Verifying term graph optimizations using Isabelle/HOL

Isabelle/HOL Theories

September 22, 2022

Abstract

Our objective is to formally verify the correctness of the hundreds of expression optimization rules used within the GraalVM compiler. When defining the semantics of a programming language, expressions naturally form abstract syntax trees, or, terms. However, in order to facilitate sharing of common subexpressions, modern compilers represent expressions as term graphs. Defining the semantics of term graphs is more complicated than defining the semantics of their equivalent term representations. More significantly, defining optimizations directly on term graphs and proving semantics preservation is considerably more complicated than on the equivalent term representations. On terms, optimizations can be expressed as conditional term rewriting rules, and proofs that the rewrites are semantics preserving are relatively straightforward. In this paper, we explore an approach to using term rewrites to verify term graph transformations of optimizations within the GraalVM compiler. This approach significantly reduces the overall verification effort and allows for simpler encoding of optimization rules.

Contents

1 Operator Semantics

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

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

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

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

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

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

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

experiment begin

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

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

setup-lifting *type-definition-IntWidth*

lift-definition *IntWidthBits* :: *IntWidth* ⇒ *nat*
  is *λw. w* **.**
end

experiment begin

Option 3: explicit type stored with each integer value.

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

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

fun int-signed :: IntType ⇒ bool where
  int-signed ILong  = True |
  int-signed IInt   = True |
  int-signed IShort = True |
  int-signed IChar  = False |
  int-signed IByte  =  True |
  int-signed IBoolean = True
end
```

Option 4: int64 with the number of significant bits.

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

datatype (discs-sels) Value  =
  UndefVal |







  IntVal iwidth int64 |

  ObjRef objref |
  ObjStr string

fun intval-bits :: Value ⇒ nat where
  intval-bits (IntVal b v) = b

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

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*

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 *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*

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

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

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

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

by (*smt* (*verit*, *best*) *assms diff-le-self diff-less int-signed-value.simps less-one mask-eq-take-bit-minus-one neg-one.simps nle-le signed-minus-1 signed-take-bit-of-minus-1 signed-take-bit-take-bit verit-comp-simplify1*(*1*))

1.1 Arithmetic Operators

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

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

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

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

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

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

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

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

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

intval-negate (*IntVal t v*) = *new-int t* (− *v*) |
intval-negate - = *UndefVal*

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

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

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

1.2 Bitwise Operators

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

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

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

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

1.3 Comparison Operators

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

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

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

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

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

1.4 Narrowing and Widening Operators

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

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

fun intval-narrow :: nat ⇒ 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

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

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 intval-narrow.simps neq0-conv intval-bits.simps
  by (metis Value.disc(2) intval-narrow.elims le-trans)

lemma intval-sign-extend-ok:
  assumes intval-sign-extend inBits outBits val ≠ UndefVal
  shows 0 < inBits ∧
        inBits ≤ outBits ∧ outBits ≤ 64 ∧
        is-IntVal val ∧
        intval-bits val = inBits
  using assms intval-sign-extend.simps neq0-conv
  by (metis intval-bits.simps intval-sign-extend.elims is-IntVal-def)

lemma intval-zero-extend-ok:
  assumes intval-zero-extend inBits outBits val ≠ UndefVal
  shows 0 < inBits ∧
        inBits ≤ outBits ∧ outBits ≤ 64 ∧
        is-IntVal val ∧
        intval-bits val = inBits
  using assms intval-sign-extend.simps neq0-conv
  by (metis intval-bits.simps intval-zero-extend.elims is-IntVal-def)
```

1.5 Bit-Shifting Operators

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

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

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

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

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

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


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

value (255 :: 8 word) >>> (2 :: nat)


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

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

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

fun *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*

1.5.1 Examples of Narrowing / Widening Functions

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

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

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

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

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

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

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

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

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

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

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

end

1.6 Fixed-width Word Theories

theory *ValueThms*
  imports *Values*
begin

1.6.1 Support Lemmas for Upper/Lower Bounds

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

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

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

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


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


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

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

lemma signed-take-bit-range:
  fixes ival :: 'a :: len word
  assumes n < LENGTH('a)
  assumes val = sint(signed-take-bit n ival)
  shows − (2 ^ n) ≤ 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 ≤ 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 ≤ 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 signed-take-bit-range assms
   by (smt (verit, ccfv-SIG) One-nat-def diff-less int-signed-value.elims len-gt-0
len-num1 power-less-imp-less-exp power-strict-increasing sint-greater-eq sint-less)
```

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

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

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

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

```
lemma take-bit-redundant[simp]:
  fixes ival :: 'a :: len word
  assumes 0 < n
  assumes n < LENGTH('a)
  shows signed-take-bit (n − 1) (take-bit n ival) = signed-take-bit (n − 1) ival
proof −
  have ¬ (n ≤ n − 1) using assms by arith
  then have ⋀i . signed-take-bit (n − 1) (take-bit n i) = signed-take-bit (n−1) i
    using signed-take-bit-take-bit by (metis (mono-tags))
  then show ?thesis
    by blast
qed
```

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

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

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

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

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

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

lemma *scast-bigger-min-bound*:
  assumes (*result* :: *'b* :: *len word*) = *scast* (*v* :: *'a* :: *len word*)

```
  shows − (2 ^ LENGTH('a) div 2) ≤ sint result
  using sint-ge lower-bounds-equiv scast-min-bound
  by (smt (verit) assms len-gt-0 nat-less-le not-less scast-max-bound)

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

Results about *new_int*.

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

1.6.2 Support lemmas for take bit and signed take bit.

Lemmas for removing redundant take_bit wrappers.

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

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


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

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

lemma *take-bit-dist-neg*[*simp*]:
  fixes *ix* :: *'a* :: *len word*
  shows *take-bit b* (− *take-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*
 by (*smt* (*verit*, *best*) *Suc-diff-1 assms lessI linorder-not-less signed-take-bit-take-bit*)

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

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

end

2 Stamp Typing

theory *Stamp*
  imports *Values*
begin

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

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

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

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

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

  is-stamp-empty x = False
```

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

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

```
fun valid-stamp :: Stamp ⇒ bool where
  valid-stamp (IntegerStamp bits lo hi) =
     (0 < bits ∧ bits ≤ 64 ∧
     fst (bit-bounds bits) ≤ lo ∧ lo ≤ snd (bit-bounds bits) ∧
     fst (bit-bounds bits) ≤ hi ∧ hi ≤ snd (bit-bounds bits)) |
  valid-stamp s = True
```

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

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

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

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

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

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

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

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

— Calculate the join stamp of two stamps

```
fun join :: Stamp ⇒ Stamp ⇒ Stamp where
  join VoidStamp VoidStamp = VoidStamp |
  join (IntegerStamp b1 l1 u1) (IntegerStamp b2 l2 u2) = (
    if b1 ≠ b2 then IllegalStamp else
    (IntegerStamp b1 (max l1 l2) (min u1 u2))
  ) |

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

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

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

— Determine if two stamps never have value overlaps i.e. their join is empty

```
fun alwaysDistinct :: Stamp ⇒ Stamp ⇒ bool where
```

```
  alwaysDistinct stamp1 stamp2 = is-stamp-empty (join stamp1 stamp2)
```

— Determine if two stamps must always be the same value i.e. two equal constants

```
fun neverDistinct :: Stamp ⇒ Stamp ⇒ bool where
   neverDistinct stamp1 stamp2 = (asConstant stamp1 = asConstant stamp2 ∧
asConstant stamp1 ≠ UndefVal)

fun constantAsStamp :: Value ⇒ Stamp where
 constantAsStamp (IntVal b v) = (IntegerStamp b (int-signed-value b v) (int-signed-value
b v)) |

  constantAsStamp - = IllegalStamp
```

— Define when a runtime value is valid for a stamp. The stamp bounds must be valid, and val must be zero-extended.

```
fun valid-value :: Value ⇒ Stamp ⇒ bool where
  valid-value (IntVal b1 val) (IntegerStamp b l h) =
     (if b1 = b then
       valid-stamp (IntegerStamp b l h) ∧
       take-bit b val = val ∧
       l ≤ int-signed-value b val ∧ int-signed-value b val ≤ h
      else False) |

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

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

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

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

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

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

3 Graph Representation

3.1 IR Graph Nodes

theory *IRNodes*
  imports
    Values
begin

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

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

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

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

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

datatype (*discs-sels*) *IRNode* =
  AbsNode (*ir-value*: *INPUT*)
  | *AddNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
  | *AndNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
  | *BeginNode* (*ir-next*: *SUCC*)

| *BytecodeExceptionNode* (*ir-arguments*: *INPUT list*) (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *ConditionalNode* (*ir-condition*: *INPUT-COND*) (*ir-trueValue*: *INPUT*) (*ir-falseValue*: *INPUT*)
| *ConstantNode* (*ir-const*: *Value*)
| *DynamicNewArrayNode* (*ir-elementType*: *INPUT*) (*ir-length*: *INPUT*) (*ir-voidClass-opt*: *INPUT option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)
| *EndNode*
| *ExceptionObjectNode* (*ir-stateAfter-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

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

| *ReturnNode* (*ir-result-opt*: *INPUT option*) (*ir-memoryMap-opt*: *INPUT-EXT option*)
| *RightShiftNode* (*ir-x*: *INPUT*) (*ir-y*: *INPUT*)
| *ShortCircuitOrNode* (*ir-x*: *INPUT-COND*) (*ir-y*: *INPUT-COND*)
| *SignExtendNode* (*ir-inputBits*: *nat*) (*ir-resultBits*: *nat*) (*ir-value*: *INPUT*)
| *SignedDivNode* (*ir-nid*: *ID*) (*ir-x*: *INPUT*) (*ir-y*: *INPUT*) (*ir-zeroCheck-opt*: *INPUT-GUARD option*) (*ir-stateBefore-opt*: *INPUT-STATE option*) (*ir-next*: *SUCC*)

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

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

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

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

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

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

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

inputs-of-BytecodeExceptionNode:
inputs-of (*BytecodeExceptionNode arguments stateAfter next*) = *arguments* @ (*opt-to-list stateAfter*) |
inputs-of-ConditionalNode:
inputs-of (*ConditionalNode condition trueValue falseValue*) = [*condition*, *trueValue*, *falseValue*] |
inputs-of-ConstantNode:
inputs-of (*ConstantNode const*) = [] |
inputs-of-DynamicNewArrayNode:
inputs-of (*DynamicNewArrayNode elementType length0 voidClass stateBefore next*) = [*elementType*, *length0*] @ (*opt-to-list voidClass*) @ (*opt-to-list stateBefore*) |
inputs-of-EndNode:
inputs-of (*EndNode*) = [] |
inputs-of-ExceptionObjectNode:
inputs-of (*ExceptionObjectNode stateAfter next*) = (*opt-to-list stateAfter*) |
inputs-of-FrameState:
inputs-of (*FrameState monitorIds outerFrameState values virtualObjectMappings*) = *monitorIds* @ (*opt-to-list outerFrameState*) @ (*opt-list-to-list values*) @ (*opt-list-to-list virtualObjectMappings*) |
inputs-of-IfNode:
inputs-of (*IfNode condition trueSuccessor falseSuccessor*) = [*condition*] |
inputs-of-IntegerBelowNode:
inputs-of (*IntegerBelowNode x y*) = [*x*, *y*] |
inputs-of-IntegerEqualsNode:
inputs-of (*IntegerEqualsNode x y*) = [*x*, *y*] |
inputs-of-IntegerLessThanNode:
inputs-of (*IntegerLessThanNode x y*) = [*x*, *y*] |
inputs-of-InvokeNode:
inputs-of (*InvokeNode nid0 callTarget classInit stateDuring stateAfter next*) = *callTarget* # (*opt-to-list classInit*) @ (*opt-to-list stateDuring*) @ (*opt-to-list stateAfter*) |
inputs-of-InvokeWithExceptionNode:
inputs-of (*InvokeWithExceptionNode nid0 callTarget classInit stateDuring stateAfter next exceptionEdge*) = *callTarget* # (*opt-to-list classInit*) @ (*opt-to-list stateDuring*) @ (*opt-to-list stateAfter*) |
inputs-of-IsNullNode:
inputs-of (*IsNullNode value*) = [*value*] |
inputs-of-KillingBeginNode:
inputs-of (*KillingBeginNode next*) = [] |
inputs-of-LeftShiftNode:
inputs-of (*LeftShiftNode x y*) = [*x*, *y*] |
inputs-of-LoadFieldNode:
inputs-of (*LoadFieldNode nid0 field object next*) = (*opt-to-list object*) |
inputs-of-LogicNegationNode:
inputs-of (*LogicNegationNode value*) = [*value*] |
inputs-of-LoopBeginNode:
inputs-of (*LoopBeginNode ends overflowGuard stateAfter next*) = *ends* @ (*opt-to-list overflowGuard*) @ (*opt-to-list stateAfter*) |

inputs-of-LoopEndNode:
inputs-of (*LoopEndNode loopBegin*) = [*loopBegin*] |
inputs-of-LoopExitNode:
inputs-of (*LoopExitNode loopBegin stateAfter next*) = *loopBegin* # (*opt-to-list stateAfter*) |
inputs-of-MergeNode:
inputs-of (*MergeNode ends stateAfter next*) = *ends* @ (*opt-to-list stateAfter*) |
inputs-of-MethodCallTargetNode:
inputs-of (*MethodCallTargetNode targetMethod arguments*) = *arguments* |
inputs-of-MulNode:
inputs-of (*MulNode x y*) = [*x*, *y*] |
inputs-of-NarrowNode:
inputs-of (*NarrowNode inputBits resultBits value*) = [*value*] |
inputs-of-NegateNode:
inputs-of (*NegateNode value*) = [*value*] |
inputs-of-NewArrayNode:
inputs-of (*NewArrayNode length0 stateBefore next*) = *length0* # (*opt-to-list stateBefore*) |
inputs-of-NewInstanceNode:
inputs-of (*NewInstanceNode nid0 instanceClass stateBefore next*) = (*opt-to-list stateBefore*) |
inputs-of-NotNode:
inputs-of (*NotNode value*) = [*value*] |
inputs-of-OrNode:
inputs-of (*OrNode x y*) = [*x*, *y*] |
inputs-of-ParameterNode:
inputs-of (*ParameterNode index*) = [] |
inputs-of-PiNode:
inputs-of (*PiNode object guard*) = *object* # (*opt-to-list guard*) |
inputs-of-ReturnNode:
inputs-of (*ReturnNode result memoryMap*) = (*opt-to-list result*) @ (*opt-to-list memoryMap*) |
inputs-of-RightShiftNode:
inputs-of (*RightShiftNode x y*) = [*x*, *y*] |
inputs-of-ShortCircuitOrNode:
inputs-of (*ShortCircuitOrNode x y*) = [*x*, *y*] |
inputs-of-SignExtendNode:
inputs-of (*SignExtendNode inputBits resultBits value*) = [*value*] |
inputs-of-SignedDivNode:
inputs-of (*SignedDivNode nid0 x y zeroCheck stateBefore next*) = [*x*, *y*] @ (*opt-to-list zeroCheck*) @ (*opt-to-list stateBefore*) |
inputs-of-SignedRemNode:
inputs-of (*SignedRemNode nid0 x y zeroCheck stateBefore next*) = [*x*, *y*] @ (*opt-to-list zeroCheck*) @ (*opt-to-list stateBefore*) |
inputs-of-StartNode:
inputs-of (*StartNode stateAfter next*) = (*opt-to-list stateAfter*) |
inputs-of-StoreFieldNode:
inputs-of (*StoreFieldNode nid0 field value stateAfter object next*) = *value* # (*opt-to-list stateAfter*) @ (*opt-to-list object*) |

inputs-of-SubNode:
inputs-of (*SubNode x y*) = [*x*, *y*] |
inputs-of-UnsignedRightShiftNode:
inputs-of (*UnsignedRightShiftNode x y*) = [*x*, *y*] |
inputs-of-UnwindNode:
inputs-of (*UnwindNode exception*) = [*exception*] |
inputs-of-ValuePhiNode:
inputs-of (*ValuePhiNode nid0 values merge*) = *merge* # *values* |
inputs-of-ValueProxyNode:
inputs-of (*ValueProxyNode value loopExit*) = [*value*, *loopExit*] |
inputs-of-XorNode:
inputs-of (*XorNode x y*) = [*x*, *y*] |
inputs-of-ZeroExtendNode:
inputs-of (*ZeroExtendNode inputBits resultBits value*) = [*value*] |
inputs-of-NoNode: *inputs-of* (*NoNode*) = [] |

inputs-of-RefNode: *inputs-of* (*RefNode ref*) = [*ref*]

fun *successors-of* :: *IRNode* ⇒ *ID list* **where**
successors-of-AbsNode:
successors-of (*AbsNode value*) = [] |
successors-of-AddNode:
successors-of (*AddNode x y*) = [] |
successors-of-AndNode:
successors-of (*AndNode x y*) = [] |
successors-of-BeginNode:
successors-of (*BeginNode next*) = [*next*] |
successors-of-BytecodeExceptionNode:
successors-of (*BytecodeExceptionNode arguments stateAfter next*) = [*next*] |
successors-of-ConditionalNode:
successors-of (*ConditionalNode condition trueValue falseValue*) = [] |
successors-of-ConstantNode:
successors-of (*ConstantNode const*) = [] |
successors-of-DynamicNewArrayNode:
successors-of (*DynamicNewArrayNode elementType length0 voidClass stateBefore next*) = [*next*] |
successors-of-EndNode:
successors-of (*EndNode*) = [] |
successors-of-ExceptionObjectNode:
successors-of (*ExceptionObjectNode stateAