ∈ *ids g*. *stamp g n* = *stamp* g' *n*
shows ($g \vdash n \simeq e$) ⟶ ($g' \vdash n \simeq e$)
⟨*proof*⟩

lemma *fresh-node-subset*:
assumes *n* ∉ *ids g*
assumes g' = *add-node n* (*k*, *s*) *g*
shows *as-set g* ⊆ *as-set* g'
⟨*proof*⟩

lemma *unique-subset*:
assumes *unique g node* (g', *n*)

shows *as-set* $g \subseteq$ *as-set* g'
⟨*proof*⟩

lemma *unrep-subset*:
assumes $(g \oplus e \rightsquigarrow (g', n))$
shows *as-set* $g \subseteq$ *as-set* g'
⟨*proof*⟩

lemma *fresh-node-preserves-other-nodes*:
assumes $n' =$ *get-fresh-id* g
assumes $g' =$ *add-node* n' (k, s) g
shows $\forall\ n \in$ *ids* g . $(g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$
⟨*proof*⟩

lemma *found-node-preserves-other-nodes*:
assumes *find-node-and-stamp* g $(k, s) =$ *Some* n
shows $\forall\ n \in$ *ids* g. $(g \vdash n \simeq e) \longleftrightarrow (g \vdash n \simeq e)$
⟨*proof*⟩

lemma *unrep-ids-subset*[*simp*]:
assumes $g \oplus e \rightsquigarrow (g', n)$
shows *ids* $g \subseteq$ *ids* g'
⟨*proof*⟩

lemma *unrep-unchanged*:
assumes $g \oplus e \rightsquigarrow (g', n)$
shows $\forall\ n \in$ *ids* g . $\forall\ e.$ $(g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$
⟨*proof*⟩

lemma *unique-kind*:
assumes *unique* g (*node*, s) $(g'$, *nid*)
assumes *node* $\neq$ *NoNode*
shows *kind* g' *nid* = *node* $\wedge$ *stamp* g' *nid* = s
⟨*proof*⟩

lemma *unique-eval*:
assumes *unique* g (n, s) $(g'$, *nid*)
shows $g \vdash nid' \simeq e \Longrightarrow g' \vdash nid' \simeq e$
⟨*proof*⟩

lemma *unrep-eval*:
assumes *unrep* g e $(g'$, *nid*)
shows $g \vdash nid' \simeq e' \Longrightarrow g' \vdash nid' \simeq e'$
⟨*proof*⟩

lemma *unary-node-nonode*:
unary-node op $x \neq$ *NoNode*
⟨*proof*⟩

lemma *bin-node-nonode*:
$bin\text{-}node\ op\ x\ y \neq NoNode$
⟨*proof*⟩

theorem *term-graph-reconstruction*:
$g \oplus e \rightsquigarrow (g', n) \Longrightarrow (g' \vdash n \simeq e) \land as\text{-}set\ g \subseteq as\text{-}set\ g'$
⟨*proof*⟩

lemma *ref-refinement*:
assumes $g \vdash n \simeq e_1$
assumes $kind\ g\ n' = RefNode\ n$
shows $g \vdash n' \trianglelefteq e_1$
⟨*proof*⟩

lemma *unrep-refines*:
assumes $g \oplus e \rightsquigarrow (g', n)$
shows $graph\text{-}refinement\ g\ g'$
⟨*proof*⟩

lemma *add-new-node-refines*:
assumes $n \notin ids\ g$
assumes $g' = add\text{-}node\ n\ (k, s)\ g$
shows $graph\text{-}refinement\ g\ g'$
⟨*proof*⟩

lemma *add-node-as-set*:
assumes $g' = add\text{-}node\ n\ (k, s)\ g$
shows $(\{n\} \trianglelefteq as\text{-}set\ g) \subseteq as\text{-}set\ g'$
⟨*proof*⟩

theorem *refined-insert*:
assumes $e_1 \geq e_2$
assumes $g_1 \oplus e_2 \rightsquigarrow (g_2, n')$
shows $(g_2 \vdash n' \trianglelefteq e_1) \land graph\text{-}refinement\ g_1\ g_2$
⟨*proof*⟩

lemma *ids-finite*: $finite\ (ids\ g)$
⟨*proof*⟩

lemma *unwrap-sorted*: $set\ (sorted\text{-}list\text{-}of\text{-}set\ (ids\ g)) = ids\ g$
⟨*proof*⟩

lemma *find-none*:
assumes $find\text{-}node\text{-}and\text{-}stamp\ g\ (k, s) = None$
shows $\forall\ n \in ids\ g.\ kind\ g\ n \neq k \lor stamp\ g\ n \neq s$
⟨*proof*⟩

method *ref-represents* **uses** *node* =
(*metis IRNode.distinct(2755) RefNode dual-order.refl find-new-kind fresh-node-subset node subset-implies-evals*)

7.8.5 Data-flow Tree to Subgraph Preserves Maximal Sharing

lemma *same-kind-stamp-encodes-equal*:
assumes $kind\ g\ n = kind\ g\ n'$
assumes $stamp\ g\ n = stamp\ g\ n'$
assumes $\neg(is\text{-}preevaluated\ (kind\ g\ n))$
shows $\forall\ e.\ (g \vdash n \simeq e) \longrightarrow (g \vdash n' \simeq e)$
⟨*proof*⟩

lemma *new-node-not-present*:
assumes $find\text{-}node\text{-}and\text{-}stamp\ g\ (node,\ s) = None$
assumes $n = get\text{-}fresh\text{-}id\ g$
assumes $g' = add\text{-}node\ n\ (node,\ s)\ g$
shows $\forall\ n' \in true\text{-}ids\ g.\ (\forall e.\ ((g \vdash n \simeq e) \wedge (g \vdash n' \simeq e)) \longrightarrow n = n')$
⟨*proof*⟩

lemma *true-ids-def*:
$true\text{-}ids\ g = \{n \in ids\ g.\ \neg(is\text{-}RefNode\ (kind\ g\ n)) \wedge ((kind\ g\ n) \neq NoNode)\}$
⟨*proof*⟩

lemma *add-node-some-node-def*:
assumes $k \neq NoNode$
assumes $g' = add\text{-}node\ nid\ (k,\ s)\ g$
shows $g' = Abs\text{-}IRGraph\ ((Rep\text{-}IRGraph\ g)(nid \mapsto (k,\ s)))$
⟨*proof*⟩

lemma *ids-add-update-v1*:
assumes $g' = add\text{-}node\ nid\ (k,\ s)\ g$
assumes $k \neq NoNode$
shows $dom\ (Rep\text{-}IRGraph\ g') = dom\ (Rep\text{-}IRGraph\ g) \cup \{nid\}$
⟨*proof*⟩

lemma *ids-add-update-v2*:
assumes $g' = add\text{-}node\ nid\ (k,\ s)\ g$
assumes $k \neq NoNode$

shows $nid \in ids\ g'$
⟨*proof*⟩

lemma *add-node-ids-subset*:
assumes $n \in ids\ g$
assumes $g' = add\text{-}node\ n\ node\ g$
shows $ids\ g' = ids\ g \cup \{n\}$
⟨*proof*⟩

lemma *convert-maximal*:
assumes $\forall\, n\ n'.\ n \in true\text{-}ids\ g \wedge n' \in true\text{-}ids\ g \longrightarrow$
$(\forall\, e\ e'.\ (g \vdash n \simeq e) \wedge (g \vdash n' \simeq e') \longrightarrow e \neq e')$
shows *maximal-sharing g*
⟨*proof*⟩

lemma *add-node-set-eq*:
assumes $k \neq NoNode$
assumes $n \notin ids\ g$
shows $as\text{-}set\ (add\text{-}node\ n\ (k,\ s)\ g) = as\text{-}set\ g \cup \{(n,\ (k,\ s))\}$
⟨*proof*⟩

lemma *add-node-as-set-eq*:
assumes $g' = add\text{-}node\ n\ (k,\ s)\ g$
assumes $n \notin ids\ g$
shows $(\{n\} \trianglelefteq as\text{-}set\ g') = as\text{-}set\ g$
⟨*proof*⟩

lemma *true-ids*:
$true\text{-}ids\ g = ids\ g - \{n \in ids\ g.\ is\text{-}RefNode\ (kind\ g\ n)\}$
⟨*proof*⟩

lemma *as-set-ids*:
assumes $as\text{-}set\ g = as\text{-}set\ g'$
shows $ids\ g = ids\ g'$
⟨*proof*⟩

lemma *ids-add-update*:
assumes $k \neq NoNode$
assumes $n \notin ids\ g$
assumes $g' = add\text{-}node\ n\ (k,\ s)\ g$
shows $ids\ g' = ids\ g \cup \{n\}$
⟨*proof*⟩

lemma *true-ids-add-update*:
assumes $k \neq NoNode$
assumes $n \notin ids\ g$
assumes $g' = add\text{-}node\ n\ (k,\ s)\ g$
assumes $\neg(is\text{-}RefNode\ k)$
shows $true\text{-}ids\ g' = true\text{-}ids\ g \cup \{n\}$

⟨*proof*⟩

lemma *new-def*:
assumes $(new \trianglelefteq as\text{-}set\ g') = as\text{-}set\ g$
shows $n \in ids\ g \longrightarrow n \notin new$
⟨*proof*⟩

lemma *add-preserves-rep*:
assumes *unchanged*: $(new \trianglelefteq as\text{-}set\ g') = as\text{-}set\ g$
assumes *closed*: *wf-closed g*
assumes *existed*: $n \in ids\ g$
assumes $g' \vdash n \simeq e$
shows $g \vdash n \simeq e$
⟨*proof*⟩

lemma *not-in-no-rep*:
$n \notin ids\ g \Longrightarrow \forall\, e.\ \neg(g \vdash n \simeq e)$
⟨*proof*⟩

lemma *unary-inputs*:
assumes $kind\ g\ n = unary\text{-}node\ op\ x$
shows $inputs\ g\ n = \{x\}$
⟨*proof*⟩

lemma *unary-succ*:
assumes $kind\ g\ n = unary\text{-}node\ op\ x$
shows $succ\ g\ n = \{\}$
⟨*proof*⟩

lemma *binary-inputs*:
assumes $kind\ g\ n = bin\text{-}node\ op\ x\ y$
shows $inputs\ g\ n = \{x,\ y\}$
⟨*proof*⟩

lemma *binary-succ*:
assumes $kind\ g\ n = bin\text{-}node\ op\ x\ y$
shows $succ\ g\ n = \{\}$
⟨*proof*⟩

lemma *unrep-contains*:
assumes $g \oplus e \rightsquigarrow (g',\ n)$
shows $n \in ids\ g'$
⟨*proof*⟩

lemma *unrep-preserves-contains*:
assumes $n \in ids\ g$
assumes $g \oplus e \rightsquigarrow (g',\ n')$

shows $n \in ids\ g'$
⟨*proof*⟩

lemma *unique-preserves-closure*:
assumes *wf-closed g*
assumes *unique g* (*node*, *s*) (g', *n*)
assumes *set* (*inputs-of node*) $\subseteq$ *ids g* $\wedge$
set (*successors-of node*) $\subseteq$ *ids g* $\wedge$
node $\neq$ *NoNode*
shows *wf-closed* g'
⟨*proof*⟩

lemma *unrep-preserves-closure*:
assumes *wf-closed g*
assumes $g \oplus e \rightsquigarrow (g', n)$
shows *wf-closed* g'
⟨*proof*⟩

inductive-cases *ConstUnrepE*: $g \oplus$ (*ConstantExpr x*) $\rightsquigarrow (g', n)$

definition *constant-value* **where**
constant-value = (*IntVal 32 0*)
definition *bad-graph* **where**
bad-graph = *irgraph* [
(*0*, *AbsNode 1*, *constantAsStamp constant-value*),
(*1*, *RefNode 2*, *constantAsStamp constant-value*),
(*2*, *ConstantNode constant-value*, *constantAsStamp constant-value*)
]

end

8 Control-flow Semantics

theory *IRStepObj*
imports
TreeToGraph
Graph.Class
begin

8.1 Object Heap

The heap model we introduce maps field references to object instances to runtime values. We use the H[f][p] heap representation. See *cite*{*heap*−*reps*−*2011*}.

We also introduce the DynamicHeap type which allocates new object references sequentially storing the next free object reference as 'Free'.

heapdef

type-synonym $('a,\ 'b)$ *Heap* = $'a \Rightarrow 'b \Rightarrow$ *Value*
type-synonym *Free* = *nat*
type-synonym $('a,\ 'b)$ *DynamicHeap* = $('a,\ 'b)$ *Heap* × *Free*

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

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

fun *h-new-inst* :: (*string*, *objref*) *DynamicHeap* ⇒ *string* ⇒ (*string*, *objref*) *DynamicHeap* × *Value* **where**
h-new-inst (*h*, *n*) *className* = (*h-store-field* ''*class*'' (*Some n*) (*ObjStr className*) (*h*,*n*+*1*), (*ObjRef* (*Some n*)))

type-synonym *FieldRefHeap* = (*string*, *objref*) *DynamicHeap*

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

8.2 Intraprocedural Semantics

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

inductive *indexof* :: $'a$ *list* ⇒ *nat* ⇒ $'a \Rightarrow$ *bool* **where**
find-index x xs = *i* ⟹ *indexof xs i x*

lemma *indexof-det*:
indexof xs i x ⟹ *indexof xs i′ x* ⟹ *i* = *i′*
⟨*proof*⟩

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

notation (*latex* **output**)
indexof (-!- = -)

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

fun *set-phis* :: *ID list* ⇒ *Value list* ⇒ *MapState* ⇒ *MapState* **where**
set-phis [] [] *m* = *m* |
set-phis (*n* # *ns*) (*v* # *vs*) *m* = (*set-phis ns vs* (*m*(*n* := *v*))) |
set-phis [] (*v* # *vs*) *m* = *m* |
set-phis (*x* # *ns*) [] *m* = *m*

definition
fun-add :: ('*a* ⇒ '*b*) ⇒ ('*a* ⇀ '*b*) ⇒ ('*a* ⇒ '*b*) (**infixl** $++_f$ *100*) **where**
f1 $++_f$ *f2* = (λ*x*. *case f2 x of None* ⇒ *f1 x* | *Some y* ⇒ *y*)

definition *upds* :: ('*a* ⇒ '*b*) ⇒ '*a list* ⇒ '*b list* ⇒ ('*a* ⇒ '*b*) (-/'(- [→] -/') *900*)
where
upds m ns vs = *m* $++_f$ (*map-of* (*rev* (*zip ns vs*)))

lemma *fun-add-empty*:
xs $++_f$ (*map-of* []) = *xs*
⟨*proof*⟩

lemma *upds-inc*:
m(*a*#*as* [→] *b*#*bs*) = (*m*(*a*:=*b*))(*as*[→]*bs*)
⟨*proof*⟩

lemma *upds-compose*:
a $++_f$ *map-of* (*rev* (*zip* (*n* # *ns*) (*v* # *vs*))) = *a*(*n* := *v*) $++_f$ *map-of* (*rev* (*zip ns vs*))
⟨*proof*⟩

lemma *set-phis ns vs* = (λ*m*. *upds m ns vs*)
⟨*proof*⟩

fun *is-PhiKind* :: *IRGraph* ⇒ *ID* ⇒ *bool* **where**
is-PhiKind g nid = *is-PhiNode* (*kind g nid*)

definition *filter-phis* :: *IRGraph* ⇒ *ID* ⇒ *ID list* **where**
filter-phis g merge = (*filter* (*is-PhiKind g*) (*sorted-list-of-set* (*usages g merge*)))

definition *phi-inputs* :: *IRGraph* ⇒ *ID list* ⇒ *nat* ⇒ *ID list* **where**
phi-inputs g phis i = (*map* (λ*n*. (*inputs-of* (*kind g n*))!(*i* + *1*)) *phis*)

Intraprocedural semantics are given as a small-step semantics.

Within the context of a graph, the configuration triple, (ID, MethodState, Heap), is related to the subsequent configuration.

inductive *step* :: *IRGraph* ⇒ *Params* ⇒ (*ID* × *MapState* × *FieldRefHeap*) ⇒ (*ID* × *MapState* × *FieldRefHeap*) ⇒ *bool*
(-, - ⊢ - → - *55*) **for** *g p* **where**

SequentialNode:
⟦*is-sequential-node* (*kind g nid*);

$nid' = (successors\text{-}of\ (kind\ g\ nid))!0]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)\ |$

FixedGuardNode:
$[\![(kind\ g\ nid) = (FixedGuardNode\ cond\ before\ next);$
$[g,\ m,\ p] \vdash cond \mapsto val;$

$\neg(val\text{-}to\text{-}bool\ val)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (next,\ m,\ h)\ |$

BytecodeExceptionNode:
$[\![(kind\ g\ nid) = (BytecodeExceptionNode\ args\ st\ nid');$
$exceptionType = stp\text{-}type\ (stamp\ g\ nid);$
$(h',\ ref) = h\text{-}new\text{-}inst\ h\ exceptionType;$
$m' = m(nid := ref)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h')\ |$

IfNode:
$[\![kind\ g\ nid = (IfNode\ cond\ tb\ fb);$
$[g,\ m,\ p] \vdash cond \mapsto val;$
$nid' = (if\ val\text{-}to\text{-}bool\ val\ then\ tb\ else\ fb)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)\ |$

EndNodes:
$[\![is\text{-}AbstractEndNode\ (kind\ g\ nid);$
$merge = any\text{-}usage\ g\ nid;$
$is\text{-}AbstractMergeNode\ (kind\ g\ merge);$

$indexof\ (inputs\text{-}of\ (kind\ g\ merge))\ i\ nid;$
$phis = filter\text{-}phis\ g\ merge;$
$inps = phi\text{-}inputs\ g\ phis\ i;$
$[g,\ m,\ p] \vdash inps\ [\mapsto]\ vs;$

$m' = (m(phis[\rightarrow]vs))]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (merge,\ m',\ h)\ |$

NewArrayNode:
$[\![kind\ g\ nid = (NewArrayNode\ len\ st\ nid');$
$[g,\ m,\ p] \vdash len \mapsto length';$

$arrayType = stp\text{-}type\ (stamp\ g\ nid);$
$(h',\ ref) = h\text{-}new\text{-}inst\ h\ arrayType;$
$ref = ObjRef\ refNo;$
$h'' = h\text{-}store\text{-}field\ ''''\ refNo\ (intval\text{-}new\text{-}array\ length'\ arrayType)\ h';$

$m' = m(nid := ref)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h'')\ |$

ArrayLengthNode:
⟦*kind g nid* = (*ArrayLengthNode x nid′*);
[*g*, *m*, *p*] ⊢ *x* ↦ *ObjRef ref*;

h-load-field '''' *ref h* = *arrayVal*;
length′ = *array-length* (*arrayVal*);

m′ = *m*(*nid* := *length′*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m′*, *h*) |

LoadIndexedNode:
⟦*kind g nid* = (*LoadIndexedNode index guard array nid′*);
[*g*, *m*, *p*] ⊢ *index* ↦ *indexVal*;
[*g*, *m*, *p*] ⊢ *array* ↦ *ObjRef ref*;

h-load-field '''' *ref h* = *arrayVal*;
loaded = *intval-load-index arrayVal indexVal*;

m′ = *m*(*nid* := *loaded*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m′*, *h*) |

StoreIndexedNode:
⟦*kind g nid* = (*StoreIndexedNode check val st index guard array nid′*);
[*g*, *m*, *p*] ⊢ *index* ↦ *indexVal*;
[*g*, *m*, *p*] ⊢ *array* ↦ *ObjRef ref*;
[*g*, *m*, *p*] ⊢ *val* ↦ *value*;

h-load-field '''' *ref h* = *arrayVal*;
updated = *intval-store-index arrayVal indexVal value*;
h′ = *h-store-field* '''' *ref updated h*;
m′ = *m*(*nid* := *updated*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m′*, *h′*) |

NewInstanceNode:
⟦*kind g nid* = (*NewInstanceNode nid cname obj nid′*);
(*h′*, *ref*) = *h-new-inst h cname*;
m′ = *m*(*nid* := *ref*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m′*, *h′*) |

LoadFieldNode:
⟦*kind g nid* = (*LoadFieldNode nid f* (*Some obj*) *nid′*);
[*g*, *m*, *p*] ⊢ *obj* ↦ *ObjRef ref*;
m′ = *m*(*nid* := *h-load-field f ref h*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m′*, *h*) |

SignedDivNode:
⟦*kind g nid* = (*SignedDivNode nid x y zero sb next*);
[*g*, *m*, *p*] ⊢ *x* ↦ *v1*;
[*g*, *m*, *p*] ⊢ *y* ↦ *v2*;

$m' = \ m(nid := intval\text{-}div\ v1\ v2)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (next,\ m',\ h)\ |$

SignedRemNode:
$[\![kind\ g\ nid = (SignedRemNode\ nid\ x\ y\ zero\ sb\ next);$
$[g,\ m,\ p] \vdash x \mapsto v1;$
$[g,\ m,\ p] \vdash y \mapsto v2;$
$m' = \ m(nid := intval\text{-}mod\ v1\ v2)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (next,\ m',\ h)\ |$

StaticLoadFieldNode:
$[\![kind\ g\ nid = (LoadFieldNode\ nid\ f\ None\ nid');$
$m' = \ m(nid := h\text{-}load\text{-}field\ f\ None\ h)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h)\ |$

StoreFieldNode:
$[\![kind\ g\ nid = (StoreFieldNode\ nid\ f\ newval\ \text{-}\ (Some\ obj)\ nid');$
$[g,\ m,\ p] \vdash newval \mapsto val;$
$[g,\ m,\ p] \vdash obj \mapsto ObjRef\ ref;$
$h' = h\text{-}store\text{-}field\ f\ ref\ val\ h;$
$m' = \ m(nid := val)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h')\ |$

StaticStoreFieldNode:
$[\![kind\ g\ nid = (StoreFieldNode\ nid\ f\ newval\ \text{-}\ None\ nid');$
$[g,\ m,\ p] \vdash newval \mapsto val;$
$h' = h\text{-}store\text{-}field\ f\ None\ val\ h;$
$m' = \ m(nid := val)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h')$

