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*
  unfolding *shiftl-def* **apply** (*induction j*)

apply *simp* **unfolding** *funpow-Suc-right*
by (*metis* (*no-types*, *opaque-lifting*) *push-bit-eq-mult*)

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

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

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

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

Unsigned shift right.

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

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

Signed shift right.

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

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

1.2 Fixed-width Word Theories

1.2.1 Support Lemmas for Upper/Lower Bounds

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

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

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

lemma *upper-bounds-equiv*:

assumes *0* < *N*
shows (*2*::*int*) ^ (*N*−*1*) = (*2*::*int*) ^ *N div 2*
by (*simp add*: *assms int-power-div-base*)

Some min/max bounds for 64-bit words

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

lemma *bit-bounds-max64*: ((*snd* (*bit-bounds 64*))) ≥ (*sint* (*v*::*int64*))
unfolding *bit-bounds.simps fst-def*
using *sint-lt*[*of v*] **by** *simp*

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

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

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

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

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

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

lemma *signed-take-bit-range*:
fixes *ival* :: ′*a* :: *len word*
assumes *n* < *LENGTH*(′*a*)
assumes *val* = *sint*(*signed-take-bit n ival*)
shows − (*2* ^ *n*) ≤ *val* ∧ *val* < *2* ^ *n*
using *signed-take-bit-int-greater-eq-minus-exp-word signed-take-bit-int-less-exp-word*
using *assms* **by** *blast*

A *bit_bounds* version of the above lemma.

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

lemma *signed-take-bit-bounds64*:
fixes *ival* :: *int64*
assumes $n \leq 64$
assumes $0 < n$
assumes *val* = *sint*(*signed-take-bit* (*n* − *1*) *ival*)
shows *fst* (*bit-bounds n*) ≤ *val* ∧ *val* ≤ *snd* (*bit-bounds n*)
using *assms signed-take-bit-bounds*
by (*metis size64 word-size*)

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

lemma *int-signed-value-range*:
fixes *ival* :: *int64*
assumes *val* = *int-signed-value n ival*
shows − (*2* ^ (*n* − *1*)) ≤ *val* ∧ *val* < *2* ^ (*n* − *1*)
using *assms* **apply** *auto*
apply (*smt* (*verit*, *ccfv-threshold*) *sint-greater-eq diff-less len-gt-0 power-strict-increasing power-less-imp-less-exp signed-take-bit-range len-num1 One-nat-def*)
by (*smt* (*verit*, *ccfv-threshold*) *neg-equal-0-iff-equal power-0 signed-minus-1 sint-0 not-gr-zero*
word-exp-length-eq-0 diff-less diff-zero len-gt-0 sint-less power-strict-increasing signed-take-bit-range power-less-imp-less-exp)

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

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

lemma *two-exp-div*:
assumes *0* < *bits*

shows $((2::int)\ \widehat{}\ bits\ div\ (2::int)) = (2::int)\ \widehat{}\ (bits - Suc\ 0)$
using *assms* **by** (*auto simp*: *int-power-div-base*)

declare [[*show-types*]]

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

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

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

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

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

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

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

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

lemma *scast-max-bound*:
  assumes *sint* (*v* :: *'a* :: *len word*) < *M*
  assumes *LENGTH*(*'a*) < *LENGTH*(*'b*)
  shows *sint* ((*scast v*) :: *'b* :: *len word*) < *M*
 using *assms* **unfolding** *Word.scast-eq Word.sint-sbintrunc'* **by** (*simp add*: *sint-uint*)

lemma *scast-min-bound*:
  assumes *M* ≤ *sint* (*v* :: *'a* :: *len word*)
  assumes *LENGTH*(*'a*) < *LENGTH*(*'b*)
  shows *M* ≤ *sint* ((*scast v*) :: *'b* :: *len word*)
 using *assms* **unfolding** *Word.scast-eq Word.sint-sbintrunc'* **by** (*simp add*: *sint-uint*)

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

lemma *scast-bigger-min-bound*:
  assumes (*result* :: *'b* :: *len word*) = *scast* (*v* :: *'a* :: *len word*)
  shows − (*2* ^ *LENGTH*(*'a*) *div 2*) ≤ *sint result*
  by (*metis upper-bounds-equiv assms len-gt-0 nat-less-le not-less scast-max-bound scast-min-bound*
    sint-ge)

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

1.2.2 Support lemmas for take bit and signed take bit.

Lemmas for removing redundant take_bit wrappers.

lemma *take-bit-dist-addL*[*simp*]:
  fixes *x* :: *'a* :: *len word*
  shows *take-bit b* (*take-bit b x* + *y*) = *take-bit b* (*x* + *y*)
proof (*induction b*)
  case *0*
  then show *?case*
    by *simp*
next
  case (*Suc b*)

then show *?case*
by (*simp add*: *add.commute mask-eqs*(*2*) *take-bit-eq-mask*)
qed

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

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

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

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

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

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

lemma *mod-dist-over-add*:
fixes *a b c* :: *int64*
fixes *n* :: *nat*
assumes *1*: *0* < *n*
assumes *2*: *n* < *64*
shows (*a mod 2*^*n* + *b*) *mod 2*^*n* = (*a* + *b*) *mod 2*^*n*
proof −
have *3*: (*0* :: *int64*) < *2* ^ *n*

using *assms* **by** (*simp add*: *size64 word-2p-lem*)
then show *?thesis*
unfolding *word-mod-2p-is-mask*[*OF 3*]
apply *transfer*
by (*metis* (*no-types*, *opaque-lifting*) *and.right-idem take-bit-add take-bit-eq-mask*)
qed

1.3 Java min and max operators on 64-bit values

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

abbreviation *javaMin64* :: *int64* ⇒ *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*:
$n < 32 \Longrightarrow bit\ (-1{::}int32)\ n$
apply *transfer* **by** *auto*

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* = {}
unfolding *MaxOrNeg-def* **by** *auto*

2.1 Long.highestOneBit

definition *highestOneBit* :: (′*a*::*len*) *word* ⇒ *int* **where**

highestOneBit v = *MaxOrNeg* {*n*. *bit v n*}

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

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

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

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

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

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

lemma *highestOneBitAtLeast*:
assumes *bit v n*
shows *highestOneBit v* ≥ *n*
proof (*induction size v*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc x*)
then have ∀ *i*. *bit v i* ⟶ *i* < *Suc x*

by (*simp add*: *bit-imp-le-length wsst-TYs*(*3*))
then show *?case*
unfolding *highestOneBit-def MaxOrNeg-def*
using *assms* **by** *auto*
qed

lemma *highestOneBitElim*:
highestOneBit v = *n*
$\Longrightarrow$ ((*n* = −*1* ∧ *v* = *0*) ∨ (*n* ≥ *0* ∧ *bit v n*))
unfolding *highestOneBit-def MaxOrNeg-def*
by (*metis Max-in finite-bit-word le0 le-minus-one-simps*(*3*) *mem-Collect-eq of-nat-0-le-iff of-nat-eq-iff*)

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

fun *highestOneBitRec* :: *nat* ⇒ (′*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* $\Longrightarrow$ *j* ≥ *0* $\Longrightarrow$ *bit v j*
proof (*induction n*)
case *0*
then show *?case*
by (*metis diff-0 highestOneBitRec.simps leD of-nat-0-eq-iff of-nat-0-le-iff zle-diff1-eq*)

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

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

lemma *highestOneBitRecMax*:
highestOneBitRec n v ≤ *n*
by (*induction n*; *simp*)

lemma *highestOneBitRecElim*:
assumes *highestOneBitRec n v* = *j*
shows ((*j* = −*1* ∧ *v* = *0*) ∨ (*j* ≥ *0* ∧ *bit v j*))
using *assms highestOneBitRecTrue* **by** *blast*

lemma *highestOneBitRecZero*:

v = *0* ⟹ *highestOneBitRec* (*size v*) *v* = −*1*
by (*induction rule*: *highestOneBitRec.induct*; *simp*)

lemma *highestOneBitRecLess*:
assumes ¬ *bit v n*
shows *highestOneBitRec n v* = *highestOneBitRec* (*n* − *1*) *v*
using *assms* **by** *force*

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*))
by (*metis assms dual-order.refl lt2p-lem mask-eq-iff size-word.rep-eq*)

lemma *maskSmaller*:
fixes *v* :: *′a* :: *len word*
assumes ¬ *bit v n*
shows *and v* (*mask* (*Suc n*)) = *and v* (*mask n*)
unfolding *bit-eq-iff*
by (*metis assms bit-and-iff bit-mask-iff less-Suc-eq*)

lemma *highestOneBitSmaller*:
assumes *size v* = *Suc n*
assumes ¬ *bit v n*
shows *highestOneBit v* = *highestOneBit* (*and v* (*mask n*))
by (*metis assms highestOneBitMask maskSmaller*)

lemma *highestOneBitRecMask*:
shows *highestOneBit* (*and v* (*mask* (*Suc n*))) = *highestOneBitRec n v*
proof (*induction n*)
case *0*
then have *highestOneBit* (*and v* (*mask* (*Suc 0*))) = *highestOneBitRec 0 v*
apply *auto*
apply (*smt* (*verit*, *ccfv-threshold*) *neg-equal-zero negative-eq-positive bit-1-iff bit-and-iff*
highestOneBitN)
by (*simp add*: *bit-iff-and-push-bit-not-eq-0 highestOneBitNeg*)
then show *?case*
by *presburger*
next
case (*Suc n*)
then show *?case*
proof (*cases bit v* (*Suc n*))
case *True*
have *1*: *highestOneBitRec* (*Suc n*) *v* = *Suc n*
by (*simp add*: *True*)
have ∀ *i*::*nat*. (*int i* > (*Suc n*) ⟶ ¬ (*bit* (*and v* (*mask* (*Suc* (*Suc n*)))) *i*))
by (*simp add*: *bit-and-iff bit-mask-iff*)

then have *2*: *highestOneBit* (*and v* (*mask* (*Suc* (*Suc n*)))) = *Suc n*
using *True highestOneBitN*
by (*metis bit-take-bit-iff lessI take-bit-eq-mask*)
then show *?thesis*
using *1 2* **by** *auto*
next
case *False*
then show *?thesis*
by (*simp add*: *Suc maskSmaller*)
qed
qed

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

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

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

2.2 Long.lowestOneBit

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

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

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

2.3 Long.numberOfLeadingZeros

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

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

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

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

lemma *highestOneBit* (*0*::*64 word*) = −*1*

by (*simp add*: *MaxOrNeg-neg highestOneBit-def*)

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

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

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

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

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

2.4 Long.numberOfTrailingZeros

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

lemma *lowestOneBit-bot*: *lowestOneBit* (*0*::*64 word*) = *64*
unfolding *lowestOneBit-def MinOrHighest-def*
by (*simp add*: *size64*)

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

lemma *nat-bot-set*: (*0*::*nat*) ∈ *xs* ⟶ (∀ *x* ∈ *xs* . *0* ≤ *x*)
by *fastforce*

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

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

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*)}
using *finite-nat-set-iff-bounded* **by** *blast*

lemma *negone-set*:
bit (−*1*::('a::len) *word*) *n* ⟷ *n* < *LENGTH*('a)
by *simp*

lemma *negone-all-bits*:
{*n* . *bit* (−*1*::('a::len) *word*) *n*} = {*n* . *0* ≤ *n* ∧ *n* < *LENGTH*('a)}
using *negone-set*
by *auto*

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

lemma *card-of-range*:
x = *card* {*n* . *0* ≤ *n* ∧ *n* < *x*}
by *simp*

lemma *range-of-nat*:

$\{(n::nat)\ .\ 0 \leq n \wedge n < x\} = \{n\ .\ n < x\}$
by *simp*

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

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

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

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

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

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

lemma *zerosBelowLowestOne*:
assumes $n <$ *lowestOneBit* a

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

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

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

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

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

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

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

apply *auto*[*1*]
by *auto*

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

lemma *intersect-bitCount-helper*:
$\mathit{card}\ \{n\ .\ n < \mathit{Nat.size}\ a\} - \mathit{bitCount}\ a = \mathit{card}\ \{n\ .\ n < \mathit{Nat.size}\ a \wedge \neg(\mathit{bit}\ a\ n)\}$
proof –
have *size-def*: $\mathit{Nat.size}\ a = \mathit{card}\ \{n\ .\ n < \mathit{Nat.size}\ a\}$
using *card-of-range* **by** *simp*
have *bitCount-def*: $\mathit{bitCount}\ a = \mathit{card}\ \{n\ .\ n < \mathit{Nat.size}\ a \wedge \mathit{bit}\ a\ n\}$
using *qualified-bitCount* **by** *auto*
have *disjoint*: $\{n\ .\ n < \mathit{Nat.size}\ a \wedge \mathit{bit}\ a\ n\} \cap \{n\ .\ n < \mathit{Nat.size}\ a \wedge \neg(\mathit{bit}\ a\ n)\} = \{\}$
using *disjoint-bit-sets* **by** *auto*
have *union*: $\{n\ .\ n < \mathit{Nat.size}\ a \wedge \mathit{bit}\ a\ n\} \cup \{n\ .\ n < \mathit{Nat.size}\ a \wedge \neg(\mathit{bit}\ a\ n)\} = \{n\ .\ n < \mathit{Nat.size}\ a\}$
using *union-bit-sets* **by** *auto*
show *?thesis*
unfolding *bitCount-def*
apply (*rule card-eq*)
using *finite-range* **apply** *simp*
using *union* **apply** *blast*
using *disjoint* **by** *simp*
qed

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

hide-fact *intersect-bitCount-helper*

end

3 Operator Semantics

theory *Values*
imports

JavaLong
begin

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

Note that Java supports 64, 32, 16, 8 signed ints, plus 1 bit (boolean) ints, and char is 16-bit unsigned. E.g. an 8-bit stamp has a default range of -128..+127. And a 1-bit stamp has a default range of -1..0, surprisingly.

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

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

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

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

  IntVal iwidth int64 |

  ObjRef objref |
  ObjStr string |
  ArrayVal length Value list

fun *intval-bits* :: *Value* ⇒ *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*)
by *simp*

lemma *neg-one-signed*[*simp*]:
assumes *0* < *b*
shows *int-signed-value b* (*neg-one b*) = −*1*
using *assms* **apply** *auto*
by (*metis* (*no-types*, *lifting*) *Suc-pred diff-Suc-1 signed-take-take-bit assms signed-minus-1 int-signed-value.simps mask-eq-take-bit-minus-one signed-take-bit-of-minus-1*)

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

3.1 Arithmetic Operators

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

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

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

```
fun intval-add :: Value ⇒ Value ⇒ Value where
  intval-add (IntVal b1 v1) (IntVal b2 v2) =
    (if b1 = b2 then IntVal b1 (take-bit b1 (v1+v2)) else UndefVal) |
  intval-add - - = UndefVal
```

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

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

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

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

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

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

       let x1 = (v1 >> 32)              in
       let x2 = (and v1 4294967295)      in
       let y1 = (v2 >> 32)              in
       let y2 = (and v2 4294967295)      in
       let z2 = (x2 ∗ y2)               in
```

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

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

        (new-int b1 result)
      ) else (

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

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

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

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

      let result = (r >> 32) in

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

fun *intval-reverse-bytes* :: *Value* ⇒ *Value* **where**
  intval-reverse-bytes (IntVal b1 v1) = (new-int b1 (reverseBytes-fun v1 b1 0)) |
  intval-reverse-bytes - = UndefVal

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

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

intval-negate - = UndefVal

fun *intval-abs* :: *Value* ⇒ *Value* **where**
intval-abs (*IntVal t v*) = *new-int t* (*if int-signed-value t v* < *0 then* − *v else v*) |
intval-abs - = UndefVal

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

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

3.2 Bitwise Operators

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

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

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

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

3.3 Comparison Operators

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

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

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

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

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

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

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

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

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

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

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

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

fun *default-value* :: *string* ⇒ *Value* **where**
default-value n = (*if* (*find-index n short-types-32*) < (*length short-types-32*)
then (*default-values*!*0*) *else*
(*if* (*find-index n short-types-64*) < (*length short-types-64*)
then (*default-values*!*1*)
else (*default-values*!*2*)))

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

fun *intval-new-array* :: *Value* ⇒ *string* ⇒ *Value* **where**

intval-new-array (*IntVal b1 v1*) *s* = (*ArrayVal* (*nat* (*int-signed-value b1 v1*))
(*populate-array* (*nat* (*int-signed-value b1 v1*)) [] *s*)) |
intval-new-array - - = *UndefVal*

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

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

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

3.4 Narrowing and Widening Operators

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

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

corollary *sint* (*signed-take-bit 0* (*1* :: *int32*)) = −*1* **by** *code-simp*
corollary *sint* (*signed-take-bit 7* ((*256* + *128*) :: *int64*)) = −*128* **by** *code-simp*
corollary *sint* (*take-bit 7* ((*256* + *128* + *64*) :: *int64*)) = *64* **by** *code-simp*
corollary *sint* (*take-bit 8* ((*256* + *128* + *64*) :: *int64*)) = *128* + *64* **by** *code-simp*

fun *intval-narrow* :: *nat* ⇒ *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*
using *assms* **apply** (*cases val*; *auto*) **apply** (*meson le-trans*)+ **by** *presburger*

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

lemma *intval-zero-extend-ok*:
assumes *intval-zero-extend inBits outBits val* ≠ *UndefVal*
shows *0* < *inBits* ∧
inBits ≤ *outBits* ∧ *outBits* ≤ *64* ∧
is-IntVal val ∧
intval-bits val = *inBits*
by (*metis intval-bits.simps intval-zero-extend.elims is-IntVal-def assms*)

3.5 Bit-Shifting Operators

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

fun *shift-amount* :: *iwidth* ⇒ *int64* ⇒ *nat* **where**
shift-amount b val = *unat* (*and val* (*if b* = *64 then 0x3F else 0x1f*))

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

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

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

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

3.5.1 Examples of Narrowing / Widening Functions

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

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

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

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

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

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

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

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

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

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

end

3.6 Fixed-width Word Theories

theory *ValueThms*
  imports *Values*
begin

3.6.1 Support Lemmas for Upper/Lower Bounds

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

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

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

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

Some min/max bounds for 64-bit words

lemma *bit-bounds-min64*: $((fst\ (bit\text{-}bounds\ 64))) \le (sint\ (v::int64))$
  using *sint-ge*[*of v*] **by** *simp*

lemma *bit-bounds-max64*: $((snd\ (bit\text{-}bounds\ 64))) \ge (sint\ (v::int64))$
  using *sint-lt*[*of v*] **by** *simp*

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

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

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

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

lemma *signed-take-bit-int-less-exp-word*:
  fixes *ival :: 'a :: len word*
  assumes $n < LENGTH('a)$
  shows $sint(signed\text{-}take\text{-}bit\ n\ ival) < (2::int)\ \hat{}\ n$

apply *transfer*
by (*smt* (*verit*) *not-take-bit-negative signed-take-bit-eq-take-bit-shift signed-take-bit-int-less-exp take-bit-int-greater-self-iff*)

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

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

A *bit_bounds* version of the above lemma.

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

lemma *signed-take-bit-bounds64*:
fixes *ival* :: *int64*
assumes $n \leq 64$
assumes $0 < n$
assumes $val = sint(signed\text{-}take\text{-}bit\ (n - 1)\ ival)$
shows $fst\ (bit\text{-}bounds\ n) \leq val \wedge val \leq snd\ (bit\text{-}bounds\ n)$
by (*metis size64 word-size signed-take-bit-bounds assms*)

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

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

using *assms int-signed-value-range* **by** *blast*

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

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

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

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

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

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

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

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

lemma *scast-max-bound*:
assumes *sint* (*v* :: *′a* :: *len word*) < *M*
assumes *LENGTH*(*′a*) < *LENGTH*(*′b*)
shows *sint* ((*scast v*) :: *′b* :: *len word*) < *M*
using *scast-max-bound assms* **by** *fast*

lemma *scast-min-bound*:
assumes *M* ≤ *sint* (*v* :: *′a* :: *len word*)
assumes *LENGTH*(*′a*) < *LENGTH*(*′b*)
shows *M* ≤ *sint* ((*scast v*) :: *′b* :: *len word*)
by (*simp add*: *scast-min-bound assms*)

lemma *scast-bigger-max-bound*:
assumes (*result* :: *′b* :: *len word*) = *scast* (*v* :: *′a* :: *len word*)
shows *sint result* < *2* ^ *LENGTH*(*′a*) *div 2*
using *assms scast-bigger-max-bound* **by** *blast*

lemma *scast-bigger-min-bound*:
assumes (*result* :: *′b* :: *len word*) = *scast* (*v* :: *′a* :: *len word*)
shows − (*2* ^ *LENGTH*(*′a*) *div 2*) ≤ *sint result*
using *scast-bigger-min-bound assms* **by** *blast*

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

Results about *new_int*.

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

3.6.2 Support lemmas for take bit and signed take bit.

Lemmas for removing redundant take_bit wrappers.

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

qed

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

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

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

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

lemma *signed-take-take-bit*[*simp*]:
  fixes *x* :: *'a* :: *len word*
  assumes *0* < *b*
  shows *signed-take-bit* (*b* − *1*) (*take-bit b x*) = *signed-take-bit* (*b* − *1*) *x*
  using *signed-take-take-bit assms* **by** *blast*

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

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

end

4 Stamp Typing

theory *Stamp*
  imports *Values*
begin

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

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

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

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

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

```
abbreviation IntegerStampM :: nat ⇒ 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*) =
($0 < bits \wedge bits \leq 64 \wedge$
fst (*bit-bounds bits*) ≤ *lo* ∧ *lo* ≤ *snd* (*bit-bounds bits*) ∧
fst (*bit-bounds bits*) ≤ *hi* ∧ *hi* ≤ *snd* (*bit-bounds bits*)) |
valid-stamp s = *True*

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

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

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

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

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

empty-stamp (*KlassPointerStamp nonNull alwaysNull*) = (*KlassPointerStamp nonNull alwaysNull*) |
empty-stamp (*MethodCountersPointerStamp nonNull alwaysNull*) = (*MethodCountersPointerStamp nonNull alwaysNull*) |

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

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

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

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

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

else (*MethodPointersStamp* (*nn1* ∨ *nn2*) (*an1* ∨ *an2*))
) |
join s1 s2 = *IllegalStamp*

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

fun *asConstant* :: *Stamp* ⇒ *Value* **where**
asConstant (*IntegerStamp b l h*) = (*if l* = *h then new-int b* (*word-of-int l*) *else UndefVal*) |
asConstant - = *UndefVal*

— Determine if two stamps never have value overlaps i.e. their join is empty
fun *alwaysDistinct* :: *Stamp* ⇒ *Stamp* ⇒ *bool* **where**
alwaysDistinct stamp1 stamp2 = *is-stamp-empty* (*join stamp1 stamp2*)

— Determine if two stamps must always be the same value i.e. two equal constants
fun *neverDistinct* :: *Stamp* ⇒ *Stamp* ⇒ *bool* **where**
neverDistinct stamp1 stamp2 = (*asConstant stamp1* = *asConstant stamp2* ∧ *asConstant stamp1* ≠ *UndefVal*)

fun *constantAsStamp* :: *Value* ⇒ *Stamp* **where**
constantAsStamp (*IntVal b v*) = (*IntegerStamp b* (*int-signed-value b v*) (*int-signed-value b v*)) |
constantAsStamp (*ObjRef* (*None*)) = *ObjectStamp* '''' *False False True* |
constantAsStamp (*ObjRef* (*Some n*)) = *ObjectStamp* '''' *False True False* |

constantAsStamp - = *IllegalStamp*

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

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

definition *wf-value* :: *Value* ⇒ *bool* **where**

wf-value v = *valid-value v* (*constantAsStamp v*)

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

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

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

— The most common type of stamp within the compiler (apart from the VoidStamp) is a 32 bit integer stamp with an unrestricted range. We use *default-stamp* as it is a frequently used stamp.
definition *default-stamp* :: *Stamp* **where**
default-stamp = (*unrestricted-stamp* (*IntegerStamp 32 0 0*))

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

5 Graph Representation

5.1 IR Graph Nodes

theory *IRNodes*
imports
Values
begin

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

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

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

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

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

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

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

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

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

| *FixedGuardNode* (*ir-condition*: *INPUT-COND*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *FrameState* (*ir-monitorIds*: *INPUT-ASSOC list*) (*ir-outerFrameState-opt*: *INPUT-STATE option*) (*ir-values-opt*: *INPUT list option*) (*ir-virtualObjectMappings-opt*: *INPUT-STATE list option*)

| *IfNode* (*ir-condition*: *INPUT-COND*) (*ir-trueSuccessor*: *SUCC*) (*ir-falseSuccessor*: *SUCC*)
| *IntegerBelowNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *IntegerEqualsNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *IntegerLessThanNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *IntegerMulHighNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *IntegerNormalizeCompareNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *IntegerTestNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *InvokeNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *InvokeWithExceptionNode* (*ir-nid*: *ID*) (*ir-callTarget*: *INPUT-EXT*) (*ir-classInit-opt*: *INPUT option*) (*ir-stateDuring-opt*: *INPUT-STATE option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*) (*ir-exceptionEdge*: *SUCC*)
| *IsNullNode* (*ir-value*: *INPUT*)
| *KillingBeginNode* (*ir-next*: *SUCC*)
| *LeftShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *LoadFieldNode* (*ir-nid*: *ID*) (*ir-field*: *string*) (*ir-object-opt*: *INPUT option*) (*ir-next*: *SUCC*)
| *LoadIndexedNode* (*ir-index*: *INPUT*) (*ir-guard-opt*: *INPUT-GUARD option*) (*ir-value*: *INPUT*) (*ir-next*: *SUCC*)
| *LogicNegationNode* (*ir-value*: *INPUT-COND*)
| *LoopBeginNode* (*ir-ends*: *INPUT-ASSOC list*) (*ir-overflowGuard-opt*: *INPUT-GUARD option*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *LoopEndNode* (*ir-loopBegin*: *INPUT-ASSOC*)
| *LoopExitNode* (*ir-loopBegin*: *INPUT-ASSOC*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *MergeNode* (*ir-ends*: *INPUT-ASSOC list*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *MethodCallTargetNode* (*ir-targetMethod*: *string*) (*ir-arguments*: *INPUT list*) (*ir-invoke-kind*: *IRInvokeKind*)
| *MulNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *NarrowNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)
| *NegateNode* (*ir-value*: *INPUT*)
| *NewArrayNode* (*ir-length*: *INPUT*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *NewInstanceNode* (*ir-nid*: *ID*) (*ir-instanceClass*: *string*) (*ir-stateBefore-opt*: *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*: *IN-*

PUT-GUARD option) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

| *SignedFloatingIntegerDivNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *SignedFloatingIntegerRemNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *SignedRemNode* (*ir-nid*: *ID*) (*ir-x*: *INPUT*) (*ir-y*: *INPUT*) (*ir-zeroCheck-opt*: *INPUT-GUARD option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *StartNode* (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *StoreFieldNode* (*ir-nid*: *ID*) (*ir-field*: *string*) (*ir-value*: *INPUT*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-object-opt*: *INPUT option*) (*ir-next*: *SUCC*)
| *StoreIndexedNode* (*ir-storeCheck*: *INPUT-GUARD option*) (*ir-value*: *ID*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-index*: *INPUT*) (*ir-guard-opt*: *INPUT-GUARD option*) (*ir-array*: *INPUT*) (*ir-next*: *SUCC*)
| *SubNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *UnsignedRightShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *UnwindNode* (*ir-exception*: *INPUT*)
| *ValuePhiNode* (*ir-nid*: *ID*) (*ir-values*: *INPUT list*) (*ir-merge*: *INPUT-ASSOC*)
| *ValueProxyNode* (*ir-value*: *INPUT*) (*ir-loopExit*: *INPUT-ASSOC*)
| *XorNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *ZeroExtendNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)
| *NoNode*

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

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

fun *opt-list-to-list* :: *'a list option* ⇒ *'a list* **where**
opt-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*
by *simp*

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

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

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

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

end

5.2 IR Graph Node Hierarchy

theory *IRNodeHierarchy*
imports *IRNodes*
begin

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

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

These functions have been automatically generated from the compiler.

fun *is-EndNode* :: *IRNode* ⇒ *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*))
∨

$((is\text{-}SignedRemNode\ n1) \wedge (is\text{-}SignedRemNode\ n2)) \vee$
$((is\text{-}SignExtendNode\ n1) \wedge (is\text{-}SignExtendNode\ n2)) \vee$
$((is\text{-}StartNode\ n1) \wedge (is\text{-}StartNode\ n2)) \vee$
$((is\text{-}StoreFieldNode\ n1) \wedge (is\text{-}StoreFieldNode\ n2)) \vee$
$((is\text{-}SubNode\ n1) \wedge (is\text{-}SubNode\ n2)) \vee$
$((is\text{-}UnsignedRightShiftNode\ n1) \wedge (is\text{-}UnsignedRightShiftNode\ n2)) \vee$
$((is\text{-}UnwindNode\ n1) \wedge (is\text{-}UnwindNode\ n2)) \vee$
$((is\text{-}ValuePhiNode\ n1) \wedge (is\text{-}ValuePhiNode\ n2)) \vee$
$((is\text{-}ValueProxyNode\ n1) \wedge (is\text{-}ValueProxyNode\ n2)) \vee$
$((is\text{-}XorNode\ n1) \wedge (is\text{-}XorNode\ n2)) \vee$
$((is\text{-}ZeroExtendNode\ n1) \wedge (is\text{-}ZeroExtendNode\ n2)))$

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

setup-lifting *type-definition-IRGraph*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

code-datatype *irgraph*

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

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

lemma *dom-eq*:
assumes ∀ *x* ∈ *set xs. fst* (*snd x*) ≠ *NoNode*
shows *filter-none* (*map-of xs*) = *dom* (*map-of xs*)
using *assms map-of-SomeD* **by** *fastforce*

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

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

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

— Get the inputs set of a given node ID
fun *inputs* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
inputs g nid = *set* (*inputs-of* (*kind g nid*))
— Get the successor set of a given node ID
fun *succ* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
succ g nid = *set* (*successors-of* (*kind g nid*))
— Gives a relation between node IDs - between a node and its input nodes
fun *input-edges* :: *IRGraph* ⇒ *ID rel* **where**
input-edges g = ($\bigcup$ *i* ∈ *ids g*. {(*i*,*j*)|*j*. *j* ∈ (*inputs g i*)})
— Find all the nodes in the graph that have nid as an input - the usages of nid
fun *usages* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
usages g nid = {*i*. *i* ∈ *ids g* ∧ *nid* ∈ *inputs g i*}
fun *successor-edges* :: *IRGraph* ⇒ *ID rel* **where**
successor-edges g = ($\bigcup$ *i* ∈ *ids g*. {(*i*,*j*)|*j* . *j* ∈ (*succ g i*)})
fun *predecessors* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**
predecessors g nid = {*i*. *i* ∈ *ids g* ∧ *nid* ∈ *succ g i*}
fun *nodes-of* :: *IRGraph* ⇒ (*IRNode* ⇒ *bool*) ⇒ *ID set* **where**
nodes-of g sel = {*nid* ∈ *ids g* . *sel* (*kind g nid*)}
fun *edge* :: (*IRNode* ⇒ *'a*) ⇒ *ID* ⇒ *IRGraph* ⇒ *'a* **where**
edge sel nid g = *sel* (*kind g nid*)

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

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

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

lemma *ids-some*[*simp*]: *x* ∈ *ids g* ⟷ *kind g x* ≠ *NoNode*
proof −
have *that*: *x* ∈ *ids g* ⟶ *kind g x* ≠ *NoNode*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *remove-node-lookup*:
gup = *remove-node nid g* ⟶ *kind gup nid* = *NoNode* ∧ *stamp gup nid* = *IllegalStamp*
by (*simp add*: *kind.rep-eq remove-node.rep-eq stamp.rep-eq*)

lemma *replace-node-lookup*[*simp*]:
gup = *replace-node nid* (*k*, *s*) *g* ∧ *k* ≠ *NoNode* ⟶ *kind gup nid* = *k* ∧ *stamp gup nid* = *s*
by (*simp add*: *replace-node.rep-eq kind.rep-eq stamp.rep-eq*)

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

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

5.3.1 Example Graphs

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

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

Example 2: public static int sq(int x) return x * x;

[1 P(0)] / [0 Start] [4 *] | / V / [5 Return]

definition *eg2-sq* :: *IRGraph* **where**
eg2-sq = *irgraph* [
(*0*, *StartNode None 5*, *VoidStamp*),
(*1*, *ParameterNode 0*, *default-stamp*),
(*4*, *MulNode 1 1*, *default-stamp*),
(*5*, *ReturnNode* (*Some 4*) *None*, *default-stamp*)
]

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

end

5.4 Structural Graph Comparison

theory
Comparison
imports
IRGraph
begin

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

fun *find-ref-nodes* :: *IRGraph* ⇒ (*ID* ⇀ *ID*) **where**
find-ref-nodes g = *map-of*
(*map* (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* (*λnid. nid* ∉ *seen*) *to-see*)
in (*case l of* [] ⇒ *None* | *xs* ⇒ *Some* (*hd xs*)))

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

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

reachables **.**

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

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

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

fun *diffNodesInfo* :: *IRGraph* ⇒ *IRGraph* ⇒ (*ID* × *IRNode* × *IRNode*) *set* (**infix** $\cap_s$ *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

$[\![kind\ g\ nid = BeginNode\ nid';$

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

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

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

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

$[\![kind\ g\ nid = EndNode;$

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

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

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

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

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

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

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

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

— 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*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *Step* **.**

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* }
using *is-ground.simps*(*6*) **by** *blast*

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*

by (*cases op*; *auto*)

lemma *binary-ops-distinct-normal*:
shows $op \in binary\text{-}normal \Longrightarrow op \notin binary\text{-}fixed\text{-}32\text{-}ops \wedge op \notin binary\text{-}fixed\text{-}ops \wedge op \notin binary\text{-}shift\text{-}ops$
by *auto*

lemma *binary-ops-distinct-fixed-32*:
shows $op \in binary\text{-}fixed\text{-}32\text{-}ops \Longrightarrow op \notin binary\text{-}normal \wedge op \notin binary\text{-}fixed\text{-}ops \wedge op \notin binary\text{-}shift\text{-}ops$
by *auto*

lemma *binary-ops-distinct-fixed*:
shows $op \in binary\text{-}fixed\text{-}ops \Longrightarrow op \notin binary\text{-}fixed\text{-}32\text{-}ops \wedge op \notin binary\text{-}normal \wedge op \notin binary\text{-}shift\text{-}ops$
by *auto*

lemma *binary-ops-distinct-shift*:
shows $op \in binary\text{-}shift\text{-}ops \Longrightarrow op \notin binary\text{-}fixed\text{-}32\text{-}ops \wedge op \notin binary\text{-}fixed\text{-}ops \wedge op \notin binary\text{-}normal$
by *auto*

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 $op \in unary\text{-}fixed\text{-}32\text{-}ops \Longrightarrow op \notin boolean\text{-}unary \wedge op \notin normal\text{-}unary$
by *auto*

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*)
by (*cases op*; *cases x*; *cases y*; *auto*)

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 **.**

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 **.**

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* (- $\doteq$ - *55*) **where**
  ($e1 \doteq e2$) = (∀ *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*
  apply (*auto simp add*: *equivp-def equiv-exprs-def*) **by** (*metis equiv-exprs-def*)+

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*]:
    ($e_2 \leq e_1$) ⟷ (∀ *m p v*. (([*m*,*p*] ⊢ e_1 ↦ *v*) ⟶ ([*m*,*p*] ⊢ e_2 ↦ *v*)))

definition
  lt-expr-def [*simp*]:
    ($e_1 < e_2$) ⟷ ($e_1 \leq e_2$ ∧ ¬ ($e_1 \doteq e_2$))

instance proof
  fix *x y z* :: *IRExpr*
  show $x < y \longleftrightarrow x \leq y \land \neg (y \leq x)$ **by** (*simp add*: *equiv-exprs-def*; *auto*)
  show $x \leq x$ **by** *simp*
  show $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$ **by** *simp*
qed

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*] ⊢ *e* ↦ *IntVal b v* ⟹ (*and v* (*not* ((*ucast* (↑*e*))))) = *0*
    and *down-spec*: [*m*, *p*] ⊢ *e* ↦ *IntVal b v* ⟹ (*and* (*not v*) (*ucast* (↓*e*))) = *0*
begin

lemma *may-implies-either*:
  [*m*, *p*] ⊢ *e* ↦ *IntVal b v* ⟹ *bit* (↑*e*) *n* ⟹ *bit v n* = *False* ∨ *bit v n* = *True*
  by *simp*

lemma *not-may-implies-false*:
  [*m*, *p*] ⊢ *e* ↦ *IntVal b v* ⟹ ¬(*bit* (↑*e*) *n*) ⟹ *bit v n* = *False*
  by (*metis* (*no-types*, *lifting*) *bit.double-compl up-spec bit-and-iff bit-not-iff bit-unsigned-iff*

    down-spec)

lemma *must-implies-true*:
  [*m*, *p*] ⊢ *e* ↦ *IntVal b v* ⟹ *bit* (↓*e*) *n* ⟹ *bit v n* = *True*
  by (*metis bit.compl-one bit-and-iff bit-minus-1-iff bit-not-iff impossible-bit ucast-id down-spec*)

lemma *not-must-implies-either*:
  [*m*, *p*] ⊢ *e* ↦ *IntVal b v* ⟹ ¬(*bit* (↓*e*) *n*) ⟹ *bit v n* = *False* ∨ *bit v n* = *True*
  by *simp*

lemma *must-implies-may*:
  [*m*, *p*] ⊢ *e* ↦ *IntVal b v* ⟹ *n* < *32* ⟹ *bit* (↓*e*) *n* ⟹ *bit* (↑*e*) *n*
  by (*meson must-implies-true not-may-implies-false*)

lemma *up-mask-and-zero-implies-zero*:
  assumes *and* (↑*x*) (↑*y*) = *0*
  assumes [*m*, *p*] ⊢ *x* ↦ *IntVal b xv*
  assumes [*m*, *p*] ⊢ *y* ↦ *IntVal b yv*
  shows *and xv yv* = *0*
  by (*smt* (*z3*) *assms and.commute and.right-neutral bit.compl-zero bit.conj-cancel-right ucast-id*
    bit.conj-disj-distribs(*1*) *up-spec word-bw-assocs*(*1*) *word-not-dist*(*2*) *word-ao-absorbs*(*8*)
    and-eq-not-not-or)

lemma *not-down-up-mask-and-zero-implies-zero*:
  assumes *and* (*not* (↓*x*)) (↑*y*) = *0*
  assumes [*m*, *p*] ⊢ *x* ↦ *IntVal b xv*

assumes $[m,\ p] \vdash y \mapsto$ *IntVal b yv*
shows *and xv yv* = *yv*
by (*metis* (*no-types*, *opaque-lifting*) *assms bit.conj-cancel-left bit.conj-disj-distribs*(*1*,*2*) *bit.de-Morgan-disj ucast-id down-spec or-eq-not-not-and up-spec word-ao-absorbs*(*2*,*8*) *word-bw-lcs*(*1*) *word-not-dist*(*2*))

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*
by *simp*

lemma *ucast-minus-one*: (*ucast* (−*1*::*int64*)::*int32*) = −*1*
apply *transfer* **by** *auto*

interpretation *simple-mask*: *stamp-mask*
IRExpr-up :: *IRExpr* ⇒ *int64*
IRExpr-down :: *IRExpr* ⇒ *int64*
apply *unfold-locales*
by (*simp add*: *ucast-minus-one IRExpr-up-def IRExpr-down-def*)+

end

6.6 Data-flow Tree Theorems

theory *IRTreeEvalThms*
imports
Graph.ValueThms
IRTreeEval
begin

6.6.1 Deterministic Data-flow Evaluation

lemma *evalDet*:
$[m,p] \vdash e \mapsto v_1 \Longrightarrow$
$[m,p] \vdash e \mapsto v_2 \Longrightarrow$
$v_1 = v_2$
apply (*induction arbitrary*: v_2 *rule*: *evaltree.induct*) **by** (*elim EvalTreeE*; *auto*)+

lemma *evalAllDet*:
$[m,p] \vdash e\ [\mapsto]\ v1 \Longrightarrow$
$[m,p] \vdash e\ [\mapsto]\ v2 \Longrightarrow$
v1 = *v2*
apply (*induction arbitrary*: *v2 rule*: *evaltrees.induct*)
apply (*elim EvalTreeE*; *auto*)

using *evalDet* **by** *force*

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*
by (*cases op*; *cases x*; *auto*)

lemma *unary-eval-not-obj-str*:
shows *unary-eval op x* ≠ *ObjStr v*
by (*cases op*; *cases x*; *auto*)

lemma *unary-eval-not-array*:
shows *unary-eval op x* ≠ *ArrayVal len v*
by (*cases op*; *cases x*; *auto*)

lemma *unary-eval-int*:
assumes *unary-eval op x* ≠ *UndefVal*
shows *is-IntVal* (*unary-eval op x*)
by (*cases unary-eval op x*; *auto simp add*: *assms unary-eval-not-obj-ref unary-eval-not-obj-str unary-eval-not-array*)

lemma *bin-eval-int*:
assumes *bin-eval op x y* ≠ *UndefVal*
shows *is-IntVal* (*bin-eval op x y*)
using *assms*
apply (*cases op*; *cases x*; *cases y*; *auto simp add*: *is-IntVal-def*)
apply *presburger+*
prefer *3* **prefer** *4*
apply (*smt* (*verit*, *del-insts*) *new-int.simps*)
apply (*smt* (*verit*, *del-insts*) *new-int.simps*)
apply (*meson new-int-bin.simps*)+
apply (*meson bool-to-val.elims*)
apply (*meson bool-to-val.elims*)
apply (*smt* (*verit*, *del-insts*) *new-int.simps*)+
by (*metis bool-to-val.elims*)+

lemma *IntVal0*:
(*IntVal 32 0*) = (*new-int 32 0*)
by *auto*

lemma *IntVal1*:
  (*IntVal 32 1*) = (*new-int 32 1*)
  by *auto*

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*)
  using *is-IntVal-def assms*
proof (*cases op*)
  case *BinAdd*
  then show *?thesis*
    using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** *presburger*
next
  case *BinMul*
  then show *?thesis*
    using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** *presburger*
next
  case *BinDiv*
  then show *?thesis*
    using *assms* **apply** (*cases x*; *cases y*; *auto*)
    by (*meson new-int-bin.simps*)
next
  case *BinMod*
  then show *?thesis*
    using *assms* **apply** (*cases x*; *cases y*; *auto*)
    by (*meson new-int-bin.simps*)
next
  case *BinSub*
  then show *?thesis*
    using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** *presburger*
next
  case *BinAnd*
  then show *?thesis*
    using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** (*metis take-bit-and*)+
next
  case *BinOr*
  then show *?thesis*
    using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** (*metis take-bit-or*)+
next
  case *BinXor*
  then show *?thesis*
    using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** (*metis take-bit-xor*)+
next
  case *BinShortCircuitOr*
  then show *?thesis*
    using *assms* **apply** (*cases x*; *cases y*; *auto*)
    by (*metis IntVal1 bits-mod-0 bool-to-val.elims new-int.simps take-bit-eq-mod*)+

```
next
  case BinLeftShift
  then show ?thesis
    using assms by (cases x; cases y; auto)
next
  case BinRightShift
  then show ?thesis
   using assms apply (cases x; cases y; auto) by (smt (verit, del-insts) new-int.simps)+
next
  case BinURightShift
  then show ?thesis
    using assms by (cases x; cases y; auto)
next
  case BinIntegerEquals
  then show ?thesis
    using assms apply (cases x; cases y; auto)
    apply (metis (full-types) IntVal0 IntVal1 bool-to-val.simps(1,2) new-int.elims)
by presburger
next
  case BinIntegerLessThan
  then show ?thesis
    using assms apply (cases x; cases y; auto)
     apply (metis (no-types, opaque-lifting) bool-to-val.simps(1,2) bool-to-val.elims
new-int.simps
          IntVal1 take-bit-of-0)
    by presburger
next
  case BinIntegerBelow
  then show ?thesis
    using assms apply (cases x; cases y; auto)
   apply (metis bool-to-val.simps(1,2) bool-to-val.elims new-int.simps IntVal0 Int-
Val1)
    by presburger
next
  case BinIntegerTest
  then show ?thesis
    using assms apply (cases x; cases y; auto)
   apply (metis bool-to-val.simps(1,2) bool-to-val.elims new-int.simps IntVal0 Int-
Val1)
    by presburger
next
  case BinIntegerNormalizeCompare
  then show ?thesis
    using assms apply (cases x; cases y; auto) using take-bit-of-0 apply blast
   by (metis IntVal1 intval-word.simps new-int.elims take-bit-minus-one-eq-mask)+
next
  case BinIntegerMulHigh
  then show ?thesis
    using assms apply (cases x; cases y; auto)
```

prefer *2* **prefer** *5* **prefer** *8*
apply *presburger+*
by *metis+*
qed

lemma *int-stamp*:
assumes *is-IntVal v*
shows *is-IntegerStamp* (*constantAsStamp v*)
using *assms is-IntVal-def* **by** *auto*

lemma *validStampIntConst*:
assumes *v = IntVal b ival*
assumes *0 < b ∧ b ≤ 64*
shows *valid-stamp* (*constantAsStamp v*)
proof −
have *bnds*: *fst* (*bit-bounds b*) ≤ *int-signed-value b ival* ∧
int-signed-value b ival ≤ *snd* (*bit-bounds b*)
using *assms*(*2*) *int-signed-value-bounds* **by** *simp*
have *s*: *constantAsStamp v = IntegerStamp b* (*int-signed-value b ival*) (*int-signed-value b ival*)
using *assms*(*1*) **by** *simp*
then show *?thesis*
unfolding *s valid-stamp.simps* **using** *assms*(*2*) *bnds* **by** *linarith*
qed

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 −
have *bnds*: *fst* (*bit-bounds b*) ≤ *int-signed-value b ival* ∧
int-signed-value b ival ≤ *snd* (*bit-bounds b*)
using *assms*(*2*) *int-signed-value-bounds* **by** *simp*
have *s*: *constantAsStamp v = IntegerStamp b* (*int-signed-value b ival*) (*int-signed-value b ival*)
using *assms*(*1*) **by** *simp*
then show *?thesis*
using *assms validStampIntConst* **by** *simp*
qed

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*
apply (*rule valid-value.elims*(*1*)[*of val s True*]) **using** *assms* **by** *auto*

lemma *valid-VoidStamp*[*elim*]:
  shows *valid-value val VoidStamp* $\Longrightarrow$ *val* = *UndefVal*
  by *simp*

lemma *valid-ObjStamp*[*elim*]:
  shows *valid-value val* (*ObjectStamp klass exact nonNull alwaysNull*) $\Longrightarrow$ ($\exists$ *v*. *val* = *ObjRef v*)
  by (*metis Value.exhaust valid-value.simps*(*3*,*11*,*12*,*18*))

lemma *valid-int*[*elim*]:
  shows *valid-value val* (*IntegerStamp b lo hi*) $\Longrightarrow$ ($\exists$ *v*. *val* = *IntVal b v*)
  using *valid-value.elims*(*2*) **by** *fastforce*

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

lemma *evaltree-not-undef*:
  fixes *m p e v*
  shows ([*m*,*p*] $\vdash$ *e* $\mapsto$ *v*) $\Longrightarrow$ *v* $\neq$ *UndefVal*
  apply (*induction rule*: *evaltree.induct*) **by** (*auto simp add*: *wf-value-def*)

lemma *leafint*:
  assumes [*m*,*p*] $\vdash$ *LeafExpr i* (*IntegerStamp b lo hi*) $\mapsto$ *val*
  shows $\exists$ *b v*. *val* = (*IntVal b v*)

proof −
  have *valid-value val* (*IntegerStamp b lo hi*)
    using *assms* **by** (*rule LeafExprE*; *simp*)
  then show *?thesis*
    by *auto*
qed

lemma *default-stamp* [*simp*]: *default-stamp* = *IntegerStamp 32* (−*2147483648*) *2147483647*
  by (*auto simp add*: *default-stamp-def*)

lemma *valid-value-signed-int-range* [*simp*]:
  assumes *valid-value val* (*IntegerStamp b lo hi*)
  assumes *lo* < *0*
  shows $\exists$ *v*. (*val* = *IntVal b v* $\wedge$
            lo $\leq$ *int-signed-value b v* $\wedge$
            int-signed-value b v $\leq$ *hi*)
  by (*metis valid-value.simps*(*1*) *assms*(*1*) *valid-int*)

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 $(\mathit{UnaryExpr\ op\ x}) \geq (\mathit{UnaryExpr\ op\ x'})$
  using *assms* **by** *auto*

lemma *mono-binary*:
  assumes $x \geq x'$
  assumes $y \geq y'$
  shows $(\mathit{BinaryExpr\ op\ x\ y}) \geq (\mathit{BinaryExpr\ op\ x'\ y'})$
  using *BinaryExpr assms* **by** *auto*

lemma *never-void*:
  assumes $[m, p] \vdash x \mapsto xv$
  assumes *valid-value xv* (*stamp-expr xe*)
  shows *stamp-expr xe* $\neq$ *VoidStamp*
  using *assms*(*2*) **by** *force*

lemma *compatible-trans*:
  compatible x y $\wedge$ *compatible y z* $\Longrightarrow$ *compatible x z*
  by (*cases x*; *cases y*; *cases z*; *auto*)

lemma *compatible-refl*:
  compatible x y $\Longrightarrow$ *compatible y x*
  using *compatible.elims*(*2*) **by** *fastforce*

lemma *mono-conditional*:
  assumes $c \geq c'$
  assumes $t \geq t'$
  assumes $f \geq f'$
  shows $(\mathit{ConditionalExpr\ c\ t\ f}) \geq (\mathit{ConditionalExpr\ c'\ t'\ f'})$
proof (*simp only*: *le-expr-def*; (*rule allI*)+; *rule impI*)

fix *m p v*
assume *a*: [*m*,*p*] ⊢ *ConditionalExpr c t f* ↦ *v*
then obtain *cond* **where** *c*: [*m*,*p*] ⊢ *c* ↦ *cond*
by *auto*
then have *c′*: [*m*,*p*] ⊢ *c′* ↦ *cond*
using *assms* **by** *simp*

then obtain *tr* **where** *tr*: [*m*,*p*] ⊢ *t* ↦ *tr*
using *a* **by** *auto*
then have *tr′*: [*m*,*p*] ⊢ *t′* ↦ *tr*
using *assms*(*2*) **by** *auto*
then obtain *fa* **where** *fa*: [*m*,*p*] ⊢ *f* ↦ *fa*
using *a* **by** *blast*
then have *fa′*: [*m*,*p*] ⊢ *f′* ↦ *fa*
using *assms*(*3*) **by** *auto*
define *branch* **where** *b*: *branch* = (*if val-to-bool cond then t else f*)
define *branch′* **where** *b′*: *branch′* = (*if val-to-bool cond then t′ else f′*)
then have *beval*: [*m*,*p*] ⊢ *branch* ↦ *v*
using *a b c evalDet* **by** *blast*

from *beval* **have** [*m*,*p*] ⊢ *branch′* ↦ *v*
using *assms* **by** (*auto simp add*: *b b′*)
then show [*m*,*p*] ⊢ *ConditionalExpr c′ t′ f′* ↦ *v*
using *c′ fa′ tr′* **by** (*simp add*: *evaltree-not-undef b′ ConditionalExpr*)
qed

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_eval / $unary_eval$ level, simply by saying $unfoldingunfold_evaltree$.

lemma *unfold-const*:
([*m*,*p*] ⊢ *ConstantExpr c* ↦ *v*) = (*wf-value v* ∧ *v* = *c*)
by *auto*

lemma *unfold-binary*:
shows ([*m*,*p*] ⊢ *BinaryExpr op xe ye* ↦ *val*) = (∃ *x y*.
(([*m*,*p*] ⊢ *xe* ↦ *x*) ∧
([*m*,*p*] ⊢ *ye* ↦ *y*) ∧
(*val* = *bin-eval op x y*) ∧
(*val* ≠ *UndefVal*)
)) (**is** *?L* = *?R*)
proof (*intro iffI*)
assume *3*: *?L*
show *?R* **by** (*rule evaltree.cases*[*OF 3*]; *blast+*)

next
  assume *?R*
  then obtain *x y* **where** $[m,p] \vdash xe \mapsto x$
        and $[m,p] \vdash ye \mapsto y$
        and *val* = *bin-eval op x y*
        and *val* ≠ *UndefVal*
    by *auto*
  then show *?L*
    by (*rule BinaryExpr*)
 qed

lemma *unfold-unary*:
  shows ($[m,p] \vdash$ *UnaryExpr op xe* ↦ *val*)
        = (∃ *x*.
            (($[m,p] \vdash xe \mapsto x$) ∧
             (*val* = *unary-eval op x*) ∧
             (*val* ≠ *UndefVal*)
            )) (**is** *?L* = *?R*)
  by *auto*