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

8.3 Interprocedural Semantics

type-synonym *Signature* = *string*
type-synonym *Program* = *Signature* ⇀ *IRGraph*
type-synonym *System* = *Program* × *Classes*

function *dynamic-lookup* :: *System* ⇒ *string* ⇒ *string* ⇒ *string list* ⇒ *IRGraph option* **where**
dynamic-lookup (*P*,*cl*) *cn mn path* = (
if (*cn* = *''None''* ∨ *cn* ∉ *set* (*Class.mapJVMFunc class-name cl*) ∨ *path* = [])
then (*P mn*)
else (

let method-index = (*find-index* (*get-simple-signature mn*) (*CLsimple-signatures cn cl*)) *in*
let parent = *hd path in*

if (*method-index* = *length* (*CLsimple-signatures cn cl*))
then (*dynamic-lookup* (*P*, *cl*) *parent mn* (*tl path*))
else (*P* (*nth* (*map method-unique-name* (*CLget-Methods cn cl*)) *method-index*))
)
)

⟨*proof*⟩
termination *dynamic-lookup* ⟨*proof*⟩

inductive *step-top* :: *System* ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap* ⇒
(*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap* ⇒ *bool*
(- ⊢ - ⟶ - *55*)
for *S* **where**

Lift:
⟦*g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid*′, *m*′, *h*′)⟧
⟹ (*S*) ⊢ ((*g*,*nid*,*m*,*p*)#*stk*, *h*) ⟶ ((*g*,*nid*′,*m*′,*p*)#*stk*, *h*′) |

InvokeNodeStepStatic:
⟦*is-Invoke* (*kind g nid*);
callTarget = *ir-callTarget* (*kind g nid*);
kind g callTarget = (*MethodCallTargetNode targetMethod actuals invoke-kind*);
¬(*hasReceiver invoke-kind*);
Some targetGraph = (*dynamic-lookup S* ′′*None*′′ *targetMethod* []);
[*g*, *m*, *p*] ⊢ *actuals* [↦] *p*′⟧
⟹ (*S*) ⊢ ((*g*,*nid*,*m*,*p*)#*stk*, *h*) ⟶ ((*targetGraph*,*0*,*new-map-state*,*p*′)#(*g*,*nid*,*m*,*p*)#*stk*, *h*) |

InvokeNodeStep:
⟦*is-Invoke* (*kind g nid*);
callTarget = *ir-callTarget* (*kind g nid*);
kind g callTarget = (*MethodCallTargetNode targetMethod arguments invoke-kind*);
hasReceiver invoke-kind;
[*g*, *m*, *p*] ⊢ *arguments* [↦] *p*′;
ObjRef self = *hd p*′;
ObjStr cname = (*h-load-field* ′′*class*′′ *self h*);
S = (*P*,*cl*);
Some targetGraph = *dynamic-lookup S cname targetMethod* (*class-parents* (*CLget-JVMClass cname cl*))⟧
⟹ (*S*) ⊢ ((*g*,*nid*,*m*,*p*)#*stk*, *h*) ⟶ ((*targetGraph*,*0*,*new-map-state*,*p*′)#(*g*,*nid*,*m*,*p*)#*stk*, *h*) |

ReturnNode:
⟦*kind g nid* = (*ReturnNode* (*Some expr*) -);

$[g,\ m,\ p] \vdash expr \mapsto v;$

$m'_c = m_c(nid_c := v);$
$nid'_c = (successors\text{-}of\ (kind\ g_c\ nid_c))!0]\!]$
$\Longrightarrow (S) \vdash ((g,nid,m,p)\#(g_c,nid_c,m_c,p_c)\#stk,\ h) \longrightarrow ((g_c,nid'_c,m'_c,p_c)\#stk,\ h)$
|

ReturnNodeVoid:
$[\![kind\ g\ nid = (ReturnNode\ None\ \text{-});$

$nid'_c = (successors\text{-}of\ (kind\ g_c\ nid_c))!0]\!]$
$\Longrightarrow (S) \vdash ((g,nid,m,p)\#(g_c,nid_c,m_c,p_c)\#stk,\ h) \longrightarrow ((g_c,nid'_c,m_c,p_c)\#stk,\ h)$ |

UnwindNode:
$[\![kind\ g\ nid = (UnwindNode\ exception);$

$[g,\ m,\ p] \vdash exception \mapsto e;$

$kind\ g_c\ nid_c = (InvokeWithExceptionNode\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ exEdge);$

$m'_c = m_c(nid_c := e)]\!]$
$\Longrightarrow (S) \vdash ((g,nid,m,p)\#(g_c,nid_c,m_c,p_c)\#stk,\ h) \longrightarrow ((g_c,exEdge,m'_c,p_c)\#stk,\ h)$

code-pred (*modes*: $i \Rightarrow i \Rightarrow o \Rightarrow bool$) *step-top* ⟨*proof*⟩

8.4 Big-step Execution

type-synonym $Trace = (IRGraph \times ID \times MapState \times Params)\ list$

fun *has-return* :: $MapState \Rightarrow bool$ **where**
has-return $m = (m\ 0 \neq UndefVal)$

inductive *exec* :: *System*
$\Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times FieldRefHeap$
$\Rightarrow Trace$
$\Rightarrow (IRGraph \times ID \times MapState \times Params)\ list \times FieldRefHeap$
$\Rightarrow Trace$
$\Rightarrow bool$
$(\text{-} \vdash \text{-} \mid \text{-} \longrightarrow\!* \text{-} \mid \text{-})$
for P **where**
$[\![P \vdash (((g,nid,m,p)\#xs),h) \longrightarrow (((g',nid',m',p')\#ys),h');$
$\neg(has\text{-}return\ m');$

$l' = (l\ @\ [(g,nid,m,p)]);$

$exec\ P\ (((g',nid',m',p')\#ys),h')\ l'\ next\text{-}state\ l'']\!]$
$\Longrightarrow exec\ P\ (((g,nid,m,p)\#xs),h)\ l\ next\text{-}state\ l''$

```
  |
  ⟦P ⊢ (((g,nid,m,p)#xs),h) ⟶ (((g',nid',m',p')#ys),h');
    has-return m';

    l' = (l @ [(g,nid,m,p)])⟧
    ⟹ exec P (((g,nid,m,p)#xs),h) l (((g',nid',m',p')#ys),h') l'
code-pred (modes: i ⇒ i ⇒ i ⇒ o ⇒ o ⇒ bool as Exec) exec ⟨proof⟩

inductive exec-debug :: System
     ⇒ (IRGraph × ID × MapState × Params) list × FieldRefHeap
     ⇒ nat
     ⇒ (IRGraph × ID × MapState × Params) list × FieldRefHeap
     ⇒ bool
  (-⊢-→*-* -)
  where
  ⟦n > 0;
    p ⊢ s ⟶ s';
    exec-debug p s' (n − 1) s''⟧
    ⟹ exec-debug p s n s'' |

  ⟦n = 0⟧
    ⟹ exec-debug p s n s
code-pred (modes: i ⇒ i ⇒ i ⇒ o ⇒ bool) exec-debug ⟨proof⟩
```

8.4.1 Heap Testing

```
definition p3:: Params where
  p3 = [IntVal 32 3]

fun graphToSystem :: IRGraph ⇒ System where
  graphToSystem graph = ((λx. Some graph), JVMClasses [])


values {(prod.fst(prod.snd (prod.snd (hd (prod.fst res))))) 0
      | res. (graphToSystem eg2-sq) ⊢ ([(eg2-sq,0,new-map-state,p3), (eg2-sq,0,new-map-state,p3)],
new-heap) →*2* res}

definition field-sq :: string where
  field-sq = ''sq''

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

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

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

values {*h-load-field field-sq* (*Some 0*) (*prod.snd res*)
    | *res*. (*graphToSystem* (*eg4-sq*)) ⊢ ([(*eg4-sq*, *0*, *new-map-state*, *p3*), (*eg4-sq*, *0*, *new-map-state*, *p3*)], *new-heap*) →∗*3*∗ *res*}

end

8.5 Control-flow Semantics Theorems

theory *IRStepThms*
  imports
    IRStepObj
    TreeToGraphThms
begin

We prove that within the same graph, a configuration triple will always transition to the same subsequent configuration. Therefore, our step semantics is deterministic.

8.5.1 Control-flow Step is Deterministic

theorem *stepDet′*:
  (*g*, *p* ⊢ *state* → *next*) ⟹
    (*g*, *p* ⊢ *state* → *next′*) ⟹ *next* = *next′*
⟨*proof*⟩

theorem *stepDet*:
  (*g*, *p* ⊢ (*nid*,*m*,*h*) → *next*) ⟹
  (∀ *next′*. ((*g*, *p* ⊢ (*nid*,*m*,*h*) → *next′*) ⟶ *next* = *next′*))
  ⟨*proof*⟩

lemma *stepRefNode*:
  $[\![kind\ g\ nid = RefNode\ nid']\!] \Longrightarrow g,\ p \vdash (nid,m,h) \rightarrow (nid',m,h)$
  ⟨*proof*⟩

lemma *IfNodeStepCases*:
  assumes $kind\ g\ nid = IfNode\ cond\ tb\ fb$
  assumes $g \vdash cond \simeq condE$
  assumes $[m,\ p] \vdash condE \mapsto v$
  assumes $g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)$
  shows $nid' \in \{tb,\ fb\}$
  ⟨*proof*⟩

lemma *IfNodeSeq*:
  shows $kind\ g\ nid = IfNode\ cond\ tb\ fb \longrightarrow \neg(is\text{-}sequential\text{-}node\ (kind\ g\ nid))$
  ⟨*proof*⟩

lemma *IfNodeCond*:
  assumes $kind\ g\ nid = IfNode\ cond\ tb\ fb$
  assumes $g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)$
  shows $\exists\ condE\ v.\ ((g \vdash cond \simeq condE) \wedge ([m,\ p] \vdash condE \mapsto v))$
  ⟨*proof*⟩

lemma *step-in-ids*:
  assumes $g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h')$
  shows $nid \in ids\ g$
  ⟨*proof*⟩

end

9 Proof Infrastructure

9.1 Bisimulation

theory *Bisimulation*
imports
  Stuttering
begin

inductive *weak-bisimilar* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$
  (- . - ∼ -) **for** *nid* **where**
  $[\![\forall P'.\ (g\ m\ p\ h \vdash nid \leadsto P') \longrightarrow (\exists Q'\ .\ (g'\ m\ p\ h \vdash nid \leadsto Q') \wedge P' = Q');$
    $\forall Q'.\ (g'\ m\ p\ h \vdash nid \leadsto Q') \longrightarrow (\exists P'\ .\ (g\ m\ p\ h \vdash nid \leadsto P') \wedge P' = Q')]\!]$
  $\Longrightarrow nid\ .\ g \sim g'$

A strong bisimilution between no-op transitions

inductive *strong-noop-bisimilar* :: $ID \Rightarrow IRGraph \Rightarrow IRGraph \Rightarrow bool$

(- | - ∼ -) **for** *nid* **where**
$[\![\forall P'. (g, p \vdash (nid, m, h) \to P') \longrightarrow (\exists Q' . (g', p \vdash (nid, m, h) \to Q') \land P' = Q');$
$\forall Q'. (g', p \vdash (nid, m, h) \to Q') \longrightarrow (\exists P' . (g, p \vdash (nid, m, h) \to P') \land P' = Q')]\!]$
$\Longrightarrow nid \mid g \sim g'$

lemma *lockstep-strong-bisimilulation*:
assumes $g' = replace\text{-}node\ nid\ node\ g$
assumes $g, p \vdash (nid, m, h) \to (nid', m, h)$
assumes $g', p \vdash (nid, m, h) \to (nid', m, h)$
shows $nid \mid g \sim g'$
⟨*proof*⟩

lemma *no-step-bisimulation*:
assumes $\forall m\ p\ h\ nid'\ m'\ h'. \neg(g, p \vdash (nid, m, h) \to (nid', m', h'))$
assumes $\forall m\ p\ h\ nid'\ m'\ h'. \neg(g', p \vdash (nid, m, h) \to (nid', m', h'))$
shows $nid \mid g \sim g'$
⟨*proof*⟩

end

9.2 Graph Rewriting

theory
Rewrites
imports
Stuttering
begin

fun *replace-usages* :: $ID \Rightarrow ID \Rightarrow IRGraph \Rightarrow IRGraph$ **where**
$replace\text{-}usages\ nid\ nid'\ g = replace\text{-}node\ nid\ (RefNode\ nid', stamp\ g\ nid')\ g$

lemma *replace-usages-effect*:
assumes $g' = replace\text{-}usages\ nid\ nid'\ g$
shows $kind\ g'\ nid = RefNode\ nid'$
⟨*proof*⟩

lemma *replace-usages-changeonly*:
assumes $nid \in ids\ g$
assumes $g' = replace\text{-}usages\ nid\ nid'\ g$
shows $changeonly\ \{nid\}\ g\ g'$
⟨*proof*⟩

lemma *replace-usages-unchanged*:
assumes $nid \in ids\ g$
assumes $g' = replace\text{-}usages\ nid\ nid'\ g$
shows $unchanged\ (ids\ g - \{nid\})\ g\ g'$
⟨*proof*⟩

fun *nextNid* :: *IRGraph* ⇒ *ID* **where**
  nextNid g = (*Max* (*ids g*)) + *1*

lemma *max-plus-one*:
  fixes *c* :: *ID set*
  shows ⟦*finite c*; *c* ≠ {}⟧ ⟹ (*Max c*) + *1* ∉ *c*
  ⟨*proof*⟩

lemma *ids-finite*:
  finite (*ids g*)
  ⟨*proof*⟩

lemma *nextNidNotIn*:
  ids g ≠ {} ⟶ *nextNid g* ∉ *ids g*
  ⟨*proof*⟩

fun *bool-to-val-width1* :: *bool* ⇒ *Value* **where**
  bool-to-val-width1 True = (*IntVal 1 1*) |
  bool-to-val-width1 False = (*IntVal 1 0*)

fun *constantCondition* :: *bool* ⇒ *ID* ⇒ *IRNode* ⇒ *IRGraph* ⇒ *IRGraph* **where**
  constantCondition val nid (*IfNode cond t f*) *g* =
   (*let* (*g′*, *nid′*) = *Predicate.the* (*unrepE g* (*ConstantExpr* (*bool-to-val-width1 val*)))
in
     replace-node nid (*IfNode nid′ t f*, *stamp g nid*) *g′*) |
  constantCondition cond nid - g = *g*

inductive-cases *unrepUnaryE*:
  unrep g (*UnaryExpr op e*) (*g′*, *nid*)
inductive-cases *unrepBinaryE*:
  unrep g (*BinaryExpr op e1 e2*) (*g′*, *nid*)
inductive-cases *unrepConditionalE*:
  unrep g (*ConditionalExpr c t f*) (*g′*, *nid*)
inductive-cases *unrepParamE*:
  unrep g (*ParameterExpr i s*) (*g′*, *nid*)
inductive-cases *unrepConstE*:
  unrep g (*ConstantExpr c*) (*g′*, *nid*)
inductive-cases *unrepLeafE*:
  unrep g (*LeafExpr n s*) (*g′*, *nid*)
inductive-cases *unrepVariableE*:
  unrep g (*VariableExpr v s*) (*g′*, *nid*)
inductive-cases *unrepConstVarE*:
  unrep g (*ConstantVar c*) (*g′*, *nid*)

lemma *uniqueDet*:
  assumes *unique g e* (g'_1, nid_1)
  assumes *unique g e* (g'_2, nid_2)
  shows $g'_1 = g'_2 \land nid_1 = nid_2$

⟨*proof*⟩

lemma *unrepDet*:
  assumes *unrep g e* (g'_1, nid_1)
  assumes *unrep g e* (g'_2, nid_2)
  shows $g'_1 = g'_2 \wedge nid_1 = nid_2$
  ⟨*proof*⟩

lemma *unwrapUnrepE*:
  assumes *unrep g e* (g', nid')
  shows (g', nid') = *Predicate.the* (*unrepE g e*)
  ⟨*proof*⟩

lemma *constantCondition-sem*:
  assumes (*unrep g* (*ConstantExpr* (*bool-to-val-width1 val*)) (g', nid'))
  shows *constantCondition val nid* (*IfNode cond t f*) *g* =
    replace-node nid (*IfNode nid' t f*, *stamp g nid*) *g'*
  ⟨*proof*⟩

fun *wf-insert* :: *IRGraph* ⇒ *IRExpr* ⇒ *bool* **where**
  wf-insert g (*LeafExpr n s*) = *is-preevaluated* (*kind g n*) |
  wf-insert g (*VariableExpr v s*) = *False* |
  wf-insert g (*ConstantVar v*) = *False* |
  wf-insert g - = *True*

lemma *insertConstUnique*:
  ∃ *g' nid'*. *unique g* (*ConstantNode c*, *s*) (g', nid')
  ⟨*proof*⟩

lemma *insertConst*:
  ∃ *g' nid'*. *unrep g* (*ConstantExpr c*) (g', nid')
  ⟨*proof*⟩

lemma *constantConditionTrue*:
  assumes *kind g ifcond* = *IfNode cond t f*
  assumes *g'* = *constantCondition True ifcond* (*kind g ifcond*) *g*
  shows $g', p \vdash (ifcond, m, h) \rightarrow (t, m, h)$
⟨*proof*⟩

lemma *constantConditionFalse*:
  assumes *kind g ifcond* = *IfNode cond t f*
  assumes *g'* = *constantCondition False ifcond* (*kind g ifcond*) *g*
  shows $g', p \vdash (ifcond, m, h) \rightarrow (f, m, h)$
⟨*proof*⟩

lemma *diff-forall*:

assumes $\forall\, n \in ids\ g - \{nid\}.\ cond\ n$
shows $\forall\, n.\ n \in ids\ g \wedge n \notin \{nid\} \longrightarrow cond\ n$
⟨*proof*⟩

lemma *replace-node-changeonly*:
assumes $g' = replace\text{-}node\ nid\ node\ g$
shows $changeonly\ \{nid\}\ g\ g'$
⟨*proof*⟩

lemma *add-node-changeonly*:
assumes $g' = add\text{-}node\ nid\ node\ g$
shows $changeonly\ \{nid\}\ g\ g'$
⟨*proof*⟩

lemma *constantConditionNoEffect*:
assumes $\neg(is\text{-}IfNode\ (kind\ g\ nid))$
shows $g = constantCondition\ b\ nid\ (kind\ g\ nid)\ g$
⟨*proof*⟩

lemma *changeonly-ConstantExpr*:
assumes $unrep\ g\ (ConstantExpr\ c)\ (g',\ nid)$
shows $changeonly\ \{\}\ g\ g'$
⟨*proof*⟩

lemma *constantCondition-changeonly*:
assumes $nid \in ids\ g$
assumes $g' = constantCondition\ b\ nid\ (kind\ g\ nid)\ g$
shows $changeonly\ \{nid\}\ g\ g'$
⟨*proof*⟩

lemma *constantConditionNoIf*:
assumes $\forall\, cond\ t\ f.\ kind\ g\ ifcond \neq IfNode\ cond\ t\ f$
assumes $g' = constantCondition\ val\ ifcond\ (kind\ g\ ifcond)\ g$
shows $\exists\, nid'\ .(g\ m\ p\ h \vdash ifcond \rightsquigarrow nid') \longleftrightarrow (g'\ m\ p\ h \vdash ifcond \rightsquigarrow nid')$
⟨*proof*⟩

lemma *constantConditionValid*:
assumes $kind\ g\ ifcond = IfNode\ cond\ t\ f$
assumes $[g,\ m,\ p] \vdash cond \mapsto v$
assumes $const = val\text{-}to\text{-}bool\ v$
assumes $g' = constantCondition\ const\ ifcond\ (kind\ g\ ifcond)\ g$
shows $\exists\, nid'\ .(g\ m\ p\ h \vdash ifcond \rightsquigarrow nid') \longleftrightarrow (g'\ m\ p\ h \vdash ifcond \rightsquigarrow nid')$
⟨*proof*⟩

end

9.3 Stuttering

theory *Stuttering*
imports
Semantics.IRStepThms
begin

inductive *stutter*:: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *FieldRefHeap* ⇒ *ID* ⇒ *ID* ⇒ *bool* (- - - - ⊢ - ⇝ - *55*)
for *g m p h* **where**

StutterStep:
$[\![g,\ p \vdash (nid,m,h) \rightarrow (nid',m,h)]\!]$
$\Longrightarrow g\ m\ p\ h \vdash nid \leadsto nid'$ |

Transitive:
$[\![g,\ p \vdash (nid,m,h) \rightarrow (nid'',m,h);$
$g\ m\ p\ h \vdash nid'' \leadsto nid']\!]$
$\Longrightarrow g\ m\ p\ h \vdash nid \leadsto nid'$

lemma *stuttering-successor*:
assumes $(g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h))$
shows $\{P'.\ (g\ m\ p\ h \vdash nid \leadsto P')\} = \{nid'\} \cup \{nid''.\ (g\ m\ p\ h \vdash nid' \leadsto nid'')\}$
⟨*proof*⟩

end

9.4 Evaluation Stamp Theorems

theory *StampEvalThms*
imports *Graph.ValueThms*
Semantics.IRTreeEvalThms
begin

lemma
assumes *take-bit b v = v*
shows *signed-take-bit b v = v*
⟨*proof*⟩

lemma *unwrap-signed-take-bit*:
fixes *v* :: *int64*
assumes $0 < b \wedge b \leq 64$
assumes *signed-take-bit* $(b - 1)$ *v = v*
shows *signed-take-bit 63* (*Word.rep* (*signed-take-bit* $(b - Suc\ 0)$ *v*)) = *sint v*
⟨*proof*⟩

lemma *unrestricted-new-int-always-valid* [*simp*]:
assumes $0 < b \wedge b \leq 64$
shows *valid-value* (*new-int b v*) (*unrestricted-stamp* (*IntegerStamp b lo hi*))
⟨*proof*⟩

lemma *unary-undef*: *val* = *UndefVal* $\Longrightarrow$ *unary-eval op val* = *UndefVal*
⟨*proof*⟩

lemma *unary-obj*:
val = *ObjRef x* $\Longrightarrow$ (*if* (*op* = *UnaryIsNull*) *then*
unary-eval op val ≠ *UndefVal else*
unary-eval op val = *UndefVal*)
⟨*proof*⟩

lemma *unrestricted-stamp-valid*:
assumes *s* = *unrestricted-stamp* (*IntegerStamp b lo hi*)
assumes $0 < b \wedge b \leq 64$
shows *valid-stamp s*
⟨*proof*⟩

lemma *unrestricted-stamp-valid-value* [*simp*]:
assumes *1*: *result* = *IntVal b ival*
assumes *take-bit b ival* = *ival*
assumes $0 < b \wedge b \leq 64$
shows *valid-value result* (*unrestricted-stamp* (*IntegerStamp b lo hi*))
⟨*proof*⟩

9.4.1 Support Lemmas for Integer Stamps and Associated IntVal values

Valid int implies some useful facts.

lemma *valid-int-gives*:
assumes *valid-value* (*IntVal b val*) *stamp*
obtains *lo hi* **where** *stamp* = *IntegerStamp b lo hi* ∧
valid-stamp (*IntegerStamp b lo hi*) ∧
take-bit b val = *val* ∧
lo ≤ *int-signed-value b val* ∧ *int-signed-value b val* ≤ *hi*
⟨*proof*⟩

And the corresponding lemma where we know the stamp rather than the value.

lemma *valid-int-stamp-gives*:
assumes *valid-value val* (*IntegerStamp b lo hi*)
obtains *ival* **where** *val* = *IntVal b ival* ∧
valid-stamp (*IntegerStamp b lo hi*) ∧
take-bit b ival = *ival* ∧
lo ≤ *int-signed-value b ival* ∧ *int-signed-value b ival* ≤ *hi*
⟨*proof*⟩

A valid int must have the expected number of bits.

lemma *valid-int-same-bits*:
assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)

shows *b* = *bits*
⟨*proof*⟩

A valid value means a valid stamp.

lemma *valid-int-valid-stamp*:
assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)
shows *valid-stamp* (*IntegerStamp bits lo hi*)
⟨*proof*⟩

A valid int means a valid non-empty stamp.

lemma *valid-int-not-empty*:
assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)
shows *lo* ≤ *hi*
⟨*proof*⟩

A valid int fits into the given number of bits (and other bits are zero).

lemma *valid-int-fits*:
assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)
shows *take-bit bits val* = *val*
⟨*proof*⟩

lemma *valid-int-is-zero-masked*:
assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)
shows *and val* (*not* (*mask bits*)) = *0*
⟨*proof*⟩

Unsigned ints have bounds 0 up to $2^b its$.

lemma *valid-int-unsigned-bounds*:
assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)

shows *uint val* < *2* ^ *bits*
⟨*proof*⟩

Signed ints have the usual two-complement bounds.

lemma *valid-int-signed-upper-bound*:
assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)
shows *int-signed-value bits val* < *2* ^ (*bits* − *1*)
⟨*proof*⟩

lemma *valid-int-signed-lower-bound*:
assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)
shows −(*2* ^ (*bits* − *1*)) ≤ *int-signed-value bits val*
⟨*proof*⟩

and *bit_bounds* versions of the above bounds.

lemma *valid-int-signed-upper-bit-bound*:
assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)
shows *int-signed-value bits val* ≤ *snd* (*bit-bounds bits*)

⟨*proof*⟩

lemma *valid-int-signed-lower-bit-bound*:
  assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)
  shows *fst* (*bit-bounds bits*) ≤ *int-signed-value bits val*
⟨*proof*⟩

Valid values satisfy their stamp bounds.

lemma *valid-int-signed-range*:
  assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)
  shows *lo* ≤ *int-signed-value bits val* ∧ *int-signed-value bits val* ≤ *hi*
  ⟨*proof*⟩

9.4.2 Validity of all Unary Operators

We split the validity proof for unary operators into two lemmas, one for normal unary operators whose output bits equals their input bits, and the other case for the widen and narrow operators.

lemma *eval-normal-unary-implies-valid-value*:
  assumes [*m*,*p*] ⊢ *expr* ↦ *val*
  assumes *result* = *unary-eval op val*
  assumes *op*: *op* ∈ *normal-unary*
  assumes *notbool*: *op* ∉ *boolean-unary*
  assumes *notfixed32*: *op* ∉ *unary-fixed-32-ops*
  assumes *result* ≠ *UndefVal*
  assumes *valid-value val* (*stamp-expr expr*)
  shows *valid-value result* (*stamp-expr* (*UnaryExpr op expr*))
⟨*proof*⟩

lemma *narrow-widen-output-bits*:
  assumes *unary-eval op val* ≠ *UndefVal*
  assumes *op* ∉ *normal-unary*
  assumes *op* ∉ *boolean-unary*
  assumes *op* ∉ *unary-fixed-32-ops*
  shows *0* < (*ir-resultBits op*) ∧ (*ir-resultBits op*) ≤ *64*
⟨*proof*⟩

lemma *eval-widen-narrow-unary-implies-valid-value*:
  assumes [*m*,*p*] ⊢ *expr* ↦ *val*
  assumes *result* = *unary-eval op val*
  assumes *op*: *op* ∉ *normal-unary*
  and *notbool*: *op* ∉ *boolean-unary*
  and *notfixed*: *op* ∉ *unary-fixed-32-ops*
  assumes *result* ≠ *UndefVal*
  assumes *valid-value val* (*stamp-expr expr*)
  shows *valid-value result* (*stamp-expr* (*UnaryExpr op expr*))
⟨*proof*⟩

lemma *eval-boolean-unary-implies-valid-value*:
assumes $[m,p] \vdash expr \mapsto val$
assumes *result* = *unary-eval op val*
assumes *op*: *op* ∈ *boolean-unary*
assumes *notnorm*: *op* ∉ *normal-unary*
assumes *result* ≠ *UndefVal*
assumes *valid-value val* (*stamp-expr expr*)
shows *valid-value result* (*stamp-expr* (*UnaryExpr op expr*))
⟨*proof*⟩

lemma *eval-fixed-unary-32-implies-valid-value*:
assumes $[m,p] \vdash expr \mapsto val$
assumes *result* = *unary-eval op val*
assumes *op*: *op* ∈ *unary-fixed-32-ops*
assumes *notnorm*: *op* ∉ *normal-unary*
assumes *notbool*: *op* ∉ *boolean-unary*
assumes *result* ≠ *UndefVal*
assumes *valid-value val* (*stamp-expr expr*)
shows *valid-value result* (*stamp-expr* (*UnaryExpr op expr*))
⟨*proof*⟩

lemma *eval-unary-implies-valid-value*:
assumes $[m,p] \vdash expr \mapsto val$
assumes *result* = *unary-eval op val*
assumes *result* ≠ *UndefVal*
assumes *valid-value val* (*stamp-expr expr*)
shows *valid-value result* (*stamp-expr* (*UnaryExpr op expr*))
⟨*proof*⟩

9.4.3 Support Lemmas for Binary Operators

lemma *binary-undef*: *v1* = *UndefVal* ∨ *v2* = *UndefVal* ⟹ *bin-eval op v1 v2* = *UndefVal*
⟨*proof*⟩

lemma *binary-obj*: *v1* = *ObjRef x* ∨ *v2* = *ObjRef y* ⟹ *bin-eval op v1 v2* = *UndefVal*
⟨*proof*⟩

Some lemmas about the three different output sizes for binary operators.

lemma *bin-eval-bits-binary-shift-ops*:
assumes *result* = *bin-eval op* (*IntVal b1 v1*) (*IntVal b2 v2*)
assumes *result* ≠ *UndefVal*
assumes *op* ∈ *binary-shift-ops*
shows ∃ *v*. *result* = *new-int b1 v*
⟨*proof*⟩

lemma *bin-eval-bits-fixed-32-ops*:
assumes *result* = *bin-eval op* (*IntVal b1 v1*) (*IntVal b2 v2*)

assumes *result* ≠ *UndefVal*
assumes *op* ∈ *binary-fixed-32-ops*
shows ∃ *v. result* = *new-int 32 v*
⟨*proof*⟩

lemma *bin-eval-bits-normal-ops*:
assumes *result* = *bin-eval op* (*IntVal b1 v1*) (*IntVal b2 v2*)
assumes *result* ≠ *UndefVal*
assumes *op* ∉ *binary-shift-ops*
assumes *op* ∉ *binary-fixed-32-ops*
shows ∃ *v. result* = *new-int b1 v*
⟨*proof*⟩

lemma *bin-eval-input-bits-equal*:
assumes *result* = *bin-eval op* (*IntVal b1 v1*) (*IntVal b2 v2*)
assumes *result* ≠ *UndefVal*
assumes *op* ∉ *binary-shift-ops*
shows *b1* = *b2*
⟨*proof*⟩

lemma *bin-eval-implies-valid-value*:
assumes [*m*,*p*] ⊢ *expr1* ↦ *val1*
assumes [*m*,*p*] ⊢ *expr2* ↦ *val2*
assumes *result* = *bin-eval op val1 val2*
assumes *result* ≠ *UndefVal*
assumes *valid-value val1* (*stamp-expr expr1*)
assumes *valid-value val2* (*stamp-expr expr2*)
shows *valid-value result* (*stamp-expr* (*BinaryExpr op expr1 expr2*))
⟨*proof*⟩

9.4.4 Validity of Stamp Meet and Join Operators

lemma *stamp-meet-integer-is-valid-stamp*:
assumes *valid-stamp stamp1*
assumes *valid-stamp stamp2*
assumes *is-IntegerStamp stamp1*
assumes *is-IntegerStamp stamp2*
shows *valid-stamp* (*meet stamp1 stamp2*)
⟨*proof*⟩

lemma *stamp-meet-is-valid-stamp*:
assumes *1*: *valid-stamp stamp1*
assumes *2*: *valid-stamp stamp2*
shows *valid-stamp* (*meet stamp1 stamp2*)
⟨*proof*⟩

lemma *stamp-meet-commutes*: *meet stamp1 stamp2* = *meet stamp2 stamp1*
⟨*proof*⟩

lemma *stamp-meet-is-valid-value1*:
  assumes *valid-value val stamp1*
  assumes *valid-stamp stamp2*
  assumes *stamp1 = IntegerStamp b1 lo1 hi1*
  assumes *stamp2 = IntegerStamp b2 lo2 hi2*
  assumes *meet stamp1 stamp2 ≠ IllegalStamp*
  shows *valid-value val* (*meet stamp1 stamp2*)
⟨*proof*⟩

and the symmetric lemma follows by the commutativity of meet.

lemma *stamp-meet-is-valid-value*:
  assumes *valid-value val stamp2*
  assumes *valid-stamp stamp1*
  assumes *stamp1 = IntegerStamp b1 lo1 hi1*
  assumes *stamp2 = IntegerStamp b2 lo2 hi2*
  assumes *meet stamp1 stamp2 ≠ IllegalStamp*
  shows *valid-value val* (*meet stamp1 stamp2*)
  ⟨*proof*⟩

9.4.5 Validity of conditional expressions

lemma *conditional-eval-implies-valid-value*:
  assumes [*m*,*p*] ⊢ *cond* ↦ *condv*
  assumes *expr* = (*if val-to-bool condv then expr1 else expr2*)
  assumes [*m*,*p*] ⊢ *expr* ↦ *val*
  assumes *val ≠ UndefVal*
  assumes *valid-value condv* (*stamp-expr cond*)
  assumes *valid-value val* (*stamp-expr expr*)
  assumes *compatible* (*stamp-expr expr1*) (*stamp-expr expr2*)
  shows *valid-value val* (*stamp-expr* (*ConditionalExpr cond expr1 expr2*))
⟨*proof*⟩

9.4.6 Validity of Whole Expression Tree Evaluation

TODO: find a way to encode that conditional expressions must have compatible (and valid) stamps? One approach would be for all the stamp_expr operators to require that all input stamps are valid.

definition *wf-stamp* :: *IRExpr* ⇒ *bool* **where**
  wf-stamp e = (∀ *m p v*. ([*m*, *p*] ⊢ *e* ↦ *v*) ⟶ *valid-value v* (*stamp-expr e*))

lemma *stamp-under-defn*:
  assumes *stamp-under* (*stamp-expr x*) (*stamp-expr y*)
  assumes *wf-stamp x* ∧ *wf-stamp y*
  assumes ([*m*, *p*] ⊢ *x* ↦ *xv*) ∧ ([*m*, *p*] ⊢ *y* ↦ *yv*)
  shows *val-to-bool* (*bin-eval BinIntegerLessThan xv yv*) ∨
        (*bin-eval BinIntegerLessThan xv yv*) = *UndefVal*
⟨*proof*⟩

lemma *stamp-under-defn-inverse*:
  assumes *stamp-under* (*stamp-expr y*) (*stamp-expr x*)
  assumes *wf-stamp x* ∧ *wf-stamp y*
  assumes ([*m*, *p*] ⊢ *x* ↦ *xv*) ∧ ([*m*, *p*] ⊢ *y* ↦ *yv*)
  shows ¬(*val-to-bool* (*bin-eval BinIntegerLessThan xv yv*)) ∨ (*bin-eval BinIntegerLessThan xv yv*) = *UndefVal*
⟨*proof*⟩

end

10 Optization DSL

10.1 Markup

theory *Markup*
  imports *Semantics.IRTreeEval Snippets.Snipping*
begin

datatype ′*a Rewrite* =
  Transform ′*a* ′*a* (- ⟼ - *10*) |
  Conditional ′*a* ′*a bool* (- ⟼ - *when* - *11*) |
  Sequential ′*a Rewrite* ′*a Rewrite* |
  Transitive ′*a Rewrite*

datatype ′*a ExtraNotation* =
  ConditionalNotation ′*a* ′*a* ′*a* (- *?* - : - *50*) |
  EqualsNotation ′*a* ′*a* (- *eq* -) |
  ConstantNotation ′*a* (*const* - *120*) |
  TrueNotation (*true*) |
  FalseNotation (*false*) |
  ExclusiveOr ′*a* ′*a* (- ⊕ -) |
  LogicNegationNotation ′*a* (!-) |
  ShortCircuitOr ′*a* ′*a* (- || -) |
  Remainder ′*a* ′*a* (- % -)

definition *word* :: (′*a*::*len*) *word* ⇒ ′*a word* **where**
  word x = *x*

ML-val @{*term* ‹*x* % *x*›}
ML-file ‹*markup.ML*›

10.1.1 Expression Markup

ML ‹
structure IRExprTranslator : *DSL-TRANSLATION* =
struct
fun markup DSL-Tokens.Add = @{*term BinaryExpr*} $ @{*term BinAdd*}
  | *markup DSL-Tokens.Sub* = @{*term BinaryExpr*} $ @{*term BinSub*}
  | *markup DSL-Tokens.Mul* = @{*term BinaryExpr*} $ @{*term BinMul*}

| *markup DSL-Tokens.Div* = @{*term BinaryExpr*} $ @{*term BinDiv*}
| *markup DSL-Tokens.Rem* = @{*term BinaryExpr*} $ @{*term BinMod*}
| *markup DSL-Tokens.And* = @{*term BinaryExpr*} $ @{*term BinAnd*}
| *markup DSL-Tokens.Or* = @{*term BinaryExpr*} $ @{*term BinOr*}
| *markup DSL-Tokens.Xor* = @{*term BinaryExpr*} $ @{*term BinXor*}
| *markup DSL-Tokens.ShortCircuitOr* = @{*term BinaryExpr*} $ @{*term BinShortCircuitOr*}
| *markup DSL-Tokens.Abs* = @{*term UnaryExpr*} $ @{*term UnaryAbs*}
| *markup DSL-Tokens.Less* = @{*term BinaryExpr*} $ @{*term BinIntegerLessThan*}
| *markup DSL-Tokens.Equals* = @{*term BinaryExpr*} $ @{*term BinIntegerEquals*}
| *markup DSL-Tokens.Not* = @{*term UnaryExpr*} $ @{*term UnaryNot*}
| *markup DSL-Tokens.Negate* = @{*term UnaryExpr*} $ @{*term UnaryNeg*}
| *markup DSL-Tokens.LogicNegate* = @{*term UnaryExpr*} $ @{*term UnaryLogicNegation*}
| *markup DSL-Tokens.LeftShift* = @{*term BinaryExpr*} $ @{*term BinLeftShift*}
| *markup DSL-Tokens.RightShift* = @{*term BinaryExpr*} $ @{*term BinRightShift*}
| *markup DSL-Tokens.UnsignedRightShift* = @{*term BinaryExpr*} $ @{*term BinURightShift*}
| *markup DSL-Tokens.Conditional* = @{*term ConditionalExpr*}
| *markup DSL-Tokens.Constant* = @{*term ConstantExpr*}
| *markup DSL-Tokens.TrueConstant* = @{*term ConstantExpr* (*IntVal 32 1*)}
| *markup DSL-Tokens.FalseConstant* = @{*term ConstantExpr* (*IntVal 32 0*)}
end
structure IRExprMarkup = *DSL-Markup*(*IRExprTranslator*);
›

ir expression translation

syntax *-expandExpr* :: *term* ⇒ *term* (*exp*[-])
parse-translation ‹ [(@{*syntax-const -expandExpr*} , *IRExprMarkup.markup-expr* [])] ›

ir expression example

value *exp*[(e_1 < e_2) ? e_1 : e_2]

ConditionalExpr (*BinaryExpr BinIntegerLessThan* (e_1::*IRExpr*) (e_2::*IRExpr*)) e_1 e_2

10.1.2 Value Markup

ML ‹
structure IntValTranslator : *DSL-TRANSLATION* =
struct
fun markup DSL-Tokens.Add = @{*term intval-add*}
| *markup DSL-Tokens.Sub* = @{*term intval-sub*}
| *markup DSL-Tokens.Mul* = @{*term intval-mul*}
| *markup DSL-Tokens.Div* = @{*term intval-div*}

```
  | markup DSL-Tokens.Rem = @{term intval-mod}
  | markup DSL-Tokens.And = @{term intval-and}
  | markup DSL-Tokens.Or = @{term intval-or}
  | markup DSL-Tokens.ShortCircuitOr = @{term intval-short-circuit-or}
  | markup DSL-Tokens.Xor = @{term intval-xor}
  | markup DSL-Tokens.Abs = @{term intval-abs}
  | markup DSL-Tokens.Less = @{term intval-less-than}
  | markup DSL-Tokens.Equals = @{term intval-equals}
  | markup DSL-Tokens.Not = @{term intval-not}
  | markup DSL-Tokens.Negate = @{term intval-negate}
  | markup DSL-Tokens.LogicNegate = @{term intval-logic-negation}
  | markup DSL-Tokens.LeftShift = @{term intval-left-shift}
  | markup DSL-Tokens.RightShift = @{term intval-right-shift}
  | markup DSL-Tokens.UnsignedRightShift = @{term intval-uright-shift}
  | markup DSL-Tokens.Conditional = @{term intval-conditional}
  | markup DSL-Tokens.Constant = @{term IntVal 32}
  | markup DSL-Tokens.TrueConstant = @{term IntVal 32 1}
  | markup DSL-Tokens.FalseConstant = @{term IntVal 32 0}
end
structure IntValMarkup = DSL-Markup(IntValTranslator);
›
```

value expression translation

syntax *-expandIntVal* :: *term* ⇒ *term* (*val*[-])
parse-translation ‹ [(@{*syntax-const* *-expandIntVal*} , *IntValMarkup.markup-expr* [])] ›

value expression example

value *val*[$(e_1 < e_2)$? e_1 : e_2]

intval-conditional (*intval-less-than* (e_1:: *Value*) (e_2:: *Value*)) e_1 e_2

10.1.3 Word Markup

ML ‹

```
structure WordTranslator : DSL-TRANSLATION =
struct
fun markup DSL-Tokens.Add = @{term plus}
  | markup DSL-Tokens.Sub = @{term minus}
  | markup DSL-Tokens.Mul = @{term times}
  | markup DSL-Tokens.Div = @{term signed-divide}
  | markup DSL-Tokens.Rem = @{term signed-modulo}
  | markup DSL-Tokens.And = @{term Bit-Operations.semiring-bit-operations-class.and}
  | markup DSL-Tokens.Or = @{term or}
  | markup DSL-Tokens.Xor = @{term xor}
  | markup DSL-Tokens.Abs = @{term abs}
  | markup DSL-Tokens.Less = @{term less}
```

| *markup DSL-Tokens.Equals* = @{*term HOL.eq*}
| *markup DSL-Tokens.Not* = @{*term not*}
| *markup DSL-Tokens.Negate* = @{*term uminus*}
| *markup DSL-Tokens.LogicNegate* = @{*term logic-negate*}
| *markup DSL-Tokens.LeftShift* = @{*term shiftl*}
| *markup DSL-Tokens.RightShift* = @{*term signed-shiftr*}
| *markup DSL-Tokens.UnsignedRightShift* = @{*term shiftr*}
| *markup DSL-Tokens.Constant* = @{*term word*}
| *markup DSL-Tokens.TrueConstant* = @{*term 1*}
| *markup DSL-Tokens.FalseConstant* = @{*term 0*}
end
structure WordMarkup = *DSL-Markup*(*WordTranslator*);
›

word expression translation

syntax *-expandWord* :: *term* ⇒ *term* (*bin*[-])
parse-translation ‹ [(@{*syntax-const -expandWord*} , *WordMarkup.markup-expr* [])] ›

word expression example

value *bin*[*x* & *y* | *z*]

intval-conditional (*intval-less-than* (e_1::*Value*) (e_2::*Value*)) e_1 e_2

value *bin*[−*x*]
value *val*[−*x*]
value *exp*[−*x*]

value *bin*[!*x*]
value *val*[!*x*]
value *exp*[!*x*]

value *bin*[¬*x*]
value *val*[¬*x*]
value *exp*[¬*x*]

value *bin*[~*x*]
value *val*[~*x*]
value *exp*[~*x*]

value ~*x*

end

10.2 Optimization Phases

theory *Phase*
imports *Main*

begin

ML-file *map.ML*
ML-file *phase.ML*

end

10.3 Canonicalization DSL

theory *Canonicalization*
 imports
  Markup
  Phase
  HOL−Eisbach.Eisbach
 keywords
  phase :: *thy-decl* **and**
  terminating :: *quasi-command* **and**
  print-phases :: *diag* **and**
  export-phases :: *thy-decl* **and**
  optimization :: *thy-goal-defn*
begin

print-methods

ML ‹
```
datatype 'a Rewrite =
  Transform of 'a * 'a |
  Conditional of 'a * 'a * term |
  Sequential of 'a Rewrite * 'a Rewrite |
  Transitive of 'a Rewrite

type rewrite = {
  name: binding,
  rewrite: term Rewrite,
  proofs: thm list,
  code: thm list,
  source: term
}

structure RewriteRule : Rule =
struct
type T = rewrite;

(*
fun pretty-rewrite ctxt (Transform (from, to)) =
      Pretty.block [
        Syntax.pretty-term ctxt from,
        Pretty.str ↦ ,
        Syntax.pretty-term ctxt to
```

```
      ]
  | pretty-rewrite ctxt (Conditional (from, to, cond)) =
      Pretty.block [
        Syntax.pretty-term ctxt from,
        Pretty.str  ↦ ,
        Syntax.pretty-term ctxt to,
        Pretty.str  when ,
        Syntax.pretty-term ctxt cond
      ]
  | pretty-rewrite - - = Pretty.str not implemented*)

fun pretty-thm ctxt thm =
  (Proof-Context.pretty-fact ctxt (, [thm]))

fun pretty ctxt obligations t =
  let
    val is-skipped = Thm-Deps.has-skip-proof (#proofs t);

    val warning = (if is-skipped
      then [Pretty.str (proof skipped), Pretty.brk 0]
      else []);

    val obligations = (if obligations
      then [Pretty.big-list
              obligations:
              (map (pretty-thm ctxt) (#proofs t)),
            Pretty.brk 0]
      else []);

    fun pretty-bind binding =
      Pretty.markup
        (Position.markup (Binding.pos-of binding) Markup.position)
        [Pretty.str (Binding.name-of binding)];

  in
  Pretty.block ([
    pretty-bind (#name t), Pretty.str : ,
    Syntax.pretty-term ctxt (#source t), Pretty.fbrk
  ] @ obligations @ warning)
  end
end

structure RewritePhase = DSL-Phase(RewriteRule);

val - =
  Outer-Syntax.command command-keyword‹phase› enter an optimization phase
   (Parse.binding --| Parse.$$$ terminating -- Parse.const --| Parse.begin
     >> (Toplevel.begin-main-target true o RewritePhase.setup));
```

```
fun print-phases print-obligations ctxt =
  let
    val thy = Proof-Context.theory-of ctxt;
    fun print phase = RewritePhase.pretty print-obligations phase ctxt
  in
    map print (RewritePhase.phases thy)
  end

fun print-optimizations print-obligations thy =
  print-phases print-obligations thy |> Pretty.writeln-chunks

val - =
  Outer-Syntax.command command-keyword ‹print-phases›
    print debug information for optimizations
    (Parse.opt-bang >>
      (fn b => Toplevel.keep ((print-optimizations b) o Toplevel.context-of)));

fun export-phases thy name =
  let
    val state = Toplevel.make-state (SOME thy);
    val ctxt = Toplevel.context-of state;
    val content = Pretty.string-of (Pretty.chunks (print-phases false ctxt));
    val cleaned = YXML.content-of content;


    val filename = Path.explode (name^.rules);
    val directory = Path.explode optimizations;
    val path = Path.binding (
               Path.append directory filename,
               Position.none);
    val thy' = thy |> Generated-Files.add-files (path, (Bytes.string content));

    val - = Export.export thy' path [YXML.parse cleaned];

    val - = writeln (Export.message thy' (Path.basic optimizations));
  in
    thy'
  end

val - =
  Outer-Syntax.command command-keyword ‹export-phases›
    export information about encoded optimizations
    (Parse.path >>
      (fn name => Toplevel.theory (fn state => export-phases state name)))
›
```

ML-file *rewrites.ML*

10.3.1 Semantic Preservation Obligation

fun *rewrite-preservation* :: *IRExpr Rewrite* ⇒ *bool* **where**
rewrite-preservation (*Transform x y*) = ($y \leq x$) |
rewrite-preservation (*Conditional x y cond*) = (*cond* ⟶ ($y \leq x$)) |
rewrite-preservation (*Sequential x y*) = (*rewrite-preservation x* ∧ *rewrite-preservation y*) |
rewrite-preservation (*Transitive x*) = *rewrite-preservation x*

10.3.2 Termination Obligation

fun *rewrite-termination* :: *IRExpr Rewrite* ⇒ (*IRExpr* ⇒ *nat*) ⇒ *bool* **where**
rewrite-termination (*Transform x y*) *trm* = (*trm x* > *trm y*) |
rewrite-termination (*Conditional x y cond*) *trm* = (*cond* ⟶ (*trm x* > *trm y*)) |
rewrite-termination (*Sequential x y*) *trm* = (*rewrite-termination x trm* ∧ *rewrite-termination y trm*) |
rewrite-termination (*Transitive x*) *trm* = *rewrite-termination x trm*

fun *intval* :: *Value Rewrite* ⇒ *bool* **where**
intval (*Transform x y*) = ($x \neq$ *UndefVal* ∧ $y \neq$ *UndefVal* ⟶ $x = y$) |
intval (*Conditional x y cond*) = (*cond* ⟶ ($x = y$)) |
intval (*Sequential x y*) = (*intval x* ∧ *intval y*) |
intval (*Transitive x*) = *intval x*

10.3.3 Standard Termination Measure

fun *size* :: *IRExpr* ⇒ *nat* **where**
unary-size:
size (*UnaryExpr op x*) = (*size x*) + *2* |

bin-const-size:
size (*BinaryExpr op x* (*ConstantExpr cy*)) = (*size x*) + *2* |
bin-size:
size (*BinaryExpr op x y*) = (*size x*) + (*size y*) + *2* |
cond-size:
size (*ConditionalExpr c t f*) = (*size c*) + (*size t*) + (*size f*) + *2* |
const-size:
size (*ConstantExpr c*) = *1* |
param-size:
size (*ParameterExpr ind s*) = *2* |
leaf-size:
size (*LeafExpr nid s*) = *2* |
size (*ConstantVar c*) = *2* |
size (*VariableExpr x s*) = *2*

10.3.4 Automated Tactics

named-theorems *size-simps size simplication rules*

method *unfold-optimization* =

(*unfold rewrite-preservation.simps*, *unfold rewrite-termination.simps*,
  unfold intval.simps,
  rule conjE, *simp*, *simp del*: *le-expr-def*, *force?*)
| (*unfold rewrite-preservation.simps*, *unfold rewrite-termination.simps*,
  rule conjE, *simp*, *simp del*: *le-expr-def*, *force?*)

method *unfold-size* =
(((*unfold size.simps*, *simp add*: *size-simps del*: *le-expr-def*)*?*
; (*simp add*: *size-simps del*: *le-expr-def*)*?*
; (*auto simp*: *size-simps*)*?*
; (*unfold size.simps*)*?*)[*1*])

print-methods

ML ‹
structure System : *RewriteSystem* =
struct
val preservation = @{*const rewrite-preservation*};
val termination = @{*const rewrite-termination*};
val intval = @{*const intval*};
end

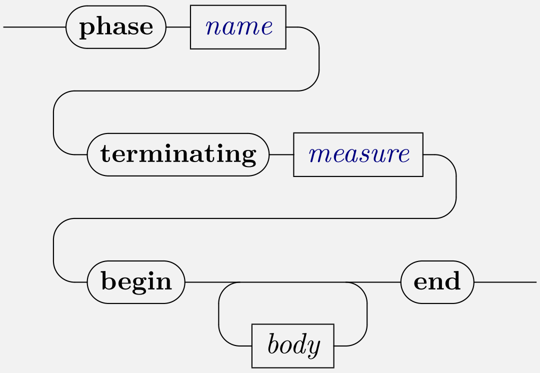
structure DSL = *DSL-Rewrites*(*System*);

val - =
  Outer-Syntax.local-theory-to-proof ***command-keyword***‹*optimization*›
    define an optimization and open proof obligation
    (*Parse-Spec.thm-name* : −− *Parse.term*
       >> *DSL.rewrite-cmd*);
›

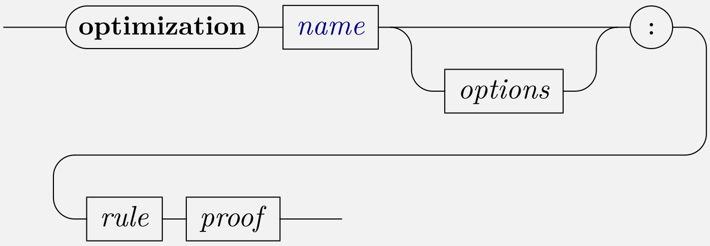
ML-file ~~/*src*/*Doc*/*antiquote-setup.ML*

PhaseRail

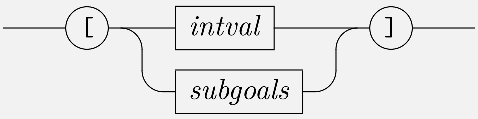
phase



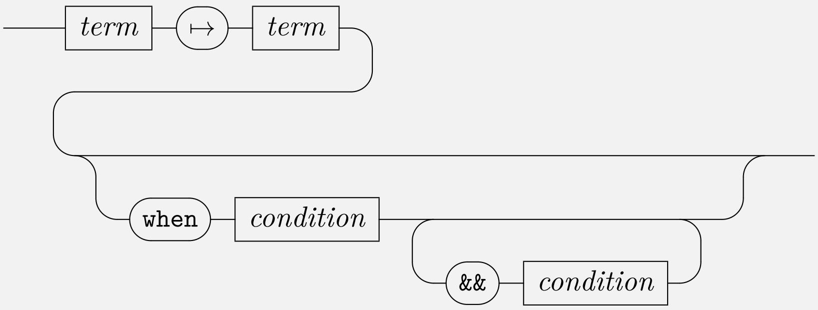
optimization



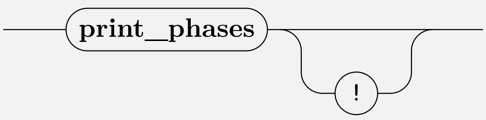
options



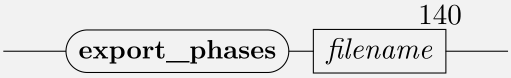
rule



print-phases



export-phases



gencode

phase *name terminating measure* opens a new optimization phase

print-syntax

end

11 Canonicalization Optimizations

theory *Common*
  imports
    OptimizationDSL.Canonicalization
    Semantics.IRTreeEvalThms
begin

lemma *size-pos*[*size-simps*]: $0 < size\ y$
  ⟨*proof*⟩

lemma *size-non-add*[*size-simps*]: $size\ (BinaryExpr\ op\ a\ b) = size\ a + size\ b + 2 \longleftrightarrow \neg(is\text{-}ConstantExpr\ b)$
  ⟨*proof*⟩

lemma *size-non-const*[*size-simps*]:
  $\neg\ is\text{-}ConstantExpr\ y \Longrightarrow 1 < size\ y$
  ⟨*proof*⟩

lemma *size-binary-const*[*size-simps*]:
  $size\ (BinaryExpr\ op\ a\ b) = size\ a + 2 \longleftrightarrow (is\text{-}ConstantExpr\ b)$
  ⟨*proof*⟩

lemma *size-flip-binary*[*size-simps*]:
  $\neg(is\text{-}ConstantExpr\ y) \longrightarrow size\ (BinaryExpr\ op\ (ConstantExpr\ x)\ y) > size\ (BinaryExpr\ op\ y\ (ConstantExpr\ x))$
  ⟨*proof*⟩

lemma *size-binary-lhs-a*[*size-simps*]:
  $size\ (BinaryExpr\ op\ (BinaryExpr\ op'\ a\ b)\ c) > size\ a$
  ⟨*proof*⟩

lemma *size-binary-lhs-b*[*size-simps*]:
  $size\ (BinaryExpr\ op\ (BinaryExpr\ op'\ a\ b)\ c) > size\ b$
  ⟨*proof*⟩

lemma *size-binary-lhs-c*[*size-simps*]:
  $size\ (BinaryExpr\ op\ (BinaryExpr\ op'\ a\ b)\ c) > size\ c$
  ⟨*proof*⟩

lemma *size-binary-rhs-a*[*size-simps*]:
  $size\ (BinaryExpr\ op\ c\ (BinaryExpr\ op'\ a\ b)) > size\ a$
  ⟨*proof*⟩

lemma *size-binary-rhs-b*[*size-simps*]:

size (*BinaryExpr op c* (*BinaryExpr op′ a b*)) > *size b*
⟨*proof*⟩

lemma *size-binary-rhs-c*[*size-simps*]:
size (*BinaryExpr op c* (*BinaryExpr op′ a b*)) > *size c*
⟨*proof*⟩

lemma *size-binary-lhs*[*size-simps*]:
size (*BinaryExpr op x y*) > *size x*
⟨*proof*⟩

lemma *size-binary-rhs*[*size-simps*]:
size (*BinaryExpr op x y*) > *size y*
⟨*proof*⟩

lemmas *arith*[*size-simps*] = *Suc-leI add-strict-increasing order-less-trans trans-less-add2*

definition *well-formed-equal* :: *Value* ⇒ *Value* ⇒ *bool*
(**infix** ≈ *50*) **where**
well-formed-equal v_1 v_2 = (v_1 ≠ *UndefVal* ⟶ v_1 = v_2)

lemma *well-formed-equal-defn* [*simp*]:
well-formed-equal v_1 v_2 = (v_1 ≠ *UndefVal* ⟶ v_1 = v_2)
⟨*proof*⟩

end

11.1 AbsNode Phase

theory *AbsPhase*
imports
Common Proofs.StampEvalThms
begin

phase *AbsNode*
terminating *size*
begin

Note:

We can't use (<*s*) for reasoning about *intval-less-than*. (<*s*) will always treat the 64^{th} bit as the sign flag while *intval-less-than* uses the b^{th} bit depending on the size of the word.

value *val*[*new-int 32 0* < *new-int 32 4294967286*] — 0 < -10 = False
value (*0*::*int64*) <*s 4294967286* — 0 < 4294967286 = True

lemma *signed-eqiv*:
assumes *b* > *0* ∧ *b* ≤ *64*

shows *val-to-bool* (*val*[*new-int b v* < *new-int b v'*]) = (*int-signed-value b v* < *int-signed-value b v'*)
⟨*proof*⟩

lemma *val-abs-pos*:
assumes *val-to-bool*(*val*[(*new-int b 0*) < (*new-int b v*)])
shows *intval-abs* (*new-int b v*) = (*new-int b v*)
⟨*proof*⟩

lemma *val-abs-neg*:
assumes *val-to-bool*(*val*[(*new-int b v*) < (*new-int b 0*)])
shows *intval-abs* (*new-int b v*) = *intval-negate* (*new-int b v*)
⟨*proof*⟩

lemma *val-bool-unwrap*:
val-to-bool (*bool-to-val v*) = *v*
⟨*proof*⟩

lemma *take-bit-64*:
assumes *0* < *b* ∧ *b* ≤ *64*
assumes *take-bit b v* = *v*
shows *take-bit 64 v* = *take-bit b v*
⟨*proof*⟩

A special value exists for the maximum negative integer as its negation is itself. We can define the value as *set-bit* ((*b*::*nat*) − (*1*::*nat*)) (*0*::*64 word*) for any bit-width, b.

value (*set-bit 1 0*)::*2 word* — 2
value −(*set-bit 1 0*)::*2 word* — 2
value (*set-bit 31 0*)::*32 word* — 2147483648
value −(*set-bit 31 0*)::*32 word* — 2147483648

lemma *negative-def*:
fixes *v* :: *'a*::*len word*
assumes *v* <*s 0*
shows *bit v* (*LENGTH*(*'a*) − *1*)
⟨*proof*⟩

lemma *positive-def*:
fixes *v* :: *'a*::*len word*
assumes *0* <*s v*
shows ¬(*bit v* (*LENGTH*(*'a*) − *1*))
⟨*proof*⟩

lemma *negative-lower-bound*:

fixes v :: $'a$::*len word*
assumes $(2\widehat{\ }(LENGTH('a) - 1)) <s\ v$
assumes $v <s\ 0$
shows $0 <s\ (-v)$
⟨*proof*⟩

lemma *min-int*:
fixes x :: $'a$::*len word*
assumes $x <s\ 0$
assumes $x \neq (2\widehat{\ }(LENGTH('a) - 1))$
shows $2\widehat{\ }(LENGTH('a) - 1) <s\ x$
⟨*proof*⟩

lemma *negate-min-int*:
fixes v :: $'a$::*len word*
assumes $v = (2\widehat{\ }(LENGTH('a) - 1))$
shows $v = (-v)$
⟨*proof*⟩

fun *abs* :: $'a$::*len word* $\Rightarrow$ $'a$ *word* **where**
abs x = (*if* $x <s\ 0$ *then* $(-x)$ *else* x)

lemma
abs(*abs*(x)) = *abs*(x)
for x :: $'a$::*len word*
⟨*proof*⟩

We need to do the same proof at the value level.

lemma *invert-intval*:
assumes *int-signed-value* $b\ v < 0$
assumes $b > 0 \wedge b \leq 64$
assumes *take-bit* $b\ v = v$
assumes $v \neq (2\widehat{\ }(b - 1))$
shows $0 <$ *int-signed-value* $b\ (-v)$
⟨*proof*⟩

lemma *negate-max-negative*:
assumes $b > 0 \wedge b \leq 64$
assumes *take-bit* $b\ v = v$
assumes $v = (2\widehat{\ }(b - 1))$
shows *new-int* $b\ v$ = *intval-negate* (*new-int* $b\ v$)
⟨*proof*⟩

lemma *val-abs-always-pos*:
assumes $b > 0 \wedge b \leq 64$
assumes *take-bit* $b\ v = v$
assumes $v \neq (2\widehat{\ }(b - 1))$

assumes *intval-abs* (*new-int b v*) = (*new-int b v'*)
shows *val-to-bool* (*val*[(*new-int b 0*) < (*new-int b v'*)]) ∨ *val-to-bool* (*val*[(*new-int b 0*) *eq* (*new-int b v'*)])
⟨*proof*⟩

lemma *intval-abs-elims*:
assumes *intval-abs x* ≠ *UndefVal*
shows ∃ *t v* . *x* = *IntVal t v* ∧
intval-abs x = *new-int t* (*if int-signed-value t v* < *0 then* − *v else v*)
⟨*proof*⟩

lemma *wf-abs-new-int*:
assumes *intval-abs* (*IntVal t v*) ≠ *UndefVal*
shows *intval-abs* (*IntVal t v*) = *new-int t v* ∨ *intval-abs* (*IntVal t v*) = *new-int t* (−*v*)
⟨*proof*⟩

lemma *mono-undef-abs*:
assumes *intval-abs* (*intval-abs x*) ≠ *UndefVal*
shows *intval-abs x* ≠ *UndefVal*
⟨*proof*⟩

lemma *val-abs-idem*:
assumes *valid-value x* (*IntegerStamp b l h*)
assumes *val*[*abs*(*abs*(*x*))] ≠ *UndefVal*
shows *val*[*abs*(*abs*(*x*))] = *val*[*abs x*]
⟨*proof*⟩

Optimisations **end**

end

11.2 AddNode Phase

theory *AddPhase*
imports
Common
begin

phase *AddNode*
terminating *size*
begin

lemma *binadd-commute*:
assumes *bin-eval BinAdd x y* ≠ *UndefVal*
shows *bin-eval BinAdd x y* = *bin-eval BinAdd y x*
⟨*proof*⟩

optimization *AddShiftConstantRight*: ((*const v*) + *y*) $\longmapsto$ *y* + (*const v*) *when* ¬(*is-ConstantExpr y*)
⟨*proof*⟩

optimization *AddShiftConstantRight2*: ((*const v*) + *y*) $\longmapsto$ *y* + (*const v*) *when* ¬(*is-ConstantExpr y*)
⟨*proof*⟩

lemma *is-neutral-0* [*simp*]:
assumes *val*[(*IntVal b x*) + (*IntVal b 0*)] ≠ *UndefVal*
shows *val*[(*IntVal b x*) + (*IntVal b 0*)] = (*new-int b x*)
⟨*proof*⟩

lemma *AddNeutral-Exp*:
shows *exp*[(*e* + (*const* (*IntVal 32 0*)))] ≥ *exp*[*e*]
⟨*proof*⟩

optimization *AddNeutral*: (*e* + (*const* (*IntVal 32 0*))) $\longmapsto$ *e*
⟨*proof*⟩

ML-val ‹@{*term* ‹*x* = *y*›}›

lemma *NeutralLeftSubVal*:
assumes *e1* = *new-int b ival*
shows *val*[(*e1* − *e2*) + *e2*] ≈ *e1*
⟨*proof*⟩

lemma *RedundantSubAdd-Exp*:
shows *exp*[((*a* − *b*) + *b*)] ≥ *a*
⟨*proof*⟩

optimization *RedundantSubAdd*: $((e_1 - e_2) + e_2) \longmapsto e_1$
⟨*proof*⟩

lemma *allE2*: (∀ *x y*. *P x y*) ⟹ (*P a b* ⟹ *R*) ⟹ *R*
⟨*proof*⟩

lemma *just-goal2*:
assumes (∀ *a b*. (*val*[(*a* − *b*) + *b*] ≠ *UndefVal* ∧ *a* ≠ *UndefVal* ⟶
val[(*a* − *b*) + *b*] = *a*))
shows $(exp[(e_1 - e_2) + e_2]) \geq e_1$
⟨*proof*⟩

optimization *RedundantSubAdd2*: $e_2 + (e_1 - e_2) \longmapsto e_1$

⟨*proof*⟩

lemma *AddToSubHelperLowLevel*:
shows *val*[$-e$ + *y*] = *val*[*y* − *e*] (**is** *?x* = *?y*)
⟨*proof*⟩

print-phases

lemma *val-redundant-add-sub*:
assumes *a* = *new-int bb ival*
assumes *val*[*b* + *a*] ≠ *UndefVal*
shows *val*[(*b* + *a*) − *b*] = *a*
⟨*proof*⟩

lemma *val-add-right-negate-to-sub*:
assumes *val*[*x* + *e*] ≠ *UndefVal*
shows *val*[*x* + (−*e*)] = *val*[*x* − *e*]
⟨*proof*⟩

lemma *exp-add-left-negate-to-sub*:
exp[−*e* + *y*] ≥ *exp*[*y* − *e*]
⟨*proof*⟩

lemma *RedundantAddSub-Exp*:
shows *exp*[(*b* + *a*) − *b*] ≥ *a*
⟨*proof*⟩

Optimisations

optimization *RedundantAddSub*: $(b + a) - b \longmapsto a$
⟨*proof*⟩

optimization *AddRightNegateToSub*: $x + -e \longmapsto x - e$
⟨*proof*⟩

optimization *AddLeftNegateToSub*: $-e + y \longmapsto y - e$
⟨*proof*⟩

end

end

11.3 AndNode Phase

theory *AndPhase*
imports
Common
Proofs.StampEvalThms
begin

context *stamp-mask*
begin

lemma *AndCommute-Val*:
assumes $val[x \& y] \neq UndefVal$
shows $val[x \& y] = val[y \& x]$
⟨*proof*⟩

lemma *AndCommute-Exp*:
shows $exp[x \& y] \geq exp[y \& x]$
⟨*proof*⟩

lemma *AndRightFallthrough*: $(((and\ (not\ (\downarrow x))\ (\uparrow y)) = 0)) \longrightarrow exp[x \& y] \geq exp[y]$
⟨*proof*⟩

lemma *AndLeftFallthrough*: $(((and\ (not\ (\downarrow y))\ (\uparrow x)) = 0)) \longrightarrow exp[x \& y] \geq exp[x]$
⟨*proof*⟩

end

phase *AndNode*
terminating *size*
begin

lemma *bin-and-nots*:
$(\sim x \& \sim y) = (\sim(x \mid y))$
⟨*proof*⟩

lemma *bin-and-neutral*:
$(x \& \sim False) = x$
⟨*proof*⟩

lemma *val-and-equal*:
assumes $x = \textit{new-int } b\ v$
and $\textit{val}[x\ \&\ x] \neq \textit{UndefVal}$
shows $\textit{val}[x\ \&\ x] = x$
⟨*proof*⟩

lemma *val-and-nots*:
$\textit{val}[\sim x\ \&\ \sim y] = \textit{val}[\sim(x\ |\ y)]$
⟨*proof*⟩

lemma *val-and-neutral*:
assumes $x = \textit{new-int } b\ v$
and $\textit{val}[x\ \&\ \sim(\textit{new-int } b'\ 0)] \neq \textit{UndefVal}$
shows $\textit{val}[x\ \&\ \sim(\textit{new-int } b'\ 0)] = x$
⟨*proof*⟩

lemma *val-and-zero*:
assumes $x = \textit{new-int } b\ v$
shows $\textit{val}[x\ \&\ (\textit{IntVal } b\ 0)] = \textit{IntVal } b\ 0$
⟨*proof*⟩

lemma *exp-and-equal*:
$\textit{exp}[x\ \&\ x] \geq \textit{exp}[x]$
⟨*proof*⟩

lemma *exp-and-nots*:
$\textit{exp}[\sim x\ \&\ \sim y] \geq \textit{exp}[\sim(x\ |\ y)]$
⟨*proof*⟩

lemma *exp-sign-extend*:
assumes $e = (1 << \textit{In}) - 1$
shows $\textit{BinaryExpr BinAnd } (\textit{UnaryExpr } (\textit{UnarySignExtend In Out})\ x)$
$(\textit{ConstantExpr } (\textit{new-int } b\ e))$
$\geq (\textit{UnaryExpr } (\textit{UnaryZeroExtend In Out})\ x)$
⟨*proof*⟩

lemma *exp-and-neutral*:
assumes $\textit{wf-stamp } x$
assumes $\textit{stamp-expr } x = \textit{IntegerStamp } b\ \textit{lo}\ \textit{hi}$
shows $\textit{exp}[(x\ \&\ \sim(\textit{const } (\textit{IntVal } b\ 0)))] \geq x$
⟨*proof*⟩

lemma *val-and-commute*[*simp*]:
  val[*x* & *y*] = *val*[*y* & *x*]
  ⟨*proof*⟩

Optimisations

optimization *AndEqual*: *x* & *x* ⟼ *x*
  ⟨*proof*⟩

optimization *AndShiftConstantRight*: ((*const x*) & *y*) ⟼ *y* & (*const x*)
  when ¬(*is-ConstantExpr y*)
  ⟨*proof*⟩

optimization *AndNots*: (~*x*) & (~*y*) ⟼ ~(*x* | *y*)
  ⟨*proof*⟩

optimization *AndSignExtend*: *BinaryExpr BinAnd* (*UnaryExpr* (*UnarySignExtend In Out*) (*x*))
  (*const* (*new-int b e*))
  ⟼ (*UnaryExpr* (*UnaryZeroExtend In Out*) (*x*))
  when (*e* = (*1* << *In*) − *1*)
  ⟨*proof*⟩

optimization *AndNeutral*: (*x* & ~(*const* (*IntVal b 0*))) ⟼ *x*
  when (*wf-stamp x* ∧ *stamp-expr x* = *IntegerStamp b lo hi*)
  ⟨*proof*⟩

optimization *AndRightFallThrough*: (*x* & *y*) ⟼ *y*
  when (((*and* (*not* (*IRExpr-down x*)) (*IRExpr-up y*)) = *0*))
  ⟨*proof*⟩

optimization *AndLeftFallThrough*: (*x* & *y*) ⟼ *x*
  when (((*and* (*not* (*IRExpr-down y*)) (*IRExpr-up x*)) = *0*))
  ⟨*proof*⟩

end

end

11.4 BinaryNode Phase

theory *BinaryNode*
  imports
    Common
begin

phase *BinaryNode*

terminating *size*
begin

optimization *BinaryFoldConstant*: *BinaryExpr op* (*const v1*) (*const v2*) $\longmapsto$ *ConstantExpr* (*bin-eval op v1 v2*)
⟨*proof*⟩

end

end

11.5 ConditionalNode Phase

theory *ConditionalPhase*
imports
Common
Proofs.StampEvalThms
begin

phase *ConditionalNode*
terminating *size*
begin

lemma *negates*: $\exists\, v\ b.\ e = IntVal\ b\ v \wedge b > 0 \Longrightarrow$ *val-to-bool* (*val*[*e*]) $\longleftrightarrow$ $\neg$(*val-to-bool* (*val*[!*e*]))
⟨*proof*⟩

lemma *negation-condition-intval*:
assumes $e = IntVal\ b\ ie$
assumes $0 < b$
shows *val*[(!*e*) *?* *x* : *y*] = *val*[*e* *?* *y* : *x*]
⟨*proof*⟩

lemma *negation-preserve-eval*:
assumes $[m, p] \vdash exp[!e] \mapsto v$
shows $\exists\, v'.\ ([m, p] \vdash exp[e] \mapsto v') \wedge v = val[!v']$
⟨*proof*⟩

lemma *negation-preserve-eval-intval*:
assumes $[m, p] \vdash exp[!e] \mapsto v$
shows $\exists\, v'\ b\ vv.\ ([m, p] \vdash exp[e] \mapsto v') \wedge v' = IntVal\ b\ vv \wedge b > 0$
⟨*proof*⟩

optimization *NegateConditionFlipBranches*: ((!*e*) *?* *x* : *y*) $\longmapsto$ (*e* *?* *y* : *x*)
⟨*proof*⟩

optimization *DefaultTrueBranch*: (*true* *?* *x* : *y*) $\longmapsto$ *x* ⟨*proof*⟩

optimization *DefaultFalseBranch*: (*false* *?* *x* : *y*) $\longmapsto$ *y* ⟨*proof*⟩

optimization *ConditionalEqualBranches*: $(e\ ?\ x : x) \longmapsto x$ ⟨*proof*⟩

optimization *condition-bounds-x*: $((u < v)\ ?\ x : y) \longmapsto x$
when (*stamp-under* (*stamp-expr u*) (*stamp-expr v*) ∧ *wf-stamp u* ∧ *wf-stamp v*)
⟨*proof*⟩

optimization *condition-bounds-y*: $((u < v)\ ?\ x : y) \longmapsto y$
when (*stamp-under* (*stamp-expr v*) (*stamp-expr u*) ∧ *wf-stamp u* ∧ *wf-stamp v*)
⟨*proof*⟩

lemma *val-optimise-integer-test*:
assumes ∃ *v*. *x* = *IntVal 32 v*
shows *val*[((*x* & (*IntVal 32 1*)) *eq* (*IntVal 32 0*)) ? (*IntVal 32 0*) : (*IntVal 32 1*)]
=
val[*x* & *IntVal 32 1*]
⟨*proof*⟩

optimization *ConditionalEliminateKnownLess*: $((x < y)\ ?\ x : y) \longmapsto x$
when (*stamp-under* (*stamp-expr x*) (*stamp-expr y*)
∧ *wf-stamp x* ∧ *wf-stamp y*)
⟨*proof*⟩

lemma *ExpIntBecomesIntVal*:
assumes *stamp-expr x* = *IntegerStamp b xl xh*
assumes *wf-stamp x*
assumes *valid-value v* (*IntegerStamp b xl xh*)
assumes [*m*,*p*] ⊢ *x* ↦ *v*
shows ∃ *xv*. *v* = *IntVal b xv*
⟨*proof*⟩

lemma *intval-self-is-true*:
assumes *yv* ≠ *UndefVal*
assumes *yv* = *IntVal b yvv*
shows *intval-equals yv yv* = *IntVal 32 1*
⟨*proof*⟩

lemma *intval-commute*:
assumes *intval-equals yv xv* ≠ *UndefVal*
assumes *intval-equals xv yv* ≠ *UndefVal*
shows *intval-equals yv xv* = *intval-equals xv yv*
⟨*proof*⟩

definition *isBoolean* :: *IRExpr* ⇒ *bool* **where**

isBoolean e = (∀ *m p cond*. ((([*m*,*p*] ⊢ *e* ↦ *cond*) ⟶ (*cond* ∈ {*IntVal 32 0*, *IntVal 32 1*})))

lemma *preserveBoolean*:
  assumes *isBoolean c*
  shows *isBoolean exp*[!*c*]
  ⟨*proof*⟩

optimization *ConditionalIntegerEquals-1*: *exp*[*BinaryExpr BinIntegerEquals* (*c ? x* : *y*) (*x*)] ⟼ *c*
  when stamp-expr x = *IntegerStamp b xl xh* ∧ *wf-stamp x* ∧
  stamp-expr y = *IntegerStamp b yl yh* ∧ *wf-stamp y* ∧
  (*alwaysDistinct* (*stamp-expr x*) (*stamp-expr y*)) ∧
  isBoolean c
  ⟨*proof*⟩

lemma *negation-preserve-eval0*:
  assumes [*m*, *p*] ⊢ *exp*[*e*] ↦ *v*
  assumes *isBoolean e*
  shows ∃ *v′*. ([*m*, *p*] ⊢ *exp*[!*e*] ↦ *v′*)
  ⟨*proof*⟩

lemma *negation-preserve-eval2*:
  assumes ([*m*, *p*] ⊢ *exp*[*e*] ↦ *v*)
  assumes (*isBoolean e*)
  shows ∃ *v′*. ([*m*, *p*] ⊢ *exp*[!*e*] ↦ *v′*) ∧ *v* = *val*[!*v′*]
  ⟨*proof*⟩

optimization *ConditionalIntegerEquals-2*: *exp*[*BinaryExpr BinIntegerEquals* (*c ? x* : *y*) (*y*)] ⟼ (!*c*)
  when stamp-expr x = *IntegerStamp b xl xh* ∧ *wf-stamp x* ∧
  stamp-expr y = *IntegerStamp b yl yh* ∧ *wf-stamp y* ∧
  (*alwaysDistinct* (*stamp-expr x*) (*stamp-expr y*)) ∧
  isBoolean c
  ⟨*proof*⟩

optimization *ConditionalExtractCondition*: *exp*[(*c ? true* : *false*)] ⟼ *c*
  when isBoolean c
  ⟨*proof*⟩

optimization *ConditionalExtractCondition2*: *exp*[(*c ? false* : *true*)] ⟼ !*c*
  when isBoolean c

⟨*proof*⟩

optimization *ConditionalEqualIsRHS*: ((*x eq y*) *?* *x* : *y*) ⟼ *y*
⟨*proof*⟩

optimization *normalizeX*: ((*x eq const* (*IntVal 32 0*)) *?*
(*const* (*IntVal 32 0*)) : (*const* (*IntVal 32 1*))) ⟼ *x*
when stamp-expr x = *IntegerStamp 32 0 1* ∧ *wf-stamp x* ∧
isBoolean x
⟨*proof*⟩

optimization *normalizeX2*: ((*x eq* (*const* (*IntVal 32 1*))) *?*
(*const* (*IntVal 32 1*)) : (*const* (*IntVal 32 0*))) ⟼ *x*
when (*x* = *ConstantExpr* (*IntVal 32 0*) |
(*x* = *ConstantExpr* (*IntVal 32 1*))) ⟨*proof*⟩

optimization *flipX*: ((*x eq* (*const* (*IntVal 32 0*))) *?*
(*const* (*IntVal 32 1*)) : (*const* (*IntVal 32 0*))) ⟼ *x* ⊕ (*const*
(*IntVal 32 1*))
when (*x* = *ConstantExpr* (*IntVal 32 0*) |
(*x* = *ConstantExpr* (*IntVal 32 1*))) ⟨*proof*⟩

optimization *flipX2*: ((*x eq* (*const* (*IntVal 32 1*))) *?*
(*const* (*IntVal 32 0*)) : (*const* (*IntVal 32 1*))) ⟼ *x* ⊕ (*const*
(*IntVal 32 1*))
when (*x* = *ConstantExpr* (*IntVal 32 0*) |
(*x* = *ConstantExpr* (*IntVal 32 1*))) ⟨*proof*⟩

lemma *stamp-of-default*:
assumes *stamp-expr x* = *default-stamp*
assumes *wf-stamp x*
shows ([*m*, *p*] ⊢ *x* ↦ *v*) ⟶ (∃ *vv*. *v* = *IntVal 32 vv*)
⟨*proof*⟩

optimization *OptimiseIntegerTest*:
(((*x* & (*const* (*IntVal 32 1*))) *eq* (*const* (*IntVal 32 0*))) *?*
(*const* (*IntVal 32 0*)) : (*const* (*IntVal 32 1*))) ⟼
x & (*const* (*IntVal 32 1*))
when (*stamp-expr x* = *default-stamp* ∧ *wf-stamp x*)
⟨*proof*⟩

optimization *opt-optimise-integer-test-2*:
(((*x* & (*const* (*IntVal 32 1*))) *eq* (*const* (*IntVal 32 0*))) *?*
(*const* (*IntVal 32 0*)) : (*const* (*IntVal 32 1*))) ⟼ *x*

when (*x* = *ConstantExpr* (*IntVal 32 0*) | (*x* = *ConstantExpr* (*IntVal 32 1*))) ⟨*proof*⟩

end

end

11.6 MulNode Phase

theory *MulPhase*
  imports
    Common
    Proofs.StampEvalThms
begin

fun *mul-size* :: *IRExpr* ⇒ *nat* **where**
  mul-size (*UnaryExpr op e*) = (*mul-size e*) + *2* |
  mul-size (*BinaryExpr BinMul x y*) = ((*mul-size x*) + (*mul-size y*) + *2*) ∗ *2* |
  mul-size (*BinaryExpr op x y*) = (*mul-size x*) + (*mul-size y*) + *2* |
  mul-size (*ConditionalExpr cond t f*) = (*mul-size cond*) + (*mul-size t*) + (*mul-size f*) + *2* |
  mul-size (*ConstantExpr c*) = *1* |
  mul-size (*ParameterExpr ind s*) = *2* |
  mul-size (*LeafExpr nid s*) = *2* |
  mul-size (*ConstantVar c*) = *2* |
  mul-size (*VariableExpr x s*) = *2*

phase *MulNode*
  terminating *mul-size*
begin

lemma *bin-eliminate-redundant-negative*:
  uminus (*x* :: ′*a*::*len word*) ∗ *uminus* (*y* :: ′*a*::*len word*) = *x* ∗ *y*
  ⟨*proof*⟩

lemma *bin-multiply-identity*:
 (*x* :: ′*a*::*len word*) ∗ *1* = *x*
  ⟨*proof*⟩

lemma *bin-multiply-eliminate*:
 (*x* :: ′*a*::*len word*) ∗ *0* = *0*
  ⟨*proof*⟩

lemma *bin-multiply-negative*:
$(x :: 'a::len\ word) * uminus\ 1 = uminus\ x$
⟨*proof*⟩

lemma *bin-multiply-power-2*:
$(x:: 'a::len\ word) * (2\hat{}j) = x << j$
⟨*proof*⟩

lemma *take-bit64*[*simp*]:
fixes $w :: int64$
shows $take\text{-}bit\ 64\ w = w$
⟨*proof*⟩

lemma *mergeTakeBit*:
fixes $a :: nat$
fixes $b\ c :: 64\ word$
shows $take\text{-}bit\ a\ (take\text{-}bit\ a\ (b) * take\text{-}bit\ a\ (c)) =$
$take\text{-}bit\ a\ (b * c)$
⟨*proof*⟩

lemma *val-eliminate-redundant-negative*:
assumes $val[-x * -y] \neq UndefVal$
shows $val[-x * -y] = val[x * y]$
⟨*proof*⟩

lemma *val-multiply-neutral*:
assumes $x = new\text{-}int\ b\ v$
shows $val[x * (IntVal\ b\ 1)] = x$
⟨*proof*⟩

lemma *val-multiply-zero*:
assumes $x = new\text{-}int\ b\ v$
shows $val[x * (IntVal\ b\ 0)] = IntVal\ b\ 0$
⟨*proof*⟩

lemma *val-multiply-negative*:
assumes $x = new\text{-}int\ b\ v$
shows $val[x * -(IntVal\ b\ 1)] = val[-x]$
⟨*proof*⟩

lemma *val-MulPower2*:
fixes $i :: 64\ word$
assumes $y = IntVal\ 64\ (2\ \hat{}\ unat(i))$
and $0 < i$

and $i < 64$
and $val[x * y] \neq UndefVal$
shows $val[x * y] = val[x << IntVal\ 64\ i]$
⟨*proof*⟩

lemma *val-MulPower2Add1*:
fixes $i :: 64\ word$
assumes $y = IntVal\ 64\ ((2\ \hat{}\ unat(i)) + 1)$
and $0 < i$
and $i < 64$
and $val\text{-}to\text{-}bool(val[IntVal\ 64\ 0 < x])$
and $val\text{-}to\text{-}bool(val[IntVal\ 64\ 0 < y])$
shows $val[x * y] = val[(x << IntVal\ 64\ i) + x]$
⟨*proof*⟩

lemma *val-MulPower2Sub1*:
fixes $i :: 64\ word$
assumes $y = IntVal\ 64\ ((2\ \hat{}\ unat(i)) - 1)$
and $0 < i$
and $i < 64$
and $val\text{-}to\text{-}bool(val[IntVal\ 64\ 0 < x])$
and $val\text{-}to\text{-}bool(val[IntVal\ 64\ 0 < y])$
shows $val[x * y] = val[(x << IntVal\ 64\ i) - x]$
⟨*proof*⟩

lemma *val-distribute-multiplication*:
assumes $x = IntVal\ b\ xx \wedge q = IntVal\ b\ qq \wedge a = IntVal\ b\ aa$
assumes $val[x * (q + a)] \neq UndefVal$
assumes $val[(x * q) + (x * a)] \neq UndefVal$
shows $val[x * (q + a)] = val[(x * q) + (x * a)]$
⟨*proof*⟩

lemma *val-distribute-multiplication64*:
assumes $x = new\text{-}int\ 64\ xx \wedge q = new\text{-}int\ 64\ qq \wedge a = new\text{-}int\ 64\ aa$
shows $val[x * (q + a)] = val[(x * q) + (x * a)]$
⟨*proof*⟩

lemma *val-MulPower2AddPower2*:
fixes $i\ j :: 64\ word$
assumes $y = IntVal\ 64\ ((2\ \hat{}\ unat(i)) + (2\ \hat{}\ unat(j)))$
and $0 < i$
and $0 < j$
and $i < 64$
and $j < 64$
and $x = new\text{-}int\ 64\ xx$

shows $val[x * y] = val[(x << IntVal\ 64\ i) + (x << IntVal\ 64\ j)]$
⟨*proof*⟩

thm-oracles *val-MulPower2AddPower2*

lemma *exp-multiply-zero-64*:
shows $exp[x * (const\ (IntVal\ b\ 0))] \geq ConstantExpr\ (IntVal\ b\ 0)$
⟨*proof*⟩

lemma *exp-multiply-neutral*:
$exp[x * (const\ (IntVal\ b\ 1))] \geq x$
⟨*proof*⟩

thm-oracles *exp-multiply-neutral*

lemma *exp-multiply-negative*:
$exp[x * -(const\ (IntVal\ b\ 1))] \geq exp[-x]$
⟨*proof*⟩

lemma *exp-MulPower2*:
fixes $i :: 64\ word$
assumes $y = ConstantExpr\ (IntVal\ 64\ (2\ \hat{}\ unat(i)))$
and $0 < i$
and $i < 64$
and $exp[x > (const\ IntVal\ b\ 0)]$
and $exp[y > (const\ IntVal\ b\ 0)]$
shows $exp[x * y] \geq exp[x << ConstantExpr\ (IntVal\ 64\ i)]$
⟨*proof*⟩

lemma *exp-MulPower2Add1*:
fixes $i :: 64\ word$
assumes $y = ConstantExpr\ (IntVal\ 64\ ((2\ \hat{}\ unat(i)) + 1))$
and $0 < i$
and $i < 64$
and $exp[x > (const\ IntVal\ b\ 0)]$
and $exp[y > (const\ IntVal\ b\ 0)]$
shows $exp[x * y] \geq exp[(x << ConstantExpr\ (IntVal\ 64\ i)) + x]$
⟨*proof*⟩

lemma *exp-MulPower2Sub1*:
fixes $i :: 64\ word$
assumes $y = ConstantExpr\ (IntVal\ 64\ ((2\ \hat{}\ unat(i)) - 1))$
and $0 < i$
and $i < 64$
and $exp[x > (const\ IntVal\ b\ 0)]$
and $exp[y > (const\ IntVal\ b\ 0)]$
shows $exp[x * y] \geq exp[(x << ConstantExpr\ (IntVal\ 64\ i)) - x]$
⟨*proof*⟩

lemma *exp-MulPower2AddPower2*:
fixes *i j* :: *64 word*
assumes $y = ConstantExpr\ (IntVal\ 64\ ((2\ \hat{}\ unat(i)) + (2\ \hat{}\ unat(j))))$
and $0 < i$
and $0 < j$
and $i < 64$
and $j < 64$
and $exp[x > (const\ IntVal\ b\ 0)]$
and $exp[y > (const\ IntVal\ b\ 0)]$
shows $exp[x * y] \geq exp[(x << ConstantExpr\ (IntVal\ 64\ i)) + (x << ConstantExpr\ (IntVal\ 64\ j))]$
⟨*proof*⟩

lemma *greaterConstant*:
fixes *a b* :: *64 word*
assumes $a > b$
and $y = ConstantExpr\ (IntVal\ 32\ a)$
and $x = ConstantExpr\ (IntVal\ 32\ b)$
shows $exp[BinaryExpr\ BinIntegerLessThan\ y\ x] \geq exp[const\ (new\text{-}int\ 32\ 0)]$
⟨*proof*⟩

lemma *exp-distribute-multiplication*:
assumes $stamp\text{-}expr\ x = IntegerStamp\ b\ xl\ xh$
assumes $stamp\text{-}expr\ q = IntegerStamp\ b\ ql\ qh$
assumes $stamp\text{-}expr\ y = IntegerStamp\ b\ yl\ yh$
assumes *wf-stamp x*
assumes *wf-stamp q*
assumes *wf-stamp y*
shows $exp[(x * q) + (x * y)] \geq exp[x * (q + y)]$
⟨*proof*⟩

Optimisations

optimization *EliminateRedundantNegative*: $-x * -y \longmapsto x * y$
⟨*proof*⟩

optimization *MulNeutral*: $x * ConstantExpr\ (IntVal\ b\ 1) \longmapsto x$
⟨*proof*⟩

optimization *MulEliminator*: $x * ConstantExpr\ (IntVal\ b\ 0) \longmapsto const\ (IntVal\ b\ 0)$
⟨*proof*⟩

optimization *MulNegate*: $x * -(const\ (IntVal\ b\ 1)) \longmapsto -x$
⟨*proof*⟩

fun *isNonZero* :: *Stamp* ⇒ *bool* **where**

isNonZero (*IntegerStamp b lo hi*) = (*lo* > *0*) |
isNonZero - = *False*

lemma *isNonZero-defn*:
assumes *isNonZero* (*stamp-expr x*)
assumes *wf-stamp x*
shows ([*m*, *p*] ⊢ *x* ↦ *v*) ⟶ (∃ *vv b*. (*v* = *IntVal b vv* ∧ *val-to-bool val*[(*IntVal b 0*) < *v*]))
⟨*proof*⟩

lemma *ExpIntBecomesIntValArbitrary*:
assumes *stamp-expr x* = *IntegerStamp b xl xh*
assumes *wf-stamp x*
assumes *valid-value v* (*IntegerStamp b xl xh*)
assumes [*m*,*p*] ⊢ *x* ↦ *v*
shows ∃ *xv*. *v* = *IntVal b xv*
⟨*proof*⟩

optimization *MulPower2*: *x* ∗ *y* ⟼ *x* << *const* (*IntVal 64 i*)
when (*i* > *0* ∧ *stamp-expr x* = *IntegerStamp 64 xl xh* ∧ *wf-stamp x* ∧
64 > *i* ∧
y = *exp*[*const* (*IntVal 64* (*2* ^ *unat*(*i*)))])
⟨*proof*⟩

optimization *MulPower2Add1*: *x* ∗ *y* ⟼ (*x* << *const* (*IntVal 64 i*)) + *x*
when (*i* > *0* ∧ *stamp-expr x* = *IntegerStamp 64 xl xh* ∧ *wf-stamp x* ∧
64 > *i* ∧
y = *ConstantExpr* (*IntVal 64* ((*2* ^ *unat*(*i*)) + *1*)))
⟨*proof*⟩

optimization *MulPower2Sub1*: *x* ∗ *y* ⟼ (*x* << *const* (*IntVal 64 i*)) − *x*
when (*i* > *0* ∧ *stamp-expr x* = *IntegerStamp 64 xl xh* ∧ *wf-stamp x* ∧
64 > *i* ∧
y = *ConstantExpr* (*IntVal 64* ((*2* ^ *unat*(*i*)) − *1*)))
⟨*proof*⟩

end

end

11.7 Experimental AndNode Phase

theory *NewAnd*
imports
Common

Graph.JavaLong
begin

lemma *intval-distribute-and-over-or*:
$val[z \ \& \ (x \mid y)] = val[(z \ \& \ x) \mid (z \ \& \ y)]$
⟨*proof*⟩

lemma *exp-distribute-and-over-or*:
$exp[z \ \& \ (x \mid y)] \geq exp[(z \ \& \ x) \mid (z \ \& \ y)]$
⟨*proof*⟩

lemma *intval-and-commute*:
$val[x \ \& \ y] = val[y \ \& \ x]$
⟨*proof*⟩

lemma *intval-or-commute*:
$val[x \mid y] = val[y \mid x]$
⟨*proof*⟩

lemma *intval-xor-commute*:
$val[x \oplus y] = val[y \oplus x]$
⟨*proof*⟩

lemma *exp-and-commute*:
$exp[x \ \& \ z] \geq exp[z \ \& \ x]$
⟨*proof*⟩

lemma *exp-or-commute*:
$exp[x \mid y] \geq exp[y \mid x]$
⟨*proof*⟩

lemma *exp-xor-commute*:
$exp[x \oplus y] \geq exp[y \oplus x]$
⟨*proof*⟩

lemma *intval-eliminate-y*:
assumes $val[y \ \& \ z] = IntVal \ b \ 0$
shows $val[(x \mid y) \ \& \ z] = val[x \ \& \ z]$
⟨*proof*⟩

lemma *intval-and-associative*:
$val[(x \ \& \ y) \ \& \ z] = val[x \ \& \ (y \ \& \ z)]$
⟨*proof*⟩

lemma *intval-or-associative*:
$val[(x \mid y) \mid z] = val[x \mid (y \mid z)]$
⟨*proof*⟩

lemma *intval-xor-associative*:

$val[(x \oplus y) \oplus z] = val[x \oplus (y \oplus z)]$
⟨*proof*⟩

lemma *exp-and-associative*:
$exp[(x \,\&\, y) \,\&\, z] \geq exp[x \,\&\, (y \,\&\, z)]$
⟨*proof*⟩

lemma *exp-or-associative*:
$exp[(x \mid y) \mid z] \geq exp[x \mid (y \mid z)]$
⟨*proof*⟩

lemma *exp-xor-associative*:
$exp[(x \oplus y) \oplus z] \geq exp[x \oplus (y \oplus z)]$
⟨*proof*⟩

lemma *intval-and-absorb-or*:
assumes $\exists\, b\ v\ .\ x = new\text{-}int\ b\ v$
assumes $val[x \,\&\, (x \mid y)] \neq UndefVal$
shows $val[x \,\&\, (x \mid y)] = val[x]$
⟨*proof*⟩

lemma *intval-or-absorb-and*:
assumes $\exists\, b\ v\ .\ x = new\text{-}int\ b\ v$
assumes $val[x \mid (x \,\&\, y)] \neq UndefVal$
shows $val[x \mid (x \,\&\, y)] = val[x]$
⟨*proof*⟩

lemma *exp-and-absorb-or*:
$exp[x \,\&\, (x \mid y)] \geq exp[x]$
⟨*proof*⟩

lemma *exp-or-absorb-and*:
$exp[x \mid (x \,\&\, y)] \geq exp[x]$
⟨*proof*⟩

lemma
assumes $y = 0$
shows $x + y = or\ x\ y$
⟨*proof*⟩

lemma *no-overlap-or*:
assumes $and\ x\ y = 0$
shows $x + y = or\ x\ y$
⟨*proof*⟩

context *stamp-mask*
begin

lemma *intval-up-and-zero-implies-zero*:
assumes $and\ (\uparrow x)\ (\uparrow y) = 0$
assumes $[m, p] \vdash x \mapsto xv$
assumes $[m, p] \vdash y \mapsto yv$
assumes $val[xv\ \&\ yv] \neq UndefVal$
shows $\exists\ b\ .\ val[xv\ \&\ yv] = new\text{-}int\ b\ 0$
⟨*proof*⟩

lemma *exp-eliminate-y*:
$and\ (\uparrow y)\ (\uparrow z) = 0 \longrightarrow exp[(x \mid y)\ \&\ z] \geq exp[x\ \&\ z]$
⟨*proof*⟩

lemma *leadingZeroBounds*:
fixes $x :: 'a::len\ word$
assumes $n = numberOfLeadingZeros\ x$
shows $0 \leq n \wedge n \leq Nat.size\ x$
⟨*proof*⟩

lemma *above-nth-not-set*:
fixes $x :: int64$
assumes $n = 64 - numberOfLeadingZeros\ x$
shows $j > n \longrightarrow \neg(bit\ x\ j)$
⟨*proof*⟩

no-notation *LogicNegationNotation* (!-)

lemma *zero-horner*:
$horner\text{-}sum\ of\text{-}bool\ 2\ (map\ (\lambda x.\ False)\ xs) = 0$
⟨*proof*⟩

lemma *zero-map*:
assumes $j \leq n$
assumes $\forall\ i.\ j \leq i \longrightarrow \neg(f\ i)$
shows $map\ f\ [0..<n] = map\ f\ [0..<j]\ @\ map\ (\lambda x.\ False)\ [j..<n]$
⟨*proof*⟩

lemma *map-join-horner*:
assumes $map\ f\ [0..<n] = map\ f\ [0..<j]\ @\ map\ (\lambda x.\ False)\ [j..<n]$
shows $horner\text{-}sum\ of\text{-}bool\ (2::'a::len\ word)\ (map\ f\ [0..<n]) = horner\text{-}sum\ of\text{-}bool\ 2\ (map\ f\ [0..<j])$
⟨*proof*⟩

lemma *split-horner*:
  assumes $j \leq n$
  assumes $\forall\, i.\ j \leq i \longrightarrow \neg(f\ i)$
  shows *horner-sum of-bool* (*2*::*'a*::*len word*) (*map f* [*0*..<*n*]) = *horner-sum of-bool* *2* (*map f* [*0*..<*j*])
  ⟨*proof*⟩

lemma *transfer-map*:
  assumes $\forall\, i.\ i < n \longrightarrow f\ i = f'\ i$
  shows (*map f* [*0*..<*n*]) = (*map f′* [*0*..<*n*])
  ⟨*proof*⟩

lemma *transfer-horner*:
  assumes $\forall\, i.\ i < n \longrightarrow f\ i = f'\ i$
  shows *horner-sum of-bool* (*2*::*'a*::*len word*) (*map f* [*0*..<*n*]) = *horner-sum of-bool* *2* (*map f′* [*0*..<*n*])
  ⟨*proof*⟩

lemma *L1*:
  assumes *n* = *64* − *numberOfLeadingZeros* (↑*z*)
  assumes [*m*, *p*] ⊢ *z* ↦ *IntVal b zv*
  shows *and v zv* = *and* (*v mod 2^n*) *zv*
⟨*proof*⟩

lemma *up-mask-upper-bound*:
  assumes [*m*, *p*] ⊢ *x* ↦ *IntVal b xv*
  shows *xv* ≤ (↑*x*)
  ⟨*proof*⟩

lemma *L2*:
  assumes *numberOfLeadingZeros* (↑*z*) + *numberOfTrailingZeros* (↑*y*) ≥ *64*
  assumes *n* = *64* − *numberOfLeadingZeros* (↑*z*)
  assumes [*m*, *p*] ⊢ *z* ↦ *IntVal b zv*
  assumes [*m*, *p*] ⊢ *y* ↦ *IntVal b yv*
  shows *yv mod 2^n* = *0*
⟨*proof*⟩

thm-oracles *L1 L2*

lemma *unfold-binary-width-add*:
  shows ([*m*,*p*] ⊢ *BinaryExpr BinAdd xe ye* ↦ *IntVal b val*) = (∃ *x y*.
      (([*m*,*p*] ⊢ *xe* ↦ *IntVal b x*) ∧
       ([*m*,*p*] ⊢ *ye* ↦ *IntVal b y*) ∧
       (*IntVal b val* = *bin-eval BinAdd* (*IntVal b x*) (*IntVal b y*)) ∧
       (*IntVal b val* ≠ *UndefVal*)
    )) (**is** *?L* = *?R*)
  ⟨*proof*⟩

lemma *unfold-binary-width-and*:
shows ([*m*,*p*] ⊢ *BinaryExpr BinAnd xe ye* ↦ *IntVal b val*) = (∃ *x y*.
(([*m*,*p*] ⊢ *xe* ↦ *IntVal b x*) ∧
([*m*,*p*] ⊢ *ye* ↦ *IntVal b y*) ∧
(*IntVal b val* = *bin-eval BinAnd* (*IntVal b x*) (*IntVal b y*)) ∧
(*IntVal b val* ≠ *UndefVal*)
)) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *mod-dist-over-add-right*:
fixes *a b c* :: *int64*
fixes *n* :: *nat*
assumes *0* < *n*
assumes *n* < *64*
shows (*a* + *b mod 2^n*) *mod 2^n* = (*a* + *b*) *mod 2^n*
⟨*proof*⟩

lemma *numberOfLeadingZeros-range*:
0 ≤ *numberOfLeadingZeros n* ∧ *numberOfLeadingZeros n* ≤ *Nat.size n*
⟨*proof*⟩

lemma *improved-opt*:
assumes *numberOfLeadingZeros* (↑*z*) + *numberOfTrailingZeros* (↑*y*) ≥ *64*
shows *exp*[(*x* + *y*) & *z*] ≥ *exp*[*x* & *z*]
⟨*proof*⟩

thm-oracles *improved-opt*

end

phase *NewAnd*
terminating *size*
begin

optimization *redundant-lhs-y-or*: ((*x* | *y*) & *z*) ⟼ *x* & *z*
when (((*and* (*IRExpr-up y*) (*IRExpr-up z*)) = *0*))
⟨*proof*⟩

optimization *redundant-lhs-x-or*: ((*x* | *y*) & *z*) ⟼ *y* & *z*
when (((*and* (*IRExpr-up x*) (*IRExpr-up z*)) = *0*))
⟨*proof*⟩

optimization *redundant-rhs-y-or*: (*z* & (*x* | *y*)) ⟼ *z* & *x*
when (((*and* (*IRExpr-up y*) (*IRExpr-up z*)) = *0*))

⟨*proof*⟩

optimization *redundant-rhs-x-or*: $(z \,\&\, (x \mid y)) \longmapsto z \,\&\, y$
when $(((and\ (IRExpr\text{-}up\ x)\ (IRExpr\text{-}up\ z)) = 0))$
⟨*proof*⟩

end

end

11.8 NotNode Phase

theory *NotPhase*
imports
Common
begin

phase *NotNode*
terminating *size*
begin

lemma *bin-not-cancel*:
$bin[\neg(\neg(e))] = bin[e]$
⟨*proof*⟩

lemma *val-not-cancel*:
assumes $val[{\sim}(new\text{-}int\ b\ v)] \neq UndefVal$
shows $val[{\sim}({\sim}(new\text{-}int\ b\ v))] = (new\text{-}int\ b\ v)$
⟨*proof*⟩

lemma *exp-not-cancel*:
$exp[{\sim}({\sim}a)] \geq exp[a]$
⟨*proof*⟩

Optimisations

optimization *NotCancel*: $exp[{\sim}({\sim}a)] \longmapsto a$
⟨*proof*⟩

end

end

11.9 OrNode Phase

theory *OrPhase*

imports
Common
begin

context *stamp-mask*
begin

Taking advantage of the truth table of or operations.

#	x	y	$x\|y$
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	1

If row 2 never applies, that is, canBeZero x & canBeOne y = 0, then $(x|y) = x$.
Likewise, if row 3 never applies, canBeZero y & canBeOne x = 0, then $(x|y) = y$.

lemma *OrLeftFallthrough*:
assumes $(and\ (not\ (\downarrow x))\ (\uparrow y)) = 0$
shows $exp[x \mid y] \geq exp[x]$
⟨*proof*⟩

lemma *OrRightFallthrough*:
assumes $(and\ (not\ (\downarrow y))\ (\uparrow x)) = 0$
shows $exp[x \mid y] \geq exp[y]$
⟨*proof*⟩

end

phase *OrNode*
terminating *size*
begin

lemma *bin-or-equal*:
$bin[x \mid x] = bin[x]$
⟨*proof*⟩

lemma *bin-shift-const-right-helper*:
$x \mid y = y \mid x$
⟨*proof*⟩

lemma *bin-or-not-operands*:
$(\sim x \mid \sim y) = (\sim(x\ \&\ y))$
⟨*proof*⟩

lemma *val-or-equal*:
  assumes $x = \textit{new-int}\ b\ v$
  and $\textit{val}[x \mid x] \neq \textit{UndefVal}$
  shows $\textit{val}[x \mid x] = \textit{val}[x]$
  ⟨*proof*⟩

lemma *val-elim-redundant-false*:
  assumes $x = \textit{new-int}\ b\ v$
  and $\textit{val}[x \mid \textit{false}] \neq \textit{UndefVal}$
  shows $\textit{val}[x \mid \textit{false}] = \textit{val}[x]$
  ⟨*proof*⟩

lemma *val-shift-const-right-helper*:
  $\textit{val}[x \mid y] = \textit{val}[y \mid x]$
  ⟨*proof*⟩

lemma *val-or-not-operands*:
 $\textit{val}[\sim x \mid \sim y] = \textit{val}[\sim(x\ \&\ y)]$
  ⟨*proof*⟩

lemma *exp-or-equal*:
  $\textit{exp}[x \mid x] \geq \textit{exp}[x]$
  ⟨*proof*⟩

lemma *exp-elim-redundant-false*:
 $\textit{exp}[x \mid \textit{false}] \geq \textit{exp}[x]$
  ⟨*proof*⟩

Optimisations

optimization *OrEqual*: $x \mid x \longmapsto x$
  ⟨*proof*⟩

optimization *OrShiftConstantRight*: $((\textit{const}\ x) \mid y) \longmapsto y \mid (\textit{const}\ x)$ *when* $\neg(\textit{is-ConstantExpr}\ y)$
  ⟨*proof*⟩

optimization *EliminateRedundantFalse*: $x \mid \textit{false} \longmapsto x$
  ⟨*proof*⟩

optimization *OrNotOperands*: $(\sim x \mid \sim y) \longmapsto \sim(x\ \&\ y)$
  ⟨*proof*⟩

optimization *OrLeftFallthrough*:
  $x \mid y \longmapsto x$ *when* $((\textit{and}\ (\textit{not}\ (\textit{IRExpr-down}\ x))\ (\textit{IRExpr-up}\ y)) = \mathit{0})$
  ⟨*proof*⟩

optimization *OrRightFallthrough*:

x | *y* ⟼ *y* *when* ((*and* (*not* (*IRExpr-down* *y*)) (*IRExpr-up* *x*)) = *0*)
⟨*proof*⟩

end

end

11.10 ShiftNode Phase

theory *ShiftPhase*
imports
Common
begin

phase *ShiftNode*
terminating *size*
begin

fun *intval-log2* :: *Value* ⇒ *Value* **where**
intval-log2 (*IntVal b v*) = *IntVal b* (*word-of-int* (*SOME e. v=2^e*)) |
intval-log2 - = *UndefVal*

fun *in-bounds* :: *Value* ⇒ *int* ⇒ *int* ⇒ *bool* **where**
in-bounds (*IntVal b v*) *l h* = (*l* < *sint v* ∧ *sint v* < *h*) |
in-bounds - *l h* = *False*

lemma
assumes *in-bounds* (*intval-log2 val-c*) *0 32*
shows *val*[*x* << (*intval-log2 val-c*)] = *val*[*x* * *val-c*]
⟨*proof*⟩

lemma *e-intval*:
n = *intval-log2 val-c* ∧ *in-bounds n 0 32* ⟶
val[*x* << (*intval-log2 val-c*)] = *val*[*x* * *val-c*]
⟨*proof*⟩

optimization *e*:
x * (*const c*) ⟼ *x* << (*const n*) *when* (*n* = *intval-log2 c* ∧ *in-bounds n 0 32*)
⟨*proof*⟩

end

end

11.11 SignedDivNode Phase

theory *SignedDivPhase*
imports
Common

begin

phase *SignedDivNode*
  terminating *size*
begin

lemma *val-division-by-one-is-self-32*:
  assumes *x* = *new-int 32 v*
  shows *intval-div x* (*IntVal 32 1*) = *x*
  ⟨*proof*⟩

end

end

11.12 SignedRemNode Phase

theory *SignedRemPhase*
  imports
    Common
begin

phase *SignedRemNode*
  terminating *size*
begin

lemma *val-remainder-one*:
  assumes *intval-mod x* (*IntVal 32 1*) ≠ *UndefVal*
  shows *intval-mod x* (*IntVal 32 1*) = *IntVal 32 0*
  ⟨*proof*⟩

value *word-of-int* (*sint* (*x2::32 word*) *smod 1*)

end

end

11.13 SubNode Phase

theory *SubPhase*
  imports
    Common
    Proofs.StampEvalThms
begin

phase *SubNode*
  terminating *size*
begin

lemma *bin-sub-after-right-add*:
  shows $((x::('a::len)\ word) + (y::('a::len)\ word)) - y = x$
  ⟨*proof*⟩

lemma *sub-self-is-zero*:
  shows $(x::('a::len)\ word) - x = 0$
  ⟨*proof*⟩

lemma *bin-sub-then-left-add*:
  shows $(x::('a::len)\ word) - (x + (y::('a::len)\ word)) = -y$
  ⟨*proof*⟩

lemma *bin-sub-then-left-sub*:
  shows $(x::('a::len)\ word) - (x - (y::('a::len)\ word)) = y$
  ⟨*proof*⟩

lemma *bin-subtract-zero*:
  shows $(x :: 'a::len\ word) - (0 :: 'a::len\ word) = x$
  ⟨*proof*⟩

lemma *bin-sub-negative-value*:
 $(x :: ('a::len)\ word) - (-(y :: ('a::len)\ word)) = x + y$
  ⟨*proof*⟩

lemma *bin-sub-self-is-zero*:
 $(x :: ('a::len)\ word) - x = 0$
  ⟨*proof*⟩

lemma *bin-sub-negative-const*:
$(x :: 'a::len\ word) - (-(y :: 'a::len\ word)) = x + y$
  ⟨*proof*⟩

lemma *val-sub-after-right-add-2*:
  assumes $x = new\text{-}int\ b\ v$
  assumes $val[(x + y) - y] \neq UndefVal$
  shows $val[(x + y) - y] = x$
  ⟨*proof*⟩

lemma *val-sub-after-left-sub*:
  assumes $val[(x - y) - x] \neq UndefVal$
  shows $val[(x - y) - x] = val[-y]$
  ⟨*proof*⟩

lemma *val-sub-then-left-sub*:
  assumes $y = new\text{-}int\ b\ v$
  assumes $val[x - (x - y)] \neq UndefVal$
  shows $val[x - (x - y)] = y$
  ⟨*proof*⟩

lemma *val-subtract-zero*:
  assumes $x = new\text{-}int\ b\ v$
  assumes $val[x - (IntVal\ b\ 0)] \neq UndefVal$
  shows $val[x - (IntVal\ b\ 0)] = x$
  ⟨*proof*⟩

lemma *val-zero-subtract-value*:
  assumes $x = new\text{-}int\ b\ v$
  assumes $val[(IntVal\ b\ 0) - x] \neq UndefVal$
  shows $val[(IntVal\ b\ 0) - x] = val[-x]$
  ⟨*proof*⟩

lemma *val-sub-then-left-add*:
  assumes $val[x - (x + y)] \neq UndefVal$
  shows $val[x - (x + y)] = val[-y]$
  ⟨*proof*⟩

lemma *val-sub-negative-value*:
  assumes $val[x - (-y)] \neq UndefVal$
  shows $val[x - (-y)] = val[x + y]$
  ⟨*proof*⟩

lemma *val-sub-self-is-zero*:
  assumes $x = new\text{-}int\ b\ v \wedge val[x - x] \neq UndefVal$
  shows $val[x - x] = new\text{-}int\ b\ 0$
  ⟨*proof*⟩

lemma *val-sub-negative-const*:
  assumes $y = new\text{-}int\ b\ v \wedge val[x - (-y)] \neq UndefVal$
  shows $val[x - (-y)] = val[x + y]$
  ⟨*proof*⟩

lemma *exp-sub-after-right-add*:
  shows $exp[(x + y) - y] \geq x$
  ⟨*proof*⟩

lemma *exp-sub-after-right-add2*:
  shows $exp[(x + y) - x] \geq y$
  ⟨*proof*⟩

lemma *exp-sub-negative-value*:
 $exp[x - (-y)] \geq exp[x + y]$

⟨*proof*⟩

lemma *exp-sub-then-left-sub*:
  $exp[x - (x - y)] \geq y$
  ⟨*proof*⟩

thm-oracles *exp-sub-then-left-sub*

lemma *SubtractZero-Exp*:
  $exp[(x - (const\ IntVal\ b\ 0))] \geq x$
  ⟨*proof*⟩

lemma *ZeroSubtractValue-Exp*:
  assumes *wf-stamp x*
  assumes *stamp-expr x = IntegerStamp b lo hi*
  assumes ¬(*is-ConstantExpr x*)
  shows $exp[(const\ IntVal\ b\ 0) - x] \geq exp[-x]$
  ⟨*proof*⟩

Optimisations

optimization *SubAfterAddRight*: $((x + y) - y) \longmapsto x$
  ⟨*proof*⟩

optimization *SubAfterAddLeft*: $((x + y) - x) \longmapsto y$
  ⟨*proof*⟩

optimization *SubAfterSubLeft*: $((x - y) - x) \longmapsto -y$
  ⟨*proof*⟩

optimization *SubThenAddLeft*: $(x - (x + y)) \longmapsto -y$
  ⟨*proof*⟩

optimization *SubThenAddRight*: $(y - (x + y)) \longmapsto -x$
  ⟨*proof*⟩

optimization *SubThenSubLeft*: $(x - (x - y)) \longmapsto y$
  ⟨*proof*⟩

optimization *SubtractZero*: $(x - (const\ IntVal\ b\ 0)) \longmapsto x$
  ⟨*proof*⟩

thm-oracles *SubtractZero*

optimization *SubNegativeValue*: $(x - (-y)) \longmapsto x + y$
  ⟨*proof*⟩

thm-oracles *SubNegativeValue*

lemma *negate-idempotent*:
assumes $x = IntVal\ b\ v \wedge take\text{-}bit\ b\ v = v$
shows $x = val[-(-x)]$
⟨*proof*⟩

optimization *ZeroSubtractValue*: $((const\ IntVal\ b\ 0) - x) \longmapsto (-x)$
when $(wf\text{-}stamp\ x \wedge stamp\text{-}expr\ x = IntegerStamp\ b\ lo\ hi \wedge \neg(is\text{-}ConstantExpr\ x))$
⟨*proof*⟩

optimization *SubSelfIsZero*: $(x - x) \longmapsto const\ IntVal\ b\ 0$ *when*
$(wf\text{-}stamp\ x \wedge stamp\text{-}expr\ x = IntegerStamp\ b\ lo\ hi)$
⟨*proof*⟩

end

end

11.14 XorNode Phase

theory *XorPhase*
imports
Common
Proofs.StampEvalThms
begin

phase *XorNode*
terminating *size*
begin

lemma *bin-xor-self-is-false*:
$bin[x \oplus x] = 0$
⟨*proof*⟩

lemma *bin-xor-commute*:
$bin[x \oplus y] = bin[y \oplus x]$
⟨*proof*⟩

lemma *bin-eliminate-redundant-false*:
$bin[x \oplus 0] = bin[x]$
⟨*proof*⟩

lemma *val-xor-self-is-false*:
assumes $val[x \oplus x] \neq UndefVal$
shows $val\text{-}to\text{-}bool\ (val[x \oplus x]) = False$
⟨*proof*⟩

lemma *val-xor-self-is-false-2*:
assumes $val[x \oplus x] \neq UndefVal$
and $x = IntVal\ 32\ v$
shows $val[x \oplus x] = bool\text{-}to\text{-}val\ False$
⟨*proof*⟩

lemma *val-xor-self-is-false-3*:
assumes $val[x \oplus x] \neq UndefVal \wedge x = IntVal\ 64\ v$
shows $val[x \oplus x] = IntVal\ 64\ 0$
⟨*proof*⟩

lemma *val-xor-commute*:
$val[x \oplus y] = val[y \oplus x]$
⟨*proof*⟩

lemma *val-eliminate-redundant-false*:
assumes $x = new\text{-}int\ b\ v$
assumes $val[x \oplus (bool\text{-}to\text{-}val\ False)] \neq UndefVal$
shows $val[x \oplus (bool\text{-}to\text{-}val\ False)] = x$
⟨*proof*⟩

lemma *exp-xor-self-is-false*:
assumes $wf\text{-}stamp\ x \wedge stamp\text{-}expr\ x = default\text{-}stamp$
shows $exp[x \oplus x] \geq exp[false]$
⟨*proof*⟩

lemma *exp-eliminate-redundant-false*:
shows $exp[x \oplus false] \geq exp[x]$
⟨*proof*⟩

Optimisations

optimization *XorSelfIsFalse*: $(x \oplus x) \longmapsto false$ *when*
$(wf\text{-}stamp\ x \wedge stamp\text{-}expr\ x = default\text{-}stamp)$
⟨*proof*⟩

optimization *XorShiftConstantRight*: $((const\ x) \oplus y) \longmapsto y \oplus (const\ x)$ *when* $\neg(is\text{-}ConstantExpr\ y)$
⟨*proof*⟩

optimization *EliminateRedundantFalse*: $(x \oplus false) \longmapsto x$

⟨*proof*⟩

end

end

12 Conditional Elimination Phase

This theory presents the specification of the `ConditionalElimination` phase within the GraalVM compiler. The `ConditionalElimination` phase simplifies any condition of an *if* statement that can be implied by the conditions that dominate it. Such that if condition A implies that condition B *must* be true, the condition B is simplified to `true`.

```
if (A) {
  if (B) {
    ...
  }
}
```

We begin by defining the individual implication rules used by the phase in 12.1. These rules are then lifted to the rewriting of a condition within an *if* statement in **??**. The traversal algorithm used by the compiler is specified in **??**.

theory *ConditionalElimination*
  imports
    Semantics.IRTreeEvalThms
    Proofs.Rewrites
    Proofs.Bisimulation
    OptimizationDSL.Markup
begin

declare [[*show-types=false*]]

12.1 Implication Rules

The set of rules used for determining whether a condition, q_1, implies another condition, q_2, must be true or false.

12.1.1 Structural Implication

The first method for determining if a condition can be implied by another condition, is structural implication. That is, by looking at the structure of the conditions, we can determine the truth value. For instance, $x \equiv y$ implies that $x < y$ cannot be true.

inductive
impliesx :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* (- ⇛ -) **and**
impliesnot :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* (- ⇛¬ -) **where**
same: q ⇛ q |
eq-not-less: *exp*[x *eq* y] ⇛¬ *exp*[$x < y$] |
eq-not-less': *exp*[x *eq* y] ⇛¬ *exp*[$y < x$] |
less-not-less: *exp*[$x < y$] ⇛¬ *exp*[$y < x$] |
less-not-eq: *exp*[$x < y$] ⇛¬ *exp*[x *eq* y] |
less-not-eq': *exp*[$x < y$] ⇛¬ *exp*[y *eq* x] |
negate-true: ⟦x ⇛¬ y⟧ ⟹ x ⇛ *exp*[!y] |
negate-false: ⟦x ⇛ y⟧ ⟹ x ⇛¬ *exp*[!y]

inductive *implies-complete* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool option* ⇒ *bool* **where**
implies:
x ⇛ y ⟹ *implies-complete* x y (*Some True*) |
impliesnot:
x ⇛¬ y ⟹ *implies-complete* x y (*Some False*) |
fail:
¬((x ⇛ y) ∨ (x ⇛¬ y)) ⟹ *implies-complete* x y *None*

The relation q_1 ⇛ q_2 requires that the implication $q_1 \longrightarrow q_2$ is known true (i.e. universally valid). The relation q_1 ⇛¬ q_2 requires that the implication $q_1 \longrightarrow q_2$ is known false (i.e. *q1* ⟶ ¬ *q2* is universally valid). If neither q_1 ⇛ q_2 nor q_1 ⇛¬ q_2 then the status is unknown and the condition cannot be simplified.

fun *implies-valid* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* (**infix** ↣ *50*) **where**
implies-valid q1 q2 =
(∀ *m p v1 v2*. ([*m*, *p*] ⊢ *q1* ↦ *v1*) ∧ ([*m*,*p*] ⊢ *q2* ↦ *v2*) ⟶
(*val-to-bool v1* ⟶ *val-to-bool v2*))

fun *impliesnot-valid* :: *IRExpr* ⇒ *IRExpr* ⇒ *bool* (**infix** ⤔ *50*) **where**
impliesnot-valid q1 q2 =
(∀ *m p v1 v2*. ([*m*, *p*] ⊢ *q1* ↦ *v1*) ∧ ([*m*,*p*] ⊢ *q2* ↦ *v2*) ⟶
(*val-to-bool v1* ⟶ ¬*val-to-bool v2*))

The relation $q_1 \rightarrowtail q_2$ means $q_1 \longrightarrow q_2$ is universally valid, and the relation q_1 ⤔ q_2 means $q_1 \longrightarrow \neg\, q_2$ is universally valid.

lemma *eq-not-less-val*:
val-to-bool(*val*[*v1* *eq* *v2*]) ⟶ ¬*val-to-bool*(*val*[*v1* < *v2*])
⟨*proof*⟩

lemma *eq-not-less'-val*:
val-to-bool(*val*[*v1* *eq* *v2*]) ⟶ ¬*val-to-bool*(*val*[*v2* < *v1*])
⟨*proof*⟩

lemma *less-not-less-val*:

$val\text{-}to\text{-}bool(val[v1 < v2]) \longrightarrow \neg val\text{-}to\text{-}bool(val[v2 < v1])$
⟨*proof*⟩

lemma *less-not-eq-val*:
$val\text{-}to\text{-}bool(val[v1 < v2]) \longrightarrow \neg val\text{-}to\text{-}bool(val[v1\ eq\ v2])$
⟨*proof*⟩

lemma *logic-negate-type*:
assumes $[m, p] \vdash UnaryExpr\ UnaryLogicNegation\ x \mapsto v$
shows $\exists\, b\ v2.\ [m, p] \vdash x \mapsto IntVal\ b\ v2$
⟨*proof*⟩

lemma *intval-logic-negation-inverse*:
assumes $b > 0$
assumes $x = IntVal\ b\ v$
shows $val\text{-}to\text{-}bool\ (intval\text{-}logic\text{-}negation\ x) \longleftrightarrow \neg(val\text{-}to\text{-}bool\ x)$
⟨*proof*⟩

lemma *logic-negation-relation-tree*:
assumes $[m, p] \vdash y \mapsto val$
assumes $[m, p] \vdash UnaryExpr\ UnaryLogicNegation\ y \mapsto invval$
shows $val\text{-}to\text{-}bool\ val \longleftrightarrow \neg(val\text{-}to\text{-}bool\ invval)$
⟨*proof*⟩

The following theorem show that the known true/false rules are valid.

theorem *implies-impliesnot-valid*:
shows $((q1 \Rrightarrow q2) \longrightarrow (q1 \rightarrowtail q2)) \wedge$
$((q1 \Rrightarrow\neg\ q2) \longrightarrow (q1 \nrightarrow q2))$
(**is** $(?imp \longrightarrow ?val) \wedge (?notimp \longrightarrow ?notval)$)
⟨*proof*⟩

12.1.2 Type Implication

The second mechanism to determine whether a condition implies another is to use the type information of the relevant nodes. For instance, $x < (4::'a)$ implies $x < (10::'a)$. We can show this by strengthening the type, stamp, of the node x such that the upper bound is $4::'a$. Then we the second condition is reached, we know that the condition must be true by the upperbound.

The following relation corresponds to the `UnaryOpLogicNode.tryFold` and `BinaryOpLogicNode.tryFold` methods and their associated concrete implementations.

We track the refined stamps by mapping nodes to Stamps, the second parameter to *tryFold*.

inductive $tryFold :: IRNode \Rightarrow (ID \Rightarrow Stamp) \Rightarrow bool \Rightarrow bool$
where
$[\![alwaysDistinct\ (stamps\ x)\ (stamps\ y)]\!]$
$\Longrightarrow tryFold\ (IntegerEqualsNode\ x\ y)\ stamps\ False\ |$

$[\![$*neverDistinct* (*stamps x*) (*stamps y*)$]\!]$
$\Longrightarrow$ *tryFold* (*IntegerEqualsNode x y*) *stamps True* |
$[\![$*is-IntegerStamp* (*stamps x*);
is-IntegerStamp (*stamps y*);
stpi-upper (*stamps x*) $<$ *stpi-lower* (*stamps y*)$]\!]$
$\Longrightarrow$ *tryFold* (*IntegerLessThanNode x y*) *stamps True* |
$[\![$*is-IntegerStamp* (*stamps x*);
is-IntegerStamp (*stamps y*);
stpi-lower (*stamps x*) $\geq$ *stpi-upper* (*stamps y*)$]\!]$
$\Longrightarrow$ *tryFold* (*IntegerLessThanNode x y*) *stamps False*

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow$ *bool*) *tryFold* ⟨*proof*⟩

Prove that, when the stamp map is valid, the *tryFold* relation correctly predicts the output value with respect to our evaluation semantics.

inductive-cases *StepE*:
g, *p* ⊢ (*nid*,*m*,*h*) → (*nid'*,*m'*,*h*)

lemma *is-stamp-empty-valid*:
assumes *is-stamp-empty s*
shows ¬(∃ *val*. *valid-value val s*)
⟨*proof*⟩

lemma *join-valid*:
assumes *is-IntegerStamp s1* ∧ *is-IntegerStamp s2*
assumes *valid-stamp s1* ∧ *valid-stamp s2*
shows (*valid-value v s1* ∧ *valid-value v s2*) = *valid-value v* (*join s1 s2*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *alwaysDistinct-evaluate*:
assumes *wf-stamp g stamps*
assumes *alwaysDistinct* (*stamps x*) (*stamps y*)
assumes *is-IntegerStamp* (*stamps x*) ∧ *is-IntegerStamp* (*stamps y*) ∧ *valid-stamp* (*stamps x*) ∧ *valid-stamp* (*stamps y*)
shows ¬(∃ *val* . ([*g*, *m*, *p*] ⊢ *x* ↦ *val*) ∧ ([*g*, *m*, *p*] ⊢ *y* ↦ *val*))
⟨*proof*⟩

lemma *alwaysDistinct-valid*:
assumes *wf-stamp g stamps*
assumes *kind g nid* = (*IntegerEqualsNode x y*)
assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
assumes *alwaysDistinct* (*stamps x*) (*stamps y*)
shows ¬(*val-to-bool v*)
⟨*proof*⟩
thm-oracles *alwaysDistinct-valid*

lemma *unwrap-valid*:

assumes $0 < b \wedge b \leq 64$
assumes *take-bit* (*b*::*nat*) (*vv*::*64 word*) = *vv*
shows (*vv*::*64 word*) = *take-bit b* (*word-of-int* (*int-signed-value* (*b*::*nat*) (*vv*::*64 word*)))
⟨*proof*⟩

lemma *asConstant-valid*:
assumes *asConstant s* = *val*
assumes *val* ≠ *UndefVal*
assumes *valid-value v s*
shows *v* = *val*
⟨*proof*⟩

lemma *neverDistinct-valid*:
assumes *wf-stamp g stamps*
assumes *kind g nid* = (*IntegerEqualsNode x y*)
assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
assumes *neverDistinct* (*stamps x*) (*stamps y*)
shows *val-to-bool v*
⟨*proof*⟩

lemma *stampUnder-valid*:
assumes *wf-stamp g stamps*
assumes *kind g nid* = (*IntegerLessThanNode x y*)
assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
assumes *stpi-upper* (*stamps x*) < *stpi-lower* (*stamps y*)
shows *val-to-bool v*
⟨*proof*⟩

lemma *stampOver-valid*:
assumes *wf-stamp g stamps*
assumes *kind g nid* = (*IntegerLessThanNode x y*)
assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
assumes *stpi-lower* (*stamps x*) ≥ *stpi-upper* (*stamps y*)
shows ¬(*val-to-bool v*)
⟨*proof*⟩

theorem *tryFoldTrue-valid*:
assumes *wf-stamp g stamps*
assumes *tryFold* (*kind g nid*) *stamps True*
assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
shows *val-to-bool v*
⟨*proof*⟩

theorem *tryFoldFalse-valid*:
assumes *wf-stamp g stamps*
assumes *tryFold* (*kind g nid*) *stamps False*
assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
shows ¬(*val-to-bool v*)

⟨*proof*⟩

12.2 Lift rules

inductive *condset-implies* :: *IRExpr set* ⇒ *IRExpr* ⇒ *bool* ⇒ *bool* **where**
impliesTrue:
$(\exists\, ce \in conds\ .\ (ce \Rrightarrow cond)) \Longrightarrow$ *condset-implies conds cond True* |
impliesFalse:
$(\exists\, ce \in conds\ .\ (ce \Rrightarrow\neg\ cond)) \Longrightarrow$ *condset-implies conds cond False*

code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *bool*) *condset-implies* ⟨*proof*⟩

The *cond-implies* function lifts the structural and type implication rules to the one relation.

fun *conds-implies* :: *IRExpr set* ⇒ (*ID* ⇒ *Stamp*) ⇒ *IRNode* ⇒ *IRExpr* ⇒ *bool option* **where**
conds-implies conds stamps condNode cond =
(*if condset-implies conds cond True* ∨ *tryFold condNode stamps True*
then Some True
else if condset-implies conds cond False ∨ *tryFold condNode stamps False*
then Some False
else None)

Perform conditional elimination rewrites on the graph for a particular node by lifting the individual implication rules to a relation that rewrites the condition of *if* statements to constant values.

In order to determine conditional eliminations appropriately the rule needs two data structures produced by static analysis. The first parameter is the set of IRNodes that we know result in a true value when evaluated. The second parameter is a mapping from node identifiers to the flow-sensitive stamp.

inductive *ConditionalEliminationStep* ::
IRExpr set ⇒ (*ID* ⇒ *Stamp*) ⇒ *ID* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool*
where
impliesTrue:
⟦*kind g ifcond* = (*IfNode cid t f*);
$g \vdash cid \simeq cond$;
condNode = *kind g cid*;
conds-implies conds stamps condNode cond = (*Some True*);
g′ = *constantCondition True ifcond* (*kind g ifcond*) *g*
⟧ ⟹ *ConditionalEliminationStep conds stamps ifcond g g′* |

impliesFalse:
⟦*kind g ifcond* = (*IfNode cid t f*);
$g \vdash cid \simeq cond$;
condNode = *kind g cid*;
conds-implies conds stamps condNode cond = (*Some False*);
g′ = *constantCondition False ifcond* (*kind g ifcond*) *g*

$\rrbracket \Longrightarrow$ *ConditionalEliminationStep conds stamps ifcond g g'* |

unknown:
$\llbracket$*kind g ifcond* = (*IfNode cid t f*);
g ⊢ *cid* ≃ *cond*;
condNode = *kind g cid*;
conds-implies conds stamps condNode cond = *None*
$\rrbracket \Longrightarrow$ *ConditionalEliminationStep conds stamps ifcond g g* |

notIfNode:
¬(*is-IfNode* (*kind g ifcond*)) $\Longrightarrow$
ConditionalEliminationStep conds stamps ifcond g g

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

thm *ConditionalEliminationStep.equation*

12.3 Control-flow Graph Traversal

type-synonym *Seen* = *ID set*
type-synonym *Condition* = *IRExpr*
type-synonym *Conditions* = *Condition list*
type-synonym *StampFlow* = (*ID* ⇒ *Stamp*) *list*
type-synonym *ToVisit* = *ID list*

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

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

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

Merge nodes represent a special case wherein 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 *preds* :: *IRGraph* ⇒ *ID* ⇒ *ID list* **where**
preds g nid = (*case kind g nid of*
(*MergeNode ends* - -) ⇒ *ends* |
- ⇒
sorted-list-of-set (*IRGraph.predecessors g nid*)

)

fun *pred* :: *IRGraph* ⇒ *ID* ⇒ *ID option* **where**
pred g nid = (*case preds g nid of* [] ⇒ *None* | *x* # *xs* ⇒ *Some x*)

When the basic block of an if statement is entered, we know that the condition of the preceding if statement must be true. As in the GraalVM compiler, we introduce the `registerNewCondition` function which roughly corresponds to `ConditionalEliminationPhase.registerNewCondition`. This method updates the flow-sensitive stamp information based on the condition which we know must be true.

fun *clip-upper* :: *Stamp* ⇒ *int* ⇒ *Stamp* **where**
clip-upper (*IntegerStamp b l h*) *c* =
(*if c* < *h then* (*IntegerStamp b l c*) *else* (*IntegerStamp b l h*)) |
clip-upper s c = *s*
fun *clip-lower* :: *Stamp* ⇒ *int* ⇒ *Stamp* **where**
clip-lower (*IntegerStamp b l h*) *c* =
(*if l* < *c then* (*IntegerStamp b c h*) *else* (*IntegerStamp b l c*)) |
clip-lower s c = *s*

fun *max-lower* :: *Stamp* ⇒ *Stamp* ⇒ *Stamp* **where**
max-lower (*IntegerStamp b1 xl xh*) (*IntegerStamp b2 yl yh*) =
(*IntegerStamp b1* (*max xl yl*) *xh*) |
max-lower xs ys = *xs*
fun *min-higher* :: *Stamp* ⇒ *Stamp* ⇒ *Stamp* **where**
min-higher (*IntegerStamp b1 xl xh*) (*IntegerStamp b2 yl yh*) =
(*IntegerStamp b1 yl* (*min xh yh*)) |
min-higher xs ys = *ys*

fun *registerNewCondition* :: *IRGraph* ⇒ *IRNode* ⇒ (*ID* ⇒ *Stamp*) ⇒ (*ID* ⇒ *Stamp*) **where**
— constrain equality by joining the stamps
registerNewCondition g (*IntegerEqualsNode x y*) *stamps* =
(*stamps*
(*x* := *join* (*stamps x*) (*stamps y*)))
(*y* := *join* (*stamps x*) (*stamps y*)) |
— constrain less than by removing overlapping stamps
registerNewCondition g (*IntegerLessThanNode x y*) *stamps* =
(*stamps*
(*x* := *clip-upper* (*stamps x*) ((*stpi-lower* (*stamps y*)) − *1*)))
(*y* := *clip-lower* (*stamps y*) ((*stpi-upper* (*stamps x*)) + *1*)) |
registerNewCondition g (*LogicNegationNode c*) *stamps* =
(*case* (*kind g c*) *of*
(*IntegerLessThanNode x y*) ⇒
(*stamps*
(*x* := *max-lower* (*stamps x*) (*stamps y*)))
(*y* := *min-higher* (*stamps x*) (*stamps y*))
| - ⇒ *stamps*) |
registerNewCondition g - stamps = *stamps*

fun *hdOr* :: *'a list* ⇒ *'a* ⇒ *'a* **where**
hdOr (*x* # *xs*) *de* = *x* |
hdOr [] *de* = *de*

type-synonym *DominatorCache* = (*ID*, *ID set*) *map*

inductive
dominators-all :: *IRGraph* ⇒ *DominatorCache* ⇒ *ID* ⇒ *ID set set* ⇒ *ID list* ⇒ *DominatorCache* ⇒ *ID set set* ⇒ *ID list* ⇒ *bool* **and**
dominators :: *IRGraph* ⇒ *DominatorCache* ⇒ *ID* ⇒ (*ID set* × *DominatorCache*) ⇒ *bool* **where**

⟦*pre* = []⟧
⟹ *dominators-all g c nid doms pre c doms pre* |

⟦*pre* = *pr* # *xs*;
(*dominators g c pr* (*doms'*, *c'*));
dominators-all g c' pr (*doms* ∪ {*doms'*}) *xs c'' doms'' pre'*⟧
⟹ *dominators-all g c nid doms pre c'' doms'' pre'* |

⟦*preds g nid* = []⟧
⟹ *dominators g c nid* ({*nid*}, *c*) |

⟦*c nid* = *None*;
preds g nid = *x* # *xs*;
dominators-all g c nid {} (*preds g nid*) *c' doms pre'*;
c'' = *c'*(*nid* ↦ ({*nid*} ∪ (⋂ *doms*)))⟧
⟹ *dominators g c nid* (({*nid*} ∪ (⋂ *doms*)), *c''*) |

⟦*c nid* = *Some doms*⟧
⟹ *dominators g c nid* (*doms*, *c*)

— Trying to simplify by removing the 3rd case won't work. A base case for root nodes is required as ⋂ ∅ = *coset* [] which swallows anything unioned with it.
value ⋂({}::*nat set set*)
value − ⋂({}::*nat set set*)
value ⋂({{}, {*0*}}::*nat set set*)
value {*0*::*nat*} ∪ (⋂{})

code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *o* ⇒ *o* ⇒ *bool*) *dominators-all* ⟨*proof*⟩
code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*) *dominators* ⟨*proof*⟩

definition *ConditionalEliminationTest13-testSnippet2-initial* :: *IRGraph* **where**
ConditionalEliminationTest13-testSnippet2-initial = *irgraph* [

(*0*, (*StartNode* (*Some 2*) *8*), *VoidStamp*),
(*1*, (*ParameterNode 0*), *IntegerStamp 32* (−*2147483648*) (*2147483647*)),
(*2*, (*FrameState* [] *None None None*), *IllegalStamp*),
(*3*, (*ConstantNode* (*new-int 32* (*0*))), *IntegerStamp 32* (*0*) (*0*)),
(*4*, (*ConstantNode* (*new-int 32* (*1*))), *IntegerStamp 32* (*1*) (*1*)),
(*5*, (*IntegerLessThanNode 1 4*), *VoidStamp*),
(*6*, (*BeginNode 13*), *VoidStamp*),
(*7*, (*BeginNode 23*), *VoidStamp*),
(*8*, (*IfNode 5 7 6*), *VoidStamp*),
(*9*, (*ConstantNode* (*new-int 32* (−*1*))), *IntegerStamp 32* (−*1*) (−*1*)),
(*10*, (*IntegerEqualsNode 1 9*), *VoidStamp*),
(*11*, (*BeginNode 17*), *VoidStamp*),
(*12*, (*BeginNode 15*), *VoidStamp*),
(*13*, (*IfNode 10 12 11*), *VoidStamp*),
(*14*, (*ConstantNode* (*new-int 32* (−*2*))), *IntegerStamp 32* (−*2*) (−*2*)),
(*15*, (*StoreFieldNode 15* ''*org.graalvm.compiler.core.test.ConditionalEliminationTestBase*::*sink2*'' *14* (*Some 16*) *None 19*), *VoidStamp*),
(*16*, (*FrameState* [] *None None None*), *IllegalStamp*),
(*17*, (*EndNode*), *VoidStamp*),
(*18*, (*MergeNode* [*17*, *19*] (*Some 20*) *21*), *VoidStamp*),
(*19*, (*EndNode*), *VoidStamp*),
(*20*, (*FrameState* [] *None None None*), *IllegalStamp*),
(*21*, (*StoreFieldNode 21* ''*org.graalvm.compiler.core.test.ConditionalEliminationTestBase*::*sink1*'' *3* (*Some 22*) *None 25*), *VoidStamp*),
(*22*, (*FrameState* [] *None None None*), *IllegalStamp*),
(*23*, (*EndNode*), *VoidStamp*),
(*24*, (*MergeNode* [*23*, *25*] (*Some 26*) *27*), *VoidStamp*),
(*25*, (*EndNode*), *VoidStamp*),
(*26*, (*FrameState* [] *None None None*), *IllegalStamp*),
(*27*, (*StoreFieldNode 27* ''*org.graalvm.compiler.core.test.ConditionalEliminationTestBase*::*sink0*'' *9* (*Some 28*) *None 29*), *VoidStamp*),
(*28*, (*FrameState* [] *None None None*), *IllegalStamp*),
(*29*, (*ReturnNode None None*), *VoidStamp*)
]

values {(*snd x*) *13*| *x*. *dominators ConditionalEliminationTest13-testSnippet2-initial Map.empty 25 x*}

inductive
condition-of :: *IRGraph* ⇒ *ID* ⇒ (*IRExpr* × *IRNode*) *option* ⇒ *bool* **where**
⟦*Some ifcond* = *pred g nid*;
kind g ifcond = *IfNode cond t f*;

i = *find-index nid* (*successors-of* (*kind g ifcond*));
c = (*if i* = *0 then kind g cond else LogicNegationNode cond*);

rep g cond ce;
ce' = (*if i* = *0 then ce else UnaryExpr UnaryLogicNegation ce*)⟧
⟹ *condition-of g nid* (*Some* (*ce'*, *c*)) |

⟦*pred g nid* = *None*⟧ ⟹ *condition-of g nid None* |
⟦*pred g nid* = *Some nid'*;
¬(*is-IfNode* (*kind g nid'*))⟧ ⟹ *condition-of g nid None*

code-pred (*modes*: *i* ⇒ *i* ⇒ *o* ⇒ *bool*) *condition-of* ⟨*proof*⟩

fun *conditions-of-dominators* :: *IRGraph* ⇒ *ID list* ⇒ *Conditions* ⇒ *Conditions*
where
conditions-of-dominators g [] *cds* = *cds* |
conditions-of-dominators g (*nid* # *nids*) *cds* =
(*case* (*Predicate.the* (*condition-of-i-i-o g nid*)) *of*
None ⇒ *conditions-of-dominators g nids cds* |
Some (*expr*, -) ⇒ *conditions-of-dominators g nids* (*expr* # *cds*))

fun *stamps-of-dominators* :: *IRGraph* ⇒ *ID list* ⇒ *StampFlow* ⇒ *StampFlow*
where
stamps-of-dominators g [] *stamps* = *stamps* |
stamps-of-dominators g (*nid* # *nids*) *stamps* =
(*case* (*Predicate.the* (*condition-of-i-i-o g nid*)) *of*
None ⇒ *stamps-of-dominators g nids stamps* |
Some (-, *node*) ⇒ *stamps-of-dominators g nids*
((*registerNewCondition g node* (*hd stamps*)) # *stamps*))

inductive
analyse :: *IRGraph* ⇒ *DominatorCache* ⇒ *ID* ⇒ (*Conditions* × *StampFlow* × *DominatorCache*) ⇒ *bool* **where**
⟦*dominators g c nid* (*doms*, *c'*);
conditions-of-dominators g (*sorted-list-of-set doms*) [] = *conds*;
stamps-of-dominators g (*sorted-list-of-set doms*) [*stamp g*] = *stamps*⟧
⟹ *analyse g c nid* (*conds*, *stamps*, *c'*)

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

values {*x*. *dominators ConditionalEliminationTest13-testSnippet2-initial Map.empty 13 x*}
values {(*conds*, *stamps*, *c*).
analyse ConditionalEliminationTest13-testSnippet2-initial Map.empty 13 (*conds*,

stamps, *c*)}
values {(*hd stamps*) *1*| *conds stamps c* .
analyse ConditionalEliminationTest13-testSnippet2-initial Map.empty 13 (*conds*, *stamps*, *c*)}
values {(*hd stamps*) *1*| *conds stamps c* .
analyse ConditionalEliminationTest13-testSnippet2-initial Map.empty 27 (*conds*, *stamps*, *c*)}

fun *next-nid* :: *IRGraph* ⇒ *ID set* ⇒ *ID* ⇒ *ID option* **where**
  next-nid g seen nid = (*case* (*kind g nid*) *of*
    (*EndNode*) ⇒ *Some* (*any-usage g nid*) |
    - ⇒ *nextEdge seen nid g*)

inductive *Step*
  :: *IRGraph* ⇒ (*ID* × *Seen*) ⇒ (*ID* × *Seen*) *option* ⇒ *bool*
  for *g* **where**
  — We can find a successor edge that is not in seen, go there
  ⟦*seen′* = {*nid*} ∪ *seen*;

    Some nid′ = *next-nid g seen′ nid*;
    nid′ ∉ *seen′*⟧
   ⟹ *Step g* (*nid*, *seen*) (*Some* (*nid′*, *seen′*)) |

  — We can cannot find a successor edge that is not in seen, give back None
  ⟦*seen′* = {*nid*} ∪ *seen*;

    None = *next-nid g seen′ nid*⟧
    ⟹ *Step g* (*nid*, *seen*) *None* |

  — We've already seen this node, give back None
  ⟦*seen′* = {*nid*} ∪ *seen*;

    Some nid′ = *next-nid g seen′ nid*;
    nid′ ∈ *seen′*⟧ ⟹ *Step g* (*nid*, *seen*) *None*

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

fun *nextNode* :: *IRGraph* ⇒ *Seen* ⇒ (*ID* × *Seen*) *option* **where**
  nextNode g seen =
    (*let toSee* = *sorted-list-of-set* {*n* ∈ *ids g*. *n* ∉ *seen*} *in*
      case toSee of [] ⇒ *None* | (*x* # *xs*) ⇒ *Some* (*x*, *seen* ∪ {*x*}))

values {*x*. *Step ConditionalEliminationTest13-testSnippet2-initial* (*17*, {*17*,*11*,*25*,*21*,*18*,*19*,*15*,*12*,*13*,*6*,*29*,*27*,*2*
x}

The *ConditionalEliminationPhase* relation is responsible for combining the individual traversal steps from the *Step* relation and the optimizations from the *ConditionalEliminationStep* relation to perform a transformation of the whole graph.

inductive *ConditionalEliminationPhase*
:: (*Seen* × *DominatorCache*) ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool*
where

— Can do a step and optimise for the current node
⟦*nextNode g seen* = *Some* (*nid*, *seen'*);

analyse g c nid (*conds*, *flow*, *c'*);
ConditionalEliminationStep (*set conds*) (*hd flow*) *nid g g'*;

ConditionalEliminationPhase (*seen'*, *c'*) *g' g''*⟧
⟹ *ConditionalEliminationPhase* (*seen*, *c*) *g g''* |

⟦*nextNode g seen* = *None*⟧
⟹ *ConditionalEliminationPhase* (*seen*, *c*) *g g*

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

definition *runConditionalElimination* :: *IRGraph* ⇒ *IRGraph* **where**
runConditionalElimination g =
(*Predicate.the* (*ConditionalEliminationPhase-i-i-o* ({}, *Map.empty*) *g*))

values {(*doms*, *c'*)| *doms c'*.
dominators ConditionalEliminationTest13-testSnippet2-initial Map.empty 6 (*doms*, *c'*)}

values {(*conds*, *stamps*, *c*)| *conds stamps c* .
analyse ConditionalEliminationTest13-testSnippet2-initial Map.empty 6 (*conds*, *stamps*, *c*)}
value
(*nextNode*
ConditionalEliminationTest13-testSnippet2-initial {*0*,*1*,*2*,*3*,*4*,*5*,*6*,*7*,*8*,*9*,*10*,*11*,*12*,*13*,*14*,*15*,*16*,*17*,*18*,*19*,*20*,*2*

lemma *IfNodeStepE*: *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid'*, *m'*, *h*) ⟹
(⋀*cond tb fb val*.
kind g nid = *IfNode cond tb fb* ⟹
nid' = (*if val-to-bool val then tb else fb*) ⟹
[*g*, *m*, *p*] ⊢ *cond* ↦ *val* ⟹ *m'* = *m*)
⟨*proof*⟩

lemma *ifNodeHasCondEvalStutter*:
assumes (*g m p h* ⊢ *nid* ⇝ *nid'*)
assumes *kind g nid* = *IfNode cond t f*
shows ∃ *v*. ([*g*, *m*, *p*] ⊢ *cond* ↦ *v*)
⟨*proof*⟩

lemma *ifNodeHasCondEval*:
assumes $(g, p \vdash (nid, m, h) \rightarrow (nid', m', h'))$
assumes *kind g nid = IfNode cond t f*
shows $\exists\ v.\ ([g, m, p] \vdash cond \mapsto v)$
⟨*proof*⟩

lemma *replace-if-t*:
assumes *kind g nid = IfNode cond tb fb*
assumes $[g, m, p] \vdash cond \mapsto bool$
assumes *val-to-bool bool*
assumes g': $g' = replace\text{-}usages\ nid\ tb\ g$
shows $\exists nid'\ .(g\ m\ p\ h \vdash nid \rightsquigarrow nid') \longleftrightarrow (g'\ m\ p\ h \vdash nid \rightsquigarrow nid')$
⟨*proof*⟩

lemma *replace-if-t-imp*:
assumes *kind g nid = IfNode cond tb fb*
assumes $[g, m, p] \vdash cond \mapsto bool$
assumes *val-to-bool bool*
assumes g': $g' = replace\text{-}usages\ nid\ tb\ g$
shows $\exists nid'\ .(g\ m\ p\ h \vdash nid \rightsquigarrow nid') \longrightarrow (g'\ m\ p\ h \vdash nid \rightsquigarrow nid')$
⟨*proof*⟩

lemma *replace-if-f*:
assumes *kind g nid = IfNode cond tb fb*
assumes $[g, m, p] \vdash cond \mapsto bool$
assumes $\neg$(*val-to-bool bool*)
assumes g': $g' = replace\text{-}usages\ nid\ fb\ g$
shows $\exists nid'\ .(g\ m\ p\ h \vdash nid \rightsquigarrow nid') \longleftrightarrow (g'\ m\ p\ h \vdash nid \rightsquigarrow nid')$
⟨*proof*⟩

Prove that the individual conditional elimination rules are correct with respect to preservation of stuttering steps.

lemma *ConditionalEliminationStepProof*:
assumes *wg*: *wf-graph g*
assumes *ws*: *wf-stamps g*
assumes *wv*: *wf-values g*
assumes *nid*: $nid \in ids\ g$
assumes *cond-valid*: $\forall\ c \in conds\ .\ \exists\ v.\ ([m, p] \vdash c \mapsto v) \land val\text{-}to\text{-}bool\ v$
assumes *ce*: *ConditionalEliminationStep conds stamps nid g g'*

shows $\exists nid'\ .(g\ m\ p\ h \vdash nid \rightsquigarrow nid') \longrightarrow (g'\ m\ p\ h \vdash nid \rightsquigarrow nid')$
⟨*proof*⟩

Prove that the individual conditional elimination rules are correct with respect to finding a bisimulation between the unoptimized and optimized graphs.

lemma *ConditionalEliminationStepProofBisimulation*:
assumes *wf*: $wf\text{-}graph\ g \land wf\text{-}stamp\ g\ stamps \land wf\text{-}values\ g$

assumes *nid*: $nid \in ids\ g$
assumes *conds-valid*: $\forall\ c \in conds\ .\ \exists\ v.\ ([m,\ p] \vdash c \mapsto v) \wedge val\text{-}to\text{-}bool\ v$
assumes *ce*: *ConditionalEliminationStep conds stamps nid g g′*
assumes *gstep*: $\exists\ h\ nid'.\ (g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h))$

shows $nid \mid g \sim g'$
⟨*proof*⟩

experiment begin

lemma *inverse-succ*:
$\forall\, n' \in (succ\ g\ n).\ n \in ids\ g \longrightarrow n \in (predecessors\ g\ n')$
⟨*proof*⟩

lemma *sequential-successors*:
assumes *is-sequential-node n*
shows $successors\text{-}of\ n \neq []$
⟨*proof*⟩

lemma *nid′-succ*:
assumes $nid \in ids\ g$
assumes $\neg(is\text{-}AbstractEndNode\ (kind\ g\ nid0))$
assumes $g,\ p \vdash (nid0,\ m0,\ h0) \rightarrow (nid,\ m,\ h)$
shows $nid \in succ\ g\ nid0$
⟨*proof*⟩

lemma *nid′-pred*:
assumes $nid \in ids\ g$
assumes $\neg(is\text{-}AbstractEndNode\ (kind\ g\ nid0))$
assumes $g,\ p \vdash (nid0,\ m0,\ h0) \rightarrow (nid,\ m,\ h)$
shows $nid0 \in predecessors\ g\ nid$
⟨*proof*⟩

definition *wf-pred*:
$wf\text{-}pred\ g = (\forall\, n \in ids\ g.\ card\ (predecessors\ g\ n) = 1)$

lemma
assumes $\neg(is\text{-}AbstractMergeNode\ (kind\ g\ n'))$
assumes *wf-pred g*
shows $\exists\, v.\ predecessors\ g\ n = \{v\} \wedge pred\ g\ n' = Some\ v$
⟨*proof*⟩

lemma *inverse-succ1*:
assumes $\neg(is\text{-}AbstractEndNode\ (kind\ g\ n'))$
assumes *wf-pred g*
shows $\forall\, n' \in (succ\ g\ n).\ n \in ids\ g \longrightarrow Some\ n = (pred\ g\ n')$

⟨*proof*⟩

lemma *BeginNodeFlow*:
assumes *g*, *p* ⊢ (*nid0*, *m0*, *h0*) → (*nid*, *m*, *h*)
assumes *Some ifcond* = *pred g nid*
assumes *kind g ifcond* = *IfNode cond t f*
assumes *i* = *find-index nid* (*successors-of* (*kind g ifcond*))
shows *i* = *0* ⟷ ([*g*, *m*, *p*] ⊢ *cond* ↦ *v*) ∧ *val-to-bool v*
⟨*proof*⟩

end

end