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-eval op x* ≠ *UndefVal*
  shows ∃ *b v*. (*unary-eval op x* = *new-int b v* ∧

          b = (*if op* ∈ *normal-unary*   *then intval-bits x* *else*
               if op ∈ *boolean-unary*   *then 32*   *else*
               if op ∈ *unary-fixed-32-ops then 32*   *else*
                                          ir-resultBits op))
proof (*cases op*)
  case *UnaryAbs*
  then show *?thesis*
    apply *auto*
      by (*metis intval-bits.simps intval-abs.simps*(*1*) *UnaryAbs def new-int.elims*
unary-eval.simps(*1*)
        intval-abs.elims)
next
  case *UnaryNeg*
  then show *?thesis*
    apply *auto*
   by (*metis def intval-bits.simps intval-negate.elims new-int.elims unary-eval.simps*(*2*))
next

```
  case UnaryNot
  then show ?thesis
    apply auto
     by (metis intval-bits.simps intval-not.elims new-int.simps unary-eval.simps(3)
def)
next
  case UnaryLogicNegation
  then show ?thesis
    apply auto
   by (metis intval-bits.simps UnaryLogicNegation intval-logic-negation.elims new-int.elims
def
        unary-eval.simps(4))
next
  case (UnaryNarrow x51 x52)
  then show ?thesis
    using assms apply auto
    subgoal premises p
    proof -
      obtain xb xvv where xvv: x = IntVal xb xvv
       by (metis UnaryNarrow def intval-logic-negation.cases intval-narrow.simps(2,3,4,5)
            unary-eval.simps(5))
      then have evalNotUndef: intval-narrow x51 x52 x ≠ UndefVal
        using p by fast
      then show ?thesis
        by (metis (no-types, lifting) new-int.elims intval-narrow.simps(1) xvv)
    qed done
next
  case (UnarySignExtend x61 x62)
  then show ?thesis
    using assms apply auto
    subgoal premises p
    proof -
      obtain xb xvv where xvv: x = IntVal xb xvv
        by (metis Value.exhaust intval-sign-extend.simps(2,3,4,5) p(2))
      then have evalNotUndef: intval-sign-extend x61 x62 x ≠ UndefVal
        using p by fast
      then show ?thesis
        by (metis intval-sign-extend.simps(1) new-int.elims xvv)
    qed done
next
  case (UnaryZeroExtend x71 x72)
  then show ?thesis
    using assms apply auto
    subgoal premises p
    proof -
      obtain xb xvv where xvv: x = IntVal xb xvv
        by (metis Value.exhaust intval-zero-extend.simps(2,3,4,5) p(2))
      then have evalNotUndef: intval-zero-extend x71 x72 x ≠ UndefVal
        using p by fast
```

then show *?thesis*
by (*metis intval-zero-extend.simps*(*1*) *new-int.elims xvv*)
qed done
next
case *UnaryIsNull*
then show *?thesis*
apply *auto*
by (*metis bool-to-val.simps*(*1*) *new-int.simps IntVal0 IntVal1 unary-eval.simps*(*8*) *assms def*
intval-is-null.elims bool-to-val.elims)
next
case *UnaryReverseBytes*
then show *?thesis*
apply *auto*
by (*metis intval-bits.simps intval-reverse-bytes.elims new-int.elims unary-eval.simps*(*9*) *def*)
next
case *UnaryBitCount*
then show *?thesis*
apply *auto*
by (*metis intval-bit-count.elims new-int.simps unary-eval.simps*(*10*) *intval-bit-count.simps*(*1*)
def)
qed

lemma *new-int-unused-bits-zero*:
assumes *IntVal b ival* = *new-int b ival0*
shows *take-bit b ival* = *ival*
by (*simp add*: *new-int-take-bits assms*)

lemma *unary-eval-unused-bits-zero*:
assumes *unary-eval op x* = *IntVal b ival*
shows *take-bit b ival* = *ival*
by (*metis unary-eval-new-int Value.inject*(*1*) *new-int.elims new-int-unused-bits-zero Value.simps*(*5*)
assms)

lemma *bin-eval-unused-bits-zero*:
assumes *bin-eval op x y* = (*IntVal b ival*)
shows *take-bit b ival* = *ival*
by (*metis bin-eval-new-int Value.distinct*(*1*) *Value.inject*(*1*) *new-int.elims new-int-take-bits*

assms)

lemma *eval-unused-bits-zero*:
$[m,p] \vdash xe \mapsto (IntVal\ b\ ix) \Longrightarrow take\text{-}bit\ b\ ix = ix$
proof (*induction xe*)
case (*UnaryExpr x1 xe*)
then show *?case*
by (*auto simp add*: *unary-eval-unused-bits-zero*)

next
  case (*BinaryExpr x1 xe1 xe2*)
  then show *?case*
    by (*auto simp add*: *bin-eval-unused-bits-zero*)
next
  case (*ConditionalExpr xe1 xe2 xe3*)
  then show *?case*
    by (*metis* (*full-types*) *EvalTreeE*(*3*))
next
  case (*ParameterExpr i s*)
  then have *valid-value* (*p*!*i*) *s*
    by *fastforce*
  then show *?case*
  by (*metis* (*no-types*, *opaque-lifting*) *Value.distinct*(*9*) *intval-bits.simps valid-value.elims*(*2*) *local.ParameterExpr ParameterExprE intval-word.simps*)
next
  case (*LeafExpr x1 x2*)
  then show *?case*
    apply *auto*
  by (*metis* (*no-types*, *opaque-lifting*) *intval-bits.simps intval-word.simps valid-value.elims*(*2*) *valid-value.simps*(*18*))
next
  case (*ConstantExpr x*)
  then show *?case*
  by (*metis EvalTreeE*(*1*) *constantAsStamp.simps*(*1*) *valid-value.simps*(*1*) *wf-value-def*)
next
  case (*ConstantVar x*)
  then show *?case*
    by *auto*
next
  case (*VariableExpr x1 x2*)
  then show *?case*
    by *auto*
qed

lemma *unary-normal-bitsize*:
  assumes *unary-eval op x* = *IntVal b ival*
  assumes *op* ∈ *normal-unary*
  shows ∃ *ix*. *x* = *IntVal b ix*
  using *assms* **apply** (*cases op*; *auto*) **prefer** *5*
 apply (*smt* (*verit*, *ccfv-threshold*) *Value.distinct*(*1*) *Value.inject*(*1*) *intval-reverse-bytes.elims new-int.simps*)
 by (*metis Value.distinct*(*1*) *Value.inject*(*1*) *intval-logic-negation.elims new-int.simps intval-not.elims intval-negate.elims intval-abs.elims*)+

lemma *unary-not-normal-bitsize*:
  assumes *unary-eval op x* = *IntVal b ival*
  assumes *op* ∉ *normal-unary* ∧ *op* ∉ *boolean-unary* ∧ *op* ∉ *unary-fixed-32-ops*
  shows *b* = *ir-resultBits op* ∧ *0* < *b* ∧ *b* ≤ *64*

apply (*cases op*) **prefer** *8* **prefer** *10* **prefer** *10* **using** *assms* **apply** *blast+*
by (*smt*(*verit*, *ccfv-SIG*) *Value.distinct*(*1*) *assms*(*1*) *intval-bits.simps intval-narrow.elims intval-narrow-ok intval-zero-extend.elims linorder-not-less neq0-conv new-int.simps unary-eval.simps*(*5*,*6*,*7*) *IRUnaryOp.sel*(*4*,*5*,*6*) *intval-sign-extend.elims*)+

lemma *unary-eval-bitsize*:
assumes *unary-eval op x* = *IntVal b ival*
assumes *2*: *x* = *IntVal bx ix*
assumes *0* < *bx* ∧ *bx* ≤ *64*
shows *0* < *b* ∧ *b* ≤ *64*
using *assms* **apply** (*cases op*; *simp*)
by (*metis Value.distinct*(*1*) *Value.inject*(*1*) *intval-narrow.simps*(*1*) *le-zero-eq int-val-narrow-ok*
new-int.simps le-zero-eq gr-zeroI)+

lemma *bin-eval-inputs-are-ints*:
assumes *bin-eval op x y* = *IntVal b ix*
obtains *xb yb xi yi* **where** *x* = *IntVal xb xi* ∧ *y* = *IntVal yb yi*
proof −
have *bin-eval op x y* ≠ *UndefVal*
by (*simp add*: *assms*)
then show *?thesis*
using *assms that* **by** (*cases op*; *cases x*; *cases y*; *auto*)
qed

lemma *eval-bits-1-64*:
[*m*,*p*] ⊢ *xe* ↦ (*IntVal b ix*) ⟹ *0* < *b* ∧ *b* ≤ *64*
proof (*induction xe arbitrary*: *b ix*)
case (*UnaryExpr op x2*)
then obtain *xv* **where**
xv: ([*m*,*p*] ⊢ *x2* ↦ *xv*) ∧
IntVal b ix = *unary-eval op xv*
by (*auto simp add*: *unfold-binary*)
then have *b* = (*if op* ∈ *normal-unary* *then intval-bits xv else*
if op ∈ *unary-fixed-32-ops then 32* *else*
if op ∈ *boolean-unary* *then 32* *else*
ir-resultBits op)
by (*metis Value.disc*(*1*) *Value.discI*(*1*) *Value.sel*(*1*) *new-int.simps unary-eval-new-int*)
then show *?case*
by (*metis xv linorder-le-cases linorder-not-less numeral-less-iff semiring-norm*(*76*,*78*) *gr0I*
unary-normal-bitsize unary-not-normal-bitsize UnaryExpr.IH)
next
case (*BinaryExpr op x y*)
then obtain *xv yv* **where**
xy: ([*m*,*p*] ⊢ *x* ↦ *xv*) ∧
([*m*,*p*] ⊢ *y* ↦ *yv*) ∧

IntVal b ix = bin-eval op xv yv
by (*auto simp add*: *unfold-binary*)
then have *def*: *bin-eval op xv yv* ≠ *UndefVal* **and** *xv*: *xv* ≠ *UndefVal* **and** *yv* ≠ *UndefVal*
using *evaltree-not-undef xy* **by** (*force*, *blast*, *blast*)
then have *b* = (*if op* ∈ *binary-fixed-32-ops then 32 else intval-bits xv*)
by (*metis xy intval-bits.simps new-int.simps bin-eval-new-int*)
then show *?case*
by (*smt* (*verit*, *best*) *Value.distinct*(*9*,*11*,*13*) *BinaryExpr.IH*(*1*) *xv bin-eval-inputs-are-ints xy*
intval-bits.elims le-add-same-cancel1 less-or-eq-imp-le numeral-Bit0 zero-less-numeral)
next
case (*ConditionalExpr xe1 xe2 xe3*)
then show *?case*
by (*metis* (*full-types*) *EvalTreeE*(*3*))
next
case (*ParameterExpr x1 x2*)
then show *?case*
apply *auto*
using *valid-value.elims*(*2*)
by (*metis valid-stamp.simps*(*1*) *intval-bits.simps valid-value.simps*(*18*))+
next
case (*LeafExpr x1 x2*)
then show *?case*
apply *auto*
using *valid-value.elims*(*1*,*2*)
by (*metis Value.inject*(*1*) *valid-stamp.simps*(*1*) *valid-value.simps*(*18*) *Value.distinct*(*9*))+
next
case (*ConstantExpr x*)
then show *?case*
by (*metis wf-value-def constantAsStamp.simps*(*1*) *valid-stamp.simps*(*1*) *valid-value.simps*(*1*) *EvalTreeE*(*1*))
next
case (*ConstantVar x*)
then show *?case*
by *auto*
next
case (*VariableExpr x1 x2*)
then show *?case*
by *auto*
qed

lemma *bin-eval-normal-bits*:
assumes *op* ∈ *binary-normal*
assumes *bin-eval op x y = xy*
assumes *xy* ≠ *UndefVal*
shows ∃ *xv yv xyv b*. (*x = IntVal b xv* ∧ *y = IntVal b yv* ∧ *xy = IntVal b xyv*)
using *assms* **apply** *simp*

```
  proof (cases op ∈ binary-normal)
  case True
  then show ?thesis
    proof -
      have operator: xy = bin-eval op x y
        by (simp add: assms(2))
      obtain xv xb where xv: x = IntVal xb xv
      by (metis assms(3) bin-eval-inputs-are-ints bin-eval-int is-IntVal-def operator)
      obtain yv yb where yv: y = IntVal yb yv
      by (metis assms(3) bin-eval-inputs-are-ints bin-eval-int is-IntVal-def operator)
       then have notUndefMeansWidthSame: bin-eval op x y ≠ UndefVal ⟹ (xb
= yb)
        using assms apply (cases op; auto)
           by (metis intval-xor.simps(1) intval-or.simps(1) intval-div.simps(1) int-
val-mod.simps(1) intval-and.simps(1) intval-sub.simps(1)
            intval-mul.simps(1) intval-add.simps(1) new-int-bin.elims xv)+
      then have inWidthsSame: xb = yb
        using assms(3) operator by auto
      obtain ob xyv where out: xy = IntVal ob xyv
        by (metis Value.collapse(1) assms(3) bin-eval-int operator)
      then have yb = ob
        using assms apply (cases op; auto)
           apply (simp add: inWidthsSame xv yv)+
           apply (metis assms(3) intval-bits.simps new-int.simps new-int-bin.elims)
            apply (metis xv yv Value.distinct(1) intval-mod.simps(1) new-int.simps
new-int-bin.elims)
          by (simp add: inWidthsSame xv yv)+
      then show ?thesis
      using xv yv inWidthsSame assms out by blast
  qed
next
  case False
  then show ?thesis
    using assms by simp
qed

lemma unfold-binary-width-bin-normal:
  assumes op ∈ binary-normal
  shows ⋀xv yv.
          IntVal b val = bin-eval op xv yv ⟹
          [m,p] ⊢ xe ↦ xv ⟹
          [m,p] ⊢ ye ↦ yv ⟹
          bin-eval op xv yv ≠ UndefVal ⟹
          ∃ xa.
          (([m,p] ⊢ xe ↦ IntVal b xa) ∧
           (∃ ya. (([m,p] ⊢ ye ↦ IntVal b ya) ∧
             bin-eval op xv yv = bin-eval op (IntVal b xa) (IntVal b ya))))
  using assms apply simp
  subgoal premises p for x y
```

proof −
obtain *xv yv* **where** *eval*: ([*m*,*p*] ⊢ *xe* ↦ *xv*) ∧ ([*m*,*p*] ⊢ *ye* ↦ *yv*)
using *p*(*2*,*3*) **by** *blast*
then obtain *xa bb* **where** *xa*: *xv* = *IntVal bb xa*
by (*metis bin-eval-inputs-are-ints evalDet p*(*1*,*2*))
then obtain *ya yb* **where** *ya*: *yv* = *IntVal yb ya*
by (*metis bin-eval-inputs-are-ints evalDet p*(*1*,*3*) *eval*)
then have *eqWidth*: *bb* = *b*
by (*metis intval-bits.simps p*(*1*,*2*,*4*) *assms eval xa bin-eval-normal-bits evalDet*)
then obtain *xy* **where** *eval0*: *bin-eval op x y* = *IntVal b xy*
by (*metis p*(*1*))
then have *sameVals*: *bin-eval op x y* = *bin-eval op xv yv*
by (*metis evalDet p*(*2*,*3*) *eval*)
then have *notUndefMeansSameWidth*: *bin-eval op xv yv* ≠ *UndefVal* ⟹ (*bb* = *yb*)
using *assms* **apply** (*cases op*; *auto*)
by (*metis intval-add.simps*(*1*) *intval-mul.simps*(*1*) *intval-div.simps*(*1*) *intval-mod.simps*(*1*) *intval-sub.simps*(*1*) *intval-and.simps*(*1*)
intval-or.simps(*1*) *intval-xor.simps*(*1*) *new-int-bin.simps xa ya*)+
have *unfoldVal*: *bin-eval op x y* = *bin-eval op* (*IntVal bb xa*) (*IntVal yb ya*)
unfolding *sameVals xa ya* **by** *simp*
then have *sameWidth*: *b* = *yb*
using *eqWidth notUndefMeansSameWidth p*(*4*) *sameVals* **by** *force*
then show *?thesis*
using *eqWidth eval xa ya unfoldVal* **by** *blast*
qed
done

lemma *unfold-binary-width*:
assumes *op* ∈ *binary-normal*
shows ([*m*,*p*] ⊢ *BinaryExpr op xe ye* ↦ *IntVal b val*) = (∃ *x y*.
(([*m*,*p*] ⊢ *xe* ↦ *IntVal b x*) ∧
([*m*,*p*] ⊢ *ye* ↦ *IntVal b y*) ∧
(*IntVal b val* = *bin-eval op* (*IntVal b x*) (*IntVal b y*)) ∧
(*IntVal b val* ≠ *UndefVal*)
)) (**is** *?L* = *?R*)
proof (*intro iffI*)
assume *3*: *?L*
show *?R*
apply (*rule evaltree.cases*[*OF 3*]) **apply** *auto*
apply (*cases op* ∈ *binary-normal*)
using *unfold-binary-width-bin-normal assms* **by** *force*+
next
assume *R*: *?R*
then obtain *x y* **where** [*m*,*p*] ⊢ *xe* ↦ *IntVal b x*
and [*m*,*p*] ⊢ *ye* ↦ *IntVal b y*
and *new-int b val* = *bin-eval op* (*IntVal b x*) (*IntVal b y*)
and *new-int b val* ≠ *UndefVal*
using *bin-eval-unused-bits-zero* **by** *force*

then show *?L*
  using *R* **by** *blast*
qed

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:
  ⟦*kind g n* = *ConstantNode c*⟧
    ⟹ *g* ⊢ *n* ≃ (*ConstantExpr c*) |

  ParameterNode:
  ⟦*kind g n* = *ParameterNode i*;

$stamp\ g\ n = s]\!]$
$\Longrightarrow g\ \vdash n \simeq (ParameterExpr\ i\ s)$ |

$ConditionalNode$:
$[\![kind\ g\ n = ConditionalNode\ c\ t\ f;$
$g \vdash c \simeq ce;$
$g \vdash t \simeq te;$
$g \vdash f \simeq fe]\!]$
$\Longrightarrow g \vdash n \simeq (ConditionalExpr\ ce\ te\ fe)$ |

$AbsNode$:
$[\![kind\ g\ n = AbsNode\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryAbs\ xe)$ |

$ReverseBytesNode$:
$[\![kind\ g\ n = ReverseBytesNode\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryReverseBytes\ xe)$ |

$BitCountNode$:
$[\![kind\ g\ n = BitCountNode\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryBitCount\ xe)$ |

$NotNode$:
$[\![kind\ g\ n = NotNode\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryNot\ xe)$ |

$NegateNode$:
$[\![kind\ g\ n = NegateNode\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryNeg\ xe)$ |

$LogicNegationNode$:
$[\![kind\ g\ n = LogicNegationNode\ x;$
$g \vdash x \simeq xe]\!]$
$\Longrightarrow g \vdash n \simeq (UnaryExpr\ UnaryLogicNegation\ xe)$ |

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

$MulNode$:

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

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

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

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

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

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

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

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

LeftShiftNode:
$[\![kind\ g\ n = LeftShiftNode\ x\ y;$

$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (\textit{BinaryExpr BinLeftShift xe ye})$ |

RightShiftNode:
$[\![\textit{kind } g\ n = \textit{RightShiftNode } x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (\textit{BinaryExpr BinRightShift xe ye})$ |

UnsignedRightShiftNode:
$[\![\textit{kind } g\ n = \textit{UnsignedRightShiftNode } x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (\textit{BinaryExpr BinURightShift xe ye})$ |

IntegerBelowNode:
$[\![\textit{kind } g\ n = \textit{IntegerBelowNode } x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (\textit{BinaryExpr BinIntegerBelow xe ye})$ |

IntegerEqualsNode:
$[\![\textit{kind } g\ n = \textit{IntegerEqualsNode } x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (\textit{BinaryExpr BinIntegerEquals xe ye})$ |

IntegerLessThanNode:
$[\![\textit{kind } g\ n = \textit{IntegerLessThanNode } x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (\textit{BinaryExpr BinIntegerLessThan xe ye})$ |

IntegerTestNode:
$[\![\textit{kind } g\ n = \textit{IntegerTestNode } x\ y;$
$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (\textit{BinaryExpr BinIntegerTest xe ye})$ |

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

IntegerMulHighNode:
$[\![\textit{kind } g\ n = \textit{IntegerMulHighNode } x\ y;$
$g \vdash x \simeq xe;$

$g \vdash y \simeq ye$⟧
$\Longrightarrow g \vdash n \simeq$ (*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-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* ⇒ *i* ⇒ *o* ⇒ *bool as exprE*) *rep* **.**

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

RepNil:
$g \vdash [] \; [\simeq] \; [] \mid$

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

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

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**
  $n \notin ids\ g \Longrightarrow fresh\text{-}id\ g\ n$

code-pred *fresh-id* **.**

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;
    $g' = add\text{-}node\ n\ node\ g$⟧
   ⟹ *unique g node* (g', n)

code-pred (*modes*: *i* ⇒ *i* ⇒ *o* ⇒ *bool as uniqueE*) *unique* **.**

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

  UnrepConstantNode:
  ⟦*unique g (ConstantNode c, constantAsStamp c)* (g_1, n)⟧
    ⟹ *g* ⊕ (*ConstantExpr c*) ⇝ (g_1, n) |

  UnrepParameterNode:
  ⟦*unique g (ParameterNode i, s)* (g_1, n)⟧
    ⟹ *g* ⊕ (*ParameterExpr i s*) ⇝ (g_1, n) |

  UnrepConditionalNode:
  ⟦*g* ⊕ *ce* ⇝ (g_1, c);
    g_1 ⊕ *te* ⇝ (g_2, t);
    g_2 ⊕ *fe* ⇝ (g_3, f);
    $s' = meet\ (stamp\ g_3\ t)\ (stamp\ g_3\ f)$;

$unique\ g_3\ (ConditionalNode\ c\ t\ f,\ s')\ (g_4,\ n)]\!]$
$\Longrightarrow g \oplus (ConditionalExpr\ ce\ te\ fe) \leadsto (g_4,\ n)\ |$

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

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

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

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

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

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

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

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

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

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

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

7.3 Lift Data-flow Tree Semantics

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

code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *i* ⇒ *o* ⇒ *bool*) *encodeeval* **.**

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

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *encodeEvalAll* **.**

7.4 Graph Refinement

definition *graph-represents-expression* :: *IRGraph* ⇒ *ID* ⇒ *IRExpr* ⇒ *bool*
  (- ⊢ - ⊴ - *50*)
  where
  $(g \vdash n \trianglelefteq e) = (\exists e' . (g \vdash n \simeq e') \wedge (e' \leq e))$

definition *graph-refinement* :: *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
  graph-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-refinement*:
  graph-refinement g1 g2 ⟹
  $(\forall n\ m\ p\ v.\ n \in ids\ g1 \longrightarrow ([g1, m, p] \vdash n \mapsto v) \longrightarrow ([g2, m, p] \vdash n \mapsto v))$
  by (*meson encodeeval.simps graph-refinement-def graph-represents-expression-def le-expr-def*)

7.5 Maximal Sharing

definition *maximal-sharing*:
  maximal-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-start* **where**
  wf-start g = (*0* ∈ *ids g* ∧
    is-StartNode (*kind g 0*))

definition *wf-closed* **where**
  wf-closed g =
    (∀ *n* ∈ *ids g* .
      inputs g n ⊆ *ids g* ∧
      succ g n ⊆ *ids g* ∧
      kind g n ≠ *NoNode*)

definition *wf-phis* **where**
  wf-phis g =

$(\forall\ n \in \mathit{ids}\ g.$
$\mathit{is\text{-}PhiNode}\ (\mathit{kind}\ g\ n) \longrightarrow$
$\mathit{length}\ (\mathit{ir\text{-}values}\ (\mathit{kind}\ g\ n))$
$= \mathit{length}\ (\mathit{ir\text{-}ends}$
$(\mathit{kind}\ g\ (\mathit{ir\text{-}merge}\ (\mathit{kind}\ g\ n)))))$

definition *wf-ends* **where**
$\mathit{wf\text{-}ends}\ g =$
$(\forall\ n \in \mathit{ids}\ g\ .$
$\mathit{is\text{-}AbstractEndNode}\ (\mathit{kind}\ g\ n) \longrightarrow$
$\mathit{card}\ (\mathit{usages}\ g\ n) > \mathit{0})$

fun *wf-graph* :: *IRGraph* ⇒ *bool* **where**
$\mathit{wf\text{-}graph}\ g = (\mathit{wf\text{-}start}\ g \wedge \mathit{wf\text{-}closed}\ g \wedge \mathit{wf\text{-}phis}\ g \wedge \mathit{wf\text{-}ends}\ g)$

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

fun *wf-stamps* :: *IRGraph* ⇒ *bool* **where**
$\mathit{wf\text{-}stamps}\ g = (\forall\ n \in \mathit{ids}\ g\ .$
$(\forall\ v\ m\ p\ e\ .\ (g \vdash n \simeq e) \wedge ([m,\ p] \vdash e \mapsto v) \longrightarrow \mathit{valid\text{-}value}\ v\ (\mathit{stamp\text{-}expr}\ e)))$

fun *wf-stamp* :: *IRGraph* ⇒ (*ID* ⇒ *Stamp*) ⇒ *bool* **where**
$\mathit{wf\text{-}stamp}\ g\ s = (\forall\ n \in \mathit{ids}\ g\ .$
$(\forall\ v\ m\ p\ e\ .\ (g \vdash n \simeq e) \wedge ([m,\ p] \vdash e \mapsto v) \longrightarrow \mathit{valid\text{-}value}\ v\ (s\ n)))$

lemma *wf-empty*: *wf-graph start-end-graph*
unfolding *wf-folds* **by** (*simp add*: *start-end-graph-def*)

lemma *wf-eg2-sq*: *wf-graph eg2-sq*
unfolding *wf-folds* **by** (*simp add*: *eg2-sq-def*)

fun *wf-logic-node-inputs* :: *IRGraph* ⇒ *ID* ⇒ *bool* **where**
$\mathit{wf\text{-}logic\text{-}node\text{-}inputs}\ g\ n =$
$(\forall\ \mathit{inp} \in \mathit{set}\ (\mathit{inputs\text{-}of}\ (\mathit{kind}\ g\ n))\ .\ (\forall\ v\ m\ p\ .\ ([g,\ m,\ p] \vdash \mathit{inp} \mapsto v) \longrightarrow \mathit{wf\text{-}bool}$
$v))$

fun *wf-values* :: *IRGraph* ⇒ *bool* **where**
$\mathit{wf\text{-}values}\ g = (\forall\ n \in \mathit{ids}\ g\ .$
$(\forall\ v\ m\ p\ .\ ([g,\ m,\ p] \vdash n \mapsto v) \longrightarrow$
$(\mathit{is\text{-}LogicNode}\ (\mathit{kind}\ g\ n) \longrightarrow$
$\mathit{wf\text{-}bool}\ v \wedge \mathit{wf\text{-}logic\text{-}node\text{-}inputs}\ g\ n)))$

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

lemma *other-node-unchanged*:
  assumes *changeonly ns g1 g2*
  assumes *nid* ∈ *ids g1*
  assumes *nid* ∉ *ns*
  shows *kind g1 nid* = *kind g2 nid*
  using *assms* **by** *simp*

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*
using *assms* **by** (*simp add*: *ids.rep-eq eval-uses.intros*(*1*))

lemma *not-in-g-inputs*:
assumes *nid* ∉ *ids g*
shows *inputs g nid* = {}
proof −
have *k*: *kind g nid* = *NoNode*
using *assms* **by** (*simp add*: *not-in-g*)
then show *?thesis*
by (*simp add*: *k*)
qed

lemma *child-member*:
assumes *n* = *kind g nid*
assumes *n* ≠ *NoNode*
assumes *List.member* (*inputs-of n*) *child*
shows *child* ∈ *inputs g nid*
by (*metis in-set-member inputs.simps assms*(*1*,*3*))

lemma *child-member-in*:
assumes *nid* ∈ *ids g*
assumes *List.member* (*inputs-of* (*kind g nid*)) *child*
shows *child* ∈ *inputs g nid*
by (*metis child-member ids-some assms*)

lemma *inp-in-g*:
assumes *n* ∈ *inputs g nid*
shows *nid* ∈ *ids g*
proof −
have *inputs g nid* ≠ {}
by (*metis empty-iff empty-set assms*)
then have *kind g nid* ≠ *NoNode*
by (*metis not-in-g-inputs ids-some*)
then show *?thesis*
by (*metis not-in-g*)
qed

lemma *inp-in-g-wf*:
assumes *wf-graph g*
assumes *n* ∈ *inputs g nid*
shows *n* ∈ *ids g*
using *assms wf-folds inp-in-g* **by** *blast*

lemma *kind-unchanged*:
  assumes *nid* ∈ *ids g1*
  assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
  shows *kind g1 nid* = *kind g2 nid*
proof −
  show *?thesis*
    using *assms eval-usages-self* **by** *simp*
qed

lemma *stamp-unchanged*:
  assumes *nid* ∈ *ids g1*
  assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
  shows *stamp g1 nid* = *stamp g2 nid*
  by (*meson assms eval-usages-self unchanged.elims*(*2*))

lemma *child-unchanged*:
  assumes *child* ∈ *inputs g1 nid*
  assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
  shows *unchanged* (*eval-usages g1 child*) *g1 g2*
 by (*smt assms eval-usages.simps mem-Collect-eq unchanged.simps use-inp use-trans*)

lemma *eval-usages*:
  assumes *us* = *eval-usages g nid*
  assumes *nid′* ∈ *ids g*
  shows *eval-uses g nid nid′* ⟷ *nid′* ∈ *us* (**is** *?P* ⟷ *?Q*)
  using *assms* **by** (*simp add*: *ids.rep-eq*)

lemma *inputs-are-uses*:
  assumes *nid′* ∈ *inputs g nid*
  shows *eval-uses g nid nid′*
  by (*metis assms use-inp*)

lemma *inputs-are-usages*:
  assumes *nid′* ∈ *inputs g nid*
  assumes *nid′* ∈ *ids g*
  shows *nid′* ∈ *eval-usages g nid*
  using *assms* **by** (*simp add*: *inputs-are-uses*)

lemma *inputs-of-are-usages*:
  assumes *List.member* (*inputs-of* (*kind g nid*)) *nid′*
  assumes *nid′* ∈ *ids g*
  shows *nid′* ∈ *eval-usages g nid*
  by (*metis assms in-set-member inputs.elims inputs-are-usages*)

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

shows *ls* ⊆ *us*
using *inputs-are-usages assms* **by** *blast*

lemma *elim-inp-set*:
assumes *k* = *kind g nid*
assumes *k* ≠ *NoNode*
assumes *child* ∈ *set* (*inputs-of k*)
shows *child* ∈ *inputs g nid*
using *assms* **by** *simp*

lemma *encode-in-ids*:
assumes *g* ⊢ *nid* ≃ *e*
shows *nid* ∈ *ids g*
using *assms* **apply** (*induction rule*: *rep.induct*) **by** *fastforce+*

lemma *eval-in-ids*:
assumes [*g*, *m*, *p*] ⊢ *nid* ↦ *v*
shows *nid* ∈ *ids g*
using *assms encode-in-ids* **by** (*auto simp add*: *encodeeval.simps*)

lemma *transitive-kind-same*:
assumes *unchanged* (*eval-usages g1 nid*) *g1 g2*
shows ∀ *nid′* ∈ (*eval-usages g1 nid*) . *kind g1 nid′* = *kind g2 nid′*
by (*meson unchanged.elims*(*1*) *assms*)

theorem *stay-same-encoding*:
assumes *nc*: *unchanged* (*eval-usages g1 nid*) *g1 g2*
assumes *g1*: *g1* ⊢ *nid* ≃ *e*
assumes *wf*: *wf-graph g1*
shows *g2* ⊢ *nid* ≃ *e*
proof −
have *dom*: *nid* ∈ *ids g1*
using *g1 encode-in-ids* **by** *simp*
show *?thesis*
using *g1 nc wf dom*
proof (*induction e rule*: *rep.induct*)
case (*ConstantNode n c*)
then have *kind g2 n* = *ConstantNode c*
by (*metis kind-unchanged*)
then show *?case*
using *rep.ConstantNode* **by** *presburger*
next
case (*ParameterNode n i s*)
then have *kind g2 n* = *ParameterNode i*
by (*metis kind-unchanged*)
then show *?case*
by (*metis ParameterNode.hyps*(*2*) *ParameterNode.prems*(*1*,*3*) *rep.ParameterNode stamp-unchanged*)
next

case (*ConditionalNode n c t f ce te fe*)
then have *kind g2 n = ConditionalNode c t f*
by (*metis kind-unchanged*)
have *c ∈ eval-usages g1 n ∧ t ∈ eval-usages g1 n ∧ f ∈ eval-usages g1 n*
by (*metis inputs-of-ConditionalNode ConditionalNode.hyps*(*1*,*2*,*3*,*4*) *encode-in-ids inputs.simps*
inputs-are-usages list.set-intros(*1*) *set-subset-Cons subset-code*(*1*))
then show *?case*
by (*metis ConditionalNode.hyps*(*1*) *ConditionalNode.prems*(*1*) *IRNodes.inputs-of-ConditionalNode*

‹*kind g2 n = ConditionalNode c t f*› *child-unchanged inputs.simps list.set-intros*(*1*)

local.ConditionalNode(*5*,*6*,*7*,*9*) *rep.ConditionalNode set-subset-Cons subset-code*(*1*)
unchanged.elims(*2*))
next
case (*AbsNode n x xe*)
then have *kind g2 n = AbsNode x*
by (*metis kind-unchanged*)
then have *x ∈ eval-usages g1 n*
by (*metis inputs-of-AbsNode AbsNode.hyps*(*1*,*2*) *encode-in-ids inputs.simps inputs-are-usages*
list.set-intros(*1*))
then show *?case*
by (*metis AbsNode.IH AbsNode.hyps*(*1*) *AbsNode.prems*(*1*,*3*) *IRNodes.inputs-of-AbsNode rep.AbsNode*
‹*kind g2 n = AbsNode x*› *child-member-in child-unchanged local.wf member-rec*(*1*)
unchanged.simps)
next
case (*ReverseBytesNode n x xe*)
then have *kind g2 n = ReverseBytesNode x*
by (*metis kind-unchanged*)
then have *x ∈ eval-usages g1 n*
by (*metis IRNodes.inputs-of-ReverseBytesNode ReverseBytesNode.hyps*(*1*,*2*) *encode-in-ids*
inputs.simps inputs-are-usages list.set-intros(*1*))
then show *?case*
by (*metis IRNodes.inputs-of-ReverseBytesNode ReverseBytesNode.IH ReverseBytesNode.hyps*(*1*,*2*)
ReverseBytesNode.prems(*1*) *child-member-in child-unchanged local.wf member-rec*(*1*)
‹*kind g2 n = ReverseBytesNode x*› *encode-in-ids rep.ReverseBytesNode*)
next
case (*BitCountNode n x xe*)
then have *kind g2 n = BitCountNode x*
by (*metis kind-unchanged*)
then have *x ∈ eval-usages g1 n*
by (*metis BitCountNode.hyps*(*1*,*2*) *IRNodes.inputs-of-BitCountNode encode-in-ids*

inputs.simps
inputs-are-usages list.set-intros(*1*))
then show *?case*
by (*metis BitCountNode.IH BitCountNode.hyps*(*1*,*2*) *BitCountNode.prems*(*1*) *member-rec*(*1*) *local.wf*
IRNodes.inputs-of-BitCountNode ‹*kind g2 n* = *BitCountNode x*› *encode-in-ids rep.BitCountNode*
child-member-in child-unchanged)
next
case (*NotNode n x xe*)
then have *kind g2 n* = *NotNode x*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n*
by (*metis inputs-of-NotNode NotNode.hyps*(*1*,*2*) *encode-in-ids inputs.simps inputs-are-usages*
list.set-intros(*1*))
then show *?case*
by (*metis NotNode.IH NotNode.hyps*(*1*) *NotNode.prems*(*1*,*3*) *IRNodes.inputs-of-NotNode rep.NotNode*
‹*kind g2 n* = *NotNode x*› *child-member-in child-unchanged local.wf member-rec*(*1*)
unchanged.simps)
next
case (*NegateNode n x xe*)
then have *kind g2 n* = *NegateNode x*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n*
by (*metis inputs-of-NegateNode NegateNode.hyps*(*1*,*2*) *encode-in-ids inputs.simps inputs-are-usages*
list.set-intros(*1*))
then show *?case*
by (*metis IRNodes.inputs-of-NegateNode NegateNode.IH NegateNode.hyps*(*1*) *NegateNode.prems*(*1*,*3*)
‹*kind g2 n* = *NegateNode x*› *child-member-in child-unchanged local.wf member-rec*(*1*)
rep.NegateNode unchanged.elims(*1*))
next
case (*LogicNegationNode n x xe*)
then have *kind g2 n* = *LogicNegationNode x*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n*
by (*metis inputs-of-LogicNegationNode inputs-of-are-usages LogicNegationNode.hyps*(*1*,*2*)
encode-in-ids member-rec(*1*))
then show *?case*
by (*metis IRNodes.inputs-of-LogicNegationNode LogicNegationNode.IH LogicNegationNode.hyps*(*1*,*2*)
LogicNegationNode.prems(*1*) ‹*kind g2 n* = *LogicNegationNode x*› *child-unchanged encode-in-ids*

inputs.simps list.set-intros(*1*) *local.wf rep.LogicNegationNode*)
next
case (*AddNode n x y xe ye*)
then have *kind g2 n* = *AddNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis AddNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-AddNode encode-in-ids in-mono inputs.simps*
inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis AddNode.IH*(*1*,*2*) *AddNode.hyps*(*1*,*2*,*3*) *AddNode.prems*(*1*) *IRNodes.inputs-of-AddNode*
‹*kind g2 n* = *AddNode x y*› *child-unchanged encode-in-ids in-set-member inputs.simps*
local.wf member-rec(*1*) *rep.AddNode*)
next
case (*MulNode n x y xe ye*)
then have *kind g2 n* = *MulNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis MulNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-MulNode encode-in-ids in-mono inputs.simps*
inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis* ‹*kind g2 n* = *MulNode x y*› *child-unchanged inputs.simps list.set-intros*(*1*) *rep.MulNode*
set-subset-Cons subset-iff unchanged.elims(*2*) *inputs-of-MulNode MulNode*(*1*,*4*,*5*,*6*,*7*))
next
case (*DivNode n x y xe ye*)
then have *kind g2 n* = *SignedFloatingIntegerDivNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis DivNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-SignedFloatingIntegerDivNode encode-in-ids in-mono inputs.simps*
inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis* ‹*kind g2 n* = *SignedFloatingIntegerDivNode x y*› *child-unchanged inputs.simps list.set-intros*(*1*) *rep.DivNode*
set-subset-Cons subset-iff unchanged.elims(*2*) *inputs-of-SignedFloatingIntegerDivNode DivNode*(*1*,*4*,*5*,*6*,*7*))
next
case (*ModNode n x y xe ye*)
then have *kind g2 n* = *SignedFloatingIntegerRemNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis ModNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-SignedFloatingIntegerRemNode encode-in-ids in-mono inputs.simps*
inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)

then show *?case*
by (*metis* ‹*kind g2 n* = *SignedFloatingIntegerRemNode x y*› *child-unchanged inputs.simps list.set-intros*(*1*) *rep.ModNode*
set-subset-Cons subset-iff unchanged.elims(*2*) *inputs-of-SignedFloatingIntegerRemNode ModNode*(*1*,*4*,*5*,*6*,*7*))
next
case (*SubNode n x y xe ye*)
then have *kind g2 n* = *SubNode x y*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
by (*metis SubNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-SubNode encode-in-ids in-mono inputs.simps*
inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis* ‹*kind g2 n* = *SubNode x y*› *child-member child-unchanged encode-in-ids ids-some SubNode*
member-rec(*1*) *rep.SubNode inputs-of-SubNode*)
next
case (*AndNode n x y xe ye*)
then have *kind g2 n* = *AndNode x y*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
by (*metis AndNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-AndNode encode-in-ids in-mono inputs.simps*
inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis AndNode*(*1*,*4*,*5*,*6*,*7*) *inputs-of-AndNode* ‹*kind g2 n* = *AndNode x y*› *child-unchanged*
inputs.simps list.set-intros(*1*) *rep.AndNode set-subset-Cons subset-iff unchanged.elims*(*2*))
next
case (*OrNode n x y xe ye*)
then have *kind g2 n* = *OrNode x y*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
by (*metis OrNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-OrNode encode-in-ids in-mono inputs.simps*
inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis inputs-of-OrNode* ‹*kind g2 n* = *OrNode x y*› *child-unchanged encode-in-ids rep.OrNode*
child-member ids-some member-rec(*1*) *OrNode*)
next
case (*XorNode n x y xe ye*)
then have *kind g2 n* = *XorNode x y*
by (*metis kind-unchanged*)
then have $x \in$ *eval-usages g1 n* $\wedge$ $y \in$ *eval-usages g1 n*
by (*metis XorNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-XorNode encode-in-ids in-mono inputs.simps*

inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis inputs-of-XorNode* ‹*kind g2 n* = *XorNode x y*› *child-member child-unchanged rep.XorNode*
encode-in-ids ids-some member-rec(*1*) *XorNode*)
next
case (*ShortCircuitOrNode n x y xe ye*)
then have *kind g2 n* = *ShortCircuitOrNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis ShortCircuitOrNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-ShortCircuitOrNode inputs-are-usages*
in-mono inputs.simps list.set-intros(*1*) *set-subset-Cons encode-in-ids*)
then show *?case*
by (*metis ShortCircuitOrNode inputs-of-ShortCircuitOrNode* ‹*kind g2 n* = *ShortCircuitOrNode x y*›
child-member child-unchanged encode-in-ids ids-some member-rec(*1*) *rep.ShortCircuitOrNode*)
next
case (*LeftShiftNode n x y xe ye*)
then have *kind g2 n* = *LeftShiftNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis LeftShiftNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-LeftShiftNode encode-in-ids inputs.simps*
inputs-are-usages list.set-intros(*1*) *set-subset-Cons in-mono*)
then show *?case*
by (*metis LeftShiftNode inputs-of-LeftShiftNode* ‹*kind g2 n* = *LeftShiftNode x y*› *child-unchanged*
encode-in-ids ids-some member-rec(*1*) *rep.LeftShiftNode child-member*)
next
case (*RightShiftNode n x y xe ye*)
then have *kind g2 n* = *RightShiftNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis RightShiftNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-RightShiftNode encode-in-ids inputs.simps*
inputs-are-usages list.set-intros(*1*) *set-subset-Cons in-mono*)
then show *?case*
by (*metis RightShiftNode inputs-of-RightShiftNode* ‹*kind g2 n* = *RightShiftNode x y*› *child-member*
child-unchanged encode-in-ids ids-some member-rec(*1*) *rep.RightShiftNode*)
next
case (*UnsignedRightShiftNode n x y xe ye*)
then have *kind g2 n* = *UnsignedRightShiftNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis UnsignedRightShiftNode.hyps*(*1*,*2*,*3*) *IRNodes.inputs-of-UnsignedRightShiftNode in-mono*
encode-in-ids inputs.simps inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)

then show *?case*
by (*metis UnsignedRightShiftNode inputs-of-UnsignedRightShiftNode child-member child-unchanged*
‹kind g2 n = UnsignedRightShiftNode x y› encode-in-ids ids-some rep.UnsignedRightShiftNode member-rec(*1*))
next
case (*IntegerBelowNode n x y xe ye*)
then have *kind g2 n = IntegerBelowNode x y*
by (*metis kind-unchanged*)
then have *x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n*
by (*metis IntegerBelowNode.hyps*(*1,2,3*) *IRNodes.inputs-of-IntegerBelowNode encode-in-ids in-mono*
inputs.simps inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis inputs-of-IntegerBelowNode ‹kind g2 n = IntegerBelowNode x y› rep.IntegerBelowNode*
child-member child-unchanged encode-in-ids ids-some member-rec(*1*) *IntegerBelowNode*)
next
case (*IntegerEqualsNode n x y xe ye*)
then have *kind g2 n = IntegerEqualsNode x y*
by (*metis kind-unchanged*)
then have *x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n*
by (*metis IntegerEqualsNode.hyps*(*1,2,3*) *IRNodes.inputs-of-IntegerEqualsNode inputs-are-usages*
in-mono inputs.simps encode-in-ids list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis inputs-of-IntegerEqualsNode ‹kind g2 n = IntegerEqualsNode x y› rep.IntegerEqualsNode*
child-member child-unchanged encode-in-ids ids-some member-rec(*1*) *IntegerEqualsNode*)
next
case (*IntegerLessThanNode n x y xe ye*)
then have *kind g2 n = IntegerLessThanNode x y*
by (*metis kind-unchanged*)
then have *x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n*
by (*metis IntegerLessThanNode.hyps*(*1,2,3*) *IRNodes.inputs-of-IntegerLessThanNode encode-in-ids*
in-mono inputs.simps inputs-are-usages list.set-intros(*1*) *set-subset-Cons*)
then show *?case*
by (*metis rep.IntegerLessThanNode inputs-of-IntegerLessThanNode child-unchanged encode-in-ids*
‹kind g2 n = IntegerLessThanNode x y› child-member member-rec(*1*) *IntegerLessThanNode*
ids-some)
next
case (*IntegerTestNode n x y xe ye*)
then have *kind g2 n = IntegerTestNode x y*
by (*metis kind-unchanged*)

then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis IntegerTestNode.hyps IRNodes.inputs-of-IntegerTestNode encode-in-ids in-mono inputs.simps inputs-are-usages list.set-intros*(*1*) *set-subset-Cons*)
then show *?case*
by (*metis rep.IntegerTestNode inputs-of-IntegerTestNode child-unchanged encode-in-ids* ‹*kind g2 n* = *IntegerTestNode x y*› *child-member member-rec*(*1*) *IntegerTestNode ids-some*)
next
case (*IntegerNormalizeCompareNode n x y xe ye*)
then have *kind g2 n* = *IntegerNormalizeCompareNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
by (*metis IRNodes.inputs-of-IntegerNormalizeCompareNode IntegerNormalizeCompareNode.hyps*(*1*,*2*,*3*) *encode-in-ids in-set-member inputs.simps inputs-are-usages member-rec*(*1*))
then show *?case*
by (*metis IRNodes.inputs-of-IntegerNormalizeCompareNode IntegerNormalizeCompareNode.IH*(*1*,*2*) *IntegerNormalizeCompareNode.hyps*(*1*,*2*,*3*) *IntegerNormalizeCompareNode.prems*(*1*) *inputs.simps* ‹*kind* (*g2*::*IRGraph*) (*n*::*nat*) = *IntegerNormalizeCompareNode* (*x*::*nat*) (*y*::*nat*)› *local.wf* *encode-in-ids list.set-intros*(*1*) *rep.IntegerNormalizeCompareNode set-subset-Cons in-mono* *child-unchanged*)
next
case (*IntegerMulHighNode n x y xe ye*)
then have *kind g2 n* = *IntegerMulHighNode x y*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n*
by (*metis IRNodes.inputs-of-IntegerMulHighNode IntegerMulHighNode.hyps*(*1*,*2*) *encode-in-ids* *inputs-of-are-usages member-rec*(*1*))
then show *?case*
by (*metis inputs-of-IntegerMulHighNode IntegerMulHighNode.IH*(*1*,*2*) *IntegerMulHighNode.hyps*(*1*,*2*,*3*) *IntegerMulHighNode.prems*(*1*) *child-unchanged encode-in-ids inputs.simps list.set-intros*(*1*,*2*) ‹*kind* (*g2*::*IRGraph*) (*n*::*nat*) = *IntegerMulHighNode* (*x*::*nat*) (*y*::*nat*)› *rep.IntegerMulHighNode* *local.wf*)
next
case (*NarrowNode n ib rb x xe*)
then have *kind g2 n* = *NarrowNode ib rb x*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n*
by (*metis NarrowNode.hyps*(*1*,*2*) *IRNodes.inputs-of-NarrowNode inputs-are-usages encode-in-ids*

list.set-intros(*1*) *inputs.simps*)
then show *?case*
by (*metis NarrowNode*(*1*,*3*,*4*,*5*) *inputs-of-NarrowNode* ‹*kind g2 n* = *NarrowNode ib rb x*› *inputs.elims*
child-unchanged list.set-intros(*1*) *rep.NarrowNode unchanged.simps*)
next
case (*SignExtendNode n ib rb x xe*)
then have *kind g2 n* = *SignExtendNode ib rb x*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n*
by (*metis inputs-of-SignExtendNode SignExtendNode.hyps*(*1*,*2*) *inputs-are-usages encode-in-ids*
list.set-intros(*1*) *inputs.simps*)
then show *?case*
by (*metis SignExtendNode*(*1*,*3*,*4*,*5*,*6*) *inputs-of-SignExtendNode in-set-member list.set-intros*(*1*)
‹*kind g2 n* = *SignExtendNode ib rb x*› *child-member-in child-unchanged rep.SignExtendNode*
unchanged.elims(*2*))
next
case (*ZeroExtendNode n ib rb x xe*)
then have *kind g2 n* = *ZeroExtendNode ib rb x*
by (*metis kind-unchanged*)
then have *x* ∈ *eval-usages g1 n*
by (*metis ZeroExtendNode.hyps*(*1*,*2*) *IRNodes.inputs-of-ZeroExtendNode encode-in-ids inputs.simps*
inputs-are-usages list.set-intros(*1*))
then show *?case*
by (*metis ZeroExtendNode*(*1*,*3*,*4*,*5*,*6*) *inputs-of-ZeroExtendNode child-unchanged unchanged.simps*
‹*kind g2 n* = *ZeroExtendNode ib rb x*› *child-member-in rep.ZeroExtendNode member-rec*(*1*))
next
case (*LeafNode n s*)
then show *?case*
by (*metis kind-unchanged rep.LeafNode stamp-unchanged*)
next
case (*PiNode n n′ gu*)
then have *kind g2 n* = *PiNode n′ gu*
by (*metis kind-unchanged*)
then show *?case*
by (*metis PiNode.IH* ‹*kind* (*g2*) (*n*) = *PiNode* (*n′*) (*gu*)› *child-unchanged encode-in-ids rep.PiNode*
inputs.elims list.set-intros(*1*)*PiNode.hyps PiNode.prems*(*1*,*2*) *IRNodes.inputs-of-PiNode*)
next
case (*RefNode n n′*)
then have *kind g2 n* = *RefNode n′*
by (*metis kind-unchanged*)
then have *n′* ∈ *eval-usages g1 n*

by (*metis IRNodes.inputs-of-RefNode RefNode.hyps*(*1*,*2*) *inputs-are-usages list.set-intros*(*1*) *inputs.elims encode-in-ids*)
then show *?case*
by (*metis IRNodes.inputs-of-RefNode RefNode.IH RefNode.hyps*(*1*,*2*) *RefNode.prems*(*1*) *inputs.elims*
‹*kind g2 n* = *RefNode n′*› *child-unchanged encode-in-ids list.set-intros*(*1*) *rep.RefNode*
local.wf)
next
case (*IsNullNode n v*)
then have *kind g2 n* = *IsNullNode v*
by (*metis kind-unchanged*)
then show *?case*
by (*metis IRNodes.inputs-of-IsNullNode IsNullNode.IH IsNullNode.hyps*(*1*,*2*) *IsNullNode.prems*(*1*)
‹*kind g2 n* = *IsNullNode v*› *child-unchanged encode-in-ids inputs.simps list.set-intros*(*1*)
local.wf rep.IsNullNode)
qed
qed

theorem *stay-same*:
assumes *nc*: *unchanged* (*eval-usages g1 nid*) *g1 g2*
assumes *g1*: [*g1*, *m*, *p*] ⊢ *nid* ↦ *v1*
assumes *wf*: *wf-graph g1*
shows [*g2*, *m*, *p*] ⊢ *nid* ↦ *v1*
proof −
have *nid*: *nid* ∈ *ids g1*
using *g1 eval-in-ids* **by** *simp*
then have *nid* ∈ *eval-usages g1 nid*
using *eval-usages-self* **by** *simp*
then have *kind-same*: *kind g1 nid* = *kind g2 nid*
using *nc node-unchanged* **by** *blast*
obtain *e* **where** *e*: (*g1* ⊢ *nid* ≃ *e*) ∧ ([*m*,*p*] ⊢ *e* ↦ *v1*)
using *g1* **by** (*auto simp add*: *encodeeval.simps*)
then have *val*: [*m*,*p*] ⊢ *e* ↦ *v1*
by (*simp add*: *g1 encodeeval.simps*)
then show *?thesis*
using *e nc* **unfolding** *encodeeval.simps*
proof (*induct e v1 arbitrary*: *nid rule*: *evaltree.induct*)
case (*ConstantExpr c*)
then show *?case*
by (*meson local.wf stay-same-encoding*)
next
case (*ParameterExpr i s*)
have *g2* ⊢ *nid* ≃ *ParameterExpr i s*
by (*meson local.wf stay-same-encoding ParameterExpr*)
then show *?case*

by (*meson ParameterExpr.hyps evaltree.ParameterExpr*)
next
case (*ConditionalExpr ce cond branch te fe v*)
then have $g2 \vdash nid \simeq$ *ConditionalExpr ce te fe*
using *local.wf stay-same-encoding* **by** *presburger*
then show *?case*
by (*meson ConditionalExpr.prems*(*1*))
next
case (*UnaryExpr xe v op*)
then show *?case*
using *local.wf stay-same-encoding* **by** *blast*
next
case (*BinaryExpr xe x ye y op*)
then show *?case*
using *local.wf stay-same-encoding* **by** *blast*
next
case (*LeafExpr val nid s*)
then show *?case*
by (*metis local.wf stay-same-encoding*)
qed
qed

lemma *add-changed*:
assumes *gup* = *add-node new k g*
shows *changeonly* {*new*} *g gup*
by (*simp add*: *assms add-node.rep-eq kind.rep-eq stamp.rep-eq*)

lemma *disjoint-change*:
assumes *changeonly change g gup*
assumes *nochange* = *ids g* − *change*
shows *unchanged nochange g gup*
using *assms* **by** *simp*

lemma *add-node-unchanged*:
assumes *new* ∉ *ids g*
assumes *nid* ∈ *ids g*
assumes *gup* = *add-node new k g*
assumes *wf-graph g*
shows *unchanged* (*eval-usages g nid*) *g gup*
proof −
have *new* ∉ (*eval-usages g nid*)
using *assms* **by** *simp*
then have *changeonly* {*new*} *g gup*
using *assms add-changed* **by** *simp*
then show *?thesis*
using *assms* **by** *auto*
qed

lemma *eval-uses-imp*:

```
  ((nid′ ∈ ids g ∧ nid = nid′)
    ∨ nid′ ∈ inputs g nid
    ∨ (∃ nid′′ . eval-uses g nid nid′′ ∧ eval-uses g nid′′ nid′))
    ⟷ eval-uses g nid nid′
  by (meson eval-uses.simps)

lemma wf-use-ids:
  assumes wf-graph g
  assumes nid ∈ ids g
  assumes eval-uses g nid nid′
  shows nid′ ∈ ids g
 using assms(3) apply (induction rule: eval-uses.induct) using assms(1) inp-in-g-wf
by auto

lemma no-external-use:
  assumes wf-graph g
  assumes nid′ ∉ ids g
  assumes nid ∈ ids g
  shows ¬(eval-uses g nid nid′)
proof −
  have 0: nid ≠ nid′
    using assms by auto
  have inp: nid′ ∉ inputs g nid
    using assms inp-in-g-wf by auto
  have rec-0: ∄ n . n ∈ ids g ∧ n = nid′
    using assms by simp
  have rec-inp: ∄ n . n ∈ ids g ∧ n ∈ inputs g nid′
    using assms(2) by (simp add: inp-in-g)
  have rec: ∄ nid′′ . eval-uses g nid nid′′ ∧ eval-uses g nid′′ nid′
    using wf-use-ids assms by blast
  from inp 0 rec show ?thesis
    using eval-uses-imp by blast
qed

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*
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-parameter* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *ParameterNode i* $\Longrightarrow$
($\exists$ *s. e* = *ParameterExpr i s*)
by (*induction rule*: *rep.induct*; *auto*)

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*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-abs* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *AbsNode x* $\Longrightarrow$
($\exists$ *xe. e* = *UnaryExpr UnaryAbs xe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-reverse-bytes* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *ReverseBytesNode x* $\Longrightarrow$
($\exists$ *xe. e* = *UnaryExpr UnaryReverseBytes xe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-bit-count* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *BitCountNode x* $\Longrightarrow$
($\exists$ *xe. e* = *UnaryExpr UnaryBitCount xe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-not* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
kind g n = *NotNode x* $\Longrightarrow$
($\exists$ *xe. e* = *UnaryExpr UnaryNot xe*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-negate* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = NegateNode\ x \Longrightarrow$
$(\exists xe.\ e = UnaryExpr\ UnaryNeg\ xe)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-logicnegation* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = LogicNegationNode\ x \Longrightarrow$
$(\exists xe.\ e = UnaryExpr\ UnaryLogicNegation\ xe)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-add* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = AddNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinAdd\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-sub* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = SubNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinSub\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-mul* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = MulNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinMul\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-div* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = SignedFloatingIntegerDivNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinDiv\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-mod* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = SignedFloatingIntegerRemNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinMod\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-and* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = AndNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinAnd\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-or* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = OrNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinOr\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-xor* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = XorNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinXor\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-short-circuit-or* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = ShortCircuitOrNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinShortCircuitOr\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-left-shift* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = LeftShiftNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinLeftShift\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-right-shift* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = RightShiftNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinRightShift\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-unsigned-right-shift* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = UnsignedRightShiftNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinURightShift\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-integer-below* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = IntegerBelowNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinIntegerBelow\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-integer-equals* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = IntegerEqualsNode\ x\ y \Longrightarrow$
$(\exists xe\ ye.\ e = BinaryExpr\ BinIntegerEquals\ xe\ ye)$
by (*induction rule*: *rep.induct*; *auto*)

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*)
by (*induction rule*: *rep.induct*; *auto*)

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*)
by (*induction rule*: *rep.induct*; *auto*)

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*)
by (*induction rule*: *rep.induct*; *auto*)

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*)
by (*induction rule*: *rep.induct*; *auto*)

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*)
by (*induction rule*: *rep.induct*; *auto*)

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*)
by (*induction rule*: *rep.induct*; *auto*)

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*)
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-load-field* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
is-preevaluated (*kind g n*) $\Longrightarrow$
($\exists$ *s*. *e* = *LeafExpr n s*)
by (*induction rule*: *rep.induct*; *auto*)

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

$(kind\ g\ n) = BytecodeExceptionNode\ gu\ st\ n' \Longrightarrow$
$(\exists\, s.\ e = LeafExpr\ n\ s)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-new-array* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$(kind\ g\ n) = NewArrayNode\ len\ st\ n' \Longrightarrow$
$(\exists\, s.\ e = LeafExpr\ n\ s)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-array-length* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$(kind\ g\ n) = ArrayLengthNode\ x\ n' \Longrightarrow$
$(\exists\, s.\ e = LeafExpr\ n\ s)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-load-index* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$(kind\ g\ n) = LoadIndexedNode\ index\ guard\ x\ n' \Longrightarrow$
$(\exists\, s.\ e = LeafExpr\ n\ s)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-store-index* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$(kind\ g\ n) = StoreIndexedNode\ check\ val\ st\ index\ guard\ x\ n' \Longrightarrow$
$(\exists\, s.\ e = LeafExpr\ n\ s)$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-ref* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = RefNode\ n' \Longrightarrow$
$g \vdash n' \simeq e$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-pi* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = PiNode\ n'\ gu \Longrightarrow$
$g \vdash n' \simeq e$
by (*induction rule*: *rep.induct*; *auto*)

lemma *rep-is-null* [*rep*]:
$g \vdash n \simeq e \Longrightarrow$
$kind\ g\ n = IsNullNode\ x \Longrightarrow$
$(\exists\, xe.\ e = (UnaryExpr\ UnaryIsNull\ xe))$
by (*induction rule*: *rep.induct*; *auto*)

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 (*induction arbitrary*: e_2 *rule*: *rep.induct*)
case (*ConstantNode n c*)
then show *?case*
using *rep-constant* **by** *simp*
next
case (*ParameterNode n i s*)
then show *?case*
by (*metis IRNode.distinct(3655) IRNode.distinct(3697) ParameterNodeE rep-parameter*)
next
case (*ConditionalNode n c t f ce te fe*)
then show *?case*
by (*metis ConditionalNodeE IRNode.distinct(925) IRNode.distinct(967) IRNode.sel(90) IRNode.sel(93) IRNode.sel(94) rep-conditional*)
next
case (*AbsNode n x xe*)
then show *?case*
by (*solve-det node*: *AbsNode*)
next

case (*ReverseBytesNode n x xe*)
then show *?case*
by (*solve-det node*: *ReverseBytesNode*)
next
case (*BitCountNode n x xe*)
then show *?case*
by (*solve-det node*: *BitCountNode*)
next
case (*NotNode n x xe*)
then show *?case*
by (*solve-det node*: *NotNode*)
next
case (*NegateNode n x xe*)
then show *?case*
by (*solve-det node*: *NegateNode*)
next
case (*LogicNegationNode n x xe*)
then show *?case*
by (*solve-det node*: *LogicNegationNode*)
next
case (*AddNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *AddNode*)
next
case (*MulNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *MulNode*)
next
case (*DivNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *DivNode*)
next
case (*ModNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *ModNode*)
next
case (*SubNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *SubNode*)
next
case (*AndNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *AndNode*)
next
case (*OrNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *OrNode*)
next
case (*XorNode n x y xe ye*)

then show *?case*
by (*solve-det node*: *XorNode*)
next
case (*ShortCircuitOrNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *ShortCircuitOrNode*)
next
case (*LeftShiftNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *LeftShiftNode*)
next
case (*RightShiftNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *RightShiftNode*)
next
case (*UnsignedRightShiftNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *UnsignedRightShiftNode*)
next
case (*IntegerBelowNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *IntegerBelowNode*)
next
case (*IntegerEqualsNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *IntegerEqualsNode*)
next
case (*IntegerLessThanNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *IntegerLessThanNode*)
next
case (*IntegerTestNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *IntegerTestNode*)
next
case (*IntegerNormalizeCompareNode n x y xe ye*)
then show *?case*
by (*solve-det node*: *IntegerNormalizeCompareNode*)
next
case (*IntegerMulHighNode n x xe*)
then show *?case*
by (*solve-det node*: *IntegerMulHighNode*)
next
case (*NarrowNode n x xe*)
then show *?case*
using *NarrowNodeE rep-narrow*
by (*metis IRNode.distinct*(*3361*) *IRNode.distinct*(*3403*) *IRNode.inject*(*36*))
next
case (*SignExtendNode n x xe*)

then show *?case*
  using *SignExtendNodeE rep-sign-extend*
  by (*metis IRNode.distinct*(*3707*) *IRNode.distinct*(*3919*) *IRNode.inject*(*48*))
next
 case (*ZeroExtendNode n x xe*)
 then show *?case*
  using *ZeroExtendNodeE rep-zero-extend*
  by (*metis IRNode.distinct*(*3735*) *IRNode.distinct*(*4157*) *IRNode.inject*(*62*))
next
 case (*LeafNode n s*)
 then show *?case*
  using *rep-load-field LeafNodeE*
  by (*metis is-preevaluated.simps*(*48*) *is-preevaluated.simps*(*65*))
next
 case (*RefNode n'*)
 then show *?case*
  using *rep-ref* **by** *blast*
next
 case (*PiNode n v*)
 then show *?case*
  using *rep-pi* **by** *blast*
next
 case (*IsNullNode n v*)
 then show *?case*
  using *IsNullNodeE rep-is-null*
  by (*metis IRNode.distinct*(*2557*) *IRNode.distinct*(*2599*) *IRNode.inject*(*24*))
qed

lemma *repAllDet*:
 $g \vdash xs\ [\simeq]\ e1 \Longrightarrow$
  $g \vdash xs\ [\simeq]\ e2 \Longrightarrow$
  $e1 = e2$
proof (*induction arbitrary*: *e2 rule*: *replist.induct*)
 case *RepNil*
 then show *?case*
  using *replist.cases* **by** *auto*
next
 case (*RepCons x xe xs xse*)
 then show *?case*
  by (*metis list.distinct*(*1*) *list.sel*(*1,3*) *repDet replist.cases*)
qed

lemma *encodeEvalDet*:
 $[g,m,p] \vdash e \mapsto v1 \Longrightarrow$
  $[g,m,p] \vdash e \mapsto v2 \Longrightarrow$
  $v1 = v2$
 by (*metis encodeeval.simps evalDet repDet*)

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

by (*auto simp add*: *encodeEvalDet*)

lemma *encodeEvalAllDet*:
$[g, m, p] \vdash nids\ [\mapsto]\ vs \Longrightarrow [g, m, p] \vdash nids\ [\mapsto]\ vs' \Longrightarrow vs = vs'$
using *repAllDet evalAllDet*
by (*metis encodeEvalAll.simps*)

7.8.2 Monotonicity of Graph Refinement

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

lemma *mono-abs*:
assumes $kind\ g1\ n = AbsNode\ x \wedge kind\ g2\ n = AbsNode\ x$
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 $e1 \geq e2$
by (*metis AbsNode assms mono-unary repDet*)

lemma *mono-not*:
assumes $kind\ g1\ n = NotNode\ x \wedge kind\ g2\ n = NotNode\ x$
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 $e1 \geq e2$
by (*metis NotNode assms mono-unary repDet*)

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 $e1 \geq e2$
by (*metis NegateNode assms mono-unary repDet*)

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 $e1 \geq e2$
by (*metis LogicNegationNode assms mono-unary repDet*)

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 $e1 \geq e2$

by (*metis NarrowNode assms mono-unary repDet*)

lemma *mono-sign-extend*:
assumes *kind g1 n = SignExtendNode ib rb x* ∧ *kind g2 n = SignExtendNode ib rb x*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes *xe1* ≥ *xe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis SignExtendNode assms mono-unary repDet*)

lemma *mono-zero-extend*:
assumes *kind g1 n = ZeroExtendNode ib rb x* ∧ *kind g2 n = ZeroExtendNode ib rb x*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes *xe1* ≥ *xe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis ZeroExtendNode assms mono-unary repDet*)

lemma *mono-conditional-graph*:
assumes *kind g1 n = ConditionalNode c t f* ∧ *kind g2 n = ConditionalNode c t f*
assumes (*g1* ⊢ *c* ≃ *ce1*) ∧ (*g2* ⊢ *c* ≃ *ce2*)
assumes (*g1* ⊢ *t* ≃ *te1*) ∧ (*g2* ⊢ *t* ≃ *te2*)
assumes (*g1* ⊢ *f* ≃ *fe1*) ∧ (*g2* ⊢ *f* ≃ *fe2*)
assumes *ce1* ≥ *ce2* ∧ *te1* ≥ *te2* ∧ *fe1* ≥ *fe2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*smt* (*verit*, *ccfv-SIG*) *ConditionalNode assms mono-conditional repDet le-expr-def*)

lemma *mono-add*:
assumes *kind g1 n = AddNode x y* ∧ *kind g2 n = AddNode x y*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes (*g1* ⊢ *y* ≃ *ye1*) ∧ (*g2* ⊢ *y* ≃ *ye2*)
assumes *xe1* ≥ *xe2* ∧ *ye1* ≥ *ye2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis* (*no-types*, *lifting*) *AddNode mono-binary assms repDet*)

lemma *mono-mul*:
assumes *kind g1 n = MulNode x y* ∧ *kind g2 n = MulNode x y*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes (*g1* ⊢ *y* ≃ *ye1*) ∧ (*g2* ⊢ *y* ≃ *ye2*)
assumes *xe1* ≥ *xe2* ∧ *ye1* ≥ *ye2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis* (*no-types*, *lifting*) *MulNode assms mono-binary repDet*)

lemma *mono-div*:

assumes *kind g1 n = SignedFloatingIntegerDivNode x y* ∧ *kind g2 n = SignedFloatingIntegerDivNode x y*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes (*g1* ⊢ *y* ≃ *ye1*) ∧ (*g2* ⊢ *y* ≃ *ye2*)
assumes *xe1* ≥ *xe2* ∧ *ye1* ≥ *ye2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis* (*no-types*, *lifting*) *DivNode assms mono-binary repDet*)

lemma *mono-mod*:
assumes *kind g1 n = SignedFloatingIntegerRemNode x y* ∧ *kind g2 n = SignedFloatingIntegerRemNode x y*
assumes (*g1* ⊢ *x* ≃ *xe1*) ∧ (*g2* ⊢ *x* ≃ *xe2*)
assumes (*g1* ⊢ *y* ≃ *ye1*) ∧ (*g2* ⊢ *y* ≃ *ye2*)
assumes *xe1* ≥ *xe2* ∧ *ye1* ≥ *ye2*
assumes (*g1* ⊢ *n* ≃ *e1*) ∧ (*g2* ⊢ *n* ≃ *e2*)
shows *e1* ≥ *e2*
by (*metis* (*no-types*, *lifting*) *ModNode assms mono-binary repDet*)

lemma *term-graph-evaluation*:
(*g* ⊢ *n* ⊴ *e*) ⟹ (∀ *m p v* . ([*m*,*p*] ⊢ *e* ↦ *v*) ⟶ ([*g*,*m*,*p*] ⊢ *n* ↦ *v*))
using *graph-represents-expression-def encodeeval.simps* **by** (*auto*; *meson*)

lemma *encodes-contains*:
g ⊢ *n* ≃ *e* ⟹
kind g n ≠ *NoNode*
apply (*induction rule*: *rep.induct*)
apply (*match IRNode.distinct* **in** *e*: *?n* ≠ *NoNode* ⇒ ‹*presburger add*: *e*›)+
by *fastforce*+

lemma *no-encoding*:
assumes *n* ∉ *ids g*
shows ¬(*g* ⊢ *n* ≃ *e*)
using *assms* **apply** *simp* **apply** (*rule notI*) **by** (*induction e*; *simp add*: *encodes-contains*)

lemma *not-excluded-keep-type*:
assumes *n* ∈ *ids g1*
assumes *n* ∉ *excluded*
assumes (*excluded* ⊴ *as-set g1*) ⊆ *as-set g2*
shows *kind g1 n = kind g2 n* ∧ *stamp g1 n = stamp g2 n*
using *assms* **by** (*auto simp add*: *domain-subtraction-def as-set-def*)

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

```
method metis-node-eq-ternary for node :: 'a ⇒ 'a ⇒ 'a ⇒ IRNode =
  (match IRNode.inject in i: (node - - - = node - - -) = - ⇒
     ‹metis i›)
```

7.8.3 Lift Data-flow Tree Refinement to Graph Refinement

```
theorem graph-semantics-preservation:
  assumes a: e1' ≥ e2'
  assumes b: ({n'} ⊴ as-set g1) ⊆ as-set g2
  assumes c: g1 ⊢ n' ≃ e1'
  assumes d: g2 ⊢ n' ≃ e2'
  shows graph-refinement g1 g2
  unfolding graph-refinement-def apply rule
  apply (metis b d ids-some no-encoding not-excluded-keep-type singleton-iff sub-
setI)
  apply (rule allI) apply (rule impI) apply (rule allI) apply (rule impI)
  unfolding graph-represents-expression-def
proof –
  fix n e1
  assume e: n ∈ ids g1
  assume f: (g1 ⊢ n ≃ e1)
  show ∃ e2. (g2 ⊢ n ≃ e2) ∧ e1 ≥ e2
  proof (cases n = n')
    case True
    have g: e1 = e1'
      using f by (simp add: repDet True c)
    have h: (g2 ⊢ n ≃ e2') ∧ e1' ≥ e2'
      using a by (simp add: d True)
    then show ?thesis
      by (auto simp add: g)
  next
    case False
    have n ∉ {n'}
      by (simp add: False)
    then have i: kind g1 n = kind g2 n ∧ stamp g1 n = stamp g2 n
      using not-excluded-keep-type b e by presburger
    show ?thesis
      using f i
    proof (induction e1)
      case (ConstantNode n c)
      then show ?case
        by (metis eq-refl rep.ConstantNode)
    next
      case (ParameterNode n i s)
      then show ?case
        by (metis eq-refl rep.ParameterNode)
    next
      case (ConditionalNode n c t f ce1 te1 fe1)
      have k: g1 ⊢ n ≃ ConditionalExpr ce1 te1 fe1
```

```
    using ConditionalNode by (simp add: ConditionalNode.hyps(2) rep.ConditionalNode
f)
     obtain cn tn fn where l: kind g1 n = ConditionalNode cn tn fn
       by (auto simp add: ConditionalNode.hyps(1))
     then have mc: g1 ⊢ cn ≃ ce1
       using ConditionalNode.hyps(1,2) by simp
     from l have mt: g1 ⊢ tn ≃ te1
       using ConditionalNode.hyps(1,3) by simp
     from l have mf: g1 ⊢ fn ≃ fe1
       using ConditionalNode.hyps(1,4) by simp
     then show ?case
     proof -
       have g1 ⊢ cn ≃ ce1
         by (simp add: mc)
       have g1 ⊢ tn ≃ te1
         by (simp add: mt)
       have g1 ⊢ fn ≃ fe1
         by (simp add: mf)
       have cer: ∃ ce2. (g2 ⊢ cn ≃ ce2) ∧ ce1 ≥ ce2
        using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
         by (metis-node-eq-ternary ConditionalNode)
       have ter: ∃ te2. (g2 ⊢ tn ≃ te2) ∧ te1 ≥ te2
        using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
         by (metis-node-eq-ternary ConditionalNode)
       have ∃ fe2. (g2 ⊢ fn ≃ fe2) ∧ fe1 ≥ fe2
        using ConditionalNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
         by (metis-node-eq-ternary ConditionalNode)
       then have ∃ ce2 te2 fe2. (g2 ⊢ n ≃ ConditionalExpr ce2 te2 fe2) ∧
             ConditionalExpr ce1 te1 fe1 ≥ ConditionalExpr ce2 te2 fe2
         apply meson
      by (smt (verit, best) mono-conditional ConditionalNode.prems l rep.ConditionalNode
cer ter)
       then show ?thesis
         by meson
     qed
   next
     case (AbsNode n x xe1)
     have k: g1 ⊢ n ≃ UnaryExpr UnaryAbs xe1
       using AbsNode by (simp add: AbsNode.hyps(2) rep.AbsNode f)
     obtain xn where l: kind g1 n = AbsNode xn
       by (auto simp add: AbsNode.hyps(1))
     then have m: g1 ⊢ xn ≃ xe1
       using AbsNode.hyps(1,2) by simp
     then show ?case
     proof (cases xn = n′)
       case True
```

```
      then have n: xe1 = e1′
        using m by (simp add: repDet c)
      then have ev: g2 ⊢ n ≃ UnaryExpr UnaryAbs e2′
        using l d by (simp add: rep.AbsNode True AbsNode.prems)
      then have r: UnaryExpr UnaryAbs e1′ ≥ UnaryExpr UnaryAbs e2′
        by (meson a mono-unary)
      then show ?thesis
        by (metis n ev)
    next
      case False
      have g1 ⊢ xn ≃ xe1
        by (simp add: m)
      have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
         using AbsNode False b encodes-contains l not-excluded-keep-type not-in-g
singleton-iff
        by (metis-node-eq-unary AbsNode)
      then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr UnaryAbs xe2) ∧
         UnaryExpr UnaryAbs xe1 ≥ UnaryExpr UnaryAbs xe2
        by (metis AbsNode.prems l mono-unary rep.AbsNode)
      then show ?thesis
        by meson
    qed
  next
    case (ReverseBytesNode n x xe1)
    have k: g1 ⊢ n ≃ UnaryExpr UnaryReverseBytes xe1
      by (simp add: ReverseBytesNode.hyps(1,2) rep.ReverseBytesNode)
    obtain xn where l: kind g1 n = ReverseBytesNode xn
      by (simp add: ReverseBytesNode.hyps(1))
    then have m: g1 ⊢ xn ≃ xe1
      by (metis IRNode.inject(45) ReverseBytesNode.hyps(1,2))
    then show ?case
    proof (cases xn = n′)
      case True
      then have n: xe1 = e1′
        using m by (simp add: repDet c)
      then have ev: g2 ⊢ n ≃ UnaryExpr UnaryReverseBytes e2′
      using ReverseBytesNode.prems True d l rep.ReverseBytesNode by presburger
       then have r: UnaryExpr UnaryReverseBytes e1′ ≥ UnaryExpr UnaryRe-
verseBytes e2′
        by (meson a mono-unary)
      then show ?thesis
        by (metis n ev)
    next
      case False
      have g1 ⊢ xn ≃ xe1
        by (simp add: m)
      have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
     by (metis False IRNode.inject(45) ReverseBytesNode.IH ReverseBytesNode.hyps(1,2)
b l
```

```
        encodes-contains ids-some not-excluded-keep-type singleton-iff)
      then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr UnaryReverseBytes xe2) ∧
  UnaryExpr UnaryReverseBytes xe1 ≥ UnaryExpr UnaryReverseBytes xe2
        by (metis ReverseBytesNode.prems l mono-unary rep.ReverseBytesNode)
      then show ?thesis
        by meson
    qed
  next
    case (BitCountNode n x xe1)
    have k: g1 ⊢ n ≃ UnaryExpr UnaryBitCount xe1
      by (simp add: BitCountNode.hyps(1,2) rep.BitCountNode)
    obtain xn where l: kind g1 n = BitCountNode xn
      by (simp add: BitCountNode.hyps(1))
    then have m: g1 ⊢ xn ≃ xe1
      by (metis BitCountNode.hyps(1,2) IRNode.inject(6))
    then show ?case
    proof (cases xn = n')
      case True
      then have n: xe1 = e1'
        using m by (simp add: repDet c)
      then have ev: g2 ⊢ n ≃ UnaryExpr UnaryBitCount e2'
        using BitCountNode.prems True d l rep.BitCountNode by presburger
      then have r: UnaryExpr UnaryBitCount e1' ≥ UnaryExpr UnaryBitCount
e2'
        by (meson a mono-unary)
      then show ?thesis
        by (metis n ev)
    next
      case False
      have g1 ⊢ xn ≃ xe1
        by (simp add: m)
      have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
        by (metis BitCountNode.IH BitCountNode.hyps(1) False IRNode.inject(6)
b emptyE insertE l m
          no-encoding not-excluded-keep-type)
      then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr UnaryBitCount xe2) ∧
    UnaryExpr UnaryBitCount xe1 ≥ UnaryExpr UnaryBitCount xe2
        by (metis BitCountNode.prems l mono-unary rep.BitCountNode)
      then show ?thesis
        by meson
    qed
  next
    case (NotNode n x xe1)
    have k: g1 ⊢ n ≃ UnaryExpr UnaryNot xe1
      using NotNode by (simp add: NotNode.hyps(2) rep.NotNode f)
    obtain xn where l: kind g1 n = NotNode xn
      by (auto simp add: NotNode.hyps(1))
    then have m: g1 ⊢ xn ≃ xe1
      using NotNode.hyps(1,2) by simp
```

then show *?case*
proof (*cases xn* = *n*′)
  case *True*
  then have *n*: *xe1* = *e1*′
    using *m* **by** (*simp add*: *repDet c*)
  then have *ev*: *g2* ⊢ *n* ≃ *UnaryExpr UnaryNot e2*′
    using *l* **by** (*simp add*: *rep.NotNode d True NotNode.prems*)
  then have *r*: *UnaryExpr UnaryNot e1*′ ≥ *UnaryExpr UnaryNot e2*′
    by (*meson a mono-unary*)
  then show *?thesis*
    by (*metis n ev*)
next
  case *False*
  have *g1* ⊢ *xn* ≃ *xe1*
    by (*simp add*: *m*)
  have ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
    using *NotNode False b l not-excluded-keep-type singletonD no-encoding*
    by (*metis-node-eq-unary NotNode*)
  then have ∃ *xe2*. (*g2* ⊢ *n* ≃ *UnaryExpr UnaryNot xe2*) ∧
      UnaryExpr UnaryNot xe1 ≥ *UnaryExpr UnaryNot xe2*
    by (*metis NotNode.prems l mono-unary rep.NotNode*)
  then show *?thesis*
    by *meson*
qed
next
case (*NegateNode n x xe1*)
have *k*: *g1* ⊢ *n* ≃ *UnaryExpr UnaryNeg xe1*
  using *NegateNode* **by** (*simp add*: *NegateNode.hyps*(*2*) *rep.NegateNode f*)
obtain *xn* **where** *l*: *kind g1 n* = *NegateNode xn*
  by (*auto simp add*: *NegateNode.hyps*(*1*))
then have *m*: *g1* ⊢ *xn* ≃ *xe1*
  using *NegateNode.hyps*(*1*,*2*) **by** *simp*
then show *?case*
proof (*cases xn* = *n*′)
  case *True*
  then have *n*: *xe1* = *e1*′
    using *m* **by** (*simp add*: *c repDet*)
  then have *ev*: *g2* ⊢ *n* ≃ *UnaryExpr UnaryNeg e2*′
    using *l* **by** (*simp add*: *rep.NegateNode True NegateNode.prems d*)
  then have *r*: *UnaryExpr UnaryNeg e1*′ ≥ *UnaryExpr UnaryNeg e2*′
    by (*meson a mono-unary*)
  then show *?thesis*
    by (*metis n ev*)
next
  case *False*
  have *g1* ⊢ *xn* ≃ *xe1*
    by (*simp add*: *m*)
  have ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
    using *NegateNode False b l not-excluded-keep-type singletonD no-encoding*

by (*metis-node-eq-unary NegateNode*)
then have ∃ *xe2*. (*g2* ⊢ *n* ≃ *UnaryExpr UnaryNeg xe2*) ∧
UnaryExpr UnaryNeg xe1 ≥ *UnaryExpr UnaryNeg xe2*
by (*metis NegateNode.prems l mono-unary rep.NegateNode*)
then show *?thesis*
by *meson*
qed
next
case (*LogicNegationNode n x xe1*)
have *k*: *g1* ⊢ *n* ≃ *UnaryExpr UnaryLogicNegation xe1*
using *LogicNegationNode* **by** (*simp add*: *LogicNegationNode.hyps*(*2*) *rep.LogicNegationNode*)
obtain *xn* **where** *l*: *kind g1 n* = *LogicNegationNode xn*
by (*simp add*: *LogicNegationNode.hyps*(*1*))
then have *m*: *g1* ⊢ *xn* ≃ *xe1*
using *LogicNegationNode.hyps*(*1*,*2*) **by** *simp*
then show *?case*
proof (*cases xn* = *n′*)
case *True*
then have *n*: *xe1* = *e1′*
using *m* **by** (*simp add*: *c repDet*)
then have *ev*: *g2* ⊢ *n* ≃ *UnaryExpr UnaryLogicNegation e2′*
using *l* **by** (*simp add*: *rep.LogicNegationNode True LogicNegationNode.prems d*
LogicNegationNode.hyps(*1*))
then have *r*: *UnaryExpr UnaryLogicNegation e1′* ≥ *UnaryExpr UnaryLogicNegation e2′*
by (*meson a mono-unary*)
then show *?thesis*
by (*metis n ev*)
next
case *False*
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *m*)
have ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *LogicNegationNode False b l not-excluded-keep-type singletonD no-encoding*
by (*metis-node-eq-unary LogicNegationNode*)
then have ∃ *xe2*. (*g2* ⊢ *n* ≃ *UnaryExpr UnaryLogicNegation xe2*) ∧
UnaryExpr UnaryLogicNegation xe1 ≥ *UnaryExpr UnaryLogicNegation xe2*
by (*metis LogicNegationNode.prems l mono-unary rep.LogicNegationNode*)
then show *?thesis*
by *meson*
qed
next
case (*AddNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinAdd xe1 ye1*
using *AddNode* **by** (*simp add*: *AddNode.hyps*(*2*) *rep.AddNode f*)
obtain *xn yn* **where** *l*: *kind g1 n* = *AddNode xn yn*
by (*simp add*: *AddNode.hyps*(*1*))

then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
  using *AddNode.hyps*(*1*,*2*) **by** *simp*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
  using *AddNode.hyps*(*1*,*3*) **by** *simp*
then show *?case*
proof −
  have *g1* ⊢ *xn* ≃ *xe1*
    by (*simp add*: *mx*)
  have *g1* ⊢ *yn* ≃ *ye1*
    by (*simp add*: *my*)
  have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
      using *AddNode a b c d l no-encoding not-excluded-keep-type repDet*
singletonD
    by (*metis-node-eq-binary AddNode*)
  have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
      using *AddNode a b c d l no-encoding not-excluded-keep-type repDet*
singletonD
    by (*metis-node-eq-binary AddNode*)
  then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinAdd xe2 ye2*) ∧
      BinaryExpr BinAdd xe1 ye1 ≥ *BinaryExpr BinAdd xe2 ye2*
    by (*metis AddNode.prems l mono-binary rep.AddNode xer*)
  then show *?thesis*
    by *meson*
qed
next
case (*MulNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinMul xe1 ye1*
  using *MulNode* **by** (*simp add*: *MulNode.hyps*(*2*) *rep.MulNode f*)
obtain *xn yn* **where** *l*: *kind g1 n* = *MulNode xn yn*
  by (*simp add*: *MulNode.hyps*(*1*))
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
  using *MulNode.hyps*(*1*,*2*) **by** *simp*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
  using *MulNode.hyps*(*1*,*3*) **by** *simp*
then show *?case*
proof −
  have *g1* ⊢ *xn* ≃ *xe1*
    by (*simp add*: *mx*)
  have *g1* ⊢ *yn* ≃ *ye1*
    by (*simp add*: *my*)
  have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
      using *MulNode a b c d l no-encoding not-excluded-keep-type repDet*
singletonD
    by (*metis-node-eq-binary MulNode*)
  have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
      using *MulNode a b c d l no-encoding not-excluded-keep-type repDet*
singletonD
    by (*metis-node-eq-binary MulNode*)
  then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinMul xe2 ye2*) ∧

```
        BinaryExpr BinMul xe1 ye1 ≥ BinaryExpr BinMul xe2 ye2
      by (metis MulNode.prems l mono-binary rep.MulNode xer)
    then show ?thesis
      by meson
  qed
next
  case (DivNode n x y xe1 ye1)
  have k: g1 ⊢ n ≃ BinaryExpr BinDiv xe1 ye1
    using DivNode by (simp add: DivNode.hyps(2) rep.DivNode f)
  obtain xn yn where l: kind g1 n = SignedFloatingIntegerDivNode xn yn
    by (simp add: DivNode.hyps(1))
  then have mx: g1 ⊢ xn ≃ xe1
    using DivNode.hyps(1,2) by simp
  from l have my: g1 ⊢ yn ≃ ye1
    using DivNode.hyps(1,3) by simp
  then show ?case
  proof −
    have g1 ⊢ xn ≃ xe1
      by (simp add: mx)
    have g1 ⊢ yn ≃ ye1
      by (simp add: my)
    have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
        using DivNode  a b c d l no-encoding not-excluded-keep-type repDet
singletonD
      by (metis-node-eq-binary SignedFloatingIntegerDivNode)
    have ∃ ye2. (g2 ⊢ yn ≃ ye2) ∧ ye1 ≥ ye2
     using DivNode a b c d l no-encoding not-excluded-keep-type repDet singletonD
      by (metis-node-eq-binary SignedFloatingIntegerDivNode)
    then have ∃ xe2 ye2. (g2 ⊢ n ≃ BinaryExpr BinDiv xe2 ye2) ∧
        BinaryExpr BinDiv xe1 ye1 ≥ BinaryExpr BinDiv xe2 ye2
      by (metis DivNode.prems l mono-binary rep.DivNode xer)
    then show ?thesis
      by meson
  qed
next
  case (ModNode n x y xe1 ye1)
  have k: g1 ⊢ n ≃ BinaryExpr BinMod xe1 ye1
    using ModNode by (simp add: ModNode.hyps(2) rep.ModNode f)
  obtain xn yn where l: kind g1 n = SignedFloatingIntegerRemNode xn yn
    by (simp add: ModNode.hyps(1))
  then have mx: g1 ⊢ xn ≃ xe1
    using ModNode.hyps(1,2) by simp
  from l have my: g1 ⊢ yn ≃ ye1
    using ModNode.hyps(1,3) by simp
  then show ?case
  proof −
    have g1 ⊢ xn ≃ xe1
      by (simp add: mx)
    have g1 ⊢ yn ≃ ye1
```

by (*simp add*: *my*)
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *ModNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary SignedFloatingIntegerRemNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *ModNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary SignedFloatingIntegerRemNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinMod xe2 ye2*) ∧
BinaryExpr BinMod xe1 ye1 ≥ *BinaryExpr BinMod xe2 ye2*
by (*metis ModNode.prems l mono-binary rep.ModNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*SubNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinSub xe1 ye1*
using *SubNode* **by** (*simp add*: *SubNode.hyps*(*2*) *rep.SubNode f*)
obtain *xn yn* **where** *l*: *kind g1 n* = *SubNode xn yn*
by (*simp add*: *SubNode.hyps*(*1*))
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *SubNode.hyps*(*1*,*2*) **by** *simp*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *SubNode.hyps*(*1*,*3*) **by** *simp*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *mx*)
have *g1* ⊢ *yn* ≃ *ye1*
by (*simp add*: *my*)
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *SubNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary SubNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *SubNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary SubNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinSub xe2 ye2*) ∧
BinaryExpr BinSub xe1 ye1 ≥ *BinaryExpr BinSub xe2 ye2*
by (*metis SubNode.prems l mono-binary rep.SubNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*AndNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinAnd xe1 ye1*
using *AndNode* **by** (*simp add*: *AndNode.hyps*(*2*) *rep.AndNode f*)
obtain *xn yn* **where** *l*: *kind g1 n* = *AndNode xn yn*
using *AndNode.hyps*(*1*) **by** *simp*

then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
  using *AndNode.hyps*(*1*,*2*) **by** *simp*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
  using *AndNode.hyps*(*1*,*3*) **by** *simp*
then show *?case*
proof −
  have *g1* ⊢ *xn* ≃ *xe1*
    by (*simp add*: *mx*)
  have *g1* ⊢ *yn* ≃ *ye1*
    by (*simp add*: *my*)
  have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
      using *AndNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
    by (*metis-node-eq-binary AndNode*)
  have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
      using *AndNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
    by (*metis-node-eq-binary AndNode*)
  then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinAnd xe2 ye2*) ∧
      BinaryExpr BinAnd xe1 ye1 ≥ *BinaryExpr BinAnd xe2 ye2*
    by (*metis AndNode.prems l mono-binary rep.AndNode xer*)
  then show *?thesis*
    by *meson*
qed
next
case (*OrNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinOr xe1 ye1*
  using *OrNode* **by** (*simp add*: *OrNode.hyps*(*2*) *rep.OrNode f*)
obtain *xn yn* **where** *l*: *kind g1 n* = *OrNode xn yn*
  using *OrNode.hyps*(*1*) **by** *simp*
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
  using *OrNode.hyps*(*1*,*2*) **by** *simp*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
  using *OrNode.hyps*(*1*,*3*) **by** *simp*
then show *?case*
proof −
  have *g1* ⊢ *xn* ≃ *xe1*
    by (*simp add*: *mx*)
  have *g1* ⊢ *yn* ≃ *ye1*
    by (*simp add*: *my*)
  have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
  using *OrNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
    by (*metis-node-eq-binary OrNode*)
  have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
  using *OrNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
    by (*metis-node-eq-binary OrNode*)
  then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinOr xe2 ye2*) ∧
      BinaryExpr BinOr xe1 ye1 ≥ *BinaryExpr BinOr xe2 ye2*
    by (*metis OrNode.prems l mono-binary rep.OrNode xer*)

```
      then show ?thesis
        by meson
    qed
  next
    case (XorNode n x y xe1 ye1)
    have k: g1 ⊢ n ≃ BinaryExpr BinXor xe1 ye1
      using XorNode by (simp add: XorNode.hyps(2) rep.XorNode f)
    obtain xn yn where l: kind g1 n = XorNode xn yn
      using XorNode.hyps(1) by simp
    then have mx: g1 ⊢ xn ≃ xe1
      using XorNode.hyps(1,2) by simp
    from l have my: g1 ⊢ yn ≃ ye1
      using XorNode.hyps(1,3) by simp
    then show ?case
    proof -
      have g1 ⊢ xn ≃ xe1
        by (simp add: mx)
      have g1 ⊢ yn ≃ ye1
        by (simp add: my)
      have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
           using XorNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
        by (metis-node-eq-binary XorNode)
      have ∃ ye2. (g2 ⊢ yn ≃ ye2) ∧ ye1 ≥ ye2
           using XorNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
        by (metis-node-eq-binary XorNode)
      then have ∃ xe2 ye2. (g2 ⊢ n ≃ BinaryExpr BinXor xe2 ye2) ∧
          BinaryExpr BinXor xe1 ye1 ≥ BinaryExpr BinXor xe2 ye2
        by (metis XorNode.prems l mono-binary rep.XorNode xer)
      then show ?thesis
        by meson
    qed
  next
  case (ShortCircuitOrNode n x y xe1 ye1)
  have k: g1 ⊢ n ≃ BinaryExpr BinShortCircuitOr xe1 ye1
  using ShortCircuitOrNode by (simp add: ShortCircuitOrNode.hyps(2) rep.ShortCircuitOrNode
f)
    obtain xn yn where l: kind g1 n = ShortCircuitOrNode xn yn
      using ShortCircuitOrNode.hyps(1) by simp
    then have mx: g1 ⊢ xn ≃ xe1
      using ShortCircuitOrNode.hyps(1,2) by simp
    from l have my: g1 ⊢ yn ≃ ye1
      using ShortCircuitOrNode.hyps(1,3) by simp
    then show ?case
    proof -
      have g1 ⊢ xn ≃ xe1
        by (simp add: mx)
      have g1 ⊢ yn ≃ ye1
```

by (*simp add*: *my*)
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *ShortCircuitOrNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary ShortCircuitOrNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *ShortCircuitOrNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary ShortCircuitOrNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinShortCircuitOr xe2 ye2*) ∧
BinaryExpr BinShortCircuitOr xe1 ye1 ≥ *BinaryExpr BinShortCircuitOr xe2 ye2*
by (*metis ShortCircuitOrNode.prems l mono-binary rep.ShortCircuitOrNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*LeftShiftNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinLeftShift xe1 ye1*
using *LeftShiftNode* **by** (*simp add*: *LeftShiftNode.hyps*(*2*) *rep.LeftShiftNode f*)
obtain *xn yn* **where** *l*: *kind g1 n* = *LeftShiftNode xn yn*
using *LeftShiftNode.hyps*(*1*) **by** *simp*
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *LeftShiftNode.hyps*(*1*,*2*) **by** *simp*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *LeftShiftNode.hyps*(*1*,*3*) **by** *simp*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *mx*)
have *g1* ⊢ *yn* ≃ *ye1*
by (*simp add*: *my*)
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *LeftShiftNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary LeftShiftNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *LeftShiftNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary LeftShiftNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinLeftShift xe2 ye2*) ∧
BinaryExpr BinLeftShift xe1 ye1 ≥ *BinaryExpr BinLeftShift xe2 ye2*
by (*metis LeftShiftNode.prems l mono-binary rep.LeftShiftNode xer*)
then show *?thesis*
by *meson*
qed
next

```
    case (RightShiftNode n x y xe1 ye1)
    have k: g1 ⊢ n ≃ BinaryExpr BinRightShift xe1 ye1
    using RightShiftNode by (simp add: RightShiftNode.hyps(2) rep.RightShiftNode)
    obtain xn yn where l: kind g1 n = RightShiftNode xn yn
      using RightShiftNode.hyps(1) by simp
    then have mx: g1 ⊢ xn ≃ xe1
      using RightShiftNode.hyps(1,2) by simp
    from l have my: g1 ⊢ yn ≃ ye1
      using RightShiftNode.hyps(1,3) by simp
    then show ?case
    proof -
      have g1 ⊢ xn ≃ xe1
        by (simp add: mx)
      have g1 ⊢ yn ≃ ye1
        by (simp add: my)
      have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
        using RightShiftNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
        by (metis-node-eq-binary RightShiftNode)
      have ∃ ye2. (g2 ⊢ yn ≃ ye2) ∧ ye1 ≥ ye2
        using RightShiftNode a b c d l no-encoding not-excluded-keep-type repDet
singletonD
        by (metis-node-eq-binary RightShiftNode)
      then have ∃ xe2 ye2. (g2 ⊢ n ≃ BinaryExpr BinRightShift xe2 ye2) ∧
    BinaryExpr BinRightShift xe1 ye1 ≥ BinaryExpr BinRightShift xe2 ye2
        by (metis RightShiftNode.prems l mono-binary rep.RightShiftNode xer)
      then show ?thesis
        by meson
    qed
  next
    case (UnsignedRightShiftNode n x y xe1 ye1)
    have k: g1 ⊢ n ≃ BinaryExpr BinURightShift xe1 ye1
    using UnsignedRightShiftNode by (simp add: UnsignedRightShiftNode.hyps(2)

                                              rep.UnsignedRightShiftNode)
    obtain xn yn where l: kind g1 n = UnsignedRightShiftNode xn yn
      using UnsignedRightShiftNode.hyps(1) by simp
    then have mx: g1 ⊢ xn ≃ xe1
      using UnsignedRightShiftNode.hyps(1,2) by simp
    from l have my: g1 ⊢ yn ≃ ye1
      using UnsignedRightShiftNode.hyps(1,3) by simp
    then show ?case
    proof -
      have g1 ⊢ xn ≃ xe1
        by (simp add: mx)
      have g1 ⊢ yn ≃ ye1
        by (simp add: my)
      have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
        using UnsignedRightShiftNode a b c d no-encoding not-excluded-keep-type
```

repDet singletonD
l
by (*metis-node-eq-binary UnsignedRightShiftNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *UnsignedRightShiftNode a b c d no-encoding not-excluded-keep-type repDet singletonD*
l
by (*metis-node-eq-binary UnsignedRightShiftNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinURightShift xe2 ye2*) ∧ *BinaryExpr BinURightShift xe1 ye1* ≥ *BinaryExpr BinURightShift xe2 ye2*
by (*metis UnsignedRightShiftNode.prems l mono-binary rep.UnsignedRightShiftNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*IntegerBelowNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinIntegerBelow xe1 ye1*
using *IntegerBelowNode* **by** (*simp add*: *IntegerBelowNode.hyps*(*2*) *rep.IntegerBelowNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *IntegerBelowNode xn yn*
using *IntegerBelowNode.hyps*(*1*) **by** *simp*
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *IntegerBelowNode.hyps*(*1*,*2*) **by** *simp*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *IntegerBelowNode.hyps*(*1*,*3*) **by** *simp*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *mx*)
have *g1* ⊢ *yn* ≃ *ye1*
by (*simp add*: *my*)
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *IntegerBelowNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary IntegerBelowNode*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *IntegerBelowNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary IntegerBelowNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinIntegerBelow xe2 ye2*) ∧ *BinaryExpr BinIntegerBelow xe1 ye1* ≥ *BinaryExpr BinIntegerBelow xe2 ye2*
by (*metis IntegerBelowNode.prems l mono-binary rep.IntegerBelowNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*IntegerEqualsNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinIntegerEquals xe1 ye1*

```
      using IntegerEqualsNode by (simp add: IntegerEqualsNode.hyps(2) rep.IntegerEqualsNode)
    obtain xn yn where l: kind g1 n = IntegerEqualsNode xn yn
      using IntegerEqualsNode.hyps(1) by simp
    then have mx: g1 ⊢ xn ≃ xe1
      using IntegerEqualsNode.hyps(1,2) by simp
    from l have my: g1 ⊢ yn ≃ ye1
      using IntegerEqualsNode.hyps(1,3) by simp
    then show ?case
    proof -
      have g1 ⊢ xn ≃ xe1
        by (simp add: mx)
      have g1 ⊢ yn ≃ ye1
        by (simp add: my)
      have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using IntegerEqualsNode a b c d l no-encoding not-excluded-keep-type
repDet singletonD
        by (metis-node-eq-binary IntegerEqualsNode)
      have ∃ ye2. (g2 ⊢ yn ≃ ye2) ∧ ye1 ≥ ye2
          using IntegerEqualsNode a b c d l no-encoding not-excluded-keep-type
repDet singletonD
        by (metis-node-eq-binary IntegerEqualsNode)
      then have ∃ xe2 ye2. (g2 ⊢ n ≃ BinaryExpr BinIntegerEquals xe2 ye2) ∧
  BinaryExpr BinIntegerEquals xe1 ye1 ≥ BinaryExpr BinIntegerEquals xe2 ye2
        by (metis IntegerEqualsNode.prems l mono-binary rep.IntegerEqualsNode
xer)
      then show ?thesis
        by meson
    qed
  next
    case (IntegerLessThanNode n x y xe1 ye1)
    have k: g1 ⊢ n ≃ BinaryExpr BinIntegerLessThan xe1 ye1
      using IntegerLessThanNode by (simp add: IntegerLessThanNode.hyps(2)
rep.IntegerLessThanNode)
    obtain xn yn where l: kind g1 n = IntegerLessThanNode xn yn
      using IntegerLessThanNode.hyps(1) by simp
    then have mx: g1 ⊢ xn ≃ xe1
      using IntegerLessThanNode.hyps(1,2) by simp
    from l have my: g1 ⊢ yn ≃ ye1
      using IntegerLessThanNode.hyps(1,3) by simp
    then show ?case
    proof -
      have g1 ⊢ xn ≃ xe1
        by (simp add: mx)
      have g1 ⊢ yn ≃ ye1
        by (simp add: my)
      have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
        using IntegerLessThanNode a b c d l no-encoding not-excluded-keep-type
repDet singletonD
        by (metis-node-eq-binary IntegerLessThanNode)
```

have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *IntegerLessThanNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis-node-eq-binary IntegerLessThanNode*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinIntegerLessThan xe2 ye2*) ∧
BinaryExpr BinIntegerLessThan xe1 ye1 ≥ *BinaryExpr BinIntegerLessThan xe2 ye2*
by (*metis IntegerLessThanNode.prems l mono-binary rep.IntegerLessThanNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*IntegerTestNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinIntegerTest xe1 ye1*
using *IntegerTestNode* **by** (*meson rep.IntegerTestNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *IntegerTestNode xn yn*
by (*simp add*: *IntegerTestNode.hyps*(*1*))
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *IRNode.inject*(*21*) *IntegerTestNode.hyps*(*1*,*2*) **by** *presburger*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
by (*metis IRNode.inject*(*21*) *IntegerTestNode.hyps*(*1*,*3*))
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *mx*)
have *g1* ⊢ *yn* ≃ *ye1*
by (*simp add*: *my*)
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *IntegerTestNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis IRNode.inject*(*21*))
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
using *IntegerLessThanNode a b c d l no-encoding not-excluded-keep-type repDet singletonD*
by (*metis IRNode.inject*(*21*) *IntegerTestNode.IH*(*2*) *IntegerTestNode.hyps*(*1*) *my*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinIntegerTest xe2 ye2*) ∧
BinaryExpr BinIntegerTest xe1 ye1 ≥ *BinaryExpr BinIntegerTest xe2 ye2*
by (*metis IntegerTestNode.prems l mono-binary xer rep.IntegerTestNode*)
then show *?thesis*
by *meson*
qed
next
case (*IntegerNormalizeCompareNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinIntegerNormalizeCompare xe1 ye1*
by (*simp add*: *IntegerNormalizeCompareNode.hyps*(*1*,*2*,*3*) *rep.IntegerNormalizeCompareNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *IntegerNormalizeCompareNode xn yn*

by (*simp add*: *IntegerNormalizeCompareNode.hyps*(*1*))
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *IRNode.inject*(*20*) *IntegerNormalizeCompareNode.hyps*(*1*,*2*) **by** *presburger*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *IRNode.inject*(*20*) *IntegerNormalizeCompareNode.hyps*(*1*,*3*) **by** *presburger*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *mx*)
have *g1* ⊢ *yn* ≃ *ye1*
by (*simp add*: *my*)
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
by (*metis IRNode.inject*(*20*) *IntegerNormalizeCompareNode.IH*(*1*) *l mx no-encoding a b c d*
IntegerNormalizeCompareNode.hyps(*1*) *emptyE insertE not-excluded-keep-type repDet*)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
by (*metis IRNode.inject*(*20*) *IntegerNormalizeCompareNode.IH*(*2*) *my no-encoding a b c d l*
IntegerNormalizeCompareNode.hyps(*1*) *emptyE insertE not-excluded-keep-type repDet*)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinIntegerNormalizeCompare xe2 ye2*) ∧
BinaryExpr BinIntegerNormalizeCompare xe1 ye1 ≥ *BinaryExpr BinIntegerNormalizeCompare xe2 ye2*
by (*metis IntegerNormalizeCompareNode.prems l mono-binary rep.IntegerNormalizeCompareNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*IntegerMulHighNode n x y xe1 ye1*)
have *k*: *g1* ⊢ *n* ≃ *BinaryExpr BinIntegerMulHigh xe1 ye1*
by (*simp add*: *IntegerMulHighNode.hyps*(*1*,*2*,*3*) *rep.IntegerMulHighNode*)
obtain *xn yn* **where** *l*: *kind g1 n* = *IntegerMulHighNode xn yn*
by (*simp add*: *IntegerMulHighNode.hyps*(*1*))
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *IRNode.inject*(*19*) *IntegerMulHighNode.hyps*(*1*,*2*) **by** *presburger*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *IRNode.inject*(*19*) *IntegerMulHighNode.hyps*(*1*,*3*) **by** *presburger*
then show *?case*
proof −
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *mx*)
have *g1* ⊢ *yn* ≃ *ye1*
by (*simp add*: *my*)
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*

by (*metis IRNode.inject*(*19*) *IntegerMulHighNode.IH*(*1*) *IntegerMulHighNode.hyps*(*1*) *a b c d*
emptyE insertE l mx no-encoding not-excluded-keep-type repDet)
have ∃ *ye2*. (*g2* ⊢ *yn* ≃ *ye2*) ∧ *ye1* ≥ *ye2*
by (*metis IRNode.inject*(*19*) *IntegerMulHighNode.IH*(*2*) *IntegerMulHighNode.hyps*(*1*) *a b c d*
emptyE insertE l my no-encoding not-excluded-keep-type repDet)
then have ∃ *xe2 ye2*. (*g2* ⊢ *n* ≃ *BinaryExpr BinIntegerMulHigh xe2 ye2*) ∧ *BinaryExpr BinIntegerMulHigh xe1 ye1* ≥ *BinaryExpr BinIntegerMulHigh xe2 ye2*
by (*metis IntegerMulHighNode.prems l mono-binary rep.IntegerMulHighNode xer*)
then show *?thesis*
by *meson*
qed
next
case (*NarrowNode n inputBits resultBits x xe1*)
have *k*: *g1* ⊢ *n* ≃ *UnaryExpr* (*UnaryNarrow inputBits resultBits*) *xe1*
using *NarrowNode* **by** (*simp add*: *NarrowNode.hyps*(*2*) *rep.NarrowNode*)
obtain *xn* **where** *l*: *kind g1 n* = *NarrowNode inputBits resultBits xn*
using *NarrowNode.hyps*(*1*) **by** *simp*
then have *m*: *g1* ⊢ *xn* ≃ *xe1*
using *NarrowNode.hyps*(*1*,*2*) **by** *simp*
then show *?case*
proof (*cases xn* = *n'*)
case *True*
then have *n*: *xe1* = *e1'*
using *m* **by** (*simp add*: *repDet c*)
then have *ev*: *g2* ⊢ *n* ≃ *UnaryExpr* (*UnaryNarrow inputBits resultBits*) *e2'*

using *l* **by** (*simp add*: *rep.NarrowNode d True NarrowNode.prems*)
then have *r*: *UnaryExpr* (*UnaryNarrow inputBits resultBits*) *e1'* ≥
UnaryExpr (*UnaryNarrow inputBits resultBits*) *e2'*
by (*meson a mono-unary*)
then show *?thesis*
by (*metis n ev*)
next
case *False*
have *g1* ⊢ *xn* ≃ *xe1*
by (*simp add*: *m*)
have ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *NarrowNode False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff*
by (*metis-node-eq-ternary NarrowNode*)
then have ∃ *xe2*. (*g2* ⊢ *n* ≃ *UnaryExpr* (*UnaryNarrow inputBits resultBits*) *xe2*) ∧
UnaryExpr (*UnaryNarrow inputBits resultBits*) *xe1* ≥
UnaryExpr (*UnaryNarrow inputBits resultBits*) *xe2*
by (*metis NarrowNode.prems l mono-unary rep.NarrowNode*)
then show *?thesis*

```
        by meson
      qed
    next
      case (SignExtendNode n inputBits resultBits x xe1)
      have k: g1 ⊢ n ≃ UnaryExpr (UnarySignExtend inputBits resultBits) xe1
     using SignExtendNode by (simp add: SignExtendNode.hyps(2) rep.SignExtendNode)
      obtain xn where l: kind g1 n = SignExtendNode inputBits resultBits xn
        using SignExtendNode.hyps(1) by simp
      then have m: g1 ⊢ xn ≃ xe1
        using SignExtendNode.hyps(1,2) by simp
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1'
          using m by (simp add: repDet c)
       then have ev: g2 ⊢ n ≃ UnaryExpr (UnarySignExtend inputBits resultBits)
e2'
         using l by (simp add: True d rep.SignExtendNode SignExtendNode.prems)
        then have r: UnaryExpr (UnarySignExtend inputBits resultBits) e1' ≥
                    UnaryExpr (UnarySignExtend inputBits resultBits) e2'
          by (meson a mono-unary)
        then show ?thesis
          by (metis n ev)
      next
        case False
        have g1 ⊢ xn ≃ xe1
          by (simp add: m)
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
           using SignExtendNode False b encodes-contains l not-excluded-keep-type
not-in-g
              singleton-iff
          by (metis-node-eq-ternary SignExtendNode)
          then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr (UnarySignExtend inputBits
resultBits) xe2) ∧
                              UnaryExpr (UnarySignExtend inputBits resultBits)
xe1 ≥
                          UnaryExpr (UnarySignExtend inputBits resultBits) xe2
          by (metis SignExtendNode.prems l mono-unary rep.SignExtendNode)
        then show ?thesis
          by meson
      qed
    next
      case (ZeroExtendNode n inputBits resultBits x xe1)
      have k: g1 ⊢ n ≃ UnaryExpr (UnaryZeroExtend inputBits resultBits) xe1
     using ZeroExtendNode by (simp add: ZeroExtendNode.hyps(2) rep.ZeroExtendNode)
      obtain xn where l: kind g1 n = ZeroExtendNode inputBits resultBits xn
        using ZeroExtendNode.hyps(1) by simp
      then have m: g1 ⊢ xn ≃ xe1
        using ZeroExtendNode.hyps(1,2) by simp
```

```
then show ?case
proof (cases xn = n′)
  case True
  then have n: xe1 = e1′
    using m by (simp add: repDet c)
 then have ev: g2 ⊢ n ≃ UnaryExpr (UnaryZeroExtend inputBits resultBits)
e2′
   using l by (simp add: ZeroExtendNode.prems True d rep.ZeroExtendNode)
  then have r: UnaryExpr (UnaryZeroExtend inputBits resultBits) e1′ ≥
           UnaryExpr (UnaryZeroExtend inputBits resultBits) e2′
    by (meson a mono-unary)
  then show ?thesis
    by (metis n ev)
next
  case False
  have g1 ⊢ xn ≃ xe1
    by (simp add: m)
  have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
   using ZeroExtendNode b encodes-contains l not-excluded-keep-type not-in-g
singleton-iff
        False
    by (metis-node-eq-ternary ZeroExtendNode)
    then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr (UnaryZeroExtend inputBits
resultBits) xe2) ∧
                      UnaryExpr (UnaryZeroExtend inputBits resultBits)
xe1 ≥
                     UnaryExpr (UnaryZeroExtend inputBits resultBits) xe2
    by (metis ZeroExtendNode.prems l mono-unary rep.ZeroExtendNode)
  then show ?thesis
    by meson
qed
next
  case (LeafNode n s)
  then show ?case
    by (metis eq-refl rep.LeafNode)
next
  case (PiNode n′ gu)
  then show ?case
   by (metis encodes-contains not-excluded-keep-type not-in-g rep.PiNode repDet
singleton-iff
     a b c d)
next
  case (RefNode n′)
  then show ?case
    by (metis a b c d no-encoding not-excluded-keep-type rep.RefNode repDet
singletonD)
next
  case (IsNullNode n)
  then show ?case
```

by (*metis insertE mono-unary no-encoding not-excluded-keep-type rep.IsNullNode repDet emptyE*
a b c d)
qed
qed
qed

lemma *graph-semantics-preservation-subscript*:
assumes *a*: $e_1{}' \geq e_2{}'$
assumes *b*: $(\{n\} \trianglelefteq \textit{as-set } g_1) \subseteq \textit{as-set } g_2$
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
using *assms* **by** (*simp add*: *graph-semantics-preservation*)

lemma *tree-to-graph-rewriting*:
$e_1 \geq e_2$
$\wedge$ $(g_1 \vdash n \simeq e_1)$ $\wedge$ *maximal-sharing* g_1
$\wedge$ $(\{n\} \trianglelefteq \textit{as-set } g_1) \subseteq \textit{as-set } g_2$
$\wedge$ $(g_2 \vdash n \simeq e_2)$ $\wedge$ *maximal-sharing* g_2
$\Longrightarrow$ *graph-refinement* g_1 g_2
by (*auto simp add*: *graph-semantics-preservation*)

declare [[*simp-trace*]]
lemma *equal-refines*:
fixes *e1 e2* :: *IRExpr*
assumes *e1* = *e2*
shows *e1* ≥ *e2*
using *assms* **by** *simp*
declare [[*simp-trace=false*]]

lemma *eval-contains-id*[*simp*]: *g1* ⊢ *n* ≃ *e* ⟹ *n* ∈ *ids g1*
using *no-encoding* **by** *auto*

lemma *subset-kind*[*simp*]: *as-set g1* ⊆ *as-set g2* ⟹ *g1* ⊢ *n* ≃ *e* ⟹ *kind g1 n* = *kind g2 n*
using *eval-contains-id as-set-def* **by** *blast*

lemma *subset-stamp*[*simp*]: *as-set g1* ⊆ *as-set g2* ⟹ *g1* ⊢ *n* ≃ *e* ⟹ *stamp g1 n* = *stamp g2 n*
using *eval-contains-id as-set-def* **by** *blast*

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*)
using *assms*(*2*)
apply (*induction e*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *ConstantNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *ParameterNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *ConditionalNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *AbsNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *ReverseBytesNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *BitCountNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *NotNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *NegateNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *LogicNegationNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *AddNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *MulNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *DivNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *ModNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *SubNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *AndNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *OrNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *XorNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *ShortCircuitOrNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *LeftShiftNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *RightShiftNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *UnsignedRightShiftNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *IntegerBelowNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *IntegerEqualsNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *IntegerLessThanNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *IntegerTestNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *IntegerNormalizeCompareNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *IntegerMulHighNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *NarrowNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *SignExtendNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *ZeroExtendNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *LeafNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *PiNode*)
apply (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *RefNode*)
by (*solve-subset-eval as-set*: *assms*(*1*) *eval*: *IsNullNode*)

lemma *subset-refines*:
assumes *as-set g1* ⊆ *as-set g2*
shows *graph-refinement g1 g2*
proof −
have *ids g1* ⊆ *ids g2*
using *assms as-set-def* **by** *blast*
then show *?thesis*
unfolding *graph-refinement-def*
apply *rule* **apply** (*rule allI*) **apply** (*rule impI*) **apply** (*rule allI*) **apply** (*rule*

impI)
  unfolding *graph-represents-expression-def*
  proof −
   fix *n e1*
   assume *1*:$n \in ids\ g1$
   assume *2*:$g1 \vdash n \simeq e1$
   show $\exists\, e2.\ (g2 \vdash n \simeq e2) \wedge e1 \geq e2$
    by (*meson equal-refines subset-implies-evals assms 1 2*)
  qed
 qed

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* $g_1\ g_2$
 by (*meson encodeeval.simps graph-represents-expression-def le-expr-def subset-refines*)

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*
 by (*metis* (*mono-tags*, *lifting*) *find-Some-iff find-node-and-stamp.simps assms*)

lemma *find-exists-stamp*:
 assumes *find-node-and-stamp g* (*node*, *s*) = *Some nid*
 shows *stamp g nid* = *s*
 by (*metis* (*mono-tags*, *lifting*) *find-Some-iff find-node-and-stamp.simps assms*)

lemma *find-new-kind*:
 assumes g' = *add-node nid* (*node*, *s*) *g*
 assumes *node* ≠ *NoNode*
 shows *kind* g' *nid* = *node*
 by (*simp add*: *add-node-lookup assms*)

lemma *find-new-stamp*:
 assumes g' = *add-node nid* (*node*, *s*) *g*
 assumes *node* ≠ *NoNode*
 shows *stamp* g' *nid* = *s*
 by (*simp add*: *assms add-node-lookup*)

lemma *sorted-bottom*:
 assumes *finite xs*
 assumes $x \in xs$
 shows $x \leq$ *last*(*sorted-list-of-set*(*xs*::*nat set*))
 proof −
 obtain *largest* **where** *largest*: *largest* = *last* (*sorted-list-of-set*(*xs*))
  by *simp*

obtain *sortedList* **where** *sortedList*: *sortedList* = *sorted-list-of-set*(*xs*)
by *simp*
have *step*: $\forall$ *i*. *0* < *i* $\wedge$ *i* < (*length* (*sortedList*)) $\longrightarrow$ *sortedList*!(*i*−*1*) $\leq$ *sortedList*!(*i*)
unfolding *sortedList* **apply** *auto*
by (*metis diff-le-self sorted-list-of-set.length-sorted-key-list-of-set sorted-nth-mono sorted-list-of-set*(*2*))
have *finalElement*: *last* (*sorted-list-of-set*(*xs*)) =
sorted-list-of-set(*xs*)!(*length* (*sorted-list-of-set*(*xs*)) − *1*)
using *assms last-conv-nth sorted-list-of-set.sorted-key-list-of-set-eq-Nil-iff* **by** *blast*
have *contains0*: (*x* $\in$ *xs*) = (*x* $\in$ *set* (*sorted-list-of-set*(*xs*)))
using *assms*(*1*) **by** *auto*
have *lastLargest*: ((*x* $\in$ *xs*) $\longrightarrow$ (*largest* $\geq$ *x*))
using *step* **unfolding** *largest finalElement* **apply** *auto*
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-pred assms*(*1*) *card-Diff1-less in-set-conv-nth sorted-list-of-set.length-sorted-key-list-of-set card-Diff-singleton-if less-Suc-eq-le sorted-list-of-set.sorted-sorted-key-list-of-set length-pos-if-in-set sorted-nth-mono contains0*)
then show *?thesis*
by (*simp add*: *assms largest*)
qed

lemma *fresh*: *finite xs* $\Longrightarrow$ *last*(*sorted-list-of-set*(*xs*::*nat set*)) + *1* $\notin$ *xs*
using *sorted-bottom not-le* **by** *auto*

lemma *fresh-ids*:
assumes *n* = *get-fresh-id g*
shows *n* $\notin$ *ids g*
proof −
have *finite* (*ids g*)
by (*simp add*: *Rep-IRGraph*)
then show *?thesis*
using *assms fresh* **unfolding** *get-fresh-id.simps* **by** *blast*
qed

lemma *graph-unchanged-rep-unchanged*:
assumes $\forall$ *n* $\in$ *ids g*. *kind g n* = *kind g′ n*
assumes $\forall$ *n* $\in$ *ids g*. *stamp g n* = *stamp g′ n*
shows (*g* $\vdash$ *n* $\simeq$ *e*) $\longrightarrow$ (*g′* $\vdash$ *n* $\simeq$ *e*)
apply (*rule impI*) **subgoal premises** *e* **using** *e assms*
apply (*induction n e*)
apply (*metis no-encoding rep.ConstantNode*)
apply (*metis no-encoding rep.ParameterNode*)
apply (*metis no-encoding rep.ConditionalNode*)
apply (*metis no-encoding rep.AbsNode*)
apply (*metis no-encoding rep.ReverseBytesNode*)

```
apply (metis no-encoding rep.BitCountNode)
apply (metis no-encoding rep.NotNode)
apply (metis no-encoding rep.NegateNode)
apply (metis no-encoding rep.LogicNegationNode)
apply (metis no-encoding rep.AddNode)
apply (metis no-encoding rep.MulNode)
apply (metis no-encoding rep.DivNode)
apply (metis no-encoding rep.ModNode)
apply (metis no-encoding rep.SubNode)
apply (metis no-encoding rep.AndNode)
apply (metis no-encoding rep.OrNode)
apply (metis no-encoding rep.XorNode)
apply (metis no-encoding rep.ShortCircuitOrNode)
apply (metis no-encoding rep.LeftShiftNode)
apply (metis no-encoding rep.RightShiftNode)
apply (metis no-encoding rep.UnsignedRightShiftNode)
apply (metis no-encoding rep.IntegerBelowNode)
apply (metis no-encoding rep.IntegerEqualsNode)
apply (metis no-encoding rep.IntegerLessThanNode)
apply (metis no-encoding rep.IntegerTestNode)
apply (metis no-encoding rep.IntegerNormalizeCompareNode)
apply (metis no-encoding rep.IntegerMulHighNode)
apply (metis no-encoding rep.NarrowNode)
apply (metis no-encoding rep.SignExtendNode)
apply (metis no-encoding rep.ZeroExtendNode)
apply (metis no-encoding rep.LeafNode)
apply (metis no-encoding rep.PiNode)
apply (metis no-encoding rep.RefNode)
by (metis no-encoding rep.IsNullNode)
done

lemma fresh-node-subset:
  assumes n ∉ ids g
  assumes g′ = add-node n (k, s) g
  shows as-set g ⊆ as-set g′
 by (smt (z3) Collect-mono-iff Diff-idemp Diff-insert-absorb add-changed as-set-def
unchanged.simps
      disjoint-change assms)

lemma unique-subset:
  assumes unique g node (g′, n)
  shows as-set g ⊆ as-set g′
  using assms fresh-ids fresh-node-subset
  by (metis Pair-inject old.prod.exhaust subsetI unique.cases)

lemma unrep-subset:
  assumes (g ⊕ e ⇝ (g′, n))
  shows as-set g ⊆ as-set g′
  using assms
```

proof (*induction g e* (g', *n*) *arbitrary*: g' *n*)
  case (*UnrepConstantNode g c n* g')
  then show *?case* **using** *unique-subset* **by** *simp*
next
  case (*UnrepParameterNode g i s n*)
  then show *?case* **using** *unique-subset* **by** *simp*
next
  case (*UnrepConditionalNode g ce g2 c te g3 t fe g4 f* s' *n*)
  then show *?case* **using** *unique-subset* **by** *blast*
next
  case (*UnrepUnaryNode g xe g2 x* s' *op n*)
  then show *?case* **using** *unique-subset* **by** *blast*
next
  case (*UnrepBinaryNode g xe g2 x ye g3 y* s' *op n*)
  then show *?case* **using** *unique-subset* **by** *blast*
next
  case (*AllLeafNodes g n s*)
  then show *?case*
    by *auto*
qed

lemma *fresh-node-preserves-other-nodes*:
  assumes $n' = $ *get-fresh-id g*
  assumes $g' = $ *add-node* n' (*k*, *s*) *g*
  shows $\forall\ n \in ids\ g\ .\ (g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$
  using *assms* **apply** *auto*
  by (*metis fresh-node-subset subset-implies-evals fresh-ids assms*)

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

lemma *unrep-ids-subset*[*simp*]:
  assumes $g \oplus e \rightsquigarrow (g', n)$
  shows $ids\ g \subseteq ids\ g'$
  by (*meson graph-refinement-def subset-refines unrep-subset assms*)

lemma *unrep-unchanged*:
  assumes $g \oplus e \rightsquigarrow (g', n)$
  shows $\forall\ n \in ids\ g\ .\ \forall\ e.\ (g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$
  by (*meson subset-implies-evals unrep-subset assms*)

lemma *unique-kind*:
  assumes *unique g* (*node*, *s*) (g', *nid*)
  assumes $node \neq NoNode$
  shows $kind\ g'\ nid = node \land stamp\ g'\ nid = s$
  using *assms find-exists-kind add-node-lookup*
  by (*smt* (*verit*, *del-insts*) *Pair-inject find-exists-stamp unique.cases*)

lemma *unique-eval*:
  assumes *unique g* (*n*, *s*) (*g'*, *nid*)
  shows $g \vdash nid' \simeq e \Longrightarrow g' \vdash nid' \simeq e$
  using *assms subset-implies-evals unique-subset* **by** *blast*

lemma *unrep-eval*:
  assumes *unrep g e* (*g'*, *nid*)
  shows $g \vdash nid' \simeq e' \Longrightarrow g' \vdash nid' \simeq e'$
  using *assms subset-implies-evals no-encoding unrep-unchanged* **by** *blast*

lemma *unary-node-nonode*:
  unary-node op x ≠ *NoNode*
  by (*cases op*; *auto*)

lemma *bin-node-nonode*:
  bin-node op x y ≠ *NoNode*
  by (*cases op*; *auto*)

theorem *term-graph-reconstruction*:
  $g \oplus e \rightsquigarrow (g', n) \Longrightarrow (g' \vdash n \simeq e) \wedge$ *as-set g* ⊆ *as-set g'*
  subgoal premises *e* **apply** (*rule conjI*) **defer**
    using *e unrep-subset* **apply** *blast* **using** *e*
  proof (*induction g e* (*g'*, *n*) *arbitrary*: *g' n*)
    case (*UnrepConstantNode g c* g_1 *n*)
    then show *?case*
      using *ConstantNode unique-kind* **by** *blast*
  next
    case (*UnrepParameterNode g i s* g_1 *n*)
    then show *?case*
      using *ParameterNode unique-kind*
      by (*metis IRNode.distinct*(*3695*))
  next
    case (*UnrepConditionalNode g ce* g_1 *c te* g_2 *t fe* g_3 *f s'* g_4 *n*)
    then show *?case*
      using *unique-kind unique-eval unrep-eval*
      by (*meson ConditionalNode IRNode.distinct*(*965*))
  next
    case (*UnrepUnaryNode g xe* g_1 *x s' op* g_2 *n*)
    then have *k*: *kind* g_2 *n* = *unary-node op x*
      using *unique-kind unary-node-nonode* **by** *simp*
    then have $g_2 \vdash x \simeq xe$
      using *UnrepUnaryNode unique-eval* **by** *blast*
    then show *?case*
      using *k* **apply** (*cases op*)
      using *unary-node.simps*(*1*,*2*,*3*,*4*,*5*,*6*,*7*,*8*,*9*,*10*)
        AbsNode NegateNode NotNode LogicNegationNode NarrowNode SignExtendNode ZeroExtendNode

IsNullNode ReverseBytesNode BitCountNode
by *presburger+*
next
case (*UnrepBinaryNode g xe* g_1 *x ye* g_2 *y s′ op* g_3 *n*)
then have *k*: *kind* g_3 *n* = *bin-node op x y*
using *unique-kind bin-node-nonode* **by** *simp*
have *x*: $g_3 \vdash x \simeq xe$
using *UnrepBinaryNode unique-eval unrep-eval* **by** *blast*
have *y*: $g_3 \vdash y \simeq ye$
using *UnrepBinaryNode unique-eval unrep-eval* **by** *blast*
then show *?case*
using *x k* **apply** (*cases op*)
using *bin-node.simps*(*1*,*2*,*3*,*4*,*5*,*6*,*7*,*8*,*9*,*10*,*11*,*12*,*13*,*14*,*15*,*16*,*17*,*18*)
AddNode MulNode DivNode ModNode SubNode AndNode OrNode ShortCircuitOrNode LeftShiftNode RightShiftNode
UnsignedRightShiftNode IntegerEqualsNode IntegerLessThanNode IntegerBelowNode XorNode
IntegerTestNode IntegerNormalizeCompareNode IntegerMulHighNode
by *metis+*
next
case (*AllLeafNodes g n s*)
then show *?case*
by (*simp add*: *rep.LeafNode*)
qed
done

lemma *ref-refinement*:
assumes $g \vdash n \simeq e_1$
assumes *kind g n′* = *RefNode n*
shows $g \vdash n' \trianglelefteq e_1$
by (*meson equal-refines graph-represents-expression-def RefNode assms*)

lemma *unrep-refines*:
assumes $g \oplus e \leadsto (g', n)$
shows *graph-refinement g g′*
using *assms* **by** (*simp add*: *unrep-subset subset-refines*)

lemma *add-new-node-refines*:
assumes $n \notin$ *ids g*
assumes *g′* = *add-node n* (*k*, *s*) *g*
shows *graph-refinement g g′*
using *assms* **by** (*simp add*: *fresh-node-subset subset-refines*)

lemma *add-node-as-set*:
assumes *g′* = *add-node n* (*k*, *s*) *g*
shows $(\{n\} \trianglelefteq$ *as-set g*$) \subseteq$ *as-set g′*
unfolding *assms*
by (*smt* (*verit*, *ccfv-SIG*) *case-prodE changeonly.simps mem-Collect-eq prod.sel*(*1*) *subsetI assms*)

add-changed as-set-def domain-subtraction-def)

theorem *refined-insert*:
assumes $e_1 \geq e_2$
assumes $g_1 \oplus e_2 \rightsquigarrow (g_2, n')$
shows $(g_2 \vdash n' \trianglelefteq e_1) \wedge$ *graph-refinement* g_1 g_2
using *assms graph-construction term-graph-reconstruction* **by** *blast*

lemma *ids-finite*: *finite* (*ids g*)
by *simp*

lemma *unwrap-sorted*: *set* (*sorted-list-of-set* (*ids g*)) = *ids g*
using *ids-finite* **by** *simp*

lemma *find-none*:
assumes *find-node-and-stamp g* (*k*, *s*) = *None*
shows $\forall$ $n \in$ *ids g*. *kind g n* $\neq$ *k* $\vee$ *stamp g n* $\neq$ *s*
proof −
have ($\nexists n$. $n \in$ *ids g* $\wedge$ (*kind g n* = *k* $\wedge$ *stamp g n* = *s*))
by (*metis* (*mono-tags*) *unwrap-sorted find-None-iff find-node-and-stamp.simps assms*)
then show *?thesis*
by *auto*
qed

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 ¬(*is-preevaluated* (*kind g n*))
shows $\forall$ *e*. $(g \vdash n \simeq e) \longrightarrow (g \vdash n' \simeq e)$
apply (*rule allI*)
subgoal for *e*

apply (*rule impI*)
subgoal premises *eval* **using** *eval assms*
apply (*induction e*)
using *ConstantNode* **apply** *presburger*
using *ParameterNode* **apply** *presburger*
apply (*metis ConditionalNode*)
apply (*metis AbsNode*)
apply (*metis ReverseBytesNode*)
apply (*metis BitCountNode*)
apply (*metis NotNode*)
apply (*metis NegateNode*)
apply (*metis LogicNegationNode*)
apply (*metis AddNode*)
apply (*metis MulNode*)
apply (*metis DivNode*)
apply (*metis ModNode*)
apply (*metis SubNode*)
apply (*metis AndNode*)
apply (*metis OrNode*)
apply (*metis XorNode*)
apply (*metis ShortCircuitOrNode*)
apply (*metis LeftShiftNode*)
apply (*metis RightShiftNode*)
apply (*metis UnsignedRightShiftNode*)
apply (*metis IntegerBelowNode*)
apply (*metis IntegerEqualsNode*)
apply (*metis IntegerLessThanNode*)
apply (*metis IntegerTestNode*)
apply (*metis IntegerNormalizeCompareNode*)
apply (*metis IntegerMulHighNode*)
apply (*metis NarrowNode*)
apply (*metis SignExtendNode*)
apply (*metis ZeroExtendNode*)
defer
apply (*metis PiNode*)
apply (*metis RefNode*)
apply (*metis IsNullNode*)
by *blast*
done
done

lemma *new-node-not-present*:
assumes *find-node-and-stamp g* (*node*, *s*) = *None*
assumes n = *get-fresh-id g*
assumes g' = *add-node n* (*node*, *s*) *g*
shows $\forall\ n' \in$ *true-ids g*. $(\forall e.\ ((g \vdash n \simeq e) \wedge (g \vdash n' \simeq e)) \longrightarrow n = n')$
using *assms encode-in-ids fresh-ids* **by** *blast*

lemma *true-ids-def*:

true-ids g = $\{n \in$ *ids* g. $\neg$(*is-RefNode* (*kind* g n)) $\wedge$ ((*kind* g n) $\neq$ *NoNode*)$\}$
using *true-ids-def* **by** (*auto simp add*: *is-RefNode-def*)

lemma *add-node-some-node-def*:
assumes $k \neq$ *NoNode*
assumes g' = *add-node nid* (k, s) g
shows g' = *Abs-IRGraph* ((*Rep-IRGraph* g)(*nid* $\mapsto$ (k, s)))
by (*metis Rep-IRGraph-inverse add-node.rep-eq fst-conv assms*)

lemma *ids-add-update-v1*:
assumes g' = *add-node nid* (k, s) g
assumes $k \neq$ *NoNode*
shows *dom* (*Rep-IRGraph* g') = *dom* (*Rep-IRGraph* g) $\cup$ $\{$*nid*$\}$
by (*simp add*: *add-node.rep-eq assms*)

lemma *ids-add-update-v2*:
assumes g' = *add-node nid* (k, s) g
assumes $k \neq$ *NoNode*
shows *nid* $\in$ *ids* g'
by (*simp add*: *find-new-kind assms*)

lemma *add-node-ids-subset*:
assumes $n \in$ *ids* g
assumes g' = *add-node* n *node* g
shows *ids* g' = *ids* $g \cup \{n\}$
using *assms replace-node.rep-eq* **by** (*auto simp add*: *replace-node-def ids.rep-eq add-node-def*)

lemma *convert-maximal*:
assumes $\forall$ n n'. $n \in$ *true-ids* g $\wedge$ $n' \in$ *true-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
using *assms* **by** (*auto simp add*: *maximal-sharing*)

lemma *add-node-set-eq*:
assumes $k \neq$ *NoNode*
assumes $n \notin$ *ids* g
shows *as-set* (*add-node* n (k, s) g) = *as-set* $g \cup \{(n, (k, s))\}$
using *assms* **unfolding** *as-set-def* **by** (*transfer*; *auto*)

lemma *add-node-as-set-eq*:
assumes g' = *add-node* n (k, s) g
assumes $n \notin$ *ids* g
shows ($\{n\}$ $\trianglelefteq$ *as-set* g') = *as-set* g
unfolding *domain-subtraction-def*
by (*smt* (*z3*) *assms add-node-set-eq Collect-cong Rep-IRGraph-inverse UnCI add-node.rep-eq le-boolE*
as-set-def case-prodE2 case-prodI2 le-boolI' mem-Collect-eq prod.sel(*1*) *singletonD singletonI*

UnE)

lemma *true-ids*:
true-ids g = *ids g* − {*n* ∈ *ids g*. *is-RefNode* (*kind g n*)}
unfolding *true-ids-def* **by** *fastforce*

lemma *as-set-ids*:
assumes *as-set g* = *as-set g′*
shows *ids g* = *ids g′*
by (*metis antisym equalityD1 graph-refinement-def subset-refines assms*)

lemma *ids-add-update*:
assumes *k* ≠ *NoNode*
assumes *n* ∉ *ids g*
assumes *g′* = *add-node n* (*k*, *s*) *g*
shows *ids g′* = *ids g* ∪ {*n*}
by (*smt* (*z3*) *Diff-idemp Diff-insert-absorb Un-commute add-node.rep-eq insert-is-Un insert-Collect*
add-node-def ids.rep-eq ids-add-update-v1 insertE assms replace-node-unchanged Collect-cong
map-upd-Some-unfold mem-Collect-eq replace-node-def ids-add-update-v2)

lemma *true-ids-add-update*:
assumes *k* ≠ *NoNode*
assumes *n* ∉ *ids g*
assumes *g′* = *add-node n* (*k*, *s*) *g*
assumes ¬(*is-RefNode k*)
shows *true-ids g′* = *true-ids g* ∪ {*n*}
by (*smt* (*z3*) *Collect-cong Diff-iff Diff-insert-absorb Un-commute add-node-def find-new-kind assms*
insert-Diff-if insert-is-Un mem-Collect-eq replace-node-def replace-node-unchanged true-ids
ids-add-update)

lemma *new-def*:
assumes (*new* ⊴ *as-set g′*) = *as-set g*
shows *n* ∈ *ids g* ⟶ *n* ∉ *new*
using *assms* **apply** *auto* **unfolding** *as-set-def*
by (*smt* (*z3*) *as-set-def case-prodD domain-subtraction-def mem-Collect-eq assms ids-some*)

lemma *add-preserves-rep*:
assumes *unchanged*: (*new* ⊴ *as-set g′*) = *as-set g*
assumes *closed*: *wf-closed g*
assumes *existed*: *n* ∈ *ids g*
assumes *g′* ⊢ *n* ≃ *e*
shows *g* ⊢ *n* ≃ *e*
proof (*cases n* ∈ *new*)
case *True*

have *n* ∉ *ids g*
using *unchanged True as-set-def* **unfolding** *domain-subtraction-def* **by** *blast*
then show *?thesis*
using *existed* **by** *simp*
next
case *False*
have *kind-eq*: ∀ *n′* . *n′* ∉ *new* ⟶ *kind g n′* = *kind g′ n′*
— can be more general than *stamp_eq* because NoNode default is equal
apply (*rule allI*; *rule impI*)
by (*smt* (*z3*) *case-prodE domain-subtraction-def ids-some mem-Collect-eq subsetI unchanged*
not-excluded-keep-type)
from *False* **have** *stamp-eq*: ∀ *n′* ∈ *ids g′* . *n′* ∉ *new* ⟶ *stamp g n′* = *stamp g′ n′*
by (*metis equalityE not-excluded-keep-type unchanged*)
show *?thesis*
using *assms*(*4*) *kind-eq stamp-eq False*
proof (*induction n e rule*: *rep.induct*)
case (*ConstantNode n c*)
then show *?case*
by (*simp add*: *rep.ConstantNode*)
next
case (*ParameterNode n i s*)
then show *?case*
by (*metis no-encoding rep.ParameterNode*)
next
case (*ConditionalNode n c t f ce te fe*)
have *kind*: *kind g n* = *ConditionalNode c t f*
by (*simp add*: *kind-eq ConditionalNode.prems*(*3*) *ConditionalNode.hyps*(*1*))
then have *isin*: *n* ∈ *ids g*
by *simp*
have *inputs*: {*c*, *t*, *f*} = *inputs g n*
by (*simp add*: *kind*)
have *c* ∈ *ids g* ∧ *t* ∈ *ids g* ∧ *f* ∈ *ids g*
using *closed wf-closed-def isin inputs* **by** *blast*
then have *c* ∉ *new* ∧ *t* ∉ *new* ∧ *f* ∉ *new*
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *rep.ConditionalNode ConditionalNode*)
next
case (*AbsNode n x xe*)
then have *kind*: *kind g n* = *AbsNode x*
by *simp*
then have *isin*: *n* ∈ *ids g*
by *simp*
have *inputs*: {*x*} = *inputs g n*
by (*simp add*: *kind*)
have *x* ∈ *ids g*
using *closed wf-closed-def isin inputs* **by** *blast*

then have $x \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *AbsNode rep.AbsNode*)
next
case (*ReverseBytesNode n x xe*)
then have *kind*: *kind g n* = *ReverseBytesNode x*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x\} = inputs\ g\ n$
by (*simp add*: *kind*)
have $x \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
using *ReverseBytesNode.IH kind kind-eq rep.ReverseBytesNode stamp-eq* **by** *blast*
next
case (*BitCountNode n x xe*)
then have *kind*: *kind g n* = *BitCountNode x*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x\} = inputs\ g\ n$
by (*simp add*: *kind*)
have $x \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
using *BitCountNode.IH kind kind-eq rep.BitCountNode stamp-eq* **by** *blast*
next
case (*NotNode n x xe*)
then have *kind*: *kind g n* = *NotNode x*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x\} = inputs\ g\ n$
by (*simp add*: *kind*)
have $x \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *NotNode rep.NotNode*)
next
case (*NegateNode n x xe*)

then have *kind*: *kind g n* = *NegateNode x*
  by *simp*
then have *isin*: *n* ∈ *ids g*
  by *simp*
have *inputs*: {*x*} = *inputs g n*
  by (*simp add*: *kind*)
have *x* ∈ *ids g*
  using *closed wf-closed-def isin inputs* **by** *blast*
then have *x* ∉ *new*
  using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
  by (*simp add*: *NegateNode rep.NegateNode*)
next
case (*LogicNegationNode n x xe*)
then have *kind*: *kind g n* = *LogicNegationNode x*
  by *simp*
then have *isin*: *n* ∈ *ids g*
  by *simp*
have *inputs*: {*x*} = *inputs g n*
  by (*simp add*: *kind*)
have *x* ∈ *ids g*
  using *closed wf-closed-def isin inputs* **by** *blast*
then have *x* ∉ *new*
  using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
  by (*simp add*: *LogicNegationNode rep.LogicNegationNode*)
next
case (*AddNode n x y xe ye*)
then have *kind*: *kind g n* = *AddNode x y*
  by *simp*
then have *isin*: *n* ∈ *ids g*
  by *simp*
have *inputs*: {*x*, *y*} = *inputs g n*
  by (*simp add*: *kind*)
have *x* ∈ *ids g* ∧ *y* ∈ *ids g*
  using *closed wf-closed-def isin inputs* **by** *blast*
then have *x* ∉ *new* ∧ *y* ∉ *new*
  using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
  by (*simp add*: *AddNode rep.AddNode*)
next
case (*MulNode n x y xe ye*)
then have *kind*: *kind g n* = *MulNode x y*
  by *simp*
then have *isin*: *n* ∈ *ids g*
  by *simp*
have *inputs*: {*x*, *y*} = *inputs g n*
  by (*simp add*: *kind*)
have *x* ∈ *ids g* ∧ *y* ∈ *ids g*

```
    using closed wf-closed-def isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using unchanged by (simp add: new-def)
  then show ?case
    by (simp add: MulNode rep.MulNode)
next
  case (DivNode n x y xe ye)
  then have kind: kind g n = SignedFloatingIntegerDivNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    by (simp add: kind)
  have x ∈ ids g ∧ y ∈ ids g
    using closed wf-closed-def isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using unchanged by (simp add: new-def)
  then show ?case
    by (simp add: DivNode rep.DivNode)
next
  case (ModNode n x y xe ye)
  then have kind: kind g n = SignedFloatingIntegerRemNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    by (simp add: kind)
  have x ∈ ids g ∧ y ∈ ids g
    using closed wf-closed-def isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using unchanged by (simp add: new-def)
  then show ?case
    by (simp add: ModNode rep.ModNode)
next
  case (SubNode n x y xe ye)
  then have kind: kind g n = SubNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    by (simp add: kind)
  have x ∈ ids g ∧ y ∈ ids g
    using closed wf-closed-def isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using unchanged by (simp add: new-def)
  then show ?case
    by (simp add: SubNode rep.SubNode)
next
  case (AndNode n x y xe ye)
```

then have *kind*: *kind g n* = *AndNode x y*
  by *simp*
then have *isin*: $n \in ids\ g$
  by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
  by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$
  using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
  using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
  by (*simp add*: *AndNode rep.AndNode*)
next
case (*OrNode n x y xe ye*)
then have *kind*: *kind g n* = *OrNode x y*
  by *simp*
then have *isin*: $n \in ids\ g$
  by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
  by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$
  using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
  using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
  by (*simp add*: *OrNode rep.OrNode*)
next
case (*XorNode n x y xe ye*)
then have *kind*: *kind g n* = *XorNode x y*
  by *simp*
then have *isin*: $n \in ids\ g$
  by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
  by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$
  using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
  using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
  by (*simp add*: *XorNode rep.XorNode*)
next
case (*ShortCircuitOrNode n x y xe ye*)
then have *kind*: *kind g n* = *ShortCircuitOrNode x y*
  by *simp*
then have *isin*: $n \in ids\ g$
  by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
  by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$

using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *ShortCircuitOrNode rep.ShortCircuitOrNode*)
next
case (*LeftShiftNode n x y xe ye*)
then have *kind*: *kind g n* = *LeftShiftNode x y*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *LeftShiftNode rep.LeftShiftNode*)
next
case (*RightShiftNode n x y xe ye*)
then have *kind*: *kind g n* = *RightShiftNode x y*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *RightShiftNode rep.RightShiftNode*)
next
case (*UnsignedRightShiftNode n x y xe ye*)
then have *kind*: *kind g n* = *UnsignedRightShiftNode x y*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *UnsignedRightShiftNode rep.UnsignedRightShiftNode*)
next
case (*IntegerBelowNode n x y xe ye*)

  then have *kind*: *kind g n* = *IntegerBelowNode x y*
    by *simp*
  then have *isin*: $n \in ids\ g$
    by *simp*
  have *inputs*: $\{x, y\} = inputs\ g\ n$
    by (*simp add*: *kind*)
  have $x \in ids\ g \wedge y \in ids\ g$
    using *closed wf-closed-def isin inputs* **by** *blast*
  then have $x \notin new \wedge y \notin new$
    using *unchanged* **by** (*simp add*: *new-def*)
  then show *?case*
    by (*simp add*: *IntegerBelowNode rep.IntegerBelowNode*)
next
  case (*IntegerEqualsNode n x y xe ye*)
  then have *kind*: *kind g n* = *IntegerEqualsNode x y*
    by *simp*
  then have *isin*: $n \in ids\ g$
    by *simp*
  have *inputs*: $\{x, y\} = inputs\ g\ n$
    by (*simp add*: *kind*)
  have $x \in ids\ g \wedge y \in ids\ g$
    using *closed wf-closed-def isin inputs* **by** *blast*
  then have $x \notin new \wedge y \notin new$
    using *unchanged* **by** (*simp add*: *new-def*)
  then show *?case*
    by (*simp add*: *IntegerEqualsNode rep.IntegerEqualsNode*)
next
  case (*IntegerLessThanNode n x y xe ye*)
  then have *kind*: *kind g n* = *IntegerLessThanNode x y*
    by *simp*
  then have *isin*: $n \in ids\ g$
    by *simp*
  have *inputs*: $\{x, y\} = inputs\ g\ n$
    by (*simp add*: *kind*)
  have $x \in ids\ g \wedge y \in ids\ g$
    using *closed wf-closed-def isin inputs* **by** *blast*
  then have $x \notin new \wedge y \notin new$
    using *unchanged* **by** (*simp add*: *new-def*)
  then show *?case*
    by (*simp add*: *IntegerLessThanNode rep.IntegerLessThanNode*)
next
  case (*IntegerTestNode n x y xe ye*)
  then have *kind*: *kind g n* = *IntegerTestNode x y*
    by *simp*
  then have *isin*: $n \in ids\ g$
    by *simp*
  have *inputs*: $\{x, y\} = inputs\ g\ n$
    by (*simp add*: *kind*)
  have $x \in ids\ g \wedge y \in ids\ g$

using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *IntegerTestNode rep.IntegerTestNode*)
next
case (*IntegerNormalizeCompareNode n x y xe ye*)
then have *kind*: *kind g n* = *IntegerNormalizeCompareNode x y*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
using *IntegerNormalizeCompareNode.IH*(*1*,*2*) *kind kind-eq rep.IntegerNormalizeCompareNode stamp-eq* **by** *blast*
next
case (*IntegerMulHighNode n x y xe ye*)
then have *kind*: *kind g n* = *IntegerMulHighNode x y*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x, y\}$ = *inputs g n*
by (*simp add*: *kind*)
have $x \in ids\ g \wedge y \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new \wedge y \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
using *IntegerMulHighNode.IH*(*1*,*2*) *kind kind-eq rep.IntegerMulHighNode stamp-eq* **by** *blast*
next
case (*NarrowNode n inputBits resultBits x xe*)
then have *kind*: *kind g n* = *NarrowNode inputBits resultBits x*
by *simp*
then have *isin*: $n \in ids\ g$
by *simp*
have *inputs*: $\{x\}$ = *inputs g n*
by (*simp add*: *kind*)
have $x \in ids\ g$
using *closed wf-closed-def isin inputs* **by** *blast*
then have $x \notin new$
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *NarrowNode rep.NarrowNode*)

next
case (*SignExtendNode n inputBits resultBits x xe*)
then have *kind*: *kind g n* = *SignExtendNode inputBits resultBits x*
by *simp*
then have *isin*: *n* ∈ *ids g*
by *simp*
have *inputs*: {*x*} = *inputs g n*
by (*simp add*: *kind*)
have *x* ∈ *ids g*
using *closed wf-closed-def isin inputs* **by** *blast*
then have *x* ∉ *new*
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *SignExtendNode rep.SignExtendNode*)
next
case (*ZeroExtendNode n inputBits resultBits x xe*)
then have *kind*: *kind g n* = *ZeroExtendNode inputBits resultBits x*
by *simp*
then have *isin*: *n* ∈ *ids g*
by *simp*
have *inputs*: {*x*} = *inputs g n*
by (*simp add*: *kind*)
have *x* ∈ *ids g*
using *closed wf-closed-def isin inputs* **by** *blast*
then have *x* ∉ *new*
using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
by (*simp add*: *ZeroExtendNode rep.ZeroExtendNode*)
next
case (*LeafNode n s*)
then show *?case*
by (*metis no-encoding rep.LeafNode*)
next
case (*PiNode n n′ gu e*)
then have *kind*: *kind g n* = *PiNode n′ gu*
by *simp*
then have *isin*: *n* ∈ *ids g*
by *simp*
have *inputs*: *set* (*n′* # (*opt-to-list gu*)) = *inputs g n*
by (*simp add*: *kind*)
have *n′* ∈ *ids g*
by (*metis in-mono list.set-intros*(*1*) *inputs isin wf-closed-def closed*)
then show *?case*
using *PiNode.IH kind kind-eq new-def rep.PiNode stamp-eq unchanged* **by** *blast*
next
case (*RefNode n n′ e*)
then have *kind*: *kind g n* = *RefNode n′*
by *simp*

then have *isin*: $n \in$ *ids g*
  by *simp*
have *inputs*: $\{n'\}$ = *inputs g n*
  by (*simp add*: *kind*)
have $n' \in$ *ids g*
  using *closed wf-closed-def isin inputs* **by** *blast*
then have $n' \notin$ *new*
  using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
  by (*simp add*: *RefNode rep.RefNode*)
next
case (*IsNullNode n v*)
then have *kind*: *kind g n* = *IsNullNode v*
  by *simp*
then have *isin*: $n \in$ *ids g*
  by *simp*
have *inputs*: $\{v\}$ = *inputs g n*
  by (*simp add*: *kind*)
have $v \in$ *ids g*
  using *closed wf-closed-def isin inputs* **by** *blast*
then have $v \notin$ *new*
  using *unchanged* **by** (*simp add*: *new-def*)
then show *?case*
  by (*simp add*: *rep.IsNullNode stamp-eq kind-eq kind IsNullNode.IH*)
qed
qed

lemma *not-in-no-rep*:
$n \notin$ *ids g* $\Longrightarrow \forall e. \neg(g \vdash n \simeq e)$
using *eval-contains-id* **by** *auto*

lemma *unary-inputs*:
assumes *kind g n* = *unary-node op x*
shows *inputs g n* = $\{x\}$
by (*cases op*; *auto simp add*: *assms*)

lemma *unary-succ*:
assumes *kind g n* = *unary-node op x*
shows *succ g n* = $\{\}$
by (*cases op*; *auto simp add*: *assms*)

lemma *binary-inputs*:
assumes *kind g n* = *bin-node op x y*
shows *inputs g n* = $\{x, y\}$
by (*cases op*; *auto simp add*: *assms*)

lemma *binary-succ*:

assumes *kind g n* = *bin-node op x y*
shows *succ g n* = {}
by (*cases op*; *auto simp add*: *assms*)

lemma *unrep-contains*:
assumes $g \oplus e \rightsquigarrow (g', n)$
shows $n \in ids\ g'$
using *assms not-in-no-rep term-graph-reconstruction* **by** *blast*

lemma *unrep-preserves-contains*:
assumes $n \in ids\ g$
assumes $g \oplus e \rightsquigarrow (g', n')$
shows $n \in ids\ g'$
by (*meson subsetD unrep-ids-subset assms*)

lemma *unique-preserves-closure*:
assumes *wf-closed g*
assumes *unique g* (*node*, *s*) (g', *n*)
assumes *set* (*inputs-of node*) ⊆ *ids g* ∧
set (*successors-of node*) ⊆ *ids g* ∧
node ≠ *NoNode*
shows *wf-closed* g'
using *assms*
by (*smt* (*verit*, *del-insts*) *Pair-inject UnE add-changed fresh-ids graph-refinement-def ids-add-update inputs.simps other-node-unchanged singletonD subset-refines subset-trans succ.simps unique.cases unique-kind unique-subset wf-closed-def*)

lemma *unrep-preserves-closure*:
assumes *wf-closed g*
assumes $g \oplus e \rightsquigarrow (g', n)$
shows *wf-closed* g'
using *assms*(*2*,*1*) *wf-closed-def*
proof (*induction g e* (g', *n*) *arbitrary*: g' *n*)
next
case (*UnrepConstantNode g c* g' *n*)
then show *?case* **using** *unique-preserves-closure*
by (*metis IRNode.distinct*(*1077*) *IRNodes.inputs-of-ConstantNode IRNodes.successors-of-ConstantNode empty-subsetI list.set*(*1*))
next
case (*UnrepParameterNode g i s n*)
then show *?case* **using** *unique-preserves-closure*
by (*metis IRNode.distinct*(*3695*) *IRNodes.inputs-of-ParameterNode IRNodes.successors-of-ParameterNode empty-subsetI list.set*(*1*))
next
case (*UnrepConditionalNode g ce* g_1 *c te* g_2 *t fe* g_3 *f* s' g_4 *n*)
then have *c*: *wf-closed* g_3
by *fastforce*
have *k*: *kind* g_4 *n* = *ConditionalNode c t f*

using *UnrepConditionalNode IRNode.distinct*(*965*) *unique-kind* **by** *presburger*
have $\{c,\ t,\ f\} \subseteq$ *ids* g_4 **using** *unrep-contains*
by (*metis UnrepConditionalNode.hyps*(*1*) *UnrepConditionalNode.hyps*(*3*) *UnrepConditionalNode.hyps*(*5*) *UnrepConditionalNode.hyps*(*8*) *empty-subsetI graph-refinement-def insert-subsetI subset-iff subset-refines unique-subset unrep-ids-subset*)
also have *inputs* g_4 $n = \{c,\ t,\ f\} \wedge$ *succ* g_4 $n = \{\}$
using *k* **by** *simp*
moreover have *inputs* g_4 $n \subseteq$ *ids* g_4 $\wedge$ *succ* g_4 $n \subseteq$ *ids* g_4 $\wedge$ *kind* g_4 $n \neq$ *NoNode*
using *k*
by (*metis IRNode.distinct*(*965*) *calculation empty-subsetI*)
ultimately show *?case* **using** *c unique-preserves-closure UnrepConditionalNode*
by (*metis empty-subsetI inputs.simps insert-subsetI k succ.simps unrep-contains unrep-preserves-contains*)
next
case (*UnrepUnaryNode g xe* g_1 *x* s' *op* g_2 *n*)
then have *c*: *wf-closed* g_1
by *fastforce*
have *k*: *kind* g_2 $n =$ *unary-node op x*
using *UnrepUnaryNode unique-kind unary-node-nonode* **by** *blast*
have $\{x\} \subseteq$ *ids* g_2 **using** *unrep-contains*
by (*metis UnrepUnaryNode.hyps*(*1*) *UnrepUnaryNode.hyps*(*4*) *encodes-contains ids-some singletonD subsetI term-graph-reconstruction unique-eval*)
also have *inputs* g_2 $n = \{x\} \wedge$ *succ* g_2 $n = \{\}$
using *k*
by (*meson unary-inputs unary-succ*)
moreover have *inputs* g_2 $n \subseteq$ *ids* g_2 $\wedge$ *succ* g_2 $n \subseteq$ *ids* g_2 $\wedge$ *kind* g_2 $n \neq$ *NoNode*
using *k*
by (*metis calculation*(*1*) *calculation*(*2*) *empty-subsetI unary-node-nonode*)
ultimately show *?case* **using** *c unique-preserves-closure UnrepUnaryNode*
by (*metis empty-subsetI inputs.simps insert-subsetI k succ.simps unrep-contains*)
next
case (*UnrepBinaryNode g xe* g_1 *x ye* g_2 *y* s' *op* g_3 *n*)
then have *c*: *wf-closed* g_2
by *fastforce*
have *k*: *kind* g_3 $n =$ *bin-node op x y*
using *UnrepBinaryNode unique-kind bin-node-nonode* **by** *blast*
have $\{x,\ y\} \subseteq$ *ids* g_3 **using** *unrep-contains*
by (*metis UnrepBinaryNode.hyps*(*1*) *UnrepBinaryNode.hyps*(*3*) *UnrepBinaryNode.hyps*(*6*) *empty-subsetI graph-refinement-def insert-absorb insert-subset subset-refines unique-subset unrep-refines*)
also have *inputs* g_3 $n = \{x,\ y\} \wedge$ *succ* g_3 $n = \{\}$
using *k*
by (*meson binary-inputs binary-succ*)
moreover have *inputs* g_3 $n \subseteq$ *ids* g_3 $\wedge$ *succ* g_3 $n \subseteq$ *ids* g_3 $\wedge$ *kind* g_3 $n \neq$ *NoNode*
using *k*
by (*metis calculation*(*1*) *calculation*(*2*) *empty-subsetI bin-node-nonode*)

```
    ultimately show ?case using c unique-preserves-closure UnrepBinaryNode
     by (metis empty-subsetI inputs.simps insert-subsetI k succ.simps unrep-contains
unrep-preserves-contains)
  next
    case (AllLeafNodes g n s)
    then show ?case
      by simp
  qed

inductive-cases ConstUnrepE: g ⊕ (ConstantExpr x) ↝ (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′*
apply (*induction rule*: *indexof.induct*)
by (*simp add*: *indexof.simps*)

code-pred (*modes*: *i* ⇒ *o* ⇒ *i* ⇒ *bool*) *indexof* **.**

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*
unfolding *fun-add-def* **by** *simp*

lemma *upds-inc*:
m(*a*#*as* [→] *b*#*bs*) = (*m*(*a*:=*b*))(*as*[→]*bs*)
unfolding *upds-def fun-add-def* **apply** *simp* **sorry**

lemma *upds-compose*:
a $++_f$ *map-of* (*rev* (*zip* (*n* # *ns*) (*v* # *vs*))) = *a*(*n* := *v*) $++_f$ *map-of* (*rev* (*zip ns vs*))
using *upds-inc*
by (*metis upds-def*)

lemma *set-phis ns vs* = (λ*m*. *upds m ns vs*)
proof (*induction rule*: *set-phis.induct*)
case (*1 m*)
then show *?case* **unfolding** *set-phis.simps upds-def*
by (*metis Nil-eq-zip-iff Nil-is-rev-conv fun-add-empty*)
next
case (*2 n xs v vs m*)
then show *?case* **unfolding** *set-phis.simps upds-def*
by (*metis upds-compose*)
next
case (*3 v vs m*)
then show *?case*
by (*metis fun-add-empty rev.simps*(*1*) *upds-def set-phis.simps*(*3*) *zip-Nil*)
next
case (*4 x xs m*)
then show *?case*
by (*metis Nil-eq-zip-iff fun-add-empty rev.simps*(*1*) *upds-def set-phis.simps*(*4*))
qed

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-of* (*kind g nid*))!*0*⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m*, *h*) |

FixedGuardNode:
⟦(*kind g nid*) = (*FixedGuardNode cond before next*);
[*g*, *m*, *p*] ⊢ *cond* ↦ *val*;

¬(*val-to-bool val*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*next*, *m*, *h*) |

BytecodeExceptionNode:
⟦(*kind g nid*) = (*BytecodeExceptionNode args st nid′*);
exceptionType = *stp-type* (*stamp g nid*);
(*h′*, *ref*) = *h-new-inst h exceptionType*;
m′ = *m*(*nid* := *ref*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m′*, *h′*) |

IfNode:
⟦*kind g nid* = (*IfNode cond tb fb*);
[*g*, *m*, *p*] ⊢ *cond* ↦ *val*;
nid′ = (*if val-to-bool val then tb else fb*)⟧
⟹ *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m*, *h*) |

EndNodes:
⟦*is-AbstractEndNode* (*kind g nid*);
merge = *any-usage g nid*;
is-AbstractMergeNode (*kind g merge*);

indexof (*inputs-of* (*kind g merge*)) *i nid*;
phis = *filter-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:
$[\![kind\ g\ nid = (ArrayLengthNode\ x\ nid');$
$[g, m, p] \vdash x \mapsto ObjRef\ ref;$

$h\text{-}load\text{-}field\ ''''\ ref\ h = arrayVal;$
$length' = array\text{-}length\ (arrayVal);$

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

LoadIndexedNode:
$[\![kind\ g\ nid = (LoadIndexedNode\ index\ guard\ array\ nid');$
$[g, m, p] \vdash index \mapsto indexVal;$
$[g, m, p] \vdash array \mapsto ObjRef\ ref;$

$h\text{-}load\text{-}field\ ''''\ ref\ h = arrayVal;$
$loaded = intval\text{-}load\text{-}index\ arrayVal\ indexVal;$

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

StoreIndexedNode:
$[\![kind\ g\ nid = (StoreIndexedNode\ check\ val\ st\ index\ guard\ array\ nid');$
$[g, m, p] \vdash index \mapsto indexVal;$
$[g, m, p] \vdash array \mapsto ObjRef\ ref;$
$[g, m, p] \vdash val \mapsto value;$

$h\text{-}load\text{-}field\ ''''\ ref\ h = arrayVal;$
$updated = intval\text{-}store\text{-}index\ arrayVal\ indexVal\ value;$
$h' = h\text{-}store\text{-}field\ ''''\ ref\ updated\ h;$
$m' = m(nid := updated)]\!]$

$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h')$ |

NewInstanceNode:
$[\![kind\ g\ nid = (NewInstanceNode\ nid\ cname\ obj\ nid');$
$(h',\ ref) = h\text{-}new\text{-}inst\ h\ cname;$
$m' = m(nid := ref)]\!]$
$\Longrightarrow g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m',\ h')$ |

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

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

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* **.**

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))
       )
     )

  by auto
termination dynamic-lookup apply (relation measure (λ(S,cn,mn,path). (length
path))) by auto

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;
$[g,\ m,\ p] \vdash$ *arguments* $[\mapsto]$ 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*))$]\!]$
$\Longrightarrow (S) \vdash ((g,nid,m,p)\#stk,\ h) \longrightarrow ((targetGraph,0,new\text{-}map\text{-}state,p')\#(g,nid,m,p)\#stk,\ h)$ |

ReturnNode:
$[\![$*kind g nid* = (*ReturnNode* (*Some expr*) -);
$[g,\ m,\ p] \vdash expr \mapsto v$;

$m'_c = m_c(nid_c := v)$;
nid'_c = (*successors-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* -);

nid'_c = (*successors-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* - - - - - - *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* **.**

8.4 Big-step Execution

type-synonym *Trace* = (*IRGraph* × *ID* × *MapState* × *Params*) *list*

fun *has-return* :: *MapState* ⇒ *bool* **where**
has-return m = ($m\ 0 \neq$ *UndefVal*)

```
inductive exec :: System
      ⇒ (IRGraph × ID × MapState × Params) list × FieldRefHeap
      ⇒ Trace
      ⇒ (IRGraph × ID × MapState × Params) list × FieldRefHeap
      ⇒ Trace
      ⇒ bool
  (- ⊢ - | - ⟶* - | -)
  for P where
  ⟦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')#ys),h') l' next-state l''⟧
    ⟹ exec P (((g,nid,m,p)#xs),h) l next-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 .

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 .
```

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 (*induction arbitrary*: *next' rule*: *step.induct*)
 case (*SequentialNode nid nid' m h*)
 have *notend*: ¬(*is-AbstractEndNode* (*kind g nid*))
  by (*metis SequentialNode.hyps*(*1*) *is-AbstractEndNode.simps is-EndNode.elims*(*2*)
is-LoopEndNode-def is-sequential-node.simps(*18*) *is-sequential-node.simps*(*36*))
 from *SequentialNode* **show** *?case* **apply** (*elim StepE*) **using** *is-sequential-node.simps*
     apply *blast*
     apply *force* **apply** *force* **apply** *force*
  using *notend*
  apply (*metis* (*no-types*, *lifting*) *Pair-inject is-AbstractEndNode.simps*)
  by *force+*
next
 case (*FixedGuardNode nid cond before next m val nid' h*)
 then show *?case* **apply** (*elim StepE*)
  by *force+*
next
 case (*BytecodeExceptionNode nid args st nid' exceptionType h' ref h m' m*)
 then show *?case* **apply** (*elim StepE*)
  by *force+*
next
 case (*IfNode nid cond tb fb m val nid' h*)
 then show *?case* **apply** (*elim StepE*)
  apply *force+*
  — IfNode rule uses expression evaluation
  using *graphDet* **apply** *fastforce*
  by *force+*
next
 case (*EndNodes nid merge i phis inps m vs m' h*)
 have *notseq*: ¬(*is-sequential-node* (*kind g nid*))
  using *EndNodes*
  by (*metis is-AbstractEndNode.simps is-EndNode.elims*(*2*) *is-LoopEndNode-def*
is-sequential-node.simps(*18*) *is-sequential-node.simps*(*36*))
 from *EndNodes* **show** *?case* **apply** (*elim StepE*)
  using *notseq* **apply** *force*
     apply *force* **apply** *force* **apply** *force*
  using *indexof-det*
  unfolding *is-AbstractEndNode.simps*
  is-AbstractMergeNode.simps any-usage.simps usages.simps inputs.simps ids-def
     apply (*smt* (*verit*, *del-insts*) *Collect-cong encodeEvalAllDet ids-def*

```
ids-some old.prod.inject)
    by force+
next
  case (NewArrayNode nid len st nid' m length' arrayType h' ref h refNo h'' m')
  then show ?case apply (elim StepE) apply force+
  — NewArrayNode rule uses expression evaluation
  using graphDet apply fastforce
  by force+
next
  case (ArrayLengthNode nid x nid' m ref h arrayVal length' m')
  then show ?case apply (elim StepE) apply force+
  — ArrayLengthNode rule uses expression evaluation
  using graphDet apply fastforce
  by force+
next
  case (LoadIndexedNode nid index guard array nid'  m indexVal ref h arrayVal
loaded m')
  then show ?case apply (elim StepE) apply force+
  — LoadIndexedNode rule uses expression evaluation
  using graphDet
  apply (metis IRNode.inject(28) Pair-inject Value.inject(2))
  by force+
next
  case (StoreIndexedNode nid check val st index guard array nid' m indexVal ref
value h arrayVal updated h' m')
  then show ?case apply (elim StepE) apply force+
  — StoreIndexedNode rule uses expression evaluation
    using graphDet
    apply (metis IRNode.inject(55) Pair-inject Value.inject(2))
  by force+
next
  case (NewInstanceNode nid cname obj nid' h' ref h m' m)
  then show ?case apply (elim StepE) by force+
next
  case (LoadFieldNode nid f obj nid' m ref h v m')
  then show ?case apply (elim StepE) apply force+
  — LoadFieldNode rule uses expression evaluation
    using graphDet apply fastforce
  by force+
next
  case (SignedDivNode nid x y zero sb nxt m v1 v2 v m' h)
  then show ?case apply (elim StepE) apply force+
  — SignedDivNode rule uses expression evaluation
    using graphDet
    apply (metis IRNode.inject(49) Pair-inject)
  by force+
next
  case (SignedRemNode nid x y zero sb nxt m v1 v2 v m' h)
  then show ?case apply (elim StepE) apply force+
```

— SignedRemNode rule uses expression evaluation
using *graphDet*
apply (*metis IRNode.inject*(*52*) *Pair-inject*)
by *force+*
next
case (*StaticLoadFieldNode nid f nid′ h v m′ m*)
then show *?case* **apply** (*elim StepE*) **by** *force+*
next
case (*StoreFieldNode nid f newval uu obj nid′ m val ref h′ h m′*)
then show *?case* **apply** (*elim StepE*) **apply** *force+*
— StoreFieldNode rule uses expression evaluation
using *graphDet*
apply (*metis IRNode.inject*(*54*) *Pair-inject Value.inject*(*2*) *option.inject*)
by *force+*
next
case (*StaticStoreFieldNode nid f newval uv nid′ m val h′ h m′*)
then show *?case* **apply** (*elim StepE*) **apply** *force+*
— StaticStoreFieldNode rule uses expression evaluation
using *graphDet* **by** *fastforce*
qed

theorem *stepDet*:
$(g,\ p \vdash (nid,m,h) \rightarrow next) \Longrightarrow$
$(\forall\ next'.\ ((g,\ p \vdash (nid,m,h) \rightarrow next') \longrightarrow next = next'))$
using *stepDet′* **by** *simp*

lemma *stepRefNode*:
$[\![kind\ g\ nid = RefNode\ nid']\!] \Longrightarrow g,\ p \vdash (nid,m,h) \rightarrow (nid',m,h)$
by (*metis IRNodes.successors-of-RefNode is-sequential-node.simps*(*7*) *nth-Cons-0 SequentialNode*)

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 $nid' \in \{tb,\ fb\}$
by (*metis insert-iff old.prod.inject step.IfNode stepDet assms encodeeval.simps*)

lemma *IfNodeSeq*:
shows $kind\ g\ nid = IfNode\ cond\ tb\ fb \longrightarrow \neg(is\text{-}sequential\text{-}node\ (kind\ g\ nid))$
using *is-sequential-node.simps*(*18*,*19*) **by** *simp*

lemma *IfNodeCond*:
assumes $kind\ g\ nid = IfNode\ cond\ tb\ fb$
assumes $g,\ p \vdash (nid,\ m,\ h) \rightarrow (nid',\ m,\ h)$
shows $\exists\ condE\ v.\ ((g \vdash cond \simeq condE) \land ([m,\ p] \vdash condE \mapsto v))$
using *assms*(*2*,*1*) *encodeeval.simps* **by** (*induct* (*nid*,*m*,*h*) (*nid′*,*m*,*h*) *rule*: *step.induct*; *auto*)

lemma *step-in-ids*:
assumes g, $p \vdash (nid, m, h) \rightarrow (nid', m', h')$
shows $nid \in ids\ g$
using *assms* **apply** (*induct* (*nid*, *m*, *h*) (*nid'*, *m'*, *h'*) *rule*: *step.induct*) **apply** *fastforce*
prefer *4* **prefer** *14* **defer defer**
using *IRNode.distinct*(*1607*) *ids-some* **apply** *presburger*
using *IRNode.distinct*(*851*) *ids-some* **apply** *presburger*

using *IRNode.distinct*(*1805*) *ids-some* **apply** *presburger*
apply (*metis IRNode.distinct*(*3507*) *not-in-g*)
apply (*metis IRNode.distinct*(*497*) *not-in-g*)
apply (*metis IRNode.distinct*(*2897*) *not-in-g*)

apply (*metis IRNode.distinct*(*4085*) *not-in-g*)
using *IRNode.distinct*(*3557*) *ids-some* **apply** *presburger*
apply (*metis IRNode.distinct*(*2825*) *not-in-g*)
apply (*metis IRNode.distinct*(*3947*) *not-in-g*)
apply (*metis IRNode.distinct*(*4025*) *not-in-g*)
using *IRNode.distinct*(*2825*) *ids-some* **apply** *presburger*
apply (*metis IRNode.distinct*(*4067*) *not-in-g*)
apply (*metis IRNode.distinct*(*4067*) *not-in-g*)
using *IRNode.disc*(*1952*) *is-EndNode.simps*(*62*) *is-AbstractEndNode.simps not-in-g*
by (*metis IRNode.disc*(*2014*) *is-EndNode.simps*(*64*))

end

9 Proof Infrastructure

9.1 Bisimulation

theory *Bisimulation*
imports
Stuttering
begin

inductive *weak-bisimilar* :: *ID* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *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* ⇒ *IRGraph* ⇒ *IRGraph* ⇒ *bool*
(- | - ∼ -) **for** *nid* **where**

$\llbracket\forall P'. (g, p \vdash (nid, m, h) \to P') \longrightarrow (\exists Q' . (g', p \vdash (nid, m, h) \to Q') \wedge P' = Q');$
$\forall Q'. (g', p \vdash (nid, m, h) \to Q') \longrightarrow (\exists P' . (g, p \vdash (nid, m, h) \to P') \wedge P' = Q')\rrbracket$
$\Longrightarrow nid \mid g \sim g'$

lemma *lockstep-strong-bisimilulation*:
assumes $g' = \textit{replace-node nid node } g$
assumes $g, p \vdash (nid, m, h) \to (nid', m, h)$
assumes $g', p \vdash (nid, m, h) \to (nid', m, h)$
shows $nid \mid g \sim g'$
by (*metis strong-noop-bisimilar.simps stepDet assms*(*2*,*3*))

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 $nid \mid g \sim g'$
by (*simp add*: *assms*(*1*,*2*) *strong-noop-bisimilar.intros*)

end

9.2 Graph Rewriting

theory
Rewrites
imports
Stuttering
begin

fun *replace-usages* :: $ID \Rightarrow ID \Rightarrow IRGraph \Rightarrow IRGraph$ **where**
replace-usages nid nid′ g = *replace-node nid* (*RefNode nid′*, *stamp g nid′*) *g*

lemma *replace-usages-effect*:
assumes *g′* = *replace-usages nid nid′ g*
shows *kind g′ nid* = *RefNode nid′*
using *replace-usages.simps replace-node-lookup assms* **by** *blast*

lemma *replace-usages-changeonly*:
assumes $nid \in ids\ g$
assumes *g′* = *replace-usages nid nid′ g*
shows *changeonly* {*nid*} *g g′*
by (*metis add-changed add-node-def replace-node-def replace-usages.simps assms*(*2*))

lemma *replace-usages-unchanged*:
assumes $nid \in ids\ g$
assumes *g′* = *replace-usages nid nid′ g*
shows *unchanged* (*ids g* − {*nid*}) *g g′*
using *assms disjoint-change replace-usages-changeonly* **by** *presburger*

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*
by (*meson Max-gr-iff less-add-one less-irrefl*)

lemma *ids-finite*:
finite (*ids g*)
by *simp*

lemma *nextNidNotIn*:
ids g ≠ {} ⟶ *nextNid g* ∉ *ids g*
unfolding *nextNid.simps* **using** *ids-finite max-plus-one* **by** *blast*

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 $g'_1 = g'_2 \wedge nid_1 = nid_2$
using *assms* **apply** (*induction*)

apply (*metis Pair-inject assms*(*1*) *assms*(*2*) *option.distinct*(*1*) *option.inject unique.cases*)
by (*metis Pair-inject assms*(*1*) *assms*(*2*) *option.discI option.inject unique.cases*)

lemma *unrepDet*:
assumes *unrep g e* (g'_1, nid_1)
assumes *unrep g e* (g'_2, nid_2)
shows $g'_1 = g'_2 \wedge nid_1 = nid_2$
using *assms* **proof** (*induction e arbitrary*: *g* g'_1 nid_1 g'_2 nid_2)
case (*UnaryExpr op e*)
then show *?case*
by (*smt* (*verit*, *best*) *uniqueDet unrepUnaryE*)
next
case (*BinaryExpr x1 e1 e2*)
then show *?case*
by (*smt* (*verit*, *best*) *uniqueDet unrepBinaryE*)
next
case (*ConditionalExpr e1 e2 e3*)
then show *?case*
by (*smt* (*verit*, *best*) *uniqueDet unrepConditionalE*)
next
case (*ParameterExpr x1 x2*)
then show *?case*
by (*smt* (*verit*, *best*) *uniqueDet unrepParamE*)
next
case (*LeafExpr x1 x2*)
then show *?case*
by (*smt* (*verit*, *best*) *uniqueDet unrepLeafE*)
next
case (*ConstantExpr x*)
then show *?case*
by (*smt* (*verit*, *best*) *uniqueDet unrepConstE*)
next
case (*ConstantVar x*)
then show *?case*
by (*smt* (*verit*, *best*) *uniqueDet unrepConstVarE*)
next
case (*VariableExpr x1 x2*)
then show *?case*
by (*smt* (*verit*, *best*) *uniqueDet unrepVariableE*)
qed

lemma *unwrapUnrepE*:
assumes *unrep g e* (g', nid')
shows (g', nid') = *Predicate.the* (*unrepE g e*)
using *assms unrepEI unrepDet* **unfolding** *Predicate.the-def*
by (*metis eval-usages.cases pred.sel the-equality unrepE-def*)

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′*
  using *assms* **unfolding** *constantCondition.simps*
  using *unwrapUnrepE* **by** *auto*

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′*)
  by (*meson not-None-eq unique.simps*)

lemma *insertConst*:
  ∃ *g′ nid′*. *unrep g* (*ConstantExpr c*) (*g′*, *nid′*)
  using *UnrepConstantNode insertConstUnique* **by** *blast*

lemma *constantConditionTrue*:
  assumes *kind g ifcond* = *IfNode cond t f*
  assumes *g′* = *constantCondition True ifcond* (*kind g ifcond*) *g*
  shows *g′*, *p* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)
proof −
  have *ifn*: ⋀ *c t f*. *IfNode c t f* ≠ *NoNode*
    by *simp*
  obtain *g″ nid′* **where** *unrep*: *unrep g* (*ConstantExpr* (*bool-to-val-width1 True*))
(*g″*, *nid′*)
    using *insertConst* **by** *blast*
  then have *kind g″ nid′* = *ConstantNode* (*bool-to-val-width1 True*)
    by (*meson ConstUnrepE IRNode.distinct*(*1077*) *unique-kind*)
  also have *nid′* ≠ *ifcond*
     by (*metis ConstUnrepE IRNode.distinct*(*981*) *assms*(*1*) *calculation fresh-ids ids-some ifn unique.cases unrep unrepDet*)
  moreover have *g′* = *replace-node ifcond* (*IfNode nid′ t f*, *stamp g ifcond*) *g″*
    using *assms constantCondition-sem unrep* **by** *presburger*
  moreover have *kind g′ nid′* = *ConstantNode* (*bool-to-val-width1 True*)
    using *assms constantCondition.simps*(*1*) *replace-node-unchanged*
     by (*metis DiffI calculation*(*1*) *calculation*(*2*) *calculation*(*3*) *emptyE insert-iff unrep unrep-contains*)
  moreover have *if′*: *kind g′ ifcond* = *IfNode nid′ t f*
    using *ifn assms constantCondition.simps*(*1*) *replace-node-lookup*
    using *calculation*(*3*) **by** *blast*
  have *truedef*: *bool-to-val True* = (*IntVal 32 1*)
    by *auto*

from *ifn* **have** *ifcond* ≠ (*nextNid g*)
by (*metis assms*(*1*) *emptyE ids-some nextNidNotIn*)
moreover have ⋀ *c. ConstantNode c* ≠ *NoNode*
by *simp*
ultimately have *kind g′ nid′* = *ConstantNode* (*bool-to-val-width1 True*)
using *add-changed*
by *fastforce*
then have *c′*: *kind g′ nid′* = *ConstantNode* (*IntVal 1 1*)
by *simp*
have *valid-value* (*IntVal 1 1*) (*constantAsStamp* (*IntVal 1 1*))
by *fastforce*
then have [*g′*, *m*, *p*] ⊢ *nid′* ↦ *IntVal 1 1*
using *Value.distinct*(*1*) ‹*kind g′ nid′* = *ConstantNode* (*bool-to-val-width1 True*)›
by (*metis bool-to-val-width1.simps*(*1*) *wf-value-def encodeeval.simps ConstantExpr ConstantNode*)
from *if′ c′* **show** *?thesis*
by (*metis* (*no-types*, *opaque-lifting*) *val-to-bool.simps*(*1*) ‹[*g′*,*m*,*p*] ⊢ *nid′* ↦ *IntVal 1 1*›
zero-neq-one IfNode)
qed

lemma *constantConditionFalse*:
assumes *kind g ifcond* = *IfNode cond t f*
assumes *g′* = *constantCondition False ifcond* (*kind g ifcond*) *g*
shows *g′*, *p* ⊢ (*ifcond*, *m*, *h*) → (*f*, *m*, *h*)
proof −
have *ifn*: ⋀ *c t f*. *IfNode c t f* ≠ *NoNode*
by *simp*
obtain *g′′ nid′* **where** *unrep*: *unrep g* (*ConstantExpr* (*bool-to-val-width1 False*)) (*g′′*, *nid′*)
using *insertConst* **by** *blast*
also have *kind g′′ nid′* = *ConstantNode* (*bool-to-val-width1 False*)
by (*meson ConstUnrepE IRNode.distinct*(*1077*) *unique-kind unrep*)
moreover have *nid′* ≠ *ifcond*
by (*metis ConstUnrepE IRNode.distinct*(*981*) *assms*(*1*) *calculation*(*2*) *fresh-ids ids-some ifn unique.cases unrep unrepDet*)
moreover have *g′* = *replace-node ifcond* (*IfNode nid′ t f*, *stamp g ifcond*) *g′′*
using *assms*(*1*) *assms*(*2*) *constantCondition-sem unrep* **by** *presburger*
moreover have *kind g′ nid′* = *ConstantNode* (*bool-to-val-width1 False*)
using *assms constantCondition.simps*(*1*) *replace-node-unchanged*
by (*metis DiffI calculation*(*2*) *calculation*(*3*) *calculation*(*4*) *emptyE insert-iff unrep unrep-contains*)
moreover have *if′*: *kind g′ ifcond* = *IfNode nid′ t f*
using *ifn assms constantCondition.simps*(*1*) *replace-node-lookup*
using *calculation*(*4*) **by** *blast*
have *falsedef*: *bool-to-val False* = (*IntVal 32 0*)
by *auto*
then have *c′*: *kind g′ nid′* = *ConstantNode* (*IntVal 1 0*)
by (*simp add*: *calculation*(*5*))

have *valid-value* (*IntVal 1 0*) (*constantAsStamp* (*IntVal 1 0*))
by *auto*
then have [*g′*, *m*, *p*] ⊢ *nid′* ↦ *IntVal 1 0*
by (*meson ConstantExpr ConstantNode c′ encodeeval.simps wf-value-def*)
from *if′ c′* **show** *?thesis*
by (*meson IfNode* ‹[*g′*::*IRGraph*,*m*::*nat* ⇒ *Value*,*p*::*Value list*] ⊢ *nid′*::*nat* ↦ *IntVal* (*1*::*nat*) (*0*::*64 word*)› *encodeeval.simps val-to-bool.simps*(*1*))
qed

lemma *diff-forall*:
assumes ∀ *n*∈*ids g* − {*nid*}. *cond n*
shows ∀ *n*. *n* ∈ *ids g* ∧ *n* ∉ {*nid*} ⟶ *cond n*
by (*meson Diff-iff assms*)

lemma *replace-node-changeonly*:
assumes *g′* = *replace-node nid node g*
shows *changeonly* {*nid*} *g g′*
by (*metis add-changed add-node-def replace-node-def assms*)

lemma *add-node-changeonly*:
assumes *g′* = *add-node nid node g*
shows *changeonly* {*nid*} *g g′*
by (*metis Rep-IRGraph-inverse add-node.rep-eq assms replace-node.rep-eq replace-node-changeonly*)

lemma *constantConditionNoEffect*:
assumes ¬(*is-IfNode* (*kind g nid*))
shows *g* = *constantCondition b nid* (*kind g nid*) *g*
using *assms constantCondition.simps*
apply (*cases kind g nid*)
prefer *15* **prefer** *16*
apply (*metis is-IfNode-def*)
apply (*metis*)
by *presburger*+

lemma *changeonly-ConstantExpr*:
assumes *unrep g* (*ConstantExpr c*) (*g′*, *nid*)
shows *changeonly* {} *g g′*
using *assms*
apply (*cases find-node-and-stamp g* (*ConstantNode c*, *constantAsStamp c*) = *None*)
apply (*smt* (*verit*, *ccfv-threshold*) *New add-node-as-set-eq changeonly.simps fresh-ids new-def not-excluded-keep-type order.refl uniqueDet unrepConstE unrep-preserves-contains*)
by (*metis changeonly.simps unique.cases unrepConstE unrepDet*)

lemma *constantCondition-changeonly*:

assumes *nid* ∈ *ids g*
assumes *g′* = *constantCondition b nid* (*kind g nid*) *g*
shows *changeonly* {*nid*} *g g′*
proof (*cases is-IfNode* (*kind g nid*))
case *True*
obtain *g′′ nid′* **where** *unrep*: *unrep g* (*ConstantExpr* (*bool-to-val-width1 b*)) (*g′′*, *nid′*)
using *insertConst* **by** *blast*
also have *changeonly* {} *g g′′*
using *changeonly-ConstantExpr unrep* **by** *blast*
moreover have ∃ *t f ifcond*. *g′* = *replace-node nid* (*IfNode nid′ t f*, *stamp g ifcond*) *g′′*
using *assms constantCondition-sem unrep*
by (*metis True is-IfNode-def*)
then show *?thesis*
using *assms replace-node-changeonly add-node-changeonly* **unfolding** *changeonly.simps*
by (*metis calculation*(*2*) *changeonly.elims*(*2*) *empty-iff*)
next
case *False*
have *g* = *g′*
using *constantConditionNoEffect False assms*(*2*) **by** *presburger*
then show *?thesis*
by *simp*
qed

lemma *constantConditionNoIf*:
assumes ∀ *cond t f*. *kind g ifcond* ≠ *IfNode cond t f*
assumes *g′* = *constantCondition val ifcond* (*kind g ifcond*) *g*
shows ∃ *nid′* .(*g m p h* ⊢ *ifcond* ⇝ *nid′*) ⟷ (*g′ m p h* ⊢ *ifcond* ⇝ *nid′*)
proof −
have *g′* = *g*
using *constantConditionNoEffect assms is-IfNode-def* **by** *presburger*
then show *?thesis*
by *simp*
qed

lemma *constantConditionValid*:
assumes *kind g ifcond* = *IfNode cond t f*
assumes [*g*, *m*, *p*] ⊢ *cond* ↦ *v*
assumes *const* = *val-to-bool v*
assumes *g′* = *constantCondition const ifcond* (*kind g ifcond*) *g*
shows ∃ *nid′* .(*g m p h* ⊢ *ifcond* ⇝ *nid′*) ⟷ (*g′ m p h* ⊢ *ifcond* ⇝ *nid′*)
proof (*cases const*)
case *True*
have *ifstep*: *g*, *p* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)
by (*meson IfNode True assms*(*1*,*2*,*3*) *encodeeval.simps*)
have *ifstep′*: *g′*, *p* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)
using *constantConditionTrue True assms*(*1*,*4*) **by** *presburger*
from *ifstep ifstep′* **show** *?thesis*

using *StutterStep* **by** *blast*
next
case *False*
have *ifstep*: $g,\ p \vdash (ifcond,\ m,\ h) \to (f,\ m,\ h)$
by (*meson IfNode False assms*(*1*,*2*,*3*) *encodeeval.simps*)
have *ifstep'*: $g',\ p \vdash (ifcond,\ m,\ h) \to (f,\ m,\ h)$
using *constantConditionFalse False assms*(*1*,*4*) **by** *presburger*
from *ifstep ifstep'* **show** *?thesis*
using *StutterStep* **by** *blast*
qed

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:
$[\![g,\ p \vdash (nid,m,h) \to (nid',m,h)]\!]$
$\Longrightarrow g\ m\ p\ h \vdash nid \leadsto nid'$ |

Transitive:
$[\![g,\ p \vdash (nid,m,h) \to (nid'',m,h);$
$g\ m\ p\ h \vdash nid'' \leadsto nid']\!]$
$\Longrightarrow g\ m\ p\ h \vdash nid \leadsto nid'$

lemma *stuttering-successor*:
assumes $(g,\ p \vdash (nid,\ m,\ h) \to (nid',\ m,\ h))$
shows $\{P'.\ (g\ m\ p\ h \vdash nid \leadsto P')\} = \{nid'\} \cup \{nid''.\ (g\ m\ p\ h \vdash nid' \leadsto nid'')\}$
proof −
have *nextin*: $nid' \in \{P'.\ (g\ m\ p\ h \vdash nid \leadsto P')\}$
using *assms StutterStep* **by** *fast*
have *nextsubset*: $\{nid''.\ (g\ m\ p\ h \vdash nid' \leadsto nid'')\} \subseteq \{P'.\ (g\ m\ p\ h \vdash nid \leadsto P')\}$
by (*metis Collect-mono assms stutter.Transitive*)
have $\forall n \in \{P'.\ (g\ m\ p\ h \vdash nid \leadsto P')\}\ .\ n = nid' \lor n \in \{nid''.\ (g\ m\ p\ h \vdash nid' \leadsto nid'')\}$
by (*metis* (*no-types*, *lifting*) *Pair-inject assms mem-Collect-eq stutter.simps stepDet*)
then show *?thesis*
using *nextin nextsubset* **by** (*auto simp add*: *mk-disjoint-insert*)
qed

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*
  by (*metis*(*full-types*) *eq-imp-le signed-take-bit-take-bit assms*)

lemma *unwrap-signed-take-bit*:
  fixes *v* :: *int64*
  assumes *0 < b ∧ b ≤ 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*
  using *assms* **by** (*simp add*: *signed-def*)

lemma *unrestricted-new-int-always-valid* [*simp*]:
  assumes *0 < b ∧ b ≤ 64*
  shows *valid-value* (*new-int b v*) (*unrestricted-stamp* (*IntegerStamp b lo hi*))
  by (*simp*; *metis One-nat-def assms int-power-div-base int-signed-value.simps int-signed-value-range*
    linorder-not-le not-exp-less-eq-0-int zero-less-numeral)

lemma *unary-undef*: *val = UndefVal ⟹ unary-eval op val = UndefVal*
  by (*cases op*; *auto*)

lemma *unary-obj*:
  val = ObjRef x ⟹ (*if* (*op = UnaryIsNull*) *then*
    unary-eval op val ≠ UndefVal else
    unary-eval op val = UndefVal)
  by (*cases op*; *auto*)

lemma *unrestricted-stamp-valid*:
  assumes *s = unrestricted-stamp* (*IntegerStamp b lo hi*)
  assumes *0 < b ∧ b ≤ 64*
  shows *valid-stamp s*
  using *assms* **apply** *auto* **by** (*simp add*: *pos-imp-zdiv-pos-iff self-le-power*)

lemma *unrestricted-stamp-valid-value* [*simp*]:
  assumes *1*: *result = IntVal b ival*
  assumes *take-bit b ival = ival*
  assumes *0 < b ∧ b ≤ 64*
  shows *valid-value result* (*unrestricted-stamp* (*IntegerStamp b lo hi*))
proof −

have *valid-stamp* (*unrestricted-stamp* (*IntegerStamp b lo hi*))
  using *assms unrestricted-stamp-valid* **by** *blast*
then show *?thesis*
  unfolding *unrestricted-stamp.simps* **using** *assms int-signed-value-bounds valid-value.simps*
  by *presburger*
qed

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*
  using *assms* **apply** (*cases stamp*; *auto*) **by** (*metis that*)

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*
  by (*metis assms valid-int valid-value.simps*(*1*))

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*
  by (*meson assms valid-value.simps*(*1*))

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*)
  by (*metis assms valid-value.simps*(*1*))

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*
  by (*metis assms order.trans valid-value.simps*(*1*))

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*
by (*metis assms valid-value.simps*(*1*))

lemma *valid-int-is-zero-masked*:
assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)
shows *and val* (*not* (*mask bits*)) = *0*
by (*metis* (*no-types*, *lifting*) *assms bit.conj-cancel-right take-bit-eq-mask valid-int-fits*

word-bw-assocs(*1*) *word-log-esimps*(*1*))

Unsigned ints have bounds 0 up to 2^bits.

lemma *valid-int-unsigned-bounds*:
assumes *valid-value* (*IntVal b val*) (*IntegerStamp bits lo hi*)

shows *uint val* < *2* ^ *bits*
by (*metis assms*(*1*) *mask-eq-iff take-bit-eq-mask valid-value.simps*(*1*))

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*)
by (*metis* (*mono-tags*, *opaque-lifting*) *diff-le-mono int-signed-value.simps less-imp-diff-less linorder-not-le one-le-numeral order-less-le-trans signed-take-bit-int-less-exp-word*
sint-lt
power-increasing)

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*
using *assms One-nat-def ValueThms.int-signed-value-range* **by** *auto*

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 −
have *b* = *bits*
using *assms valid-int-same-bits* **by** *blast*
then show *?thesis*
using *assms* **by** *auto*
qed

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 −
have *b* = *bits*

using *assms valid-int-same-bits* **by** *blast*
then show *?thesis*
using *assms* **by** *auto*
qed

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*
by (*metis assms valid-value.simps*(*1*))

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 −
obtain *b1 v1* **where** *v1*: *val* = *IntVal b1 v1*
using *assms* **by** (*meson is-IntVal-def unary-eval-int unary-normal-bitsize*)
then obtain *b2 v2* **where** *v2*: *result* = *IntVal b2 v2*
by (*metis Value.collapse*(*1*) *assms*(*2*,*6*) *unary-eval-int*)
then have *result* = *unary-eval op* (*IntVal b1 v1*)
using *assms*(*2*) *v1* **by** *blast*
then obtain *vtmp* **where** *vtmp*: *result* = *new-int b2 vtmp*
using *assms*(*3*) **by** (*auto simp add*: *v2*)
obtain *b′ lo′ hi′* **where** *stamp-expr expr* = *IntegerStamp b′ lo′ hi′*
by (*metis assms*(*7*) *v1 valid-int-gives*)
then have *stamp-unary op* (*stamp-expr expr*) =
unrestricted-stamp
(*IntegerStamp* (*if op* ∈ *normal-unary then b′ else ir-resultBits op*) *lo′ hi′*)
using *op* **by** *force*
then obtain *lo2 hi2* **where** *s*: (*stamp-expr* (*UnaryExpr op expr*)) =
unrestricted-stamp (*IntegerStamp b2 lo2 hi2*)
unfolding *stamp-expr.simps*
by (*metis* (*full-types*) *assms*(*2*,*7*) *unary-normal-bitsize v2 valid-int-same-bits op* ‹*stamp-expr expr* = *IntegerStamp b′ lo′ hi′*›)
then have *bitRange*: *0* < *b1* ∧ *b1* ≤ *64*
using *assms*(*1*) *eval-bits-1-64 v1* **by** *blast*
then have *fst* (*bit-bounds b2*) ≤ *int-signed-value b2 v2* ∧

int-signed-value b2 v2 ≤ *snd* (*bit-bounds b2*)
using *assms*(*2*) *int-signed-value-bounds unary-eval-bitsize v1 v2* **by** *blast*
then show *?thesis*
apply *auto*
by (*metis stamp-expr.simps*(*1*) *unrestricted-new-int-always-valid bitRange assms*(*2*) *s v1 vtmp v2*
unary-eval-bitsize)
qed

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 −
consider *ib ob* **where** *op* = *UnaryNarrow ib ob*
| *ib ob* **where** *op* = *UnarySignExtend ib ob*
| *ib ob* **where** *op* = *UnaryZeroExtend ib ob*
using *IRUnaryOp.exhaust-sel assms*(*2*,*3*,*4*) **by** *blast*
then show *?thesis*
proof (*cases*)
case *1*
then show *?thesis*
using *assms intval-narrow-ok* **by** *force*
next
case *2*
then show *?thesis*
using *assms intval-sign-extend-ok* **by** *force*
next
case *3*
then show *?thesis*
using *assms intval-zero-extend-ok* **by** *force*
qed
qed

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 −
obtain *b1 v1* **where** *v1*: *val* = *IntVal b1 v1*
by (*metis Value.exhaust-disc insertCI is-ArrayVal-def is-IntVal-def is-ObjRef-def is-ObjStr-def*

unary-obj valid-value.simps(3,11,12) assms(2,4,6,7))
then have *result = unary-eval op (IntVal b1 v1)*
using *assms(2)* **by** *blast*
then obtain *v2* **where** *v2: result = new-int (ir-resultBits op) v2*
using *assms unary-eval-new-int* **by** *presburger*
then obtain *v3* **where** *v3: result = IntVal (ir-resultBits op) v3*
using *assms* **by** *(cases op; simp; (meson new-int.simps)+)*
then obtain *b lo2 hi2* **where** *eval: stamp-expr expr = IntegerStamp b lo2 hi2*
by *(metis assms(7) v1 valid-int-gives)*
then have *s: (stamp-expr (UnaryExpr op expr)) =*
unrestricted-stamp (IntegerStamp (ir-resultBits op) lo2 hi2)
using *op notbool notfixed* **by** *(cases op; auto)*
then have *outBits: 0 < (ir-resultBits op) ∧ (ir-resultBits op) ≤ 64*
using *assms narrow-widen-output-bits* **by** *blast*
then have *fst (bit-bounds (ir-resultBits op)) ≤ int-signed-value (ir-resultBits op) v3 ∧*
int-signed-value (ir-resultBits op) v3 ≤ snd (bit-bounds (ir-resultBits op))
using *ValueThms.int-signed-value-bounds outBits* **by** *blast*
then show *?thesis*
using *v2 s* **by** *(simp add: v3 outBits)*
qed

lemma *eval-boolean-unary-implies-valid-value*:
assumes *[m,p] ⊢ expr ↦ 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 −
obtain *b1* **where** *v1: val = ObjRef (b1)*
by *(metis singletonD unary-eval.simps(8) intval-is-null.elims assms(2,3,5))*
then have *eval: result = unary-eval op (ObjRef (b1))*
using *assms(2)* **by** *blast*
then obtain *v2* **where** *v2: result = IntVal 32 v2*
by *(metis op singleton-iff unary-eval.simps(8) intval-is-null.simps(1) bool-to-val.simps(1,2))*
have *vBounds: result ∈ {bool-to-val True, bool-to-val False}*
by *(metis insertI1 insertI2 intval-is-null.simps(1) op singleton-iff unary-eval.simps(8) eval)*
then have *boolstamp: (stamp-expr (UnaryExpr op expr)) = (IntegerStamp 32 0 1)*
using *op* **by** *(cases op; auto)*
then show *?thesis*
using *vBounds* **by** *(cases result; auto)*
qed

lemma *eval-fixed-unary-32-implies-valid-value*:
assumes *[m,p] ⊢ expr ↦ 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 −
obtain *b1 v1* **where** *v1*: *val* = *IntVal b1 v1*
by (*metis Value.exhaust-sel insert-iff intval-bit-count.simps*(*3*,*4*,*5*) *unary-eval.simps*(*10*) *valid-value.simps*(*3*) *assms*(*2*,*3*,*5*,*6*,*7*))
then obtain *v2* **where** *v2*: *result* = *new-int 32 v2*
using *assms unary-eval-new-int* **by** *presburger*
then obtain *v3* **where** *v3*: *result* = *IntVal 32 v3*
using *assms* **by** (*cases op*; *simp*; (*meson new-int.simps*)+)
then obtain *b lo2 hi2* **where** *eval*: *stamp-expr expr* = *IntegerStamp b lo2 hi2*
by (*metis assms*(*7*) *v1 valid-int-gives*)
then have *s*: (*stamp-expr* (*UnaryExpr op expr*)) = *unrestricted-stamp* (*IntegerStamp 32 lo2 hi2*)
using *op notbool* **by** (*cases op*; *auto*)
then have *fst* (*bit-bounds 32*) ≤ *int-signed-value 32 v3* ∧
int-signed-value 32 v3 ≤ *snd* (*bit-bounds 32*)
by (*metis ValueThms.int-signed-value-bounds leI not-numeral-le-zero semiring-norm*(*68*,*71*)
numeral-le-iff)
then show *?thesis*
using *s v2 v3* **by** *force*
qed

lemma *eval-unary-implies-valid-value*:
assumes [*m*,*p*] ⊢ *expr* ↦ *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 (*cases op* ∈ *normal-unary*)
case *True*
then show *?thesis*
using *assms eval-normal-unary-implies-valid-value* **by** *blast*
next
case *False*
then show *?thesis*
proof (*cases op* ∈ *boolean-unary*)
case *True*
then show *?thesis*
using *assms eval-boolean-unary-implies-valid-value* **by** *blast*
next
case *False*
then show *?thesis*

proof (*cases op* ∈ *unary-fixed-32-ops*)
case *True*
then show *?thesis*
using *assms eval-fixed-unary-32-implies-valid-value* **by** *auto*
next
case *False*
then show *?thesis*
using *assms*
by (*meson eval-boolean-unary-implies-valid-value eval-normal-unary-implies-valid-value eval-widen-narrow-unary-implies-valid-value unary-ops-distinct*(*2*))
qed
qed
qed

9.4.3 Support Lemmas for Binary Operators

lemma *binary-undef*: *v1* = *UndefVal* ∨ *v2* = *UndefVal* ⟹ *bin-eval op v1 v2* = *UndefVal*
by (*cases op*; *auto*)

lemma *binary-obj*: *v1* = *ObjRef x* ∨ *v2* = *ObjRef y* ⟹ *bin-eval op v1 v2* = *UndefVal*
by (*cases op*; *auto*)

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*
using *assms* **by** (*cases op*; *simp*; *smt* (*verit*, *best*) *new-int.simps*)+

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*
apply (*cases op*; *simp*)
using *assms* **by** (*metis new-int.simps bin-eval-new-int*)+

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*
using *assms* **apply** (*cases op*; *simp*)
apply *metis*+
apply (*metis new-int-bin.simps*)+

```
  by (metis take-bit-xor take-bit-and take-bit-or)+

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
  using assms apply (cases op; simp) by (meson new-int-bin.simps)+

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 −
  obtain b1 v1 where v1: val1 = IntVal b1 v1
    by (metis Value.collapse(1) assms(3,4) bin-eval-inputs-are-ints bin-eval-int)
  obtain b2 v2 where v2: val2 = IntVal b2 v2
    by (metis Value.collapse(1) assms(3,4) bin-eval-inputs-are-ints bin-eval-int)
  then obtain lo1 hi1 where s1: stamp-expr expr1 = IntegerStamp b1 lo1 hi1
    by (metis assms(5) v1 valid-int-gives)
  then obtain lo2 hi2 where s2: stamp-expr expr2 = IntegerStamp b2 lo2 hi2
    by (metis assms(6) v2 valid-int-gives)
  then have r: result = bin-eval op (IntVal b1 v1) (IntVal b2 v2)
    using assms(3) v1 v2 by presburger
  then obtain bres vtmp where vtmp: result = new-int bres vtmp
    using assms by (meson bin-eval-new-int)
  then obtain vres where vres: result = IntVal bres vres
    by force

  then have sres: stamp-expr (BinaryExpr op expr1 expr2) =
             unrestricted-stamp (IntegerStamp bres lo1 hi1)
           ∧ 0 < bres ∧ bres ≤ 64
    proof (cases op ∈ binary-shift-ops)
      case True
      then show ?thesis
        unfolding stamp-expr.simps
     by (metis Value.inject(1) eval-bits-1-64 new-int.simps r assms(1,4) stamp-binary.simps(1)
            bin-eval-bits-binary-shift-ops s2 s1 v1 vres)
    next
      case False
      then have op ∉ binary-shift-ops
        by blast
      then have beq: b1 = b2
        using v1 v2 assms bin-eval-input-bits-equal by blast
      then show ?thesis
```

proof (*cases op* ∈ *binary-fixed-32-ops*)
case *True*
then show *?thesis*
unfolding *stamp-expr.simps*
by (*metis False Value.inject*(*1*) *beq bin-eval-new-int le-add-same-cancel1 new-int.simps s2 s1*
numeral-Bit0 vres zero-le-numeral zero-less-numeral assms(*3*,*4*) *stamp-binary.simps*(*1*))
next
case *False*
then show *?thesis*
unfolding *s1 s2 stamp-binary.simps stamp-expr.simps*
by (*metis beq bin-eval-new-int eval-bits-1-64 intval-bits.simps assms*(*1*,*3*,*4*) *vres v1*
unrestricted-new-int-always-valid unrestricted-stamp.simps(*2*) *valid-int-same-bits*)
qed
qed
then show *?thesis*
using *unrestricted-new-int-always-valid vres vtmp* **by** *presburger*
qed

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*)
using *assms* **apply** (*cases stamp1*; *cases stamp2*; *auto*)
using *meet.simps*(*2*) *valid-stamp.simps*(*1*,*8*) *is-IntegerStamp-def assms* **by** *linarith+*

lemma *stamp-meet-is-valid-stamp*:
assumes *1*: *valid-stamp stamp1*
assumes *2*: *valid-stamp stamp2*
shows *valid-stamp* (*meet stamp1 stamp2*)
by (*cases stamp1*; *cases stamp2*; *insert stamp-meet-integer-is-valid-stamp*[*OF 1 2*]; *auto*)

lemma *stamp-meet-commutes*: *meet stamp1 stamp2* = *meet stamp2 stamp1*
by (*cases stamp1*; *cases stamp2*; *auto*)

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 –
  have *m*: *meet stamp1 stamp2* = *IntegerStamp b1* (*min lo1 lo2*) (*max hi1 hi2*)
    by (*metis assms*(*3*,*4*,*5*) *meet.simps*(*2*))
  obtain *ival* **where** *val*: *val* = *IntVal b1 ival*
    using *assms valid-int* **by** *blast*
  then have *v*: *valid-stamp* (*IntegerStamp b1 lo1 hi1*) ∧
      take-bit b1 ival = *ival* ∧
      lo1 ≤ *int-signed-value b1 ival* ∧ *int-signed-value b1 ival* ≤ *hi1*
    by (*metis assms*(*1*,*3*) *valid-value.simps*(*1*))
  then have *mm*: *min lo1 lo2* ≤ *int-signed-value b1 ival* ∧ *int-signed-value b1 ival* ≤ *max hi1 hi2*
    by *linarith*
  then have *valid-stamp* (*IntegerStamp b1* (*min lo1 lo2*) (*max hi1 hi2*))
    by (*metis meet.simps*(*2*) *stamp-meet-is-valid-stamp v assms*(*2*,*3*,*4*,*5*))
  then show *?thesis*
    using *mm v valid-value.simps val m* **by** *presburger*
qed

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*)
  by (*metis stamp-meet-is-valid-value1 stamp-meet-commutes assms*)

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 –
  have *def*: *meet* (*stamp-expr expr1*) (*stamp-expr expr2*) ≠ *IllegalStamp*
    using *assms* **apply** *auto*
   by (*smt* (*verit*, *ccfv-threshold*) *Stamp.distinct*(*13*,*25*) *compatible.elims*(*2*) *meet.simps*(*1*,*2*))
  then have *valid-stamp* (*meet* (*stamp-expr expr1*) (*stamp-expr expr2*))
    using *assms* **apply** *auto*
   by (*metis compatible-refl compatible.elims*(*2*) *stamp-meet-is-valid-stamp valid-stamp.simps*(*2*)
      assms(*7*))
  then show *?thesis*
    using *assms* **apply** *auto*

by (*smt* (*verit*, *ccfv-SIG*) *Stamp.distinct*(*1*) *assms*(*6*,*7*) *compatible.elims*(*2*) *compatible.simps*(*1*)
def compatible-refl stamp-meet-commutes stamp-meet-is-valid-value1 valid-value.simps(*13*))
qed

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 −
have *yval*: *valid-value yv* (*stamp-expr y*)
using *assms wf-stamp-def* **by** *blast*
obtain *b lx hi* **where** *xstamp*: *stamp-expr x* = *IntegerStamp b lx hi*
by (*metis stamp-under.elims*(*2*) *assms*(*1*))
then obtain *b′ lo hy* **where** *ystamp*: *stamp-expr y* = *IntegerStamp b′ lo hy*
by (*meson stamp-under.elims*(*2*) *assms*(*1*))
obtain *xvv* **where** *xvv*: *xv* = *IntVal b xvv*
by (*metis assms*(*2*,*3*) *valid-int wf-stamp-def xstamp*)
then have *xval*: *valid-value* (*IntVal b xvv*) (*stamp-expr x*)
using *assms*(*2*,*3*) *wf-stamp-def* **by** *blast*
obtain *yvv* **where** *yvv*: *yv* = *IntVal b′ yvv*
by (*metis valid-int ystamp yval*)
then have *xval*: *valid-value* (*IntVal b′ yvv*) (*stamp-expr y*)
using *yval* **by** *blast*
have *xunder*: *int-signed-value b xvv* ≤ *hi*
by (*metis assms*(*2*,*3*) *wf-stamp-def xstamp valid-value.simps*(*1*) *xvv*)
have *yunder*: *lo* ≤ *int-signed-value b′ yvv*
by (*metis ystamp valid-value.simps*(*1*) *yval yvv*)
have *unwrap*: ∀ *cond*. *bool-to-val-bin b b cond* = *bool-to-val cond*
by *simp*
from *xunder yunder* **have** *int-signed-value b xvv* < *int-signed-value b′ yvv*
using *assms*(*1*) *xstamp ystamp* **by** *force*
then have (*intval-less-than xv yv*) = *IntVal 32 1* ∨ (*intval-less-than xv yv*) = *UndefVal*
by (*simp add*: *yvv xvv*)
then show *?thesis*
by *force*
qed

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 −
have *yval*: *valid-value yv* (*stamp-expr y*)
using *assms wf-stamp-def* **by** *blast*
obtain *b lo hx* **where** *xstamp*: *stamp-expr x* = *IntegerStamp b lo hx*
by (*metis stamp-under.elims*(*2*) *assms*(*1*))
then obtain *b′ ly hi* **where** *ystamp*: *stamp-expr y* = *IntegerStamp b′ ly hi*
by (*meson stamp-under.elims*(*2*) *assms*(*1*))
obtain *xvv* **where** *xvv*: *xv* = *IntVal b xvv*
by (*metis assms*(*2*,*3*) *valid-int wf-stamp-def xstamp*)
then have *xval*: *valid-value* (*IntVal b xvv*) (*stamp-expr x*)
using *assms*(*2*,*3*) *wf-stamp-def* **by** *blast*
obtain *yvv* **where** *yvv*: *yv* = *IntVal b′ yvv*
by (*metis valid-int ystamp yval*)
then have *xval*: *valid-value* (*IntVal b′ yvv*) (*stamp-expr y*)
using *yval* **by** *simp*
have *yunder*: *int-signed-value b′ yvv* ≤ *hi*
by (*metis ystamp valid-value.simps*(*1*) *yval yvv*)
have *xover*: *lo* ≤ *int-signed-value b xvv*
by (*metis assms*(*2*,*3*) *wf-stamp-def xstamp valid-value.simps*(*1*) *xvv*)
have *unwrap*: ∀ *cond*. *bool-to-val-bin b b cond* = *bool-to-val cond*
by *simp*
from *xover yunder* **have** *int-signed-value b′ yvv* < *int-signed-value b xvv*
using *assms*(*1*) *xstamp ystamp* **by** *force*
then have (*intval-less-than xv yv*) = *IntVal 32 0* ∨ (*intval-less-than xv yv*) = *UndefVal*
by (*auto simp add*: *yvv xvv*)
then show *?thesis*
by *force*
qed

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 Bin-ShortCircuitOr*}
| *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 UnaryLog-icNegation*}
| *markup DSL-Tokens.LeftShift* = @{*term BinaryExpr*} $ @{*term BinLeftShift*}
| *markup DSL-Tokens.RightShift* = @{*term BinaryExpr*} $ @{*term BinRightShift*}
| *markup DSL-Tokens.UnsignedRightShift* = @{*term BinaryExpr*} $ @{*term Bin-URightShift*}
| *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*} , *IntVal-Markup.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*} , *Word-Markup.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*[$\neg x$]
value *val*[$\neg x$]
value *exp*[$\neg x$]

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*) = ($y \le x$) |
rewrite-preservation (*Conditional x y cond*) = (*cond* ⟶ ($y \le x$)) |
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*) = ($x \neq UndefVal \land y \neq UndefVal \longrightarrow x = y$) |
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

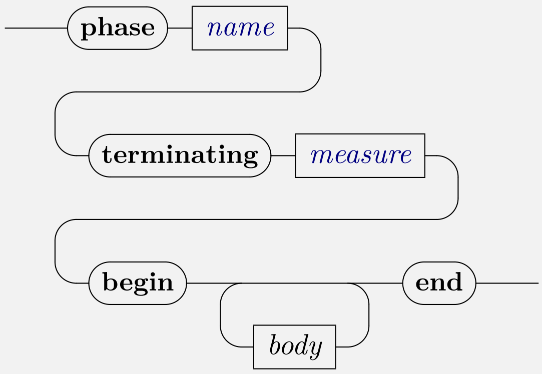
structure DSL = DSL-Rewrites(System);

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);
›

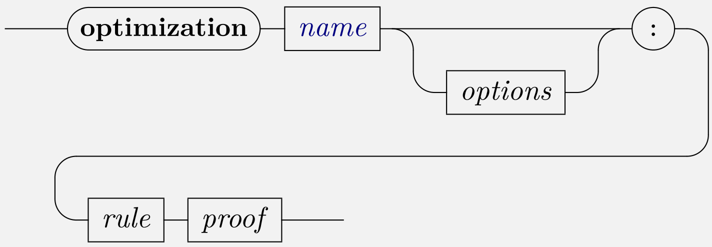
ML-file ~~/*src*/*Doc*/*antiquote-setup.ML*

PhaseRail

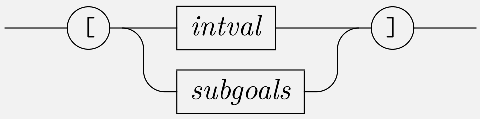
phase



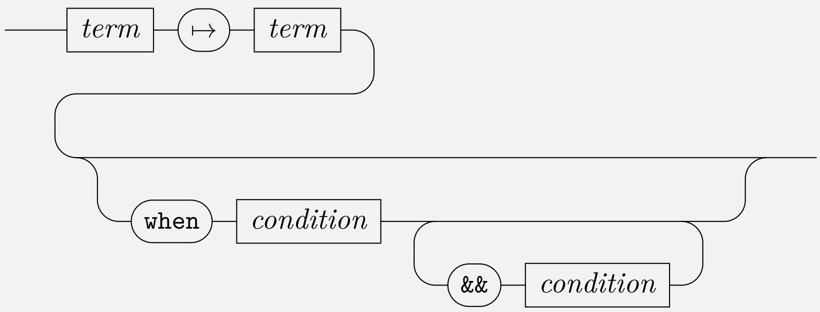
optimization



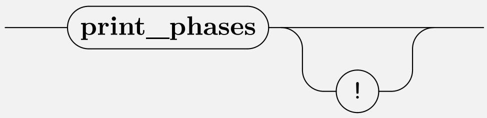
options



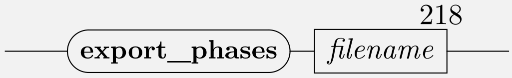
rule



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*
  apply (*induction y*; *auto?*)
  subgoal for *op*
    apply (*cases op*)
    by (*smt* (*z3*) *gr0I one-neq-zero pos2 size.elims trans-less-add2*)+
  done

lemma *size-non-add*[*size-simps*]: *size* (*BinaryExpr op a b*) = *size a* + *size b* + *2* ⟷ ¬(*is-ConstantExpr b*)
  by (*induction b*; *induction op*; *auto simp*: *is-ConstantExpr-def*)

lemma *size-non-const*[*size-simps*]:
  ¬ *is-ConstantExpr y* ⟹ *1* < *size y*
  using *size-pos* **apply** (*induction y*; *auto*)
  by (*metis Suc-lessI add-is-1 is-ConstantExpr-def le-less linorder-not-le n-not-Suc-n numeral-2-eq-2 pos2 size.simps*(*2*) *size-non-add*)

lemma *size-binary-const*[*size-simps*]:
  size (*BinaryExpr op a b*) = *size a* + *2* ⟷ (*is-ConstantExpr b*)
  by (*induction b*; *auto simp*: *is-ConstantExpr-def size-pos*)

lemma *size-flip-binary*[*size-simps*]:
  ¬(*is-ConstantExpr y*) ⟶ *size* (*BinaryExpr op* (*ConstantExpr x*) *y*) > *size* (*BinaryExpr op y* (*ConstantExpr x*))
  by (*metis add-Suc not-less-eq order-less-asym plus-1-eq-Suc size.simps*(*2*,*11*) *size-non-add*)

lemma *size-binary-lhs-a*[*size-simps*]:
  size (*BinaryExpr op* (*BinaryExpr op′ a b*) *c*) > *size a*
  by (*metis add-lessD1 less-add-same-cancel1 pos2 size-binary-const size-non-add*)

lemma *size-binary-lhs-b*[*size-simps*]:
  size (*BinaryExpr op* (*BinaryExpr op′ a b*) *c*) > *size b*
  by (*metis IRExpr.disc*(*42*) *One-nat-def add.left-commute add.right-neutral is-ConstantExpr-def less-add-Suc2 numeral-2-eq-2 plus-1-eq-Suc size.simps*(*11*) *size-binary-const size-non-add size-non-const trans-less-add1*)

lemma *size-binary-lhs-c*[*size-simps*]:
size (*BinaryExpr op* (*BinaryExpr op′ a b*) *c*) > *size c*
by (*metis IRExpr.disc*(*42*) *add.left-commute add.right-neutral is-ConstantExpr-def less-Suc-eq numeral-2-eq-2 plus-1-eq-Suc size.simps*(*11*) *size-non-add size-non-const trans-less-add2*)

lemma *size-binary-rhs-a*[*size-simps*]:
size (*BinaryExpr op c* (*BinaryExpr op′ a b*)) > *size a*
apply *auto*
by (*metis trans-less-add2 less-Suc-eq less-add-same-cancel1 linorder-neqE-nat not-add-less1 pos2*
order-less-trans size-binary-const size-non-add)

lemma *size-binary-rhs-b*[*size-simps*]:
size (*BinaryExpr op c* (*BinaryExpr op′ a b*)) > *size b*
by (*metis add.left-commute add.right-neutral is-ConstantExpr-def lessI numeral-2-eq-2 plus-1-eq-Suc size.simps*(*4*,*11*) *size-non-add trans-less-add2*)

lemma *size-binary-rhs-c*[*size-simps*]:
size (*BinaryExpr op c* (*BinaryExpr op′ a b*)) > *size c*
by *simp*

lemma *size-binary-lhs*[*size-simps*]:
size (*BinaryExpr op x y*) > *size x*
by (*metis One-nat-def Suc-eq-plus1 add-Suc-right less-add-Suc1 numeral-2-eq-2 size-binary-const size-non-add*)

lemma *size-binary-rhs*[*size-simps*]:
size (*BinaryExpr op x y*) > *size y*
by (*metis IRExpr.disc*(*42*) *add-strict-increasing is-ConstantExpr-def linorder-not-le not-add-less1 size.simps*(*11*) *size-non-add size-non-const size-pos*)

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)
unfolding *well-formed-equal-def* **by** *simp*

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′*)
  using *assms*
  by (*metis* (*no-types*, *lifting*) *ValueThms.signed-take-take-bit bool-to-val.elims bool-to-val-bin.elims int-signed-value.simps intval-less-than.simps*(*1*) *new-int.simps one-neq-zero val-to-bool.simps*(*1*))

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*)
  using *assms* **by** *force*

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*)
  using *assms* **by** *force*

lemma *val-bool-unwrap*:
  val-to-bool (*bool-to-val v*) = *v*
  by (*metis bool-to-val.elims one-neq-zero val-to-bool.simps*(*1*))

lemma *take-bit-64*:
  assumes *0* < *b* ∧ *b* ≤ *64*
  assumes *take-bit b v* = *v*
  shows *take-bit 64 v* = *take-bit b v*
  using *assms*
  by (*metis min-def nle-le take-bit-take-bit*)

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*)
using *assms*
by (*simp add*: *bit-last-iff word-sless-alt*)

lemma *positive-def*:
fixes *v* :: *'a*::*len word*
assumes *0 <s v*
shows ¬(*bit v* (*LENGTH*(*'a*) $-$ *1*))
using *assms*
by (*simp add*: *bit-last-iff word-sless-alt*)

lemma *negative-lower-bound*:
fixes *v* :: *'a*::*len word*
assumes (*2*^(*LENGTH*(*'a*) $-$ *1*)) *<s v*
assumes *v <s 0*
shows *0 <s* ($-$*v*)
using *assms*
by (*smt* (*verit*) *signed-0 signed-take-bit-int-less-self-iff sint-ge sint-word-ariths*(*4*) *word-sless-alt*)

lemma *min-int*:
fixes *x* :: *'a*::*len word*
assumes *x <s 0*
assumes *x* ≠ (*2*^(*LENGTH*(*'a*) $-$ *1*))
shows *2*^(*LENGTH*(*'a*) $-$ *1*) *<s x*
using *assms* **sorry**

lemma *negate-min-int*:
fixes *v* :: *'a*::*len word*
assumes *v* = (*2*^(*LENGTH*(*'a*) $-$ *1*))
shows *v* = ($-$*v*)
using *assms*
by (*metis One-nat-def add.inverse-neutral double-eq-zero-iff mult-minus-right verit-minus-simplify*(*4*))

fun *abs* :: *'a*::*len word* ⇒ *'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 (*cases* $0 \leq s$ x)
  case *True*
  then show *?thesis*
    by *force*
next
  case *neg*: *False*
  then show *?thesis*
  proof (*cases* $x = (2\hat{\ }LENGTH('a) - 1)$)
    case *True*
    then show *?thesis*
      using *negate-min-int*
      by (*simp add*: *word-sless-alt*)
  next
    case *False*
    then show *?thesis* **using** *min-int negative-lower-bound*
      using *negate-min-int* **by** *force*
  qed
qed

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 $v \neq (2\hat{\ }(b - 1))$
  shows $0 <$ *int-signed-value* b $(-v)$
  using *assms* **apply** *simp* **sorry**

lemma *negate-max-negative*:
  assumes $b > 0 \wedge b \leq 64$
  assumes *take-bit* b $v = v$
  assumes $v = (2\hat{\ }(b - 1))$
  shows *new-int* b v = *intval-negate* (*new-int* b v)
  using *assms* **apply** *simp* **using** *negate-min-int* **sorry**

lemma *val-abs-always-pos*:
  assumes $b > 0 \wedge b \leq 64$
  assumes *take-bit* b $v = v$
  assumes $v \neq (2\hat{\ }(b - 1))$
  assumes *intval-abs* (*new-int* b v) = (*new-int* b v')
  shows *val-to-bool* (*val*[(*new-int* b 0) < (*new-int* b v')]) $\vee$ *val-to-bool* (*val*[(*new-int* b 0) *eq* (*new-int* b v')])
proof (*cases* $v = 0$)
  case *True*
  then have *isZero*: *intval-abs* (*new-int* b 0) = *new-int* b 0

```
    by auto
  then have IntVal b 0 = new-int b v'
    using True assms by auto
  then have val-to-bool (val[(new-int b 0) eq (new-int b v')])
    by simp
  then show ?thesis by simp
next
  case neq0: False
  have zero: int-signed-value b 0 = 0
    by simp
  then show ?thesis
  proof (cases int-signed-value b v > 0)
    case True
    then have val-to-bool(val[(new-int b 0) < (new-int b v)])
      using zero apply simp
    by (metis One-nat-def ValueThms.signed-take-take-bit assms(1) val-bool-unwrap)
    then have val-to-bool (val[new-int b 0 < new-int b v'])
      by (metis assms(4) val-abs-pos)
    then show ?thesis
      by blast
  next
    case neg: False
    then have val-to-bool (val[new-int b 0 < new-int b v'])
    proof -
      have int-signed-value b v ≤ 0
        using assms neg neq0 by simp
      then show ?thesis
      proof (cases int-signed-value b v = 0)
        case True
        then have v = 0
              by (metis One-nat-def Suc-pred assms(1) assms(2) dual-order.refl
int-signed-value.simps signed-eq-0-iff take-bit-of-0 take-bit-signed-take-bit)
        then show ?thesis
          using neq0 by simp
      next
        case False
        then have int-signed-value b v < 0
          using ‹int-signed-value (b::nat) (v::64 word) ⊑ (0::int)› by linarith
        then have new-int b v' = new-int b (−v)
          using assms using intval-abs.elims
          by simp
        then have 0 < int-signed-value b (−v)
          using assms(3) invert-intval
         using ‹int-signed-value (b::nat) (v::64 word) < (0::int)› assms(1) assms(2)
by blast
        then show ?thesis
        using ‹new-int (b::nat) (v'::64 word) = new-int b (− (v::64 word))› assms(1)
signed-eqiv zero by presburger
      qed
```

qed
then show *?thesis*
by *simp*
qed
qed

lemma *intval-abs-elims*:
assumes *intval-abs x* $\neq$ *UndefVal*
shows $\exists$ *t v* . *x* = *IntVal t v* $\wedge$
intval-abs x = *new-int t* (*if int-signed-value t v* < *0 then* − *v else v*)
by (*meson intval-abs.elims assms*)

lemma *wf-abs-new-int*:
assumes *intval-abs* (*IntVal t v*) $\neq$ *UndefVal*
shows *intval-abs* (*IntVal t v*) = *new-int t v* $\vee$ *intval-abs* (*IntVal t v*) = *new-int t* (−*v*)
by *simp*

lemma *mono-undef-abs*:
assumes *intval-abs* (*intval-abs x*) $\neq$ *UndefVal*
shows *intval-abs x* $\neq$ *UndefVal*
using *assms* **by** *force*

lemma *val-abs-idem*:
assumes *valid-value x* (*IntegerStamp b l h*)
assumes *val*[*abs*(*abs*(*x*))] $\neq$ *UndefVal*
shows *val*[*abs*(*abs*(*x*))] = *val*[*abs x*]
proof −
obtain *b v* **where** *in-def*: *x* = *IntVal b v*
using *assms intval-abs-elims mono-undef-abs* **by** *blast*
then have *bInRange*: *b* > *0* $\wedge$ *b* $\leq$ *64*
using *assms*(*1*)
by (*metis valid-stamp.simps*(*1*) *valid-value.simps*(*1*))
then show *?thesis*
proof (*cases int-signed-value b v* < *0*)
case *neg*: *True*
then show *?thesis*
proof (*cases v* = (*2*^(*b* − *1*)))
case *min*: *True*
then show *?thesis*
by (*smt* (*z3*) *assms*(*1*) *bInRange in-def intval-abs.simps*(*1*) *intval-negate.simps*(*1*) *negate-max-negative new-int.simps valid-value.simps*(*1*))
next
case *notMin*: *False*
then have *nested*: (*intval-abs x*) = *new-int b* (−*v*)
using *neg val-abs-neg in-def* **by** *simp*
also have *int-signed-value b* (−*v*) > *0*
using *neg notMin invert-intval bInRange*

by (*metis assms*(*1*) *in-def valid-value.simps*(*1*))
then have (*intval-abs* (*new-int b* (−*v*))) = *new-int b* (−*v*)
by (*smt* (*verit*, *best*) *ValueThms.signed-take-take-bit bInRange int-signed-value.simps intval-abs.simps*(*1*) *new-int.simps new-int-unused-bits-zero*)
then show *?thesis*
using *nested* **by** *presburger*
qed
next
case *False*
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) *assms*(*1*) *in-def intval-abs.simps*(*1*) *new-int.simps valid-value.simps*(*1*))
qed
qed

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*
by (*simp add*: *intval-add-sym*)

optimization *AddShiftConstantRight*: ((*const v*) + *y*) ⟼ *y* + (*const v*) *when* ¬(*is-ConstantExpr y*)
apply (*metis add-2-eq-Suc′ less-Suc-eq plus-1-eq-Suc size.simps*(*11*) *size-non-add*)
using *le-expr-def binadd-commute* **by** *blast*

optimization *AddShiftConstantRight2*: ((*const v*) + *y*) ⟼ *y* + (*const v*) *when* ¬(*is-ConstantExpr y*)
using *AddShiftConstantRight* **by** *auto*

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)
  by simp

lemma AddNeutral-Exp:
  shows exp[(e + (const (IntVal 32 0)))] ≥ exp[e]
  apply auto
  subgoal premises p for m p x
  proof -
    obtain ev where ev: [m,p] ⊢ e ↦ ev
      using p by auto
    then obtain b evx where evx: ev = IntVal b evx
    by (metis evalDet evaltree-not-undef intval-add.simps(3,4,5) intval-logic-negation.cases
         p(1,2))
    then have additionNotUndef: val[ev + (IntVal 32 0)] ≠ UndefVal
      using p evalDet ev by blast
    then have sameWidth: b = 32
      by (metis evx additionNotUndef intval-add.simps(1))
    then have unfolded: val[ev + (IntVal 32 0)] = IntVal 32 (take-bit 32 (evx+0))
      by (simp add: evx)
    then have eqE: IntVal 32 (take-bit 32 (evx+0)) = IntVal 32 (take-bit 32 (evx))
      by auto
    then show ?thesis
      by (metis ev evalDet eval-unused-bits-zero evx p(1) sameWidth unfolded)
  qed
  done

optimization AddNeutral: (e + (const (IntVal 32 0))) ⟼ e
  using AddNeutral-Exp by presburger

ML-val ‹@{term ‹x = y›}›

lemma NeutralLeftSubVal:
  assumes e1 = new-int b ival
  shows val[(e1 − e2) + e2] ≈ e1
  using assms by (cases e1; cases e2; auto)

lemma RedundantSubAdd-Exp:
  shows exp[((a − b) + b)] ≥ a
  apply auto
  subgoal premises p for m p y xa ya
  proof -
    obtain bv where bv: [m,p] ⊢ b ↦ bv
      using p(1) by auto
    obtain av where av: [m,p] ⊢ a ↦ av
      using p(3) by auto
    then have subNotUndef: val[av − bv] ≠ UndefVal
      by (metis bv evalDet p(3,4,5))
    then obtain bb bvv where bInt: bv = IntVal bb bvv
```

by (*metis bv evaltree-not-undef intval-logic-negation.cases intval-sub.simps*(*7*,*8*,*9*))
then obtain *ba avv* **where** *aInt*: *av* = *IntVal ba avv*
by (*metis av evaltree-not-undef intval-logic-negation.cases intval-sub.simps*(*3*,*4*,*5*) *subNotUndef*)
then have *widthSame*: *bb*=*ba*
by (*metis av bInt bv evalDet intval-sub.simps*(*1*) *new-int-bin.simps p*(*3*,*4*,*5*))
then have *valEval*: *val*[((*av*−*bv*)+*bv*)] = *val*[*av*]
using *aInt av eval-unused-bits-zero widthSame bInt* **by** *simp*
then show *?thesis*
by (*metis av bv evalDet p*(*1*,*3*,*4*))
qed
done

optimization *RedundantSubAdd*: ((e_1 − e_2) + e_2) ⟼ e_1
using *RedundantSubAdd-Exp* **by** *blast*

lemma *allE2*: (∀ *x y*. *P x y*) ⟹ (*P a b* ⟹ *R*) ⟹ *R*
by *simp*

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]) ≥ e_1
unfolding *le-expr-def unfold-binary bin-eval.simps* **by** (*metis assms evalDet evaltree-not-undef*)

optimization *RedundantSubAdd2*: e_2 + (e_1 − e_2) ⟼ e_1
using *size-binary-rhs-a* **apply** *simp* **apply** *auto*
by (*smt* (*z3*) *NeutralLeftSubVal evalDet eval-unused-bits-zero intval-add-sym intval-sub.elims new-int.simps well-formed-equal-defn*)

lemma *AddToSubHelperLowLevel*:
shows *val*[−*e* + *y*] = *val*[*y* − *e*] (**is** *?x* = *?y*)
by (*induction y*; *induction e*; *auto*)

print-phases

lemma *val-redundant-add-sub*:

assumes *a* = *new-int bb ival*
assumes *val*[*b* + *a*] ≠ *UndefVal*
shows *val*[(*b* + *a*) − *b*] = *a*
using *assms* **apply** (*cases a*; *cases b*; *auto*) **by** *presburger*

lemma *val-add-right-negate-to-sub*:
assumes *val*[*x* + *e*] ≠ *UndefVal*
shows *val*[*x* + (−*e*)] = *val*[*x* − *e*]
by (*cases x*; *cases e*; *auto simp*: *assms*)

lemma *exp-add-left-negate-to-sub*:
exp[−*e* + *y*] ≥ *exp*[*y* − *e*]
by (*cases e*; *cases y*; *auto simp*: *AddToSubHelperLowLevel*)

lemma *RedundantAddSub-Exp*:
shows *exp*[(*b* + *a*) − *b*] ≥ *a*
apply *auto*
subgoal premises *p* **for** *m p y xa ya*
proof −
obtain *bv* **where** *bv*: [*m*,*p*] ⊢ *b* ↦ *bv*
using *p*(*1*) **by** *auto*
obtain *av* **where** *av*: [*m*,*p*] ⊢ *a* ↦ *av*
using *p*(*4*) **by** *auto*
then have *addNotUndef*: *val*[*av* + *bv*] ≠ *UndefVal*
by (*metis bv evalDet intval-add-sym intval-sub.simps*(*2*) *p*(*2*,*3*,*4*))
then obtain *bb bvv* **where** *bInt*: *bv* = *IntVal bb bvv*
by (*metis bv evalDet evaltree-not-undef intval-add.simps*(*3*,*5*) *intval-logic-negation.cases intval-sub.simps*(*8*) *p*(*1*,*2*,*3*,*5*))
then obtain *ba avv* **where** *aInt*: *av* = *IntVal ba avv*
by (*metis addNotUndef intval-add.simps*(*2*,*3*,*4*,*5*) *intval-logic-negation.cases*)
then have *widthSame*: *bb*=*ba*
by (*metis addNotUndef bInt intval-add.simps*(*1*))
then have *valEval*: *val*[((*bv*+*av*)−*bv*)] = *val*[*av*]
using *aInt av eval-unused-bits-zero widthSame bInt* **by** *simp*
then show *?thesis*
by (*metis av bv evalDet p*(*1*,*3*,*4*))
qed
done

Optimisations

optimization *RedundantAddSub*: (*b* + *a*) − *b* ⟼ *a*
using *RedundantAddSub-Exp* **by** *blast*

optimization *AddRightNegateToSub*: *x* + −*e* ⟼ *x* − *e*
apply (*metis Nat.add-0-right add-2-eq-Suc′ add-less-mono1 add-mono-thms-linordered-field*(*2*) *less-SucI not-less-less-Suc-eq size-binary-const size-non-add size-pos*)
using *AddToSubHelperLowLevel intval-add-sym* **by** *auto*

optimization *AddLeftNegateToSub*: $-e + y \longmapsto y - e$
apply (*smt* (*verit*, *best*) *One-nat-def add.commute add-Suc-right is-ConstantExpr-def less-add-Suc2*
numeral-2-eq-2 plus-1-eq-Suc size.simps(*1*) *size.simps*(*11*) *size-binary-const size-non-add*)
using *exp-add-left-negate-to-sub* **by** *simp*

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*]
using *assms* **apply** (*cases x*; *cases y*; *auto*) **by** (*simp add*: *and.commute*)

lemma *AndCommute-Exp*:
shows *exp*[*x* & *y*] $\geq$ *exp*[*y* & *x*]
using *AndCommute-Val unfold-binary* **by** *auto*

lemma *AndRightFallthrough*: (((*and* (*not* ($\downarrow$ *x*)) ($\uparrow$ *y*)) = *0*)) $\longrightarrow$ *exp*[*x* & *y*] $\geq$ *exp*[*y*]
apply *simp* **apply** (*rule impI*; (*rule allI*)+; *rule impI*)
subgoal premises *p* **for** *m p v*
proof –
obtain *xv* **where** *xv*: [*m*, *p*] $\vdash$ *x* $\mapsto$ *xv*
using *p*(*2*) **by** *blast*
obtain *yv* **where** *yv*: [*m*, *p*] $\vdash$ *y* $\mapsto$ *yv*
using *p*(*2*) **by** *blast*
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis bin-eval-inputs-are-ints bin-eval-int evalDet is-IntVal-def p*(*2*) *unfold-binary xv*)
obtain *yb yvv* **where** *yvv*: *yv* = *IntVal yb yvv*
by (*metis bin-eval-inputs-are-ints bin-eval-int evalDet is-IntVal-def p*(*2*)

unfold-binary yv)
    have *equalAnd*: *v* = *val*[*xv* & *yv*]
      by (*metis BinaryExprE bin-eval.simps*(*6*) *evalDet p*(*2*) *xv yv*)
   then have *andUnfold*: *val*[*xv* & *yv*] = (*if xb*=*yb then new-int xb* (*and xvv yvv*) *else UndefVal*)
      by (*simp add*: *xvv yvv*)
    have *v* = *yv*
      apply (*cases v*; *cases yv*; *auto*)
      using *p*(*2*) **apply** *auto*[*1*] **using** *yvv* **apply** *simp-all*
    by (*metis Value.distinct*(*1*,*3*,*5*,*7*,*9*,*11*,*13*) *Value.inject*(*1*) *andUnfold equalAnd new-int.simps*
      xv xvv yv eval-unused-bits-zero new-int.simps not-down-up-mask-and-zero-implies-zero
        equalAnd p(*1*))+
    then show *?thesis*
      by (*simp add*: *yv*)
  qed
 done

lemma *AndLeftFallthrough*: (((*and* (*not* (↓ *y*)) (↑ *x*)) = *0*)) ⟶ *exp*[*x* & *y*] ≥ *exp*[*x*]
 using *AndRightFallthrough AndCommute-Exp* **by** *simp*

end

phase *AndNode*
 terminating *size*
begin

lemma *bin-and-nots*:
 (~*x* & ~*y*) = (~(*x* | *y*))
 by *simp*

lemma *bin-and-neutral*:
 (*x* & ~*False*) = *x*
 by *simp*

lemma *val-and-equal*:
 assumes *x* = *new-int b v*
 and *val*[*x* & *x*] ≠ *UndefVal*
 shows *val*[*x* & *x*] = *x*
 by (*auto simp*: *assms*)

lemma *val-and-nots*:
 val[~*x* & ~*y*] = *val*[~(*x* | *y*)]
 by (*cases x*; *cases y*; *auto simp*: *take-bit-not-take-bit*)

lemma *val-and-neutral*:

assumes *x* = *new-int b v*
and *val*[*x* & ~(*new-int b′ 0*)] ≠ *UndefVal*
shows *val*[*x* & ~(*new-int b′ 0*)] = *x*
using *assms* **apply** (*simp add*: *take-bit-eq-mask*) **by** *presburger*

lemma *val-and-zero*:
assumes *x* = *new-int b v*
shows *val*[*x* & (*IntVal b 0*)] = *IntVal b 0*
by (*auto simp*: *assms*)

lemma *exp-and-equal*:
exp[*x* & *x*] ≥ *exp*[*x*]
apply *auto*
subgoal premises *p* **for** *m p xv yv*
proof–
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*1*) **by** *auto*
obtain *yv* **where** *yv*: [*m*,*p*] ⊢ *x* ↦ *yv*
using *p*(*1*) **by** *auto*
then have *evalSame*: *xv* = *yv*
using *evalDet xv* **by** *auto*
then have *notUndef*: *xv* ≠ *UndefVal* ∧ *yv* ≠ *UndefVal*
using *evaltree-not-undef xv* **by** *blast*
then have *andNotUndef*: *val*[*xv* & *yv*] ≠ *UndefVal*
by (*metis evalDet evalSame p*(*1*,*2*,*3*) *xv*)
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis Value.exhaust-sel andNotUndef evalSame intval-and.simps*(*3*,*4*,*9*) *notUndef*)
obtain *yb yvv* **where** *yvv*: *yv* = *IntVal yb yvv*
using *evalSame xvv* **by** *auto*
then have *widthSame*: *xb*=*yb*
using *evalSame xvv* **by** *auto*
then have *valSame*: *yvv*=*xvv*
using *evalSame xvv yvv* **by** *blast*
then have *evalSame0*: *val*[*xv* & *yv*] = *new-int xb* (*xvv*)
using *evalSame xvv* **by** *auto*
then show *?thesis*
by (*metis eval-unused-bits-zero new-int.simps evalDet p*(*1*,*2*) *valSame widthSame xv xvv yvv*)
qed
done

lemma *exp-and-nots*:
exp[~*x* & ~*y*] ≥ *exp*[~(*x* | *y*)]

```
  using val-and-nots by force

lemma exp-sign-extend:
  assumes e = (1 << In) − 1
  shows  BinaryExpr BinAnd (UnaryExpr (UnarySignExtend In Out) x)
                          (ConstantExpr (new-int b e))
                        ≥ (UnaryExpr (UnaryZeroExtend In Out) x)
  apply auto
  subgoal premises p for m p va
    proof −
      obtain va where va: [m,p] ⊢ x ↦ va
        using p(2) by auto
      then have notUndef: va ≠ UndefVal
        by (simp add: evaltree-not-undef)
      then have 1: intval-and (intval-sign-extend In Out va) (IntVal b (take-bit b
e)) ≠ UndefVal
        using evalDet p(1) p(2) va by blast
      then have 2: intval-sign-extend In Out va ≠ UndefVal
        by auto
      then have 21: (0::nat) < b
        using eval-bits-1-64 p(4) by blast
      then have 3: b ⊑ (64::nat)
        using eval-bits-1-64 p(4) by blast
      then have 4: − ((2::int) ^ b div (2::int)) ⊑ sint (signed-take-bit (b − Suc
(0::nat)) (take-bit b e))
     by (simp add: 21 int-power-div-base signed-take-bit-int-greater-eq-minus-exp-word)
     then have 5: sint (signed-take-bit (b − Suc (0::nat)) (take-bit b e)) < (2::int)
^ b div (2::int)
     by (simp add: 21 3 Suc-le-lessD int-power-div-base signed-take-bit-int-less-exp-word)
      then have 6: [m,p] ⊢ UnaryExpr (UnaryZeroExtend In Out)
              x ↦ intval-and (intval-sign-extend In Out va) (IntVal b (take-bit b e))
        apply (cases va; simp)
        apply (simp add: notUndef) defer
        using 2 apply fastforce+
        sorry
      then show ?thesis
        by (metis evalDet p(2) va)
    qed
  done

lemma exp-and-neutral:
  assumes wf-stamp x
  assumes stamp-expr x = IntegerStamp b lo hi
  shows exp[(x & ~(const (IntVal b 0)))] ≥ x
  using assms apply auto
  subgoal premises p for m p xa
  proof−
    obtain xv where xv: [m,p] ⊢ x ↦ xv
      using p(3) by auto
```

obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis assms valid-int wf-stamp-def xv*)
then have *widthSame*: *xb*=*b*
by (*metis p*(*1*,*2*) *valid-int-same-bits wf-stamp-def xv*)
then show *?thesis*
by (*metis evalDet eval-unused-bits-zero intval-and.simps*(*1*) *new-int.elims new-int-bin.elims*
p(*3*) *take-bit-eq-mask xv xvv*)
qed
done

lemma *val-and-commute*[*simp*]:
val[*x* & *y*] = *val*[*y* & *x*]
by (*cases x*; *cases y*; *auto simp*: *word-bw-comms*(*1*))

Optimisations

optimization *AndEqual*: *x* & *x* ⟼ *x*
using *exp-and-equal* **by** *blast*

optimization *AndShiftConstantRight*: ((*const x*) & *y*) ⟼ *y* & (*const x*)
when ¬(*is-ConstantExpr y*)
using *size-flip-binary* **by** *auto*

optimization *AndNots*: (~*x*) & (~*y*) ⟼ ~(*x* | *y*)
by (*metis add-2-eq-Suc′ less-SucI less-add-Suc1 not-less-eq size-binary-const size-non-add exp-and-nots*)+

optimization *AndSignExtend*: *BinaryExpr BinAnd* (*UnaryExpr* (*UnarySignExtend In Out*) (*x*))
(*const* (*new-int b e*))
⟼ (*UnaryExpr* (*UnaryZeroExtend In Out*) (*x*))
when (*e* = (*1* << *In*) − *1*)
using *exp-sign-extend* **by** *simp*

optimization *AndNeutral*: (*x* & ~(*const* (*IntVal b 0*))) ⟼ *x*
when (*wf-stamp x* ∧ *stamp-expr x* = *IntegerStamp b lo hi*)
using *exp-and-neutral* **by** *fast*

optimization *AndRightFallThrough*: (*x* & *y*) ⟼ *y*
when (((*and* (*not* (*IRExpr-down x*)) (*IRExpr-up y*)) = *0*))
by (*simp add*: *IRExpr-down-def IRExpr-up-def*)

optimization *AndLeftFallThrough*: (*x* & *y*) ⟼ *x*
when (((*and* (*not* (*IRExpr-down y*)) (*IRExpr-up x*)) = *0*))
by (*simp add*: *IRExpr-down-def IRExpr-up-def*)

end

end

11.4 BinaryNode Phase

```
theory BinaryNode
  imports
    Common
begin

phase BinaryNode
  terminating size
begin

optimization BinaryFoldConstant: BinaryExpr op (const v1) (const v2) ⟼ ConstantExpr (bin-eval op v1 v2)
  unfolding le-expr-def
  apply (rule allI impI)+
  subgoal premises bin for m p v
    apply (rule BinaryExprE[OF bin])
    subgoal premises prems for x y
    proof −
      have x: x = v1
        using prems by auto
      have y: y = v2
        using prems by auto
      have xy: v = bin-eval op x y
        by (simp add: prems x y)
      have int: ∃ b vv . v = new-int b vv
        using bin-eval-new-int prems by fast
      show ?thesis
         by (metis ConstantExpr prems(1) x y int bin eval-bits-1-64 new-int.simps
new-int-take-bits
            wf-value-def validDefIntConst)
      qed
    done
  done

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*]))
  by (*metis* (*mono-tags*, *lifting*) *intval-logic-negation.simps*(*1*) *logic-negate-def new-int.simps*

    of-bool-eq(*2*) *one-neq-zero take-bit-of-0 take-bit-of-1 val-to-bool.simps*(*1*))

lemma *negation-condition-intval*:
  assumes *e* = *IntVal b ie*
  assumes *0* < *b*
  shows *val*[(!*e*) *?* *x* : *y*] = *val*[*e* *?* *y* : *x*]
  by (*metis assms intval-conditional.simps negates*)

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'*]
  using *assms* **by** *auto*

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*
  by (*metis assms eval-bits-1-64 intval-logic-negation.elims negation-preserve-eval unfold-unary*)

optimization *NegateConditionFlipBranches*: ((!*e*) *?* *x* : *y*) $\longmapsto$ (*e* *?* *y* : *x*)
  apply *simp* **apply** (*rule allI*; *rule allI*; *rule allI*; *rule impI*)
  subgoal premises *p* **for** *m p v*
  proof −
    obtain *ev* **where** *ev*: [*m*,*p*] $\vdash$ *e* $\mapsto$ *ev*
      using *p* **by** *blast*
    obtain *notEv* **where** *notEv*: *notEv* = *intval-logic-negation ev*
      by *simp*
    obtain *lhs* **where** *lhs*: [*m*,*p*] $\vdash$ *ConditionalExpr* (*UnaryExpr UnaryLogicNegation e*) *x y* $\mapsto$ *lhs*
      using *p* **by** *auto*
    obtain *xv* **where** *xv*: [*m*,*p*] $\vdash$ *x* $\mapsto$ *xv*
      using *lhs* **by** *blast*
    obtain *yv* **where** *yv*: [*m*,*p*] $\vdash$ *y* $\mapsto$ *yv*
      using *lhs* **by** *blast*
    then show *?thesis*
      by (*smt* (*z3*) *le-expr-def ConditionalExpr ConditionalExprE Value.distinct*(*1*) *evalDet negates p*
        negation-preserve-eval negation-preserve-eval-intval)
  qed

done

optimization *DefaultTrueBranch*: (*true* ? *x* : *y*) $\longmapsto$ *x* .

optimization *DefaultFalseBranch*: (*false* ? *x* : *y*) $\longmapsto$ *y* .

optimization *ConditionalEqualBranches*: (*e* ? *x* : *x*) $\longmapsto$ *x* .

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*)
using *stamp-under-defn* **by** *fastforce*

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*)
using *stamp-under-defn-inverse* **by** *fastforce*

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*]
using *assms* **apply** *auto*
apply (*metis* (*full-types*) *bool-to-val.simps*(*2*) *val-to-bool.simps*(*1*))
by (*metis* (*mono-tags*, *lifting*) *bool-to-val.simps*(*1*) *val-to-bool.simps*(*1*) *even-iff-mod-2-eq-zero odd-iff-mod-2-eq-one and-one-eq*)

optimization *ConditionalEliminateKnownLess*: ((*x* < *y*) ? *x* : *y*) $\longmapsto$ *x*
when (*stamp-under* (*stamp-expr x*) (*stamp-expr y*)
∧ *wf-stamp x* ∧ *wf-stamp y*)
using *stamp-under-defn* **by** *fastforce*

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*
using *assms* **by** (*simp add*: *IRTreeEvalThms.valid-value-elims*(*3*))

lemma *intval-self-is-true*:
assumes *yv* ≠ *UndefVal*
assumes *yv* = *IntVal b yvv*
shows *intval-equals yv yv* = *IntVal 32 1*
using *assms* **by** (*cases yv*; *auto*)

lemma *intval-commute*:
  assumes *intval-equals yv xv* ≠ *UndefVal*
  assumes *intval-equals xv yv* ≠ *UndefVal*
  shows *intval-equals yv xv* = *intval-equals xv yv*
  using *assms* **apply** (*cases yv*; *cases xv*; *auto*) **by** (*smt* (*verit*, *best*))

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*]
  using *assms isBoolean-def* **apply** *auto*
 by (*metis* (*no-types*, *lifting*) *IntVal0 IntVal1 intval-logic-negation.simps*(*1*) *logic-negate-def*)

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
  apply (*metis Canonicalization.cond-size add-lessD1 size-binary-lhs*) **apply** *auto*
  subgoal premises *p* **for** *m p cExpr xv cond*
  proof −
    obtain *cond* **where** *cond*: [*m*,*p*] ⊢ *c* ↦ *cond*
      using *p* **by** *blast*
    have *cRange*: *cond* = *IntVal 32 0* ∨ *cond* = *IntVal 32 1*
      using *p cond isBoolean-def* **by** *blast*
    then obtain *yv* **where** *yVal*: [*m*,*p*] ⊢ *y* ↦ *yv*
      using *p*(*15*) **by** *auto*
    obtain *xvv* **where** *xvv*: *xv* = *IntVal b xvv*
      by (*metis p*(*1*,*2*,*7*) *valid-int wf-stamp-def*)
    obtain *yvv* **where** *yvv*: *yv* = *IntVal b yvv*
      by (*metis ExpIntBecomesIntVal p*(*3*,*4*) *wf-stamp-def yVal*)
    have *yxDiff*: *xvv* ≠ *yvv*
      by (*smt* (*verit*, *del-insts*) *yVal xvv wf-stamp-def valid-int-signed-range p yvv*)
    have *eqEvalFalse*: *intval-equals yv xv* = (*IntVal 32 0*)
       unfolding *xvv yvv* **apply** *auto* **by** (*metis* (*mono-tags*) *bool-to-val.simps*(*2*) *yxDiff*)
    then have *valEvalSame*: *cond* = *intval-equals val*[*cond* ? *xv* : *yv*] *xv*
      apply (*cases cond* = *IntVal 32 0*; *simp*) **using** *cRange xvv* **by** *auto*
    then have *condTrue*: *val-to-bool cond* ⟹ *cExpr* = *xv*
      by (*metis* (*mono-tags*, *lifting*) *cond evalDet p*(*11*) *p*(*7*) *p*(*9*))
    then have *condFalse*: ¬(*val-to-bool cond*) ⟹ *cExpr* = *yv*

by (*metis* (*full-types*) *cond evalDet p*(*11*) *p*(*9*) *yVal*)
then have $[m,p] \vdash c \mapsto$ *intval-equals cExpr xv*
using *cond condTrue valEvalSame* **by** *fastforce*
then show *?thesis*
by *blast*
qed
done

lemma *negation-preserve-eval0*:
assumes $[m, p] \vdash exp[e] \mapsto v$
assumes *isBoolean e*
shows $\exists v'. ([m, p] \vdash exp[!e] \mapsto v')$
using *assms*
proof –
obtain *b vv* **where** *vIntVal*: $v = IntVal\ b\ vv$
using *isBoolean-def assms* **by** *blast*
then have *negationDefined*: *intval-logic-negation* $v \neq$ *UndefVal*
by *simp*
show *?thesis*
using *assms*(*1*) *negationDefined* **by** *fastforce*
qed

lemma *negation-preserve-eval2*:
assumes $([m, p] \vdash exp[e] \mapsto v)$
assumes (*isBoolean e*)
shows $\exists v'. ([m, p] \vdash exp[!e] \mapsto v') \wedge v = val[!v']$
using *assms*
proof –
obtain *notEval* **where** *notEval*: $([m, p] \vdash exp[!e] \mapsto notEval)$
by (*metis assms negation-preserve-eval0*)
then have *logicNegateEquiv*: *notEval* = *intval-logic-negation v*
using *evalDet assms*(*1*) *unary-eval.simps*(*4*) **by** *blast*
then have *vRange*: $v = IntVal\ 32\ 0 \vee v = IntVal\ 32\ 1$
using *assms* **by** (*auto simp add*: *isBoolean-def*)
have *evaluateNot*: *v* = *intval-logic-negation notEval*
by (*metis IntVal0 IntVal1 intval-logic-negation.simps*(*1*) *logicNegateEquiv logic-negate-def vRange*)
then show *?thesis*
using *notEval* **by** *auto*
qed

optimization *ConditionalIntegerEquals-2*: *exp*[*BinaryExpr BinIntegerEquals* (*c ? x : y*) (*y*)] $\longmapsto$ (!*c*)
when stamp-expr x = *IntegerStamp b xl xh* $\wedge$
wf-stamp x $\wedge$
stamp-expr y = *IntegerStamp b yl yh* $\wedge$
wf-stamp y $\wedge$
(*alwaysDistinct* (*stamp-expr x*) (*stamp-expr*

y)) ∧
isBoolean c
apply (*smt* (*verit*) *not-add-less1 max-less-iff-conj max.absorb3 linorder-less-linear add-2-eq-Suc′*
add-less-cancel-right size-binary-lhs add-lessD1 Canonicalization.cond-size)
apply *auto*
subgoal premises *p* **for** *m p cExpr yv cond trE faE*
proof −
obtain *cond* **where** *cond*: [*m*,*p*] ⊢ *c* ↦ *cond*
using *p* **by** *blast*
then have *condNotUndef*: *cond* ≠ *UndefVal*
by (*simp add*: *evaltree-not-undef*)
then obtain *notCond* **where** *notCond*: [*m*,*p*] ⊢ *exp*[!*c*] ↦ *notCond*
by (*meson p*(*6*) *negation-preserve-eval2 cond*)
have *cRange*: *cond* = *IntVal 32 0* ∨ *cond* = *IntVal 32 1*
using *p cond* **by** (*simp add*: *isBoolean-def*)
then have *cNotRange*: *notCond* = *IntVal 32 0* ∨ *notCond* = *IntVal 32 1*
by (*metis* (*no-types*, *lifting*) *IntVal0 IntVal1 cond evalDet intval-logic-negation.simps*(*1*)
logic-negate-def negation-preserve-eval notCond)
then obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p* **by** *auto*
then have *trueCond*: (*notCond* = *IntVal 32 1*) ⟹ [*m*,*p*] ⊢ (*ConditionalExpr c x y*) ↦ *yv*
by (*smt* (*verit*, *best*) *cRange evalDet negates negation-preserve-eval notCond p*(*7*) *cond*
zero-less-numeral val-to-bool.simps(*1*) *evaltree-not-undef ConditionalExpr ConditionalExprE*)
obtain *xvv* **where** *xvv*: *xv* = *IntVal b xvv*
by (*metis p*(*1*,*2*) *valid-int wf-stamp-def xv*)
then have *opposites*: *notCond* = *intval-logic-negation cond*
by (*metis cond evalDet negation-preserve-eval notCond*)
then have *negate*: (*intval-logic-negation cond* = *IntVal 32 0*) ⟹ (*cond* = *IntVal 32 1*)
using *cRange intval-logic-negation.simps negates* **by** *fastforce*
have *falseCond*: (*notCond* = *IntVal 32 0*) ⟹ [*m*,*p*] ⊢ (*ConditionalExpr c x y*) ↦ *xv*
unfolding *opposites* **using** *negate cond evalDet p*(*13*,*14*,*15*,*16*) *xv* **by** *auto*
obtain *yvv* **where** *yvv*: *yv* = *IntVal b yvv*
by (*metis p*(*3*,*4*,*7*) *wf-stamp-def ExpIntBecomesIntVal*)
have *yxDiff*: *xv* ≠ *yv*
by (*metis linorder-not-less max.absorb1 max.absorb4 max-less-iff-conj min-def xv yvv*
wf-stamp-def valid-int-signed-range p(*1*,*2*,*3*,*4*,*5*,*7*))
then have *trueEvalCond*: (*cond* = *IntVal 32 0*) ⟹
[*m*,*p*] ⊢ *exp*[*BinaryExpr BinIntegerEquals* (*c ? x : y*) (*y*)]
↦ *intval-equals yv yv*
by (*smt* (*verit*) *cNotRange trueCond ConditionalExprE cond bin-eval.simps*(*13*) *evalDet p*
falseCond unfold-binary val-to-bool.simps(*1*))

then have *falseEval*: (*notCond* = *IntVal 32 0*) ⟹
[*m*,*p*] ⊢ *exp*[*BinaryExpr BinIntegerEquals* (*c* ? *x* : *y*) (*y*)]
↦ *intval-equals xv yv*
using *p* **by** (*metis ConditionalExprE bin-eval.simps*(*13*) *evalDet falseCond unfold-binary*)
have *eqEvalFalse*: *intval-equals yv xv* = (*IntVal 32 0*)
unfolding *xvv yvv* **apply** *auto* **by** (*metis* (*mono-tags*) *bool-to-val.simps*(*2*) *yxDiff yvv xvv*)
have *trueEvalEquiv*: [*m*,*p*] ⊢ *exp*[*BinaryExpr BinIntegerEquals* (*c* ? *x* : *y*) (*y*)] ↦ *notCond*
apply (*cases notCond*) **prefer** *2*
apply (*metis IntVal0 Value.distinct*(*1*) *eqEvalFalse evalDet evaltree-not-undef falseEval p*(*6*)
intval-commute intval-logic-negation.simps(*1*) *intval-self-is-true logic-negate-def*
negation-preserve-eval2 notCond trueEvalCond yvv cNotRange cond)
using *notCond cNotRange* **by** *auto*
show *?thesis*
using *ConditionalExprE*
by (*metis cNotRange falseEval notCond trueEvalEquiv trueCond falseCond intval-self-is-true*
yvv p(*9*,*11*) *evalDet*)
qed
done

optimization *ConditionalExtractCondition*: *exp*[(*c* ? *true* : *false*)] ⟼ *c*
when isBoolean c
using *isBoolean-def* **by** *fastforce*

optimization *ConditionalExtractCondition2*: *exp*[(*c* ? *false* : *true*)] ⟼ !*c*
when isBoolean c
apply *auto*
subgoal premises *p* **for** *m p cExpr cond*
proof−
obtain *cond* **where** *cond*: [*m*,*p*] ⊢ *c* ↦ *cond*
using *p*(*2*) **by** *auto*
obtain *notCond* **where** *notCond*: [*m*,*p*] ⊢ *exp*[!*c*] ↦ *notCond*
by (*metis cond negation-preserve-eval2 p*(*1*))
then have *cRange*: *cond* = *IntVal 32 0* ∨ *cond* = *IntVal 32 1*
using *isBoolean-def cond p*(*1*) **by** *auto*
then have *cExprRange*: *cExpr* = *IntVal 32 0* ∨ *cExpr* = *IntVal 32 1*
by (*metis* (*full-types*) *ConstantExprE p*(*4*))
then have *condTrue*: *cond* = *IntVal 32 1* ⟹ *cExpr* = *IntVal 32 0*
using *cond evalDet p*(*2*) *p*(*4*) **by** *fastforce*
then have *condFalse*: *cond* = *IntVal 32 0* ⟹ *cExpr* = *IntVal 32 1*
using *p cond evalDet* **by** *fastforce*
then have *opposite*: *cond* = *intval-logic-negation cExpr*
by (*metis* (*full-types*) *IntVal0 IntVal1 cRange condTrue intval-logic-negation.simps*(*1*)
logic-negate-def)
then have *eq*: *notCond* = *cExpr*

by (*metis* (*no-types*, *lifting*) *IntVal0 IntVal1 cExprRange cond evalDet negation-preserve-eval*
intval-logic-negation.simps(*1*) *logic-negate-def notCond*)
then show *?thesis*
using *notCond* **by** *auto*
qed
done

optimization *ConditionalEqualIsRHS*: ((*x eq y*) *? x* : *y*) ⟼ *y*
apply *auto*
subgoal premises *p* **for** *m p v true false xa ya*
proof−
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*8*) **by** *auto*
obtain *yv* **where** *yv*: [*m*,*p*] ⊢ *y* ↦ *yv*
using *p*(*9*) **by** *auto*
have *notUndef*: *xv* ≠ *UndefVal* ∧ *yv* ≠ *UndefVal*
using *evaltree-not-undef xv yv* **by** *blast*
have *evalNotUndef*: *intval-equals xv yv* ≠ *UndefVal*
by (*metis evalDet p*(*1*,*8*,*9*) *xv yv*)
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis Value.exhaust evalNotUndef intval-equals.simps*(*3*,*4*,*5*) *notUndef*)
obtain *yb yvv* **where** *yvv*: *yv* = *IntVal yb yvv*
by (*metis evalNotUndef intval-equals.simps*(*7*,*8*,*9*) *intval-logic-negation.cases notUndef*)
obtain *vv* **where** *evalLHS*: [*m*,*p*] ⊢ *if val-to-bool* (*intval-equals xv yv*) *then x else y* ↦ *vv*
by (*metis* (*full-types*) *p*(*4*) *yv*)
obtain *equ* **where** *equ*: *equ* = *intval-equals xv yv*
by *fastforce*
have *trueEval*: *equ* = *IntVal 32 1* ⟹ *vv* = *xv*
using *evalLHS* **by** (*simp add*: *evalDet xv equ*)
have *falseEval*: *equ* = *IntVal 32 0* ⟹ *vv* = *yv*
using *evalLHS* **by** (*simp add*: *evalDet yv equ*)
then have *vv* = *v*
by (*metis evalDet evalLHS p*(*2*,*8*,*9*) *xv yv*)
then show *?thesis*
by (*metis* (*full-types*) *bool-to-val.simps*(*1*,*2*) *bool-to-val-bin.simps equ evalNotUndef falseEval*
intval-equals.simps(*1*) *trueEval xvv yv yvv*)
qed
done

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
apply *auto*

```
  subgoal premises p for m p v
    proof -
      obtain xa where xa: [m,p] ⊢ x ↦ xa
        using p by blast
       have eval: [m,p] ⊢ if val-to-bool (intval-equals xa (IntVal 32 0))
                        then ConstantExpr (IntVal 32 0)
                        else ConstantExpr (IntVal 32 1) ↦ v
         using evalDet p(3,4,5,6,7) xa by blast
       then have xaRange: xa = IntVal 32 0 ∨ xa = IntVal 32 1
         using isBoolean-def p(3) xa by blast
      then have 6: v = xa
        using eval xaRange by auto
      then show ?thesis
        by (auto simp: xa)
    qed
  done


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))) .


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))) .


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))) .

lemma stamp-of-default:
  assumes stamp-expr x = default-stamp
  assumes wf-stamp x
  shows ([m, p] ⊢ x ↦ v) ⟶ (∃ vv. v = IntVal 32 vv)
  by (metis assms default-stamp valid-value-elims(3) wf-stamp-def)

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)
  apply (simp; rule impI; (rule allI)+; rule impI)
```

```
  subgoal premises eval for m p v
proof -
  obtain xv where xv: [m, p] ⊢ x ↦ xv
    using eval by fast
  then have x32: ∃ v. xv = IntVal 32 v
    using stamp-of-default eval by auto
  obtain lhs where lhs: [m, p] ⊢ exp[(((x & (const (IntVal 32 1))) eq (const (IntVal
32 0))) ?
                                (const (IntVal 32 0)) : (const (IntVal 32 1)))] ↦ lhs
    using eval(2) by auto
  then have lhsV: lhs = val[((xv & (IntVal 32 1)) eq (IntVal 32 0)) ?
                        (IntVal 32 0) : (IntVal 32 1)]
     using ConditionalExprE ConstantExprE bin-eval.simps(4,11) evalDet xv un-
fold-binary
          intval-conditional.simps
    by fastforce
  obtain rhs where rhs: [m, p] ⊢ exp[x & (const (IntVal 32 1))] ↦ rhs
    using eval(2) by blast
  then have rhsV: rhs = val[xv & IntVal 32 1]
    by (metis BinaryExprE ConstantExprE bin-eval.simps(6) evalDet xv)
  have lhs = rhs
    using val-optimise-integer-test x32 lhsV rhsV by presburger
  then show ?thesis
    by (metis eval(2) evalDet lhs rhs)
qed
  done


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))) .


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*
by *simp*

lemma *bin-multiply-identity*:
(*x* :: *'a*::*len word*) ∗ *1* = *x*
by *simp*

lemma *bin-multiply-eliminate*:
(*x* :: *'a*::*len word*) ∗ *0* = *0*
by *simp*

lemma *bin-multiply-negative*:
(*x* :: *'a*::*len word*) ∗ *uminus 1* = *uminus x*
by *simp*

lemma *bin-multiply-power-2*:
(*x*:: *'a*::*len word*) ∗ (*2*^*j*) = *x* << *j*
by *simp*

lemma *take-bit64*[*simp*]:
fixes *w* :: *int64*
shows *take-bit 64 w* = *w*
proof −
have *Nat.size w* = *64*
by (*simp add*: *size64*)
then show *?thesis*
by (*metis lt2p-lem mask-eq-iff take-bit-eq-mask verit-comp-simplify1*(*2*) *wsst-TYs*(*3*))

qed

lemma *mergeTakeBit*:
  fixes *a* :: *nat*
  fixes *b c* :: *64 word*
  shows *take-bit a* (*take-bit a* (*b*) ∗ *take-bit a* (*c*)) =
      take-bit a (*b* ∗ *c*)
 by (*smt* (*verit*, *ccfv-SIG*) *take-bit-mult take-bit-of-int unsigned-take-bit-eq word-mult-def*)

lemma *val-eliminate-redundant-negative*:
  assumes *val*[−*x* ∗ −*y*] ≠ *UndefVal*
  shows *val*[−*x* ∗ −*y*] = *val*[*x* ∗ *y*]
  by (*cases x*; *cases y*; *auto simp*: *mergeTakeBit*)

lemma *val-multiply-neutral*:
  assumes *x* = *new-int b v*
  shows *val*[*x* ∗ (*IntVal b 1*)] = *x*
  by (*auto simp*: *assms*)

lemma *val-multiply-zero*:
  assumes *x* = *new-int b v*
  shows *val*[*x* ∗ (*IntVal b 0*)] = *IntVal b 0*
  by (*simp add*: *assms*)

lemma *val-multiply-negative*:
  assumes *x* = *new-int b v*
  shows *val*[*x* ∗ −(*IntVal b 1*)] = *val*[−*x*]
  unfolding *assms*(*1*) **apply** *auto*
  by (*metis bin-multiply-negative mergeTakeBit take-bit-minus-one-eq-mask*)

lemma *val-MulPower2*:
  fixes *i* :: *64 word*
  assumes *y* = *IntVal 64* (*2* ^ *unat*(*i*))
  and   *0* < *i*
  and   *i* < *64*
  and   *val*[*x* ∗ *y*] ≠ *UndefVal*
  shows *val*[*x* ∗ *y*] = *val*[*x* << *IntVal 64 i*]
  using *assms* **apply** (*cases x*; *cases y*; *auto*)
   subgoal premises *p* **for** *x2*
   proof −
    have *63*: (*63* :: *int64*) = *mask 6*
     by *eval*
    then have (*2*::*int*) ^ *6* = *64*
     by *eval*
    then have *uint i* < (*2*::*int*) ^ *6*
       by (*metis linorder-not-less lt2p-lem of-int-numeral p*(*4*) *word-2p-lem*

word-of-int-2p
wsst-TYs(*3*))
then have *and i* (*mask 6*) = *i*
using *mask-eq-iff* **by** *blast*
then show *x2* << *unat i* = *x2* << *unat* (*and i* (*63*::*64 word*))
by (*auto simp*: *63*)
qed
by *presburger*

lemma *val-MulPower2Add1*:
fixes *i* :: *64 word*
assumes *y* = *IntVal 64* ((*2* ^ *unat*(*i*)) + *1*)
and *0* < *i*
and *i* < *64*
and *val-to-bool*(*val*[*IntVal 64 0* < *x*])
and *val-to-bool*(*val*[*IntVal 64 0* < *y*])
shows *val*[*x* ∗ *y*] = *val*[(*x* << *IntVal 64 i*) + *x*]
using *assms* **apply** (*cases x*; *cases y*; *auto*)
subgoal premises *p* **for** *x2*
proof −
have *63*: (*63* :: *int64*) = *mask 6*
by *eval*
then have (*2* :: *int*) ^ *6* = *64*
by *eval*
then have *and i* (*mask 6*) = *i*
by (*simp add*: *less-mask-eq p*(*6*))
then have *x2* ∗ (*2* ^ *unat i* + *1*) = (*x2* ∗ (*2* ^ *unat i*)) + *x2*
by (*simp add*: *distrib-left*)
then show *x2* ∗ (*2* ^ *unat i* + *1*) = *x2* << *unat* (*and i 63*) + *x2*
by (*simp add*: *63* ‹*and i* (*mask 6*) = *i*›)
qed
using *val-to-bool.simps*(*2*) **by** *presburger*

lemma *val-MulPower2Sub1*:
fixes *i* :: *64 word*
assumes *y* = *IntVal 64* ((*2* ^ *unat*(*i*)) − *1*)
and *0* < *i*
and *i* < *64*
and *val-to-bool*(*val*[*IntVal 64 0* < *x*])
and *val-to-bool*(*val*[*IntVal 64 0* < *y*])
shows *val*[*x* ∗ *y*] = *val*[(*x* << *IntVal 64 i*) − *x*]
using *assms* **apply** (*cases x*; *cases y*; *auto*)
subgoal premises *p* **for** *x2*
proof −
have *63*: (*63* :: *int64*) = *mask 6*
by *eval*
then have (*2* :: *int*) ^ *6* = *64*

by *eval*
then have *and i* (*mask 6*) = *i*
by (*simp add*: *less-mask-eq p*(*6*))
then have *x2* ∗ (*2* ^ *unat i* − *1*) = (*x2* ∗ (*2* ^ *unat i*)) − *x2*
by (*simp add*: *right-diff-distrib′*)
then show *x2* ∗ (*2* ^ *unat i* − *1*) = *x2* << *unat* (*and i 63*) − *x2*
by (*simp add*: *63* ‹*and i* (*mask 6*) = *i*›)
qed
using *val-to-bool.simps*(*2*) **by** *presburger*

lemma *val-distribute-multiplication*:
assumes *x* = *IntVal b xx* ∧ *q* = *IntVal b qq* ∧ *a* = *IntVal b aa*
assumes *val*[*x* ∗ (*q* + *a*)] ≠ *UndefVal*
assumes *val*[(*x* ∗ *q*) + (*x* ∗ *a*)] ≠ *UndefVal*
shows *val*[*x* ∗ (*q* + *a*)] = *val*[(*x* ∗ *q*) + (*x* ∗ *a*)]
using *assms* **apply** (*cases x*; *cases q*; *cases a*; *auto*)
by (*metis* (*no-types*, *opaque-lifting*) *distrib-left new-int.elims new-int-unused-bits-zero mergeTakeBit*)

lemma *val-distribute-multiplication64*:
assumes *x* = *new-int 64 xx* ∧ *q* = *new-int 64 qq* ∧ *a* = *new-int 64 aa*
shows *val*[*x* ∗ (*q* + *a*)] = *val*[(*x* ∗ *q*) + (*x* ∗ *a*)]
using *assms* **apply** (*cases x*; *cases q*; *cases a*; *auto*)
using *distrib-left* **by** *blast*

lemma *val-MulPower2AddPower2*:
fixes *i j* :: *64 word*
assumes *y* = *IntVal 64* ((*2* ^ *unat*(*i*)) + (*2* ^ *unat*(*j*)))
and *0* < *i*
and *0* < *j*
and *i* < *64*
and *j* < *64*
and *x* = *new-int 64 xx*
shows *val*[*x* ∗ *y*] = *val*[(*x* << *IntVal 64 i*) + (*x* << *IntVal 64 j*)]
proof −
have *63*: (*63* :: *int64*) = *mask 6*
by *eval*
then have (*2* :: *int*) ^ *6* = *64*
by *eval*
then have *n*: *IntVal 64* ((*2* ^ *unat*(*i*)) + (*2* ^ *unat*(*j*))) =
val[(*IntVal 64* (*2* ^ *unat*(*i*))) + (*IntVal 64* (*2* ^ *unat*(*j*)))]

by *auto*
then have *1*: *val*[*x* ∗ ((*IntVal 64* (*2* ^ *unat*(*i*))) + (*IntVal 64* (*2* ^ *unat*(*j*))))] =
val[(*x* ∗ *IntVal 64* (*2* ^ *unat*(*i*))) + (*x* ∗ *IntVal 64* (*2* ^ *unat*(*j*)))]

using *assms val-distribute-multiplication64* **by** *simp*

then have *2*: *val*[(*x* ∗ *IntVal 64* (*2* ^ *unat*(*i*)))] = *val*[*x* << *IntVal 64* *i*]
by (*metis* (*no-types*, *opaque-lifting*) *Value.distinct*(*1*) *intval-mul.simps*(*1*)
new-int.simps
new-int-bin.simps *assms*(*2*,*4*,*6*) *val-MulPower2*)
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *1* *Value.distinct*(*1*) *n* *intval-mul.simps*(*1*) *new-int-bin.elims*
new-int.simps *val-MulPower2* *assms*(*1*,*3*,*5*,*6*))
qed

thm-oracles *val-MulPower2AddPower2*

lemma *exp-multiply-zero-64*:
shows *exp*[*x* ∗ (*const* (*IntVal b 0*))] ≥ *ConstantExpr* (*IntVal b 0*)
apply *auto*
subgoal premises *p* **for** *m p xa*
proof −
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*1*) **by** *auto*
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis* *evalDet* *p*(*1*,*2*) *xv* *evaltree-not-undef* *intval-is-null.cases* *intval-mul.simps*(*3*,*4*,*5*))
then have *evalNotUndef*: *val*[*xv* ∗ (*IntVal b 0*)] ≠ *UndefVal*
using *p* *evalDet* *xv* **by** *blast*
then have *mulUnfold*: *val*[*xv* ∗ (*IntVal b 0*)] = *IntVal xb* (*take-bit xb* (*xvv*∗*0*))
by (*metis* *new-int.simps* *xvv* *new-int-bin.simps* *intval-mul.simps*(*1*))
then have *isZero*: *val*[*xv* ∗ (*IntVal b 0*)] = (*new-int xb* (*0*))
by (*simp add*: *mulUnfold*)
then have *eq*: (*IntVal b 0*) = (*IntVal xb* (*0*))
by (*metis* *Value.distinct*(*1*) *intval-mul.simps*(*1*) *mulUnfold* *new-int-bin.elims*
xvv)
then show *?thesis*
using *evalDet* *isZero* *p*(*1*,*3*) *xv* **by** *fastforce*
qed
done

lemma *exp-multiply-neutral*:
exp[*x* ∗ (*const* (*IntVal b 1*))] ≥ *x*
apply *auto*
subgoal premises *p* **for** *m p xa*
proof −
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*1*) **by** *auto*
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*smt* (*z3*) *evalDet* *intval-mul.elims* *p*(*1*,*2*) *xv*)
then have *evalNotUndef*: *val*[*xv* ∗ (*IntVal b 1*)] ≠ *UndefVal*
using *p* *evalDet* *xv* **by** *blast*
then have *mulUnfold*: *val*[*xv* ∗ (*IntVal b 1*)] = *IntVal xb* (*take-bit xb* (*xvv*∗*1*))
by (*metis* *new-int.simps* *xvv* *new-int-bin.simps* *intval-mul.simps*(*1*))
then show *?thesis*

```
    by (metis bin-multiply-identity evalDet eval-unused-bits-zero p(1) xv xvv)
  qed
  done

thm-oracles exp-multiply-neutral

lemma exp-multiply-negative:
 exp[x ∗ −(const (IntVal b 1))] ≥ exp[−x]
  apply auto
  subgoal premises p for m p xa
  proof −
    obtain xv where xv: [m,p] ⊢ x ↦ xv
      using p(1) by auto
    obtain xb xvv where xvv: xv = IntVal xb xvv
     by (metis array-length.cases evalDet evaltree-not-undef intval-mul.simps(3,4,5)
p(1,2) xv)
    then have rewrite: val[−(IntVal b 1)] = IntVal b (mask b)
      by simp
    then have evalNotUndef: val[xv ∗ −(IntVal b 1)] ≠ UndefVal
      unfolding rewrite using evalDet p(1,2) xv by blast
    then have mulUnfold: val[xv ∗ (IntVal b (mask b))] =
                          (if xb=b then (IntVal xb (take-bit xb (xvv∗(mask xb)))) else
UndefVal)
      by (metis new-int.simps xvv new-int-bin.simps intval-mul.simps(1))
    then have sameWidth: xb=b
      by (metis evalNotUndef rewrite)
    then show ?thesis
    by (metis evalDet eval-unused-bits-zero new-int.elims p(1,2) rewrite unary-eval.simps(2)
xvv
          unfold-unary val-multiply-negative xv)
  qed
  done

lemma exp-MulPower2:
  fixes i :: 64 word
  assumes y = ConstantExpr (IntVal 64 (2 ^ 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] ≥ exp[x << ConstantExpr (IntVal 64 i)]
  using ConstantExprE equiv-exprs-def unfold-binary assms by fastforce

lemma exp-MulPower2Add1:
  fixes i :: 64 word
  assumes y = ConstantExpr (IntVal 64 ((2 ^ 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*] ≥ *exp*[(*x* << *ConstantExpr* (*IntVal 64 i*)) + *x*]
using *ConstantExprE equiv-exprs-def unfold-binary assms* **by** *fastforce*

lemma *exp-MulPower2Sub1*:
fixes *i* :: *64 word*
assumes *y* = *ConstantExpr* (*IntVal 64* ((*2* ^ *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*] ≥ *exp*[(*x* << *ConstantExpr* (*IntVal 64 i*)) − *x*]
using *ConstantExprE equiv-exprs-def unfold-binary assms* **by** *fastforce*

lemma *exp-MulPower2AddPower2*:
fixes *i j* :: *64 word*
assumes *y* = *ConstantExpr* (*IntVal 64* ((*2* ^ *unat*(*i*)) + (*2* ^ *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*] ≥ *exp*[(*x* << *ConstantExpr* (*IntVal 64 i*)) + (*x* << *ConstantExpr* (*IntVal 64 j*))]
using *ConstantExprE equiv-exprs-def unfold-binary assms* **by** *fastforce*

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*] ≥ *exp*[*const* (*new-int 32 0*)]
using *assms*
apply *simp* **unfolding** *equiv-exprs-def* **apply** *auto*
sorry

lemma *exp-distribute-multiplication*:
assumes *stamp-expr x* = *IntegerStamp b xl xh*
assumes *stamp-expr q* = *IntegerStamp b ql qh*
assumes *stamp-expr y* = *IntegerStamp b yl yh*
assumes *wf-stamp x*
assumes *wf-stamp q*
assumes *wf-stamp y*
shows *exp*[(*x* ∗ *q*) + (*x* ∗ *y*)] ≥ *exp*[*x* ∗ (*q* + *y*)]
apply *auto*
subgoal premises *p* **for** *m p xa qa xb aa*

proof −
  obtain *xv* **where** *xv*: $[m,p] \vdash x \mapsto xv$
    using *p* **by** *simp*
  obtain *qv* **where** *qv*: $[m,p] \vdash q \mapsto qv$
    using *p* **by** *simp*
  obtain *yv* **where** *yv*: $[m,p] \vdash y \mapsto yv$
    using *p* **by** *simp*
  then obtain *xvv* **where** *xvv*: *xv* = *IntVal b xvv*
    by (*metis assms*(*1*,*4*) *valid-int wf-stamp-def xv*)
  then obtain *qvv* **where** *qvv*: *qv* = *IntVal b qvv*
    by (*metis qv valid-int assms*(*2*,*5*) *wf-stamp-def*)
  then obtain *yvv* **where** *yvv*: *yv* = *IntVal b yvv*
    by (*metis yv valid-int assms*(*3*,*6*) *wf-stamp-def*)
  then have *rhsDefined*: *val*[*xv* ∗ (*qv* + *yv*)] ≠ *UndefVal*
    by (*simp add*: *xvv qvv*)
  have *val*[*xv* ∗ (*qv* + *yv*)] = *val*[(*xv* ∗ *qv*) + (*xv* ∗ *yv*)]
    using *val-distribute-multiplication* **by** (*simp add*: *yvv qvv xvv*)
  then show *?thesis*
    by (*metis bin-eval.simps*(*1*,*3*) *BinaryExpr p*(*1*,*2*,*3*,*5*,*6*) *qv xv evalDet yv qvv Value.distinct*(*1*)
      yvv intval-add.simps(*1*))
  qed
done

Optimisations

optimization *EliminateRedundantNegative*: $-x * -y \longmapsto x * y$
  apply *auto*
  by (*metis BinaryExpr val-eliminate-redundant-negative bin-eval.simps*(*3*))

optimization *MulNeutral*: *x* ∗ *ConstantExpr* (*IntVal b 1*) ⟼ *x*
  using *exp-multiply-neutral* **by** *blast*

optimization *MulEliminator*: *x* ∗ *ConstantExpr* (*IntVal b 0*) ⟼ *const* (*IntVal b 0*)
  using *exp-multiply-zero-64* **by** *fast*

optimization *MulNegate*: *x* ∗ −(*const* (*IntVal b 1*)) ⟼ −*x*
  using *exp-multiply-negative* **by** *presburger*

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 ($[m, p] \vdash x \mapsto v$) ⟶ (∃ *vv b*. (*v* = *IntVal b vv* ∧ *val-to-bool val*[(*IntVal b 0*) < *v*]))
  apply (*rule impI*) **subgoal premises** *eval*

proof −
  obtain *b lo hi* **where** *xstamp*: *stamp-expr x* = *IntegerStamp b lo hi*
    by (*meson isNonZero.elims*(*2*) *assms*)
  then obtain *vv* **where** *vdef*: *v* = *IntVal b vv*
    by (*metis assms*(*2*) *eval valid-int wf-stamp-def*)
  have *lo* > *0*
    using *assms*(*1*) *xstamp* **by** *force*
  then have *signed-above*: *int-signed-value b vv* > *0*
    using *assms eval vdef xstamp wf-stamp-def* **by** *fastforce*
  have *take-bit b vv* = *vv*
    using *eval eval-unused-bits-zero vdef* **by** *auto*
  then have *vv* > *0*
   by (*metis bit-take-bit-iff int-signed-value.simps signed-eq-0-iff take-bit-of-0 signed-above verit-comp-simplify1*(*1*) *word-gt-0 signed-take-bit-eq-if-positive*)
  then show *?thesis*
    using *vdef signed-above* **by** *simp*
qed
  done

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*
  using *assms* **by** (*simp add*: *IRTreeEvalThms.valid-value-elims*(*3*))

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*)))])
   apply *simp* **apply** (*rule impI*; (*rule allI*)+; *rule impI*)
  subgoal premises *eval* **for** *m p v*
proof −
  obtain *xv* **where** *xv*: [*m*, *p*] ⊢ *x* ↦ *xv*
    using *eval*(*2*) **by** *blast*
  then have *notUndef*: *xv* ≠ *UndefVal*
    by (*simp add*: *evaltree-not-undef*)
  obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
    by (*metis wf-stamp-def eval*(*1*) *ExpIntBecomesIntValArbitrary xv*)
  then have *w64*: *xb* = *64*
      by (*metis wf-stamp-def intval-bits.simps ExpIntBecomesIntValArbitrary xv eval*(*1*))
  obtain *yv* **where** *yv*: [*m*, *p*] ⊢ *y* ↦ *yv*
    using *eval*(*1*,*2*) **by** *blast*
  then have *lhs*: [*m*, *p*] ⊢ *exp*[*x* ∗ *y*] ↦ *val*[*xv* ∗ *yv*]
    by (*metis bin-eval.simps*(*3*) *eval*(*1*,*2*) *evalDet unfold-binary xv*)
  have [*m*, *p*] ⊢ *exp*[*const* (*IntVal 64 i*)] ↦ *val*[(*IntVal 64 i*)]

by (*smt* (*verit*, *ccfv-SIG*) *ConstantExpr constantAsStamp.simps*(*1*) *eval-bits-1-64 take-bit64 xv xvv*
validStampIntConst wf-value-def valid-value.simps(*1*) *w64*)
then have *rhs*: [*m*, *p*] ⊢ *exp*[*x* << *const* (*IntVal 64 i*)] ↦ *val*[*xv* << (*IntVal 64 i*)]
by (*metis Value.simps*(*5*) *bin-eval.simps*(*10*) *intval-left-shift.simps*(*1*) *new-int.simps xv xvv*
evaltree.BinaryExpr)
have *val*[*xv* ∗ *yv*] = *val*[*xv* << (*IntVal 64 i*)]
by (*metis ConstantExprE eval*(*1*) *evaltree-not-undef lhs yv val-MulPower2*)
then show *?thesis*
by (*metis eval*(*1*,*2*) *evalDet lhs rhs*)
qed
done

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*)))
apply *simp* **apply** (*rule impI*; (*rule allI*)+; *rule impI*)
subgoal premises *p* **for** *m p v*
proof −
obtain *xv* **where** *xv*: [*m*, *p*] ⊢ *x* ↦ *xv*
using *p* **by** *fast*
then obtain *xvv* **where** *xvv*: *xv* = *IntVal 64 xvv*
using *p* **by** (*metis valid-int wf-stamp-def*)
obtain *yv* **where** *yv*: [*m*, *p*] ⊢ *y* ↦ *yv*
using *p* **by** *blast*
have *ygezero*: *y* > *ConstantExpr* (*IntVal 64 0*)
using *greaterConstant p wf-value-def* **sorry**
then have *1*: *0* < *i* ∧
i < *64* ∧
y = *ConstantExpr* (*IntVal 64* ((*2* ^ *unat*(*i*)) + *1*))
using *p* **by** *blast*
then have *lhs*: [*m*, *p*] ⊢ *exp*[*x* ∗ *y*] ↦ *val*[*xv* ∗ *yv*]
by (*metis bin-eval.simps*(*3*) *evalDet p*(*2*) *xv yv unfold-binary*)
then have [*m*, *p*] ⊢ *exp*[*const* (*IntVal 64 i*)] ↦ *val*[(*IntVal 64 i*)]
by (*metis wf-value-def verit-comp-simplify1*(*2*) *zero-less-numeral ConstantExpr take-bit64*
constantAsStamp.simps(*1*) *validStampIntConst valid-value.simps*(*1*))
then have *rhs2*: [*m*, *p*] ⊢ *exp*[*x* << *const* (*IntVal 64 i*)] ↦ *val*[*xv* << (*IntVal 64 i*)]
by (*metis Value.simps*(*5*) *bin-eval.simps*(*10*) *intval-left-shift.simps*(*1*) *new-int.simps xv xvv*
evaltree.BinaryExpr)
then have *rhs*: [*m*, *p*] ⊢ *exp*[(*x* << *const* (*IntVal 64 i*)) + *x*] ↦ *val*[(*xv* << (*IntVal 64 i*)) + *xv*]
by (*metis* (*no-types*, *lifting*) *intval-add.simps*(*1*) *bin-eval.simps*(*1*) *Value.simps*(*5*)

xv xvv
evaltree.BinaryExpr intval-left-shift.simps(*1*) *new-int.simps*)
then have *simple*: *val*[*xv* ∗ (*IntVal 64* (*2* ^ *unat*(*i*)))] = *val*[*xv* << (*IntVal 64* *i*)]
using *val-MulPower2* **sorry**
then have *val*[*xv* ∗ *yv*] = *val*[(*xv* << (*IntVal 64 i*)) + *xv*]
using *val-MulPower2Add1* **sorry**
then show *?thesis*
by (*metis 1 evalDet lhs p*(*2*) *rhs*)
qed
done

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*)))
apply *simp* **apply** (*rule impI*; (*rule allI*)+; *rule impI*)
subgoal premises *p* **for** *m p v*
proof −
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p* **by** *fast*
then obtain *xvv* **where** *xvv*: *xv* = *IntVal 64 xvv*
using *p* **by** (*metis valid-int wf-stamp-def*)
obtain *yv* **where** *yv*: [*m*,*p*] ⊢ *y* ↦ *yv*
using *p* **by** *blast*
have *ygezero*: *y* > *ConstantExpr* (*IntVal 64 0*) **sorry**
then have *1*: *0* < *i* ∧
i < *64* ∧
y = *ConstantExpr* (*IntVal 64* ((*2* ^ *unat*(*i*)) − *1*))
using *p* **by** *blast*
then have *lhs*: [*m*, *p*] ⊢ *exp*[*x* ∗ *y*] ↦ *val*[*xv* ∗ *yv*]
by (*metis bin-eval.simps*(*3*) *evalDet p*(*2*) *xv yv unfold-binary*)
then have [*m*, *p*] ⊢ *exp*[*const* (*IntVal 64 i*)] ↦ *val*[(*IntVal 64 i*)]
by (*metis wf-value-def verit-comp-simplify1*(*2*) *zero-less-numeral ConstantExpr take-bit64*
constantAsStamp.simps(*1*) *validStampIntConst valid-value.simps*(*1*))
then have *rhs2*: [*m*, *p*] ⊢ *exp*[*x* << *const* (*IntVal 64 i*)] ↦ *val*[*xv* << (*IntVal 64 i*)]
by (*metis Value.simps*(*5*) *bin-eval.simps*(*10*) *intval-left-shift.simps*(*1*) *new-int.simps xv xvv*
evaltree.BinaryExpr)
then have *rhs*: [*m*, *p*] ⊢ *exp*[(*x* << *const* (*IntVal 64 i*)) − *x*] ↦ *val*[(*xv* << (*IntVal 64 i*)) − *xv*]
using *1 equiv-exprs-def ygezero yv* **by** *fastforce*
then have *val*[*xv* ∗ *yv*] = *val*[(*xv* << (*IntVal 64 i*)) − *xv*]
using *1 exp-MulPower2Sub1 ygezero* **sorry**
then show *?thesis*

by (*metis evalDet lhs p*(*1*) *p*(*2*) *rhs*)
qed
done

end

end

11.7 Experimental AndNode Phase

theory *NewAnd*
imports
Common
Graph.JavaLong
begin

lemma *intval-distribute-and-over-or*:
val[*z* & (*x* | *y*)] = *val*[(*z* & *x*) | (*z* & *y*)]
by (*cases x*; *cases y*; *cases z*; *auto simp add*: *bit.conj-disj-distrib*)

lemma *exp-distribute-and-over-or*:
exp[*z* & (*x* | *y*)] ≥ *exp*[(*z* & *x*) | (*z* & *y*)]
apply *auto*
by (*metis bin-eval.simps*(*6*,*7*) *intval-or.simps*(*2*,*6*) *intval-distribute-and-over-or* *BinaryExpr*)

lemma *intval-and-commute*:
val[*x* & *y*] = *val*[*y* & *x*]
by (*cases x*; *cases y*; *auto simp*: *and.commute*)

lemma *intval-or-commute*:
val[*x* | *y*] = *val*[*y* | *x*]
by (*cases x*; *cases y*; *auto simp*: *or.commute*)

lemma *intval-xor-commute*:
val[*x* ⊕ *y*] = *val*[*y* ⊕ *x*]
by (*cases x*; *cases y*; *auto simp*: *xor.commute*)

lemma *exp-and-commute*:
exp[*x* & *z*] ≥ *exp*[*z* & *x*]
by (*auto simp*: *intval-and-commute*)

lemma *exp-or-commute*:
exp[*x* | *y*] ≥ *exp*[*y* | *x*]
by (*auto simp*: *intval-or-commute*)

lemma *exp-xor-commute*:
exp[*x* ⊕ *y*] ≥ *exp*[*y* ⊕ *x*]
by (*auto simp*: *intval-xor-commute*)

lemma *intval-eliminate-y*:
  assumes *val*[*y* & *z*] = *IntVal b 0*
  shows *val*[(*x* | *y*) & *z*] = *val*[*x* & *z*]
  using *assms* **by** (*cases x*; *cases y*; *cases z*; *auto simp add*: *bit.conj-disj-distrib2*)

lemma *intval-and-associative*:
  val[(*x* & *y*) & *z*] = *val*[*x* & (*y* & *z*)]
  by (*cases x*; *cases y*; *cases z*; *auto simp*: *and.assoc*)

lemma *intval-or-associative*:
  val[(*x* | *y*) | *z*] = *val*[*x* | (*y* | *z*)]
  by (*cases x*; *cases y*; *cases z*; *auto simp*: *or.assoc*)

lemma *intval-xor-associative*:
  val[(*x* ⊕ *y*) ⊕ *z*] = *val*[*x* ⊕ (*y* ⊕ *z*)]
  by (*cases x*; *cases y*; *cases z*; *auto simp*: *xor.assoc*)

lemma *exp-and-associative*:
  exp[(*x* & *y*) & *z*] ≥ *exp*[*x* & (*y* & *z*)]
  using *intval-and-associative* **by** *fastforce*

lemma *exp-or-associative*:
  exp[(*x* | *y*) | *z*] ≥ *exp*[*x* | (*y* | *z*)]
  using *intval-or-associative* **by** *fastforce*

lemma *exp-xor-associative*:
  exp[(*x* ⊕ *y*) ⊕ *z*] ≥ *exp*[*x* ⊕ (*y* ⊕ *z*)]
  using *intval-xor-associative* **by** *fastforce*

lemma *intval-and-absorb-or*:
  assumes ∃ *b v* . *x* = *new-int b v*
  assumes *val*[*x* & (*x* | *y*)] ≠ *UndefVal*
  shows *val*[*x* & (*x* | *y*)] = *val*[*x*]
  using *assms* **apply** (*cases x*; *cases y*; *auto*)
  by (*metis* (*full-types*) *intval-and.simps*(*6*))

lemma *intval-or-absorb-and*:
  assumes ∃ *b v* . *x* = *new-int b v*
  assumes *val*[*x* | (*x* & *y*)] ≠ *UndefVal*
  shows *val*[*x* | (*x* & *y*)] = *val*[*x*]
  using *assms* **apply** (*cases x*; *cases y*; *auto*)
  by (*metis* (*full-types*) *intval-or.simps*(*6*))

lemma *exp-and-absorb-or*:
  exp[*x* & (*x* | *y*)] ≥ *exp*[*x*]
  apply *auto*
  subgoal premises *p* **for** *m p xa xaa ya*
  proof−

obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*1*) **by** *auto*
obtain *yv* **where** *yv*: [*m*,*p*] ⊢ *y* ↦ *yv*
using *p*(*4*) **by** *auto*
then have *lhsDefined*: *val*[*xv* & (*xv* | *yv*)] ≠ *UndefVal*
by (*metis evalDet p*(*1*,*2*,*3*,*4*) *xv*)
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis Value.exhaust-sel intval-and.simps*(*2*,*3*,*4*,*5*) *lhsDefined*)
obtain *yb yvv* **where** *yvv*: *yv* = *IntVal yb yvv*
by (*metis Value.exhaust-sel intval-and.simps*(*6*) *intval-or.simps*(*6*,*7*,*8*,*9*) *lhsDefined*)
then have *valEval*: *val*[*xv* & (*xv* | *yv*)] = *val*[*xv*]
by (*metis eval-unused-bits-zero intval-and-absorb-or lhsDefined new-int.elims xv xvv*)
then show *?thesis*
by (*metis evalDet p*(*1*,*3*,*4*) *xv yv*)
qed
done

lemma *exp-or-absorb-and*:
exp[*x* | (*x* & *y*)] ≥ *exp*[*x*]
apply *auto*
subgoal premises *p* **for** *m p xa xaa ya*
proof−
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*1*) **by** *auto*
obtain *yv* **where** *yv*: [*m*,*p*] ⊢ *y* ↦ *yv*
using *p*(*4*) **by** *auto*
then have *lhsDefined*: *val*[*xv* | (*xv* & *yv*)] ≠ *UndefVal*
by (*metis evalDet p*(*1*,*2*,*3*,*4*) *xv*)
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis Value.exhaust-sel intval-and.simps*(*3*,*4*,*5*) *intval-or.simps*(*2*,*6*) *lhsDefined*)
obtain *yb yvv* **where** *yvv*: *yv* = *IntVal yb yvv*
by (*metis Value.exhaust-sel intval-and.simps*(*6*,*7*,*8*,*9*) *intval-or.simps*(*6*) *lhsDefined*)
then have *valEval*: *val*[*xv* | (*xv* & *yv*)] = *val*[*xv*]
by (*metis eval-unused-bits-zero intval-or-absorb-and lhsDefined new-int.elims xv xvv*)
then show *?thesis*
by (*metis evalDet p*(*1*,*3*,*4*) *xv yv*)
qed
done

lemma
assumes *y* = *0*
shows *x* + *y* = *or x y*
by (*simp add*: *assms*)

lemma *no-overlap-or*:
  assumes *and x y = 0*
  shows *x + y = or x y*
  by (*metis bit-and-iff bit-xor-iff disjunctive-add xor-self-eq assms*)

context *stamp-mask*
begin

lemma *intval-up-and-zero-implies-zero*:
  assumes *and* (↑*x*) (↑*y*) = *0*
  assumes [*m*, *p*] ⊢ *x* ↦ *xv*
  assumes [*m*, *p*] ⊢ *y* ↦ *yv*
  assumes *val*[*xv* & *yv*] ≠ *UndefVal*
  shows ∃ *b* . *val*[*xv* & *yv*] = *new-int b 0*
  using *assms* **apply** (*cases xv*; *cases yv*; *auto*)
   apply (*metis eval-unused-bits-zero stamp-mask.up-mask-and-zero-implies-zero stamp-mask-axioms*)
  by *presburger*

lemma *exp-eliminate-y*:
  and (↑*y*) (↑*z*) = *0* ⟶ *exp*[(*x* | *y*) & *z*] ≥ *exp*[*x* & *z*]
  apply *simp* **apply** (*rule impI*; *rule allI*; *rule allI*; *rule allI*)
  subgoal premises *p* **for** *m p v* **apply** (*rule impI*) **subgoal premises** *e*
  proof −
    obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
      using *e* **by** *auto*
    obtain *yv* **where** *yv*: [*m*,*p*] ⊢ *y* ↦ *yv*
      using *e* **by** *auto*
    obtain *zv* **where** *zv*: [*m*,*p*] ⊢ *z* ↦ *zv*
      using *e* **by** *auto*
    have *lhs*: *v* = *val*[(*xv* | *yv*) & *zv*]
      by (*smt* (*verit*, *best*) *BinaryExprE bin-eval.simps*(*6*,*7*) *e evalDet xv yv zv*)
    then have *v* = *val*[(*xv* & *zv*) | (*yv* & *zv*)]
      by (*simp add*: *intval-and-commute intval-distribute-and-over-or*)
    also have ∃ *b*. *val*[*yv* & *zv*] = *new-int b 0*
    by (*metis calculation e intval-or.simps*(*6*) *p unfold-binary intval-up-and-zero-implies-zero yv*
        zv)
    ultimately have *rhs*: *v* = *val*[*xv* & *zv*]
      by (*auto simp*: *intval-eliminate-y lhs*)

from *lhs rhs* **show** *?thesis*
by (*metis BinaryExpr BinaryExprE bin-eval.simps*(*6*) *e xv zv*)
qed
done
done

lemma *leadingZeroBounds*:
fixes *x* :: *'a::len word*
assumes *n* = *numberOfLeadingZeros x*
shows *0* ≤ *n* ∧ *n* ≤ *Nat.size x*
by (*simp add*: *MaxOrNeg-def highestOneBit-def nat-le-iff numberOfLeadingZeros-def assms*)

lemma *above-nth-not-set*:
fixes *x* :: *int64*
assumes *n* = *64* − *numberOfLeadingZeros x*
shows *j* > *n* ⟶ ¬(*bit x j*)
by (*smt* (*verit*, *ccfv-SIG*) *highestOneBit-def int-nat-eq int-ops*(*6*) *less-imp-of-nat-less size64*
max-set-bit zerosAboveHighestOne assms numberOfLeadingZeros-def)

no-notation *LogicNegationNotation* (!-)

lemma *zero-horner*:
horner-sum of-bool 2 (*map* (λ*x. False*) *xs*) = *0*
by (*induction xs*; *auto*)

lemma *zero-map*:
assumes *j* ≤ *n*
assumes ∀ *i. j* ≤ *i* ⟶ ¬(*f i*)
shows *map f* [*0..*<*n*] = *map f* [*0..*<*j*] @ *map* (λ*x. False*) [*j..*<*n*]
by (*smt* (*verit*, *del-insts*) *add-diff-inverse-nat atLeastLessThan-iff bot-nat-0.extremum leD assms*
map-append map-eq-conv set-upt upt-add-eq-append)

lemma *map-join-horner*:
assumes *map f* [*0..*<*n*] = *map f* [*0..*<*j*] @ *map* (λ*x. False*) [*j..*<*n*]
shows *horner-sum of-bool* (*2::'a::len word*) (*map f* [*0..*<*n*]) = *horner-sum of-bool 2* (*map f* [*0..*<*j*])
proof −
have *horner-sum of-bool* (*2::'a::len word*) (*map f* [*0..*<*n*]) = *horner-sum of-bool 2* (*map f* [*0..*<*j*]) + *2* ^ *length* [*0..*<*j*] ∗ *horner-sum of-bool 2* (*map f* [*j..*<*n*])
using *assms* **apply** *auto*
by (*smt* (*verit*) *assms diff-le-self diff-zero le-add-same-cancel2 length-append length-map*
length-upt map-append upt-add-eq-append horner-sum-append)
also have ... = *horner-sum of-bool 2* (*map f* [*0..*<*j*]) + *2* ^ *length* [*0..*<*j*] ∗ *horner-sum of-bool 2* (*map* (λ*x. False*) [*j..*<*n*])
by (*metis calculation horner-sum-append length-map assms*)

also have ... = *horner-sum of-bool 2* (*map f* [*0..*<*j*])
using *zero-horner mult-not-zero* **by** *auto*
finally show *?thesis*
by *simp*
qed

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*])
by (*auto simp*: *assms zero-map map-join-horner*)

lemma *transfer-map*:
assumes $\forall i.\ i < n \longrightarrow f\ i = f'\ i$
shows (*map f* [*0..*<*n*]) = (*map f*′ [*0..*<*n*])
by (*simp add*: *assms*)

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*])
by (*smt* (*verit*, *best*) *assms transfer-map*)

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 −
have *nle*: $n \leq 64$
using *assms diff-le-self* **by** *blast*
also have *and v zv* = *horner-sum of-bool 2* (*map* (*bit* (*and v zv*)) [*0..*<*64*])
by (*metis size-word.rep-eq take-bit-length-eq horner-sum-bit-eq-take-bit size64*)
also have ... = *horner-sum of-bool 2* (*map* (λ*i*. *bit* (*and v zv*) *i*) [*0..*<*64*])
by *blast*
also have ... = *horner-sum of-bool 2* (*map* (λ*i*. ((*bit v i*) ∧ (*bit zv i*))) [*0..*<*64*])
by (*metis bit-and-iff*)
also have ... = *horner-sum of-bool 2* (*map* (λ*i*. ((*bit v i*) ∧ (*bit zv i*))) [*0..*<*n*])
proof −
have $\forall i.\ i \geq n \longrightarrow \neg(bit\ zv\ i)$
by (*smt* (*verit*, *ccfv-SIG*) *One-nat-def diff-less int-ops*(*6*) *leadingZerosAddHighestOne assms*
linorder-not-le nat-int-comparison(*2*) *not-numeral-le-zero size64 zero-less-Suc*

zerosAboveHighestOne not-may-implies-false)
then have $\forall i.\ i \geq n \longrightarrow \neg((bit\ v\ i) \wedge (bit\ zv\ i))$
by *auto*
then show *?thesis* **using** *nle split-horner*
by (*metis* (*no-types*, *lifting*))

qed
also have ... = *horner-sum of-bool 2* (*map* (*λi.* ((*bit* (*v mod 2^n*) *i*) ∧ (*bit zv i*))) [*0..<n*])
proof −
have ∀ *i. i* < *n* ⟶ *bit* (*v mod 2^n*) *i* = *bit v i*
by (*metis bit-take-bit-iff take-bit-eq-mod*)
then have ∀ *i. i* < *n* ⟶ ((*bit v i*) ∧ (*bit zv i*)) = ((*bit* (*v mod 2^n*) *i*) ∧ (*bit zv i*))
by *force*
then show *?thesis*
by (*rule transfer-horner*)
qed
also have ... = *horner-sum of-bool 2* (*map* (*λi.* ((*bit* (*v mod 2^n*) *i*) ∧ (*bit zv i*))) [*0..<64*])
proof −
have ∀ *i. i* ≥ *n* ⟶ ¬(*bit zv i*)
by (*smt* (*verit, ccfv-SIG*) *One-nat-def diff-less int-ops*(*6*) *leadingZerosAddHighestOne assms*
linorder-not-le nat-int-comparison(*2*) *not-numeral-le-zero size64 zero-less-Suc*
zerosAboveHighestOne not-may-implies-false)
then show *?thesis*
by (*metis* (*no-types, lifting*) *assms*(*1*) *diff-le-self split-horner*)
qed
also have ... = *horner-sum of-bool 2* (*map* (*bit* (*and* (*v mod 2^n*) *zv*)) [*0..<64*])
by (*meson bit-and-iff*)
also have ... = *and* (*v mod 2^n*) *zv*
by (*metis size-word.rep-eq take-bit-length-eq horner-sum-bit-eq-take-bit size64*)
finally show *?thesis*
using ‹*and* (*v*::*64 word*) (*zv*::*64 word*) = *horner-sum of-bool* (*2*::*64 word*) (*map* (*bit* (*and v zv*)) [*0*::*nat..<64*::*nat*])› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat. bit* ((*v*::*64 word*) *mod* (*2*::*64 word*) ^ (*n*::*nat*)) *i* ∧ *bit* (*zv*::*64 word*) *i*) [*0*::*nat..<64*::*nat*]) = *horner-sum of-bool* (*2*::*64 word*) (*map* (*bit* (*and* (*v mod* (*2*::*64 word*) ^ *n*) *zv*)) [*0*::*nat..<64*::*nat*])› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat. bit* ((*v*::*64 word*) *mod* (*2*::*64 word*) ^ (*n*::*nat*)) *i* ∧ *bit* (*zv*::*64 word*) *i*) [*0*::*nat..<n*]) = *horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat. bit* (*v mod* (*2*::*64 word*) ^ *n*) *i* ∧ *bit zv i*) [*0*::*nat..<64*::*nat*])› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat. bit* (*v*::*64 word*) *i* ∧ *bit* (*zv*::*64 word*) *i*) [*0*::*nat..<64*::*nat*]) = *horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat. bit v i* ∧ *bit zv i*) [*0*::*nat..<n*::*nat*])› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat. bit* (*v*::*64 word*) *i* ∧ *bit* (*zv*::*64 word*) *i*) [*0*::*nat..<n*::*nat*]) = *horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat. bit* (*v mod* (*2*::*64 word*) ^ *n*) *i* ∧ *bit zv i*) [*0*::*nat..<n*])› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (*bit* (*and* ((*v*::*64 word*) *mod* (*2*::*64 word*) ^ (*n*::*nat*)) (*zv*::*64 word*))) [*0*::*nat..<64*::*nat*]) = *and* (*v mod* (*2*::*64 word*) ^ *n*) *zv*› ‹*horner-sum of-bool* (*2*::*64 word*) (*map* (*bit* (*and* (*v*::*64 word*) (*zv*::*64 word*))) [*0*::*nat..<64*::*nat*]) = *horner-sum of-bool* (*2*::*64 word*) (*map* (*λi*::*nat. bit v i* ∧ *bit zv i*) [*0*::*nat..<64*::*nat*])› **by** *presburger*
qed

lemma *up-mask-upper-bound*:

assumes [*m*, *p*] ⊢ *x* ↦ *IntVal b xv*
shows *xv* ≤ (↑*x*)
by (*metis* (*no-types*, *lifting*) *and.right-neutral bit.conj-cancel-left bit.conj-disj-distribs*(*1*)
bit.double-compl ucast-id up-spec word-and-le1 word-not-dist(*2*) *assms*)

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 −
have *yv mod 2^n* = *horner-sum of-bool 2* (*map* (*bit yv*) [*0..<n*])
by (*simp add*: *horner-sum-bit-eq-take-bit take-bit-eq-mod*)
also have ... ≤ *horner-sum of-bool 2* (*map* (*bit* (↑*y*)) [*0..<n*])
by (*metis* (*no-types*, *opaque-lifting*) *and.right-neutral bit.conj-cancel-right word-not-dist*(*2*)
bit.conj-disj-distribs(*1*) *bit.double-compl horner-sum-bit-eq-take-bit take-bit-and*
ucast-id
up-spec word-and-le1 assms(*4*))
also have *horner-sum of-bool 2* (*map* (*bit* (↑*y*)) [*0..<n*]) = *horner-sum of-bool 2*
(*map* (λ*x*. *False*) [*0..<n*])
proof −
have ∀ *i* < *n*. ¬(*bit* (↑*y*) *i*)
by (*metis add.commute add-diff-inverse-nat add-lessD1 leD le-diff-conv zeros-*
BelowLowestOne
numberOfTrailingZeros-def assms(*1*,*2*))
then show *?thesis*
by (*metis* (*full-types*) *transfer-map*)
qed
also have *horner-sum of-bool 2* (*map* (λ*x*. *False*) [*0..<n*]) = *0*
by (*auto simp*: *zero-horner*)
finally show *?thesis*
by *auto*
qed

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*)
using *unfold-binary-width* **by** *simp*

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*)
using *unfold-binary-width* **by** *simp*

lemma *mod-dist-over-add-right*:
fixes *a b c* :: *int64*
fixes *n* :: *nat*
assumes $0 < n$
assumes $n < 64$
shows $(a + b \bmod 2\hat{~}n) \bmod 2\hat{~}n = (a + b) \bmod 2\hat{~}n$
using *mod-dist-over-add* **by** (*simp add*: *assms add.commute*)

lemma *numberOfLeadingZeros-range*:
0 ≤ *numberOfLeadingZeros n* ∧ *numberOfLeadingZeros n* ≤ *Nat.size n*
by (*simp add*: *leadingZeroBounds*)

lemma *improved-opt*:
assumes *numberOfLeadingZeros* (↑*z*) + *numberOfTrailingZeros* (↑*y*) ≥ *64*
shows *exp*[(*x* + *y*) & *z*] ≥ *exp*[*x* & *z*]
apply *simp* **apply** ((*rule allI*)+; *rule impI*)
subgoal premises *eval* **for** *m p v*
proof −
obtain *n* **where** *n*: *n* = *64* − *numberOfLeadingZeros* (↑*z*)
by *simp*
obtain *b val* **where** *val*: [*m*, *p*] ⊢ *exp*[(*x* + *y*) & *z*] ↦ *IntVal b val*
by (*metis BinaryExprE bin-eval-new-int eval new-int.simps*)
then obtain *xv yv* **where** *addv*: [*m*, *p*] ⊢ *exp*[*x* + *y*] ↦ *IntVal b* (*xv* + *yv*)
apply (*subst* (*asm*) *unfold-binary-width-and*) **by** (*metis add.right-neutral*)
then obtain *yv* **where** *yv*: [*m*, *p*] ⊢ *y* ↦ *IntVal b yv*
apply (*subst* (*asm*) *unfold-binary-width-add*) **by** *blast*
from *addv* **obtain** *xv* **where** *xv*: [*m*, *p*] ⊢ *x* ↦ *IntVal b xv*
apply (*subst* (*asm*) *unfold-binary-width-add*) **by** *blast*
from *val* **obtain** *zv* **where** *zv*: [*m*, *p*] ⊢ *z* ↦ *IntVal b zv*
apply (*subst* (*asm*) *unfold-binary-width-and*) **by** *blast*
have *addv*: [*m*, *p*] ⊢ *exp*[*x* + *y*] ↦ *new-int b* (*xv* + *yv*)
using *xv yv evaltree.BinaryExpr* **by** *auto*
have *lhs*: [*m*, *p*] ⊢ *exp*[(*x* + *y*) & *z*] ↦ *new-int b* (*and* (*xv* + *yv*) *zv*)
using *addv zv* **apply** (*rule evaltree.BinaryExpr*) **by** *simp*+
have *rhs*: [*m*, *p*] ⊢ *exp*[*x* & *z*] ↦ *new-int b* (*and xv zv*)
using *xv zv evaltree.BinaryExpr* **by** *auto*
then show *?thesis*
proof (*cases numberOfLeadingZeros* (↑*z*) > *0*)
case *True*
have *n-bounds*: $0 \le n \wedge n < 64$
by (*simp add*: *True n*)
have *and* (*xv* + *yv*) *zv* = *and* ((*xv* + *yv*) *mod 2^n*) *zv*
using *L1 n zv* **by** *blast*

also have ... = *and* ((*xv* + (*yv mod 2^n*)) *mod 2^n*) *zv*
by (*metis take-bit-0 take-bit-eq-mod zero-less-iff-neq-zero mod-dist-over-add-right n-bounds*)
also have ... = *and* (((*xv mod 2^n*) + (*yv mod 2^n*)) *mod 2^n*) *zv*
by (*metis bits-mod-by-1 mod-dist-over-add n-bounds order-le-imp-less-or-eq power-0*)
also have ... = *and* ((*xv mod 2^n*) *mod 2^n*) *zv*
using *L2 n zv yv assms* **by** *auto*
also have ... = *and* (*xv mod 2^n*) *zv*
by (*smt* (*verit*, *best*) *and.idem take-bit-eq-mask take-bit-eq-mod word-bw-assocs*(*1*)

mod-mod-trivial)
also have ... = *and xv zv*
by (*metis L1 n zv*)
finally show *?thesis*
by (*metis evalDet eval lhs rhs*)
next
case *False*
then have *numberOfLeadingZeros* (↑*z*) = *0*
by *simp*
then have *numberOfTrailingZeros* (↑*y*) ≥ *64*
using *assms* **by** *fastforce*
then have *yv* = *0*
by (*metis* (*no-types*, *lifting*) *L1 L2 add-diff-cancel-left′ and.comm-neutral linorder-not-le*
bit.conj-cancel-right bit.conj-disj-distribs(*1*) *bit.double-compl less-imp-diff-less yv*
word-not-dist(*2*))
then show *?thesis*
by (*metis add.right-neutral eval evalDet lhs rhs*)
qed
qed
done

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*))

```
  by (simp add: IRExpr-up-def)+

optimization redundant-lhs-x-or: ((x | y) & z) ⟼ y & z
                              when (((and (IRExpr-up x) (IRExpr-up z)) = 0))
  by (simp add: IRExpr-up-def)+

optimization redundant-rhs-y-or: (z & (x | y)) ⟼ z & x
                              when (((and (IRExpr-up y) (IRExpr-up z)) = 0))
  by (simp add: IRExpr-up-def)+

optimization redundant-rhs-x-or: (z & (x | y)) ⟼ z & y
                              when (((and (IRExpr-up x) (IRExpr-up z)) = 0))
  by (simp add: IRExpr-up-def)+



end

end
```

11.8 NotNode Phase

```
theory NotPhase
  imports
    Common
begin

phase NotNode
  terminating size
begin


lemma bin-not-cancel:
 bin[¬(¬(e))] = bin[e]
  by auto


lemma val-not-cancel:
  assumes val[~(new-int b v)] ≠ UndefVal
  shows   val[~(~(new-int b v))] = (new-int b v)
  by (simp add: take-bit-not-take-bit)


lemma exp-not-cancel:
   exp[~(~a)] ≥ exp[a]
  apply auto
  subgoal premises p for m p x
  proof −
    obtain av where av: [m,p] ⊢ a ↦ av
```

using *p*(*2*) **by** *auto*
obtain *bv avv* **where** *avv*: *av* = *IntVal bv avv*
by (*metis Value.exhaust av evalDet evaltree-not-undef intval-not.simps*(*3*,*4*,*5*) *p*(*2*,*3*))
then have *valEval*: *val*[~(~*av*)] = *val*[*av*]
by (*metis av avv evalDet eval-unused-bits-zero new-int.elims p*(*2*,*3*) *val-not-cancel*)
then show *?thesis*
by (*metis av evalDet p*(*2*))
qed
done

Optimisations

optimization *NotCancel*: *exp*[~(~*a*)] ⟼ *a*
by (*metis exp-not-cancel*)

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	$x \vert y$
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* (↓*x*)) (↑*y*)) = *0*
shows *exp*[*x* | *y*] ≥ *exp*[*x*]
using *assms*
apply *simp* **apply** ((*rule allI*)+; *rule impI*)
subgoal premises *eval* **for** *m p v*
proof −

```
    obtain b vv where e: [m, p] ⊢ exp[x | y] ↦ IntVal b vv
      by (metis BinaryExprE bin-eval-new-int new-int.simps eval(2))
    from e obtain xv where xv: [m, p] ⊢ x ↦ IntVal b xv
      apply (subst (asm) unfold-binary-width) by force+
    from e obtain yv where yv: [m, p] ⊢ y ↦ IntVal b yv
      apply (subst (asm) unfold-binary-width) by force+
    have vdef: v = val[(IntVal b xv) | (IntVal b yv)]
      by (metis bin-eval.simps(7) eval(2) evalDet unfold-binary xv yv)
    have ∀ i. (bit xv i) | (bit yv i) = (bit xv i)
      by (metis assms bit-and-iff not-down-up-mask-and-zero-implies-zero xv yv)
    then have IntVal b xv = val[(IntVal b xv) | (IntVal b yv)]
    by (metis (no-types, lifting) and.idem assms bit.conj-disj-distrib eval-unused-bits-zero
yv xv
       intval-or.simps(1) new-int.simps new-int-bin.simps not-down-up-mask-and-zero-implies-zero

          word-ao-absorbs(3))
    then show ?thesis
      using xv vdef by presburger
  qed
  done

lemma OrRightFallthrough:
  assumes (and (not (↓y)) (↑x)) = 0
  shows exp[x | y] ≥ exp[y]
  using assms
  apply simp apply ((rule allI)+; rule impI)
  subgoal premises eval for m p v
  proof −
    obtain b vv where e: [m, p] ⊢ exp[x | y] ↦ IntVal b vv
      by (metis BinaryExprE bin-eval-new-int new-int.simps eval(2))
    from e obtain xv where xv: [m, p] ⊢ x ↦ IntVal b xv
      apply (subst (asm) unfold-binary-width) by force+
    from e obtain yv where yv: [m, p] ⊢ y ↦ IntVal b yv
      apply (subst (asm) unfold-binary-width) by force+
    have vdef: v = val[(IntVal b xv) | (IntVal b yv)]
      by (metis bin-eval.simps(7) eval(2) evalDet unfold-binary xv yv)
    have ∀ i. (bit xv i) | (bit yv i) = (bit yv i)
      by (metis assms bit-and-iff not-down-up-mask-and-zero-implies-zero xv yv)
    then have IntVal b yv = val[(IntVal b xv) | (IntVal b yv)]
       by (metis (no-types, lifting) assms eval-unused-bits-zero intval-or.simps(1)
new-int.elims yv
             new-int-bin.elims stamp-mask.not-down-up-mask-and-zero-implies-zero
stamp-mask-axioms xv
          word-ao-absorbs(8))
    then show ?thesis
      using vdef yv by presburger
  qed
  done
```

end

phase *OrNode*
  terminating *size*
begin

lemma *bin-or-equal*:
  bin[*x* | *x*] = *bin*[*x*]
  by *simp*

lemma *bin-shift-const-right-helper*:
 x | *y* = *y* | *x*
  by *simp*

lemma *bin-or-not-operands*:
 (~*x* | ~*y*) = (~(*x* & *y*))
  by *simp*

lemma *val-or-equal*:
  assumes *x* = *new-int b v*
  and *val*[*x* | *x*] ≠ *UndefVal*
  shows *val*[*x* | *x*] = *val*[*x*]
  by (*auto simp*: *assms*)

lemma *val-elim-redundant-false*:
  assumes *x* = *new-int b v*
  and *val*[*x* | *false*] ≠ *UndefVal*
  shows *val*[*x* | *false*] = *val*[*x*]
  using *assms* **by** (*cases x*; *auto*; *presburger*)

lemma *val-shift-const-right-helper*:
   val[*x* | *y*] = *val*[*y* | *x*]
  by (*cases x*; *cases y*; *auto simp*: *or.commute*)

lemma *val-or-not-operands*:
 val[~*x* | ~*y*] = *val*[~(*x* & *y*)]
  by (*cases x*; *cases y*; *auto simp*: *take-bit-not-take-bit*)

lemma *exp-or-equal*:
  exp[*x* | *x*] ≥ *exp*[*x*]
  apply *auto*[*1*]
  subgoal premises *p* **for** *m p xa ya*
  proof−
    obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
      using *p*(*1*) **by** *auto*
    obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*

by (*metis evalDet evaltree-not-undef intval-is-null.cases intval-or.simps*(*3*,*4*,*5*) *p*(*1*,*3*) *xv*)
then have *evalNotUndef*: *val*[*xv* | *xv*] ≠ *UndefVal*
using *p evalDet xv* **by** *blast*
then have *orUnfold*: *val*[*xv* | *xv*] = (*new-int xb* (*or xvv xvv*))
by (*simp add*: *xvv*)
then have *simplify*: *val*[*xv* | *xv*] = (*new-int xb* (*xvv*))
by (*simp add*: *orUnfold*)
then have *eq*: (*xv*) = (*new-int xb* (*xvv*))
using *eval-unused-bits-zero xv xvv* **by** *auto*
then show *?thesis*
by (*metis evalDet p*(*1*,*2*) *simplify xv*)
qed
done

lemma *exp-elim-redundant-false*:
exp[*x* | *false*] ≥ *exp*[*x*]
apply *auto*[*1*]
subgoal premises *p* **for** *m p xa*
proof–
obtain *xv* **where** *xv*: [*m*,*p*] ⊢ *x* ↦ *xv*
using *p*(*1*) **by** *auto*
obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
by (*metis evalDet evaltree-not-undef intval-is-null.cases intval-or.simps*(*3*,*4*,*5*) *p*(*1*,*2*) *xv*)
then have *evalNotUndef*: *val*[*xv* | (*IntVal 32 0*)] ≠ *UndefVal*
using *p evalDet xv* **by** *blast*
then have *widthSame*: *xb*=*32*
by (*metis intval-or.simps*(*1*) *new-int-bin.simps xvv*)
then have *orUnfold*: *val*[*xv* | (*IntVal 32 0*)] = (*new-int xb* (*or xvv 0*))
by (*simp add*: *xvv*)
then have *simplify*: *val*[*xv* | (*IntVal 32 0*)] = (*new-int xb* (*xvv*))
by (*simp add*: *orUnfold*)
then have *eq*: (*xv*) = (*new-int xb* (*xvv*))
using *eval-unused-bits-zero xv xvv* **by** *auto*
then show *?thesis*
by (*metis evalDet p*(*1*) *simplify xv*)
qed
done

Optimisations

optimization *OrEqual*: *x* | *x* ⟼ *x*
by (*meson exp-or-equal*)

optimization *OrShiftConstantRight*: ((*const x*) | *y*) ⟼ *y* | (*const x*) *when* ¬(*is-ConstantExpr y*)
using *size-flip-binary* **by** (*auto simp*: *BinaryExpr unfold-const val-shift-const-right-helper*)

optimization *EliminateRedundantFalse*: *x* | *false* ⟼ *x*

by (*meson exp-elim-redundant-false*)

optimization *OrNotOperands*: (~*x* | ~*y*) ⟼ ~(*x* & *y*)
apply (*metis add-2-eq-Suc′ less-SucI not-add-less1 not-less-eq size-binary-const size-non-add*)
using *BinaryExpr UnaryExpr bin-eval.simps*(*4*) *intval-not.simps*(*2*) *unary-eval.simps*(*3*)

val-or-not-operands **by** *fastforce*

optimization *OrLeftFallthrough*:
x | *y* ⟼ *x when* ((*and* (*not* (*IRExpr-down x*)) (*IRExpr-up y*)) = *0*)
using *simple-mask.OrLeftFallthrough* **by** *blast*

optimization *OrRightFallthrough*:
x | *y* ⟼ *y when* ((*and* (*not* (*IRExpr-down y*)) (*IRExpr-up x*)) = *0*)
using *simple-mask.OrRightFallthrough* **by** *blast*

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*]
apply (*cases val-c*; *auto*) **using** *intval-left-shift.simps*(*1*) *intval-mul.simps*(*1*) *intval-log2.simps*(*1*)
sorry

lemma *e-intval*:

n = *intval-log2 val-c* ∧ *in-bounds n 0 32* ⟶
val[*x* << (*intval-log2 val-c*)] = *val*[*x* ∗ *val-c*]
proof (*rule impI*)
assume *n* = *intval-log2 val-c* ∧ *in-bounds n 0 32*
show *val*[*x* << (*intval-log2 val-c*)] = *val*[*x* ∗ *val-c*]
proof (*cases* ∃ *v* . *val-c* = *IntVal 32 v*)
case *True*
obtain *vc* **where** *val-c* = *IntVal 32 vc*
using *True* **by** *blast*
then have *n* = *IntVal 32* (*word-of-int* (*SOME e. vc=2^e*))
using ‹*n* = *intval-log2 val-c* ∧ *in-bounds n 0 32*› *intval-log2.simps*(*1*) **by**
presburger
then show *?thesis* **sorry**
next
case *False*
then have ∃ *v* . *val-c* = *IntVal 64 v*
sorry
then obtain *vc* **where** *val-c* = *IntVal 64 vc*
by *auto*
then have *n* = *IntVal 64* (*word-of-int* (*SOME e. vc=2^e*))
using ‹*n* = *intval-log2 val-c* ∧ *in-bounds n 0 32*› *intval-log2.simps*(*1*) **by**
presburger
then show *?thesis* **sorry**
qed
qed

optimization *e*:
x ∗ (*const c*) ⟼ *x* << (*const n*) *when* (*n* = *intval-log2 c* ∧ *in-bounds n 0 32*)
using *e-intval BinaryExprE ConstantExprE bin-eval.simps*(*2*,*7*) **sorry**

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*

using *assms* **apply** (*cases x*; *auto*)
by (*simp add*: *take-bit-signed-take-bit*)

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*) $\neq$ *UndefVal*
shows *intval-mod x* (*IntVal 32 1*) = *IntVal 32 0*
using *assms* **apply** (*cases x*; *auto*) **sorry**

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*
by *simp*

lemma *sub-self-is-zero*:
shows $(x::('a::len)\ word) - x = 0$
by *simp*

lemma *bin-sub-then-left-add*:
shows $(x::('a::len)\ word) - (x + (y::('a::len)\ word)) = -y$
by *simp*

lemma *bin-sub-then-left-sub*:
shows $(x::('a::len)\ word) - (x - (y::('a::len)\ word)) = y$
by *simp*

lemma *bin-subtract-zero*:
shows $(x :: 'a::len\ word) - (0 :: 'a::len\ word) = x$
by *simp*

lemma *bin-sub-negative-value*:
$(x :: ('a::len)\ word) - (-(y :: ('a::len)\ word)) = x + y$
by *simp*

lemma *bin-sub-self-is-zero*:
$(x :: ('a::len)\ word) - x = 0$
by *simp*

lemma *bin-sub-negative-const*:
$(x :: 'a::len\ word) - (-(y :: 'a::len\ word)) = x + y$
by *simp*

lemma *val-sub-after-right-add-2*:
assumes $x = new\text{-}int\ b\ v$
assumes $val[(x + y) - y] \neq UndefVal$
shows $val[(x + y) - y] = x$
using *assms* **apply** (*cases x*; *cases y*; *auto*)
by (*metis* (*full-types*) *intval-sub.simps*(*2*))

lemma *val-sub-after-left-sub*:
assumes $val[(x - y) - x] \neq UndefVal$
shows $val[(x - y) - x] = val[-y]$
using *assms intval-sub.elims* **apply** (*cases x*; *cases y*; *auto*)
by *fastforce*

lemma *val-sub-then-left-sub*:
assumes $y = new\text{-}int\ b\ v$
assumes $val[x - (x - y)] \neq UndefVal$
shows $val[x - (x - y)] = y$
using *assms* **apply** (*cases x*; *auto*)
by (*metis* (*mono-tags*) *intval-sub.simps*(*6*))

lemma *val-subtract-zero*:
assumes $x = new\text{-}int\ b\ v$
assumes $val[x - (IntVal\ b\ 0)] \neq UndefVal$
shows $val[x - (IntVal\ b\ 0)] = x$
by (*cases x*; *simp add*: *assms*)

lemma *val-zero-subtract-value*:
assumes $x = new\text{-}int\ b\ v$
assumes $val[(IntVal\ b\ 0) - x] \neq UndefVal$
shows $val[(IntVal\ b\ 0) - x] = val[-x]$
by (*cases x*; *simp add*: *assms*)

lemma *val-sub-then-left-add*:
assumes $val[x - (x + y)] \neq UndefVal$
shows $val[x - (x + y)] = val[-y]$
using *assms* **apply** (*cases x*; *cases y*; *auto*)
by (*metis* (*mono-tags*, *lifting*) *intval-sub.simps*(*6*))

lemma *val-sub-negative-value*:
assumes $val[x - (-y)] \neq UndefVal$
shows $val[x - (-y)] = val[x + y]$
by (*cases x*; *cases y*; *simp add*: *assms*)

lemma *val-sub-self-is-zero*:
assumes $x = new\text{-}int\ b\ v \wedge val[x - x] \neq UndefVal$
shows $val[x - x] = new\text{-}int\ b\ 0$
by (*cases x*; *simp add*: *assms*)

lemma *val-sub-negative-const*:
assumes $y = new\text{-}int\ b\ v \wedge val[x - (-y)] \neq UndefVal$
shows $val[x - (-y)] = val[x + y]$
by (*cases x*; *simp add*: *assms*)

lemma *exp-sub-after-right-add*:
shows $exp[(x + y) - y] \geq x$
apply *auto*
subgoal premises *p* **for** *m p ya xa yaa*
proof−
obtain *xv* **where** *xv*: $[m,p] \vdash x \mapsto xv$
using *p*(*3*) **by** *auto*
obtain *yv* **where** *yv*: $[m,p] \vdash y \mapsto yv$
using *p*(*1*) **by** *auto*
obtain *xb xvv* **where** *xvv*: $xv = IntVal\ xb\ xvv$
by (*metis Value.exhaust evalDet evaltree-not-undef intval-add.simps*(*3*,*4*,*5*)
intval-sub.simps(*2*)
p(*2*,*3*) *xv*)
obtain *yb yvv* **where** *yvv*: $yv = IntVal\ yb\ yvv$
by (*metis evalDet evaltree-not-undef intval-add.simps*(*7*,*8*,*9*) *intval-logic-negation.cases*

```
yv
          intval-sub.simps(2) p(2,4))
    then have lhsDefined: val[(xv + yv) − yv] ≠ UndefVal
      using xvv yvv apply (cases xv; cases yv; auto)
      by (metis evalDet intval-add.simps(1) p(3,4,5) xv yv)
     then show ?thesis
      by (metis ‹⋀thesis. (⋀(xb) xvv. (xv) = IntVal xb xvv ⟹ thesis) ⟹ thesis›
evalDet xv yv
        eval-unused-bits-zero lhsDefined new-int.simps p(1,3,4) val-sub-after-right-add-2)
  qed
  done

lemma exp-sub-after-right-add2:
  shows exp[(x + y) − x] ≥ y
  using exp-sub-after-right-add apply auto
  by (metis bin-eval.simps(1,2) intval-add-sym unfold-binary)

lemma exp-sub-negative-value:
 exp[x − (−y)] ≥ exp[x + y]
  apply auto
  subgoal premises p for m p xa ya
  proof −
    obtain xv where xv: [m,p] ⊢ x ↦ xv
      using p(1) by auto
    obtain yv where yv: [m,p] ⊢ y ↦ yv
      using p(3) by auto
    then have rhsEval: [m,p] ⊢ exp[x + y] ↦ val[xv + yv]
    by (metis bin-eval.simps(1) evalDet p(1,2,3) unfold-binary val-sub-negative-value
xv)
    then show ?thesis
      by (metis evalDet p(1,2,3) val-sub-negative-value xv yv)
  qed
  done

lemma exp-sub-then-left-sub:
  exp[x − (x − y)] ≥ y
  using val-sub-then-left-sub apply auto
  subgoal premises p for m p xa xaa ya
    proof−
      obtain xa where xa: [m, p] ⊢ x ↦ xa
        using p(2) by blast
      obtain ya where ya: [m, p] ⊢ y ↦ ya
        using p(5) by auto
      obtain xaa where xaa: [m, p] ⊢ x ↦ xaa
        using p(2) by blast
      have 1: val[xa − (xaa − ya)] ≠ UndefVal
        by (metis evalDet p(2,3,4,5) xa xaa ya)
      then have val[xaa − ya] ≠ UndefVal
        by auto
```

then have $[m, p] \vdash y \mapsto val[xa - (xaa - ya)]$
by (*metis 1 Value.exhaust eval-unused-bits-zero evaltree-not-undef xa xaa ya new-int.simps intval-sub.simps*(*6*,*7*,*8*,*9*) *evalDet val-sub-then-left-sub*)
then show *?thesis*
by (*metis evalDet p*(*2*,*4*,*5*) *xa xaa ya*)
qed
done

thm-oracles *exp-sub-then-left-sub*

lemma *SubtractZero-Exp*:
exp$[(x - (const\ IntVal\ b\ 0))] \geq x$
apply *auto*
subgoal premises *p* **for** *m p xa*
proof–
obtain *xv* **where** *xv*: $[m,p] \vdash x \mapsto xv$
using *p*(*1*) **by** *auto*
obtain *xb xvv* **where** *xvv*: $xv = IntVal\ xb\ xvv$
by (*metis array-length.cases evalDet evaltree-not-undef intval-sub.simps*(*3*,*4*,*5*) *p*(*1*,*2*) *xv*)
then have *widthSame*: $xb=b$
by (*metis evalDet intval-sub.simps*(*1*) *new-int-bin.simps p*(*1*) *p*(*2*) *xv*)
then have *unfoldSub*: $val[xv - (IntVal\ b\ 0)] = (new\text{-}int\ xb\ (xvv-0))$
by (*simp add*: *xvv*)
then have *rhsSame*: $val[xv] = (new\text{-}int\ xb\ (xvv))$
using *eval-unused-bits-zero xv xvv* **by** *auto*
then show *?thesis*
by (*metis diff-zero evalDet p*(*1*) *unfoldSub xv*)
qed
done

lemma *ZeroSubtractValue-Exp*:
assumes *wf-stamp x*
assumes *stamp-expr* $x = IntegerStamp\ b\ lo\ hi$
assumes $\neg$(*is-ConstantExpr x*)
shows *exp*$[(const\ IntVal\ b\ 0) - x] \geq exp[-x]$
using *assms* **apply** *auto*
subgoal premises *p* **for** *m p xa*
proof–
obtain *xv* **where** *xv*: $[m,p] \vdash x \mapsto xv$
using *p*(*4*) **by** *auto*
obtain *xb xvv* **where** *xvv*: $xv = IntVal\ xb\ xvv$
by (*metis constantAsStamp.cases evalDet evaltree-not-undef intval-sub.simps*(*7*,*8*,*9*) *p*(*4*,*5*) *xv*)
then have *unfoldSub*: $val[(IntVal\ b\ 0) - xv] = (new\text{-}int\ xb\ (0-xvv))$
by (*metis intval-sub.simps*(*1*) *new-int-bin.simps p*(*1*,*2*) *valid-int-same-bits wf-stamp-def xv*)
then show *?thesis*

by (*metis UnaryExpr intval-negate.simps*(*1*) *p*(*4*,*5*) *unary-eval.simps*(*2*) *verit-minus-simplify*(*3*)
evalDet xv xvv)
qed
done

Optimisations

optimization *SubAfterAddRight*: $((x + y) - y) \longmapsto x$
using *exp-sub-after-right-add* **by** *blast*

optimization *SubAfterAddLeft*: $((x + y) - x) \longmapsto y$
using *exp-sub-after-right-add2* **by** *blast*

optimization *SubAfterSubLeft*: $((x - y) - x) \longmapsto -y$
by (*smt* (*verit*) *Suc-lessI add-2-eq-Suc′ add-less-cancel-right less-trans-Suc not-add-less1 evalDet*
size-binary-const size-binary-lhs size-binary-rhs size-non-add BinaryExprE bin-eval.simps(*2*)
le-expr-def unary-eval.simps(*2*) *unfold-unary val-sub-after-left-sub*)+

optimization *SubThenAddLeft*: $(x - (x + y)) \longmapsto -y$
apply *auto*
by (*metis evalDet unary-eval.simps*(*2*) *unfold-unary val-sub-then-left-add*)

optimization *SubThenAddRight*: $(y - (x + y)) \longmapsto -x$
apply *auto*
by (*metis evalDet intval-add-sym unary-eval.simps*(*2*) *unfold-unary val-sub-then-left-add*)

optimization *SubThenSubLeft*: $(x - (x - y)) \longmapsto y$
using *size-simps exp-sub-then-left-sub* **by** *auto*

optimization *SubtractZero*: $(x - (const\ IntVal\ b\ 0)) \longmapsto x$
using *SubtractZero-Exp* **by** *fast*

thm-oracles *SubtractZero*

optimization *SubNegativeValue*: $(x - (-y)) \longmapsto x + y$
apply (*metis add-2-eq-Suc′ less-SucI less-add-Suc1 not-less-eq size-binary-const size-non-add*)
using *exp-sub-negative-value* **by** *blast*

thm-oracles *SubNegativeValue*

lemma *negate-idempotent*:
assumes $x = IntVal\ b\ v \wedge take\text{-}bit\ b\ v = v$
shows $x = val[-(-x)]$

by (*auto simp*: *assms is-IntVal-def*)

optimization *ZeroSubtractValue*: ((*const IntVal b 0*) − *x*) ⟼ (−*x*)
when (*wf-stamp x* ∧ *stamp-expr x* = *IntegerStamp b lo hi* ∧ ¬(*is-ConstantExpr x*))
using *size-flip-binary ZeroSubtractValue-Exp* **by** *simp+*

optimization *SubSelfIsZero*: (*x* − *x*) ⟼ *const IntVal b 0 when*
(*wf-stamp x* ∧ *stamp-expr x* = *IntegerStamp b lo hi*)
using *size-non-const* **apply** *auto*
by (*smt* (*verit*) *wf-value-def ConstantExpr eval-bits-1-64 eval-unused-bits-zero new-int.simps*
take-bit-of-0 val-sub-self-is-zero validDefIntConst valid-int wf-stamp-def One-nat-def evalDet)

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* ⊕ *x*] = *0*
by *simp*

lemma *bin-xor-commute*:
bin[*x* ⊕ *y*] = *bin*[*y* ⊕ *x*]
by (*simp add*: *xor.commute*)

lemma *bin-eliminate-redundant-false*:
bin[*x* ⊕ *0*] = *bin*[*x*]
by *simp*

lemma *val-xor-self-is-false*:
  assumes *val*[$x \oplus x$] $\neq$ *UndefVal*
  shows *val-to-bool* (*val*[$x \oplus x$]) = *False*
  by (*cases x*; *auto simp*: *assms*)

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-to-val False*
  by (*auto simp*: *assms*)

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*
  by (*auto simp*: *assms*)

lemma *val-xor-commute*:
  val[$x \oplus y$] = *val*[$y \oplus x$]
  by (*cases x*; *cases y*; *auto simp*: *xor.commute*)

lemma *val-eliminate-redundant-false*:
  assumes x = *new-int b v*
  assumes *val*[$x \oplus$ (*bool-to-val False*)] $\neq$ *UndefVal*
  shows *val*[$x \oplus$ (*bool-to-val False*)] = x
  using *assms* **by** (*auto*; *meson*)

lemma *exp-xor-self-is-false*:
 assumes *wf-stamp x* $\wedge$ *stamp-expr x* = *default-stamp*
 shows *exp*[$x \oplus x$] $\geq$ *exp*[*false*]
  using *assms* **apply** *auto*
  subgoal premises *p* **for** *m p xa ya*
  proof−
    obtain *xv* **where** *xv*: [*m*,*p*] $\vdash$ *x* $\mapsto$ *xv*
      using *p*(*3*) **by** *auto*
    obtain *xb xvv* **where** *xvv*: *xv* = *IntVal xb xvv*
    by (*metis Value.exhaust-sel assms evalDet evaltree-not-undef intval-xor.simps*(*5*,*7*)
p(*3*,*4*,*5*) *xv*
        valid-value.simps(*11*) *wf-stamp-def*)
    then have *unfoldXor*: *val*[$xv \oplus xv$] = (*new-int xb* (*xor xvv xvv*))
      by *simp*
    then have *isZero*: *xor xvv xvv* = *0*
      by *simp*
    then have *width*: *xb* = *32*
      by (*metis valid-int-same-bits xv xvv p*(*1*,*2*) *wf-stamp-def*)
    then have *isFalse*: *val*[$xv \oplus xv$] = *bool-to-val False*
      unfolding *unfoldXor isZero width* **by** *fastforce*

then show *?thesis*
by (*metis* (*no-types*, *lifting*) *eval-bits-1-64* *p*(*3*,*4*) *width* *xv* *xvv* *validDefIntConst* *IntVal0*
Value.inject(*1*) *bool-to-val.simps*(*2*) *evalDet* *new-int.simps* *unfold-const* *wf-value-def*)
qed
done

lemma *exp-eliminate-redundant-false*:
shows *exp*[*x* ⊕ *false*] ≥ *exp*[*x*]
using *val-eliminate-redundant-false* **apply** *auto*
subgoal premises *p* **for** *m p xa*
proof −
obtain *xa* **where** *xa*: [*m*, *p*] ⊢ *x* ↦ *xa*
using *p*(*2*) **by** *blast*
then have *val*[*xa* ⊕ (*IntVal 32 0*)] ≠ *UndefVal*
using *evalDet* *p*(*2*,*3*) **by** *blast*
then have [*m*, *p*] ⊢ *x* ↦ *val*[*xa* ⊕ (*IntVal 32 0*)]
using *eval-unused-bits-zero* *xa* **by** (*cases* *xa*; *auto*)
then show *?thesis*
using *evalDet* *p*(*2*) *xa* **by** *blast*
qed
done

Optimisations

optimization *XorSelfIsFalse*: (*x* ⊕ *x*) ⟼ *false* *when*
(*wf-stamp* *x* ∧ *stamp-expr* *x* = *default-stamp*)
using *size-non-const* *exp-xor-self-is-false* **by** *auto*

optimization *XorShiftConstantRight*: ((*const* *x*) ⊕ *y*) ⟼ *y* ⊕ (*const* *x*) *when* ¬(*is-ConstantExpr* *y*)
using *size-flip-binary* *val-xor-commute* **by** *auto*

optimization *EliminateRedundantFalse*: (*x* ⊕ *false*) ⟼ *x*
using *exp-eliminate-redundant-false* **by** *auto*

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 ↣ 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 −
have *unfoldEqualDefined*: (*intval-equals v1 v2* ≠ *UndefVal*) ⟹
(*val-to-bool*(*intval-equals v1 v2*) ⟶ (¬(*val-to-bool*(*intval-less-than v1 v2*))))
subgoal premises *p*
proof −
obtain *v1b v1v* **where** *v1v*: *v1* = *IntVal v1b v1v*
by (*metis array-length.cases intval-equals.simps*(*2*,*3*,*4*,*5*) *p*)
obtain *v2b v2v* **where** *v2v*: *v2* = *IntVal v2b v2v*
by (*metis Value.exhaust-sel intval-equals.simps*(*6*,*7*,*8*,*9*) *p*)
have *sameWidth*: *v1b*=*v2b*
by (*metis bool-to-val-bin.simps intval-equals.simps*(*1*) *p v1v v2v*)
have *unfoldEqual*: *intval-equals v1 v2* = (*bool-to-val* (*v1v*=*v2v*))
by (*simp add*: *sameWidth v1v v2v*)
have *unfoldLessThan*: *intval-less-than v1 v2* = (*bool-to-val* (*int-signed-value v1b v1v* < *int-signed-value v2b v2v*))
by (*simp add*: *sameWidth v1v v2v*)
have *val*: ((*v1v*=*v2v*)) ⟶ (¬((*int-signed-value v1b v1v* < *int-signed-value v2b v2v*)))
using *sameWidth* **by** *auto*
have *doubleCast0*: *val-to-bool* (*bool-to-val* ((*v1v* = *v2v*))) = (*v1v* = *v2v*)
using *bool-to-val.elims val-to-bool.simps*(*1*) **by** *fastforce*

have *doubleCast1*: *val-to-bool* (*bool-to-val* ((*int-signed-value v1b v1v* < *int-signed-value v2b v2v*))) =
(*int-signed-value v1b v1v* < *int-signed-value v2b v2v*)
using *bool-to-val.elims val-to-bool.simps*(*1*) **by** *fastforce*
then show *?thesis*
using *p val* **unfolding** *unfoldEqual unfoldLessThan doubleCast0 doubleCast1* **by** *blast*
qed done
show *?thesis*
by (*metis Value.distinct*(*1*) *val-to-bool.elims*(*2*) *unfoldEqualDefined*)
qed

lemma *eq-not-less′-val*:
val-to-bool(*val*[*v1 eq v2*]) ⟶ ¬*val-to-bool*(*val*[*v2* < *v1*])
proof –
have *a*: *intval-equals v1 v2* = *intval-equals v2 v1*
apply (*cases intval-equals v1 v2* = *UndefVal*)
apply (*smt* (*z3*) *bool-to-val-bin.simps intval-equals.elims intval-equals.simps*)
subgoal premises *p*
proof –
obtain *v1b v1v* **where** *v1v*: *v1* = *IntVal v1b v1v*
by (*metis Value.exhaust-sel intval-equals.simps*(*2*,*3*,*4*,*5*) *p*)
obtain *v2b v2v* **where** *v2v*: *v2* = *IntVal v2b v2v*
by (*metis Value.exhaust-sel intval-equals.simps*(*6*,*7*,*8*,*9*) *p*)
then show *?thesis*
by (*smt* (*verit*) *bool-to-val-bin.simps intval-equals.simps*(*1*) *v1v*)
qed done
show *?thesis*
using *a eq-not-less-val* **by** *presburger*
qed

lemma *less-not-less-val*:
val-to-bool(*val*[*v1* < *v2*]) ⟶ ¬*val-to-bool*(*val*[*v2* < *v1*])
apply (*rule impI*)
subgoal premises *p*
proof –
obtain *v1b v1v* **where** *v1v*: *v1* = *IntVal v1b v1v*
by (*metis Value.exhaust-sel intval-less-than.simps*(*2*,*3*,*4*,*5*) *p val-to-bool.simps*(*2*))
obtain *v2b v2v* **where** *v2v*: *v2* = *IntVal v2b v2v*
by (*metis Value.exhaust-sel intval-less-than.simps*(*6*,*7*,*8*,*9*) *p val-to-bool.simps*(*2*))
then have *unfoldLessThanRHS*: *intval-less-than v2 v1* =
(*bool-to-val* (*int-signed-value v2b v2v* < *int-signed-value v1b v1v*))
using *p v1v* **by** *force*
then have *unfoldLessThanLHS*: *intval-less-than v1 v2* =
(*bool-to-val* (*int-signed-value v1b v1v* < *int-signed-value v2b v2v*))
using *bool-to-val-bin.simps intval-less-than.simps*(*1*) *p v1v v2v val-to-bool.simps*(*2*)

by *auto*
then have *symmetry*: (*int-signed-value v2b v2v* < *int-signed-value v1b v1v*) ⟶
(¬(*int-signed-value v1b v1v* < *int-signed-value v2b v2v*))
by *simp*
then show *?thesis*
using *p unfoldLessThanLHS unfoldLessThanRHS* **by** *fastforce*
qed done

lemma *less-not-eq-val*:
val-to-bool(*val*[*v1* < *v2*]) ⟶ ¬*val-to-bool*(*val*[*v1 eq v2*])
using *eq-not-less-val* **by** *blast*

lemma *logic-negate-type*:
assumes [*m*, *p*] ⊢ *UnaryExpr UnaryLogicNegation x* ↦ *v*
shows ∃ *b v2*. [*m*, *p*] ⊢ *x* ↦ *IntVal b v2*
using *assms*
by (*metis UnaryExprE intval-logic-negation.elims unary-eval.simps*(*4*))

lemma *intval-logic-negation-inverse*:
assumes *b* > *0*
assumes *x* = *IntVal b v*
shows *val-to-bool* (*intval-logic-negation x*) ⟷ ¬(*val-to-bool x*)
using *assms* **by** (*cases x*; *auto simp*: *logic-negate-def*)

lemma *logic-negation-relation-tree*:
assumes [*m*, *p*] ⊢ *y* ↦ *val*
assumes [*m*, *p*] ⊢ *UnaryExpr UnaryLogicNegation y* ↦ *invval*
shows *val-to-bool val* ⟷ ¬(*val-to-bool invval*)
using *assms* **using** *intval-logic-negation-inverse*
by (*metis UnaryExprE evalDet eval-bits-1-64 logic-negate-type unary-eval.simps*(*4*))

The following theorem show that the known true/false rules are valid.

theorem *implies-impliesnot-valid*:
shows ((*q1* ⇛ *q2*) ⟶ (*q1* ↣ *q2*)) ∧
((*q1* ⇛¬ *q2*) ⟶ (*q1* ⤔ *q2*))
(**is** (*?imp* ⟶ *?val*) ∧ (*?notimp* ⟶ *?notval*))
proof (*induct q1 q2 rule*: *impliesx-impliesnot.induct*)
case (*same q*)
then show *?case*
using *evalDet* **by** *fastforce*
next
case (*eq-not-less x y*)
then show *?case* **apply** *auto*[*1*] **using** *eq-not-less-val evalDet* **by** *blast*
next
case (*eq-not-less′ x y*)
then show *?case* **apply** *auto*[*1*] **using** *eq-not-less′-val evalDet* **by** *blast*
next
case (*less-not-less x y*)
then show *?case* **apply** *auto*[*1*] **using** *less-not-less-val evalDet* **by** *blast*

next
  case (*less-not-eq x y*)
  then show *?case* **apply** *auto*[*1*] **using** *less-not-eq-val evalDet* **by** *blast*
next
  case (*less-not-eq′ x y*)
  then show *?case* **apply** *auto*[*1*] **using** *eq-not-less′-val evalDet* **by** *metis*
next
  case (*negate-true x y*)
  then show *?case* **apply** *auto*[*1*]
    by (*metis logic-negation-relation-tree unary-eval.simps*(*4*) *unfold-unary*)
next
  case (*negate-false x y*)
  then show *?case* **apply** *auto*[*1*]
    by (*metis UnaryExpr logic-negation-relation-tree unary-eval.simps*(*4*))
qed

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* ⇒ (*ID* ⇒ *Stamp*) ⇒ *bool* ⇒ *bool*
  where
  ⟦*alwaysDistinct* (*stamps x*) (*stamps y*)⟧
    ⟹ *tryFold* (*IntegerEqualsNode x y*) *stamps False* |
  ⟦*neverDistinct* (*stamps x*) (*stamps y*)⟧
    ⟹ *tryFold* (*IntegerEqualsNode x y*) *stamps True* |
  ⟦*is-IntegerStamp* (*stamps x*);
    is-IntegerStamp (*stamps y*);
    stpi-upper (*stamps x*) < *stpi-lower* (*stamps y*)⟧
    ⟹ *tryFold* (*IntegerLessThanNode x y*) *stamps True* |
  ⟦*is-IntegerStamp* (*stamps x*);
    is-IntegerStamp (*stamps y*);
    stpi-lower (*stamps x*) ≥ *stpi-upper* (*stamps y*)⟧
    ⟹ *tryFold* (*IntegerLessThanNode x y*) *stamps False*

code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *bool*) *tryFold* **.**

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*)
  using *assms is-stamp-empty.simps* **apply** (*cases s*; *auto*)
  by (*metis linorder-not-le not-less-iff-gr-or-eq order.strict-trans valid-value.elims*(*2*) *valid-value.simps*(*1*) *valid-value.simps*(*5*))

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
  assume *?lhs*
  then show *?rhs*
   using *assms*(*1*) **apply** (*cases s1*; *cases s2*; *auto*)
   apply (*metis Value.inject*(*1*) *valid-int*)
  by (*smt* (*z3*) *valid-int valid-stamp.simps*(*1*) *valid-value.simps*(*1*))
  next
  assume *?rhs*
  then show *?lhs*
    using *assms* **apply** (*cases s1*; *cases s2*; *simp*)
  by (*smt* (*verit*, *best*) *assms*(*2*) *valid-int valid-value.simps*(*1*) *valid-value.simps*(*22*))
qed

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 −
  obtain *stampx stampy* **where** *stampdef*: *stampx* = *stamps x* ∧ *stampy* = *stamps y*
    by *simp*
  then have *xv*: ∀ *xv* . ([*g*, *m*, *p*] ⊢ *x* ↦ *xv*) ⟶ *valid-value xv stampx*
    by (*meson assms*(*1*) *encodeeval.simps eval-in-ids wf-stamp.elims*(*2*))
  from *stampdef* **have** *yv*: ∀ *yv* . ([*g*, *m*, *p*] ⊢ *y* ↦ *yv*) ⟶ *valid-value yv stampy*
    by (*meson assms*(*1*) *encodeeval.simps eval-in-ids wf-stamp.elims*(*2*))
  have ∀ *v*. *valid-value v* (*join stampx stampy*) = (*valid-value v stampx* ∧ *valid-value v stampy*)
    using *assms*(*3*)
    by (*simp add*: *join-valid stampdef*)
  then show *?thesis*
    using *assms* **unfolding** *alwaysDistinct.simps*

using *is-stamp-empty-valid stampdef xv yv* **by** *blast*
qed

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 –
have *no-valid*: ∀ *val*. ¬(*valid-value val* (*join* (*stamps x*) (*stamps y*)))
by (*smt* (*verit*, *best*) *is-stamp-empty.elims*(*2*) *valid-int valid-value.simps*(*1*) *assms*(*1*,*4*)
alwaysDistinct.simps)
obtain *xe ye* **where** *repr*: *rep g nid* (*BinaryExpr BinIntegerEquals xe ye*)
by (*metis assms*(*2*) *assms*(*3*) *encodeeval.simps rep-integer-equals*)
moreover have *evale*: [*m*, *p*] ⊢ (*BinaryExpr BinIntegerEquals xe ye*) ↦ *v*
by (*metis assms*(*3*) *calculation encodeeval.simps repDet*)
moreover have *repsub*: *rep g x xe* ∧ *rep g y ye*
by (*metis IRNode.distinct*(*1955*) *IRNode.distinct*(*1997*) *IRNode.inject*(*17*) *IntegerEqualsNodeE assms*(*2*) *calculation*)
ultimately obtain *xv yv* **where** *evalsub*: [*g*, *m*, *p*] ⊢ *x* ↦ *xv* ∧ [*g*, *m*, *p*] ⊢ *y* ↦ *yv*
by (*meson BinaryExprE encodeeval.simps*)
have *xvalid*: *valid-value xv* (*stamps x*)
using *assms*(*1*) *encode-in-ids encodeeval.simps evalsub wf-stamp.simps* **by** *blast*
then have *xint*: *is-IntegerStamp* (*stamps x*)
using *assms*(*4*) *valid-value.elims*(*2*) **by** *fastforce*
then have *xstamp*: *valid-stamp* (*stamps x*)
using *xvalid* **apply** (*cases xv*; *auto*)
apply (*smt* (*z3*) *valid-stamp.simps*(*6*) *valid-value.elims*(*1*))
using *is-IntegerStamp-def* **by** *fastforce*
have *yvalid*: *valid-value yv* (*stamps y*)
using *assms*(*1*) *encode-in-ids encodeeval.simps evalsub wf-stamp.simps* **by** *blast*
then have *yint*: *is-IntegerStamp* (*stamps y*)
using *assms*(*4*) *valid-value.elims*(*2*) **by** *fastforce*
then have *ystamp*: *valid-stamp* (*stamps y*)
using *yvalid* **apply** (*cases yv*; *auto*)
apply (*smt* (*z3*) *valid-stamp.simps*(*6*) *valid-value.elims*(*1*))
using *is-IntegerStamp-def* **by** *fastforce*
have *disjoint*: ¬(∃ *val* . ([*g*, *m*, *p*] ⊢ *x* ↦ *val*) ∧ ([*g*, *m*, *p*] ⊢ *y* ↦ *val*))
using *alwaysDistinct-evaluate*
using *assms*(*1*) *assms*(*4*) *xint yint xvalid yvalid xstamp ystamp* **by** *simp*
have *v* = *bin-eval BinIntegerEquals xv yv*
by (*metis BinaryExprE encodeeval.simps evale evalsub graphDet repsub*)
also have *v* ≠ *UndefVal*
using *evale* **by** *auto*
ultimately have ∃ *b1 b2*. *v* = *bool-to-val-bin b1 b2* (*xv* = *yv*)
unfolding *bin-eval.simps*

by (*smt* (*z3*) *Value.inject*(*1*) *bool-to-val-bin.simps intval-equals.elims*)
then show *?thesis*
by (*metis* (*mono-tags*, *lifting*) ‹(*v*::*Value*) ≠ *UndefVal*› *bool-to-val.elims bool-to-val-bin.simps disjoint evalsub val-to-bool.simps*(*1*))
qed
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*)))
using *assms* **apply** *auto*[*1*]
by (*simp add*: *take-bit-signed-take-bit*)

lemma *asConstant-valid*:
assumes *asConstant s* = *val*
assumes *val* ≠ *UndefVal*
assumes *valid-value v s*
shows *v* = *val*
proof −
obtain *b l h* **where** *s*: *s* = *IntegerStamp b l h*
using *assms*(*1*,*2*) **by** (*cases s*; *auto*)
obtain *vv* **where** *vdef*: *v* = *IntVal b vv*
using *assms*(*3*) *s valid-int* **by** *blast*
have *l* ≤ *int-signed-value b vv* ∧ *int-signed-value b vv* ≤ *h*
by (*metis* ‹(*v*::*Value*) = *IntVal* (*b*::*nat*) (*vv*::*64 word*)› *assms*(*3*) *s valid-value.simps*(*1*))
then have *veq*: *int-signed-value b vv* = *l*
by (*smt* (*verit*) *asConstant.simps*(*1*) *assms*(*1*) *assms*(*2*) *s*)
have *valdef*: *val* = *new-int b* (*word-of-int l*)
by (*metis asConstant.simps*(*1*) *assms*(*1*) *assms*(*2*) *s*)
have *take-bit b vv* = *vv*
by (*metis* ‹(*v*::*Value*) = *IntVal* (*b*::*nat*) (*vv*::*64 word*)› *assms*(*3*) *s valid-value.simps*(*1*))
then show *?thesis*
using *veq vdef valdef*
using *assms*(*3*) *s unwrap-valid* **by** *force*
qed

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 −
obtain *val* **where** *constx*: *asConstant* (*stamps x*) = *val*
by *simp*
moreover have *val* ≠ *UndefVal*
using *assms*(*4*) *calculation* **by** *auto*

then have *constx*: *val* = *asConstant* (*stamps y*)
  using *calculation assms*(*4*) **by** *force*
obtain *xe ye* **where** *repr*: *rep g nid* (*BinaryExpr BinIntegerEquals xe ye*)
  by (*metis assms*(*2*) *assms*(*3*) *encodeeval.simps rep-integer-equals*)
moreover have *evale*: [*m*, *p*] ⊢ (*BinaryExpr BinIntegerEquals xe ye*) ↦ *v*
  by (*metis assms*(*3*) *calculation encodeeval.simps repDet*)
moreover have *repsub*: *rep g x xe* ∧ *rep g y ye*
  by (*metis IRNode.distinct*(*1955*) *IRNode.distinct*(*1997*) *IRNode.inject*(*17*) *IntegerEqualsNodeE assms*(*2*) *calculation*)
ultimately obtain *xv yv* **where** *evalsub*: [*g*, *m*, *p*] ⊢ *x* ↦ *xv* ∧ [*g*, *m*, *p*] ⊢ *y* ↦ *yv*
  by (*meson BinaryExprE encodeeval.simps*)
have *xvalid*: *valid-value xv* (*stamps x*)
  using *assms*(*1*) *encode-in-ids encodeeval.simps evalsub wf-stamp.simps* **by** *blast*
then have *xint*: *is-IntegerStamp* (*stamps x*)
  using *assms*(*4*) *valid-value.elims*(*2*) **by** *fastforce*
have *yvalid*: *valid-value yv* (*stamps y*)
  using *assms*(*1*) *encode-in-ids encodeeval.simps evalsub wf-stamp.simps* **by** *blast*
then have *yint*: *is-IntegerStamp* (*stamps y*)
  using *assms*(*4*) *valid-value.elims*(*2*) **by** *fastforce*
have *eq*: ∀ *v1 v2*. (([*g*, *m*, *p*] ⊢ *x* ↦ *v1*) ∧ ([*g*, *m*, *p*] ⊢ *y* ↦ *v2*)) ⟶ *v1* = *v2*
  by (*metis asConstant-valid assms*(*4*) *encodeEvalDet evalsub neverDistinct.elims*(*1*) *xvalid yvalid*)
have *v* = *bin-eval BinIntegerEquals xv yv*
  by (*metis BinaryExprE encodeeval.simps evale evalsub graphDet repsub*)
also have *v* ≠ *UndefVal*
  using *evale* **by** *auto*
ultimately have ∃ *b1 b2*. *v* = *bool-to-val-bin b1 b2* (*xv* = *yv*)
  unfolding *bin-eval.simps*
  by (*smt* (*z3*) *Value.inject*(*1*) *bool-to-val-bin.simps intval-equals.elims*)
then show *?thesis*
  using ‹(*v*::*Value*) ≠ *UndefVal*› *eq evalsub* **by** *fastforce*
qed

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 −
obtain *xe ye* **where** *repr*: *rep g nid* (*BinaryExpr BinIntegerLessThan xe ye*)
  by (*metis assms*(*2*) *assms*(*3*) *encodeeval.simps rep-integer-less-than*)
moreover have *evale*: [*m*, *p*] ⊢ (*BinaryExpr BinIntegerLessThan xe ye*) ↦ *v*
  by (*metis assms*(*3*) *calculation encodeeval.simps repDet*)
moreover have *repsub*: *rep g x xe* ∧ *rep g y ye*
  by (*metis IRNode.distinct*(*2047*) *IRNode.distinct*(*2089*) *IRNode.inject*(*18*) *IntegerLessThanNodeE assms*(*2*) *repr*)
ultimately obtain *xv yv* **where** *evalsub*: [*g*, *m*, *p*] ⊢ *x* ↦ *xv* ∧ [*g*, *m*, *p*] ⊢ *y* ↦

yv
  by (*meson BinaryExprE encodeeval.simps*)
 have *vval*: *v* = *intval-less-than xv yv*
  by (*metis BinaryExprE bin-eval.simps*(*14*) *encodeEvalDet encodeeval.simps evale evalsub repsub*)
 then obtain *b xvv* **where** *xv* = *IntVal b xvv*
  by (*metis bin-eval.simps*(*14*) *defined-eval-is-intval evale evaltree-not-undef is-IntVal-def*)
 also have *xvalid*: *valid-value xv* (*stamps x*)
  by (*meson assms*(*1*) *encodeeval.simps eval-in-ids evalsub wf-stamp.elims*(*2*))
 then obtain *xl xh* **where** *xstamp*: *stamps x* = *IntegerStamp b xl xh*
  using *calculation valid-value.simps* **apply** (*cases stamps x*; *auto*)
  by *presburger*
 from *vval* **obtain** *yvv* **where** *yint*: *yv* = *IntVal b yvv*
  by (*metis Value.collapse*(*1*) *bin-eval.simps*(*14*) *bool-to-val-bin.simps calculation defined-eval-is-intval evale evaltree-not-undef intval-less-than.simps*(*1*))
 then have *yvalid*: *valid-value yv* (*stamps y*)
  using *assms*(*1*) *encodeeval.simps evalsub no-encoding wf-stamp.simps* **by** *blast*
 then obtain *yl yh* **where** *ystamp*: *stamps y* = *IntegerStamp b yl yh*
  using *calculation yint valid-value.simps* **apply** (*cases stamps y*; *auto*)
  by *presburger*
 have *int-signed-value b xvv* ≤ *xh*
  using *calculation valid-value.simps*(*1*) *xstamp xvalid* **by** *presburger*
 moreover have *yl* ≤ *int-signed-value b yvv*
  using *valid-value.simps*(*1*) *yint ystamp yvalid* **by** *presburger*
 moreover have *xh* < *yl*
  using *assms*(*4*) *xstamp ystamp* **by** *auto*
 ultimately have *int-signed-value b xvv* < *int-signed-value b yvv*
  by *linarith*
 then have *val-to-bool* (*intval-less-than xv yv*)
  by (*simp add*: ‹(*xv*::*Value*) = *IntVal* (*b*::*nat*) (*xvv*::*64 word*)› *yint*)
 then show *?thesis*
  by (*simp add*: *vval*)
qed

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 −
 obtain *xe ye* **where** *repr*: *rep g nid* (*BinaryExpr BinIntegerLessThan xe ye*)
  by (*metis assms*(*2*) *assms*(*3*) *encodeeval.simps rep-integer-less-than*)
 moreover have *evale*: [*m*, *p*] ⊢ (*BinaryExpr BinIntegerLessThan xe ye*) ↦ *v*
  by (*metis assms*(*3*) *calculation encodeeval.simps repDet*)
 moreover have *repsub*: *rep g x xe* ∧ *rep g y ye*
  by (*metis IRNode.distinct*(*2047*) *IRNode.distinct*(*2089*) *IRNode.inject*(*18*) *IntegerLessThanNodeE assms*(*2*) *repr*)

```
  ultimately obtain xv yv where evalsub: [g, m, p] ⊢ x ↦ xv ∧ [g, m, p] ⊢ y ↦
yv
    by (meson BinaryExprE encodeeval.simps)
  have vval: v = intval-less-than xv yv
   by (metis BinaryExprE bin-eval.simps(14) encodeEvalDet encodeeval.simps evale
evalsub repsub)
  then obtain b xvv where xv = IntVal b xvv
      by (metis bin-eval.simps(14) defined-eval-is-intval evale evaltree-not-undef
is-IntVal-def)
  also have xvalid: valid-value xv (stamps x)
    by (meson assms(1) encodeeval.simps eval-in-ids evalsub wf-stamp.elims(2))
  then obtain xl xh where xstamp: stamps x = IntegerStamp b xl xh
    using calculation valid-value.simps apply (cases stamps x; auto)
    by presburger
  from vval obtain yvv where yint: yv = IntVal b yvv
    by (metis Value.collapse(1) bin-eval.simps(14) bool-to-val-bin.simps calculation
defined-eval-is-intval evale evaltree-not-undef intval-less-than.simps(1))
  then have yvalid: valid-value yv (stamps y)
    using assms(1) encodeeval.simps evalsub no-encoding wf-stamp.simps by blast
  then obtain yl yh where ystamp: stamps y = IntegerStamp b yl yh
    using calculation yint valid-value.simps apply (cases stamps y; auto)
    by presburger
  have xl ≤ int-signed-value b xvv
    using calculation valid-value.simps(1) xstamp xvalid by presburger
  moreover have int-signed-value b yvv ≤ yh
    using valid-value.simps(1) yint ystamp yvalid by presburger
  moreover have xl ≥ yh
    using assms(4) xstamp ystamp by auto
  ultimately have int-signed-value b xvv ≥ int-signed-value b yvv
    by linarith
  then have ¬(val-to-bool (intval-less-than xv yv))
    by (simp add: ‹(xv::Value) = IntVal (b::nat) (xvv::64 word)› yint)
  then show ?thesis
    by (simp add: vval)
qed

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
  using assms(2) proof (induction kind g nid stamps True rule: tryFold.induct)
case (1 stamps x y)
  then show ?case
    using alwaysDistinct-valid assms by force
next
  case (2 stamps x y)
  then show ?case
    by (smt (verit, best) one-neq-zero tryFold.cases neverDistinct-valid assms
```

stampUnder-valid val-to-bool.simps(1))
next
case (*3 stamps x y*)
then show *?case*
by (*smt* (*verit, best*) *one-neq-zero tryFold.cases neverDistinct-valid assms val-to-bool.simps(1) stampUnder-valid*)
next
case (*4 stamps x y*)
then show *?case*
by *force*
qed

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*)
using *assms*(*2*) **proof** (*induction kind g nid stamps False rule*: *tryFold.induct*)
case (*1 stamps x y*)
then show *?case*
by (*smt* (*verit*) *stampOver-valid alwaysDistinct-valid tryFold.cases neverDistinct-valid val-to-bool.simps(1) assms*)
next
case (*2 stamps x y*)
then show *?case*
by *blast*
next
case (*3 stamps x y*)
then show *?case*
by *blast*
next
case (*4 stamps x y*)
then show *?case*
by (*smt* (*verit, del-insts*) *tryFold.cases alwaysDistinct-valid val-to-bool.simps(1) stampOver-valid assms*)
qed

12.2 Lift rules

inductive *condset-implies* :: *IRExpr set* ⇒ *IRExpr* ⇒ *bool* ⇒ *bool* **where**
impliesTrue:
(∃ *ce* ∈ *conds* . (*ce* ⇛ *cond*)) ⟹ *condset-implies conds cond True* |
impliesFalse:
(∃ *ce* ∈ *conds* . (*ce* ⇛¬ *cond*)) ⟹ *condset-implies conds cond False*

code-pred (*modes*: *i* ⇒ *i* ⇒ *i* ⇒ *bool*) *condset-implies* **.**

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 ⊢ *cid* ≃ *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 ⊢ *cid* ≃ *cond*;
condNode = *kind g cid*;
conds-implies conds stamps condNode cond = (*Some False*);
g' = *constantCondition False ifcond* (*kind g ifcond*) *g*
⟧ ⟹ *ConditionalEliminationStep conds stamps ifcond g* g' |

unknown:
⟦*kind g ifcond* = (*IfNode cid t f*);
g ⊢ *cid* ≃ *cond*;
condNode = *kind g cid*;
conds-implies conds stamps condNode cond = *None*
⟧ ⟹ *ConditionalEliminationStep conds stamps ifcond g g* |

notIfNode:
¬(*is-IfNode* (*kind g ifcond*)) ⟹
ConditionalEliminationStep conds stamps ifcond g g

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *ConditionalEliminationStep* **.**

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**

$[\![pre = []]\!]$
$\Longrightarrow$ *dominators-all g c nid doms pre c doms pre* |

$[\![pre = pr \# xs;$
$(dominators\ g\ c\ pr\ (doms',\ c'));$
$dominators\text{-}all\ g\ c'\ pr\ (doms \cup \{doms'\})\ xs\ c''\ doms''\ pre']\!]$
$\Longrightarrow$ *dominators-all g c nid doms pre c'' doms'' pre'* |

$[\![preds\ g\ nid = []]\!]$
$\Longrightarrow$ *dominators g c nid* $(\{nid\},\ c)$ |

$[\![c\ nid = None;$
$preds\ g\ nid = x \# xs;$
$dominators\text{-}all\ g\ c\ nid\ \{\}\ (preds\ g\ nid)\ c'\ doms\ pre';$
$c'' = c'(nid \mapsto (\{nid\} \cup (\bigcap doms)))]\!]$
$\Longrightarrow$ *dominators g c nid* $((\{nid\} \cup (\bigcap doms)),\ c'')$ |

$[\![c\ nid = Some\ doms]\!]$
$\Longrightarrow$ *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 $\bigcap\ \emptyset = coset$ [] which swallows anything unioned with it.
value $\bigcap(\{\}::nat\ set\ set)$
value $- \bigcap(\{\}::nat\ set\ set)$
value $\bigcap(\{\{\},\ \{0\}\}::nat\ set\ set)$
value $\{0::nat\} \cup (\bigcap\{\})$

code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow o \Rightarrow o \Rightarrow bool$) *dominators-all* **.**
code-pred (*modes*: $i \Rightarrow i \Rightarrow i \Rightarrow o \Rightarrow bool$) *dominators* **.**

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* **.**

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* **.**

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* **.**

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* **.**

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*)
using *StepE*
by (*smt* (*verit*, *best*) *IfNode Pair-inject stepDet*)

lemma *ifNodeHasCondEvalStutter*:
assumes (*g m p h* ⊢ *nid* ⇝ *nid′*)
assumes *kind g nid* = *IfNode cond t f*
shows ∃ *v*. ([*g*, *m*, *p*] ⊢ *cond* ↦ *v*)
using *IfNodeStepE assms*(*1*) *assms*(*2*) *stutter.cases* **unfolding** *encodeeval.simps*
by (*smt* (*verit*, *ccfv-SIG*) *IfNodeCond*)

lemma *ifNodeHasCondEval*:
assumes (*g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid′*, *m′*, *h′*))
assumes *kind g nid* = *IfNode cond t f*
shows ∃ *v*. ([*g*, *m*, *p*] ⊢ *cond* ↦ *v*)
using *IfNodeStepE assms*(*1*) *assms*(*2*) **apply** *auto*[*1*]
by (*smt* (*verit*) *IRNode.disc*(*1966*) *IRNode.distinct*(*1733*) *IRNode.distinct*(*1735*) *IRNode.distinct*(*1755*) *IRNode.distinct*(*1757*) *IRNode.distinct*(*1777*) *IRNode.distinct*(*1783*) *IRNode.distinct*(*1787*) *IRNode.distinct*(*1789*) *IRNode.distinct*(*401*) *IRNode.distinct*(*755*) *StutterStep fst-conv ifNodeHasCondEvalStutter is-AbstractEndNode.simps is-EndNode.simps*(*16*) *snd-conv step.cases*)

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-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 −
have *g1step*: $g, p \vdash (nid, m, h) \rightarrow (tb, m, h)$
by (*meson IfNode assms*(*1*) *assms*(*2*) *assms*(*3*) *encodeeval.simps*)
have *g2step*: $g', p \vdash (nid, m, h) \rightarrow (tb, m, h)$
using g' **unfolding** *replace-usages.simps*
by (*simp add*: *stepRefNode*)
from *g1step g2step* **show** *?thesis*
using *StutterStep* **by** *blast*
qed

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-usages nid tb g*
shows $\exists nid' .(g\ m\ p\ h \vdash nid \rightsquigarrow nid') \longrightarrow (g'\ m\ p\ h \vdash nid \rightsquigarrow nid')$
using *replace-if-t assms* **by** *blast*

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-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 −
have *g1step*: $g, p \vdash (nid, m, h) \rightarrow (fb, m, h)$
by (*meson IfNode assms*(*1*) *assms*(*2*) *assms*(*3*) *encodeeval.simps*)
have *g2step*: $g', p \vdash (nid, m, h) \rightarrow (fb, m, h)$
using g' **unfolding** *replace-usages.simps*
by (*simp add*: *stepRefNode*)
from *g1step g2step* **show** *?thesis*
using *StutterStep* **by** *blast*
qed

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 *conds-valid*: $\forall\ c \in conds\ .\ \exists\ v.\ ([m, p] \vdash c \mapsto v) \land$ *val-to-bool v*
assumes *ce*: *ConditionalEliminationStep conds stamps nid g* g'

shows $\exists$ *nid'* .(*g m p h* $\vdash$ *nid* $\leadsto$ *nid'*) $\longrightarrow$ (*g' m p h* $\vdash$ *nid* $\leadsto$ *nid'*)
using *ce* **using** *assms*
proof (*induct nid g g' rule*: *ConditionalEliminationStep.induct*)
case (*impliesTrue g ifcond cid t f cond conds g'*)
show *?case* **proof** (*cases* $\exists$ *nid'*. (*g m p h* $\vdash$ *ifcond* $\leadsto$ *nid'*))
case *True*
show *?thesis*
by (*metis StutterStep constantConditionNoIf constantConditionTrue impliesTrue.hyps*(*5*))
next
case *False*
then show *?thesis* **by** *auto*
qed
next
case (*impliesFalse g ifcond cid t f cond conds g'*)
then show *?case*
proof (*cases* $\exists$ *nid'*. (*g m p h* $\vdash$ *ifcond* $\leadsto$ *nid'*))
case *True*
then show *?thesis*
by (*metis StutterStep constantConditionFalse constantConditionNoIf impliesFalse.hyps*(*5*))
next
case *False*
then show *?thesis*
by *auto*
qed
next
case (*unknown g ifcond cid t f cond condNode conds stamps*)
then show *?case*
by *blast*
next
case (*notIfNode g ifcond conds stamps*)
then show *?case*
by *blast*
qed

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-graph g* $\wedge$ *wf-stamp g stamps* $\wedge$ *wf-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-to-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* | *g* $\sim$ *g'*
using *ce gstep* **using** *assms*
proof (*induct nid g g' rule*: *ConditionalEliminationStep.induct*)

case (*impliesTrue g ifcond cid t f cond condNode conds stamps g'*)
from *impliesTrue*(*5*) **obtain** *h* **where** *gstep*: *g*, *p* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)
using *IfNode encodeeval.simps ifNodeHasCondEval impliesTrue.hyps*(*1*) *impliesTrue.hyps*(*2*) *impliesTrue.hyps*(*3*) *impliesTrue.prems*(*4*) *implies-impliesnot-valid implies-valid.simps repDet*
by (*smt* (*verit*) *conds-implies.elims condset-implies.simps impliesTrue.hyps*(*4*) *impliesTrue.prems*(*1*) *impliesTrue.prems*(*2*) *option.distinct*(*1*) *option.inject tryFoldTrue-valid*)
have *g'*, *p* ⊢ (*ifcond*, *m*, *h*) → (*t*, *m*, *h*)
using *constantConditionTrue impliesTrue.hyps*(*1*) *impliesTrue.hyps*(*5*) **by** *blast*
then show *?case* **using** *gstep*
by (*metis stepDet strong-noop-bisimilar.intros*)
next
case (*impliesFalse g ifcond cid t f cond condNode conds stamps g'*)
from *impliesFalse*(*5*) **obtain** *h* **where** *gstep*: *g*, *p* ⊢ (*ifcond*, *m*, *h*) → (*f*, *m*, *h*)
using *IfNode encodeeval.simps ifNodeHasCondEval impliesFalse.hyps*(*1*) *impliesFalse.hyps*(*2*) *impliesFalse.hyps*(*3*) *impliesFalse.prems*(*4*) *implies-impliesnot-valid impliesnot-valid.simps repDet*
by (*smt* (*verit*) *conds-implies.elims condset-implies.simps impliesFalse.hyps*(*4*) *impliesFalse.prems*(*1*) *impliesFalse.prems*(*2*) *option.distinct*(*1*) *option.inject tryFoldFalse-valid*)
have *g'*, *p* ⊢ (*ifcond*, *m*, *h*) → (*f*, *m*, *h*)
using *constantConditionFalse impliesFalse.hyps*(*1*) *impliesFalse.hyps*(*5*) **by** *blast*
then show *?case* **using** *gstep*
by (*metis stepDet strong-noop-bisimilar.intros*)
next
case (*unknown g ifcond cid t f cond condNode conds stamps*)
then show *?case*
using *strong-noop-bisimilar.simps* **by** *presburger*
next
case (*notIfNode g ifcond conds stamps*)
then show *?case*
using *strong-noop-bisimilar.simps* **by** *presburger*
qed

experiment begin

lemma *inverse-succ*:
∀ *n'* ∈ (*succ g n*). *n* ∈ *ids g* ⟶ *n* ∈ (*predecessors g n'*)
by *simp*

lemma *sequential-successors*:
assumes *is-sequential-node n*
shows *successors-of n* ≠ []
using *assms* **by** (*cases n*; *auto*)

lemma *nid'-succ*:

assumes *nid* ∈ *ids g*
assumes ¬(*is-AbstractEndNode* (*kind g nid0*))
assumes *g*, *p* ⊢ (*nid0*, *m0*, *h0*) → (*nid*, *m*, *h*)
shows *nid* ∈ *succ g nid0*
using *assms*(*3*) **proof** (*induction* (*nid0*, *m0*, *h0*) (*nid*, *m*, *h*) *rule*: *step.induct*)
case *SequentialNode*
then show *?case*
by (*metis length-greater-0-conv nth-mem sequential-successors succ.simps*)
next
case (*FixedGuardNode cond before val*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-FixedGuardNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
using *FixedGuardNode.hyps*(*5*) **by** *blast*
next
case (*BytecodeExceptionNode args st exceptionType ref*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-BytecodeExceptionNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*IfNode cond tb fb val*)
then have {*tb*, *fb*} = *succ g nid0*
using *IRNodes.successors-of-IfNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by (*metis IfNode.hyps*(*3*) *insert-iff*)
next
case (*EndNodes i phis inps vs*)
then show *?case* **using** *assms*(*2*) **by** *blast*
next
case (*NewArrayNode len st length′ arrayType h′ ref refNo*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-NewArrayNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*ArrayLengthNode x ref arrayVal length′*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-ArrayLengthNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*LoadIndexedNode index guard array indexVal ref arrayVal loaded*)
then have {*nid*} = *succ g nid0*

using *IRNodes.successors-of-LoadIndexedNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*StoreIndexedNode check val st index guard array indexVal ref value arrayVal updated*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-StoreIndexedNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*NewInstanceNode cname obj ref*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-NewInstanceNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*LoadFieldNode f obj ref*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-LoadFieldNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*SignedDivNode x y zero sb v1 v2*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-SignedDivNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*SignedRemNode x y zero sb v1 v2*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-SignedRemNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*StaticLoadFieldNode f*)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-LoadFieldNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*StoreFieldNode* - - - - - -)

then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-StoreFieldNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
next
case (*StaticStoreFieldNode* - - - -)
then have {*nid*} = *succ g nid0*
using *IRNodes.successors-of-StoreFieldNode* **unfolding** *succ.simps*
by (*metis empty-set list.simps*(*15*))
then show *?case*
by *blast*
qed

lemma *nid′-pred*:
assumes *nid* ∈ *ids g*
assumes ¬(*is-AbstractEndNode* (*kind g nid0*))
assumes *g*, *p* ⊢ (*nid0*, *m0*, *h0*) → (*nid*, *m*, *h*)
shows *nid0* ∈ *predecessors g nid*
using *assms*
by (*meson inverse-succ nid′-succ step-in-ids*)

definition *wf-pred*:
wf-pred g = (∀ *n* ∈ *ids g*. *card* (*predecessors g n*) = *1*)

lemma
assumes ¬(*is-AbstractMergeNode* (*kind g n′*))
assumes *wf-pred g*
shows ∃ *v*. *predecessors g n* = {*v*} ∧ *pred g n′* = *Some v*
using *assms* **unfolding** *pred.simps* **sorry**

lemma *inverse-succ1*:
assumes ¬(*is-AbstractEndNode* (*kind g n′*))
assumes *wf-pred g*
shows ∀ *n′* ∈ (*succ g n*). *n* ∈ *ids g* ⟶ *Some n* = (*pred g n′*)
using *assms* **sorry**

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 −
obtain *tb fb* **where** [*tb*, *fb*] = *successors-of* (*kind g ifcond*)
by (*simp add*: *assms*(*3*))
have *nid0* = *ifcond*
using *assms step.IfNode* **sorry**
show *?thesis* **sorry**

qed

end

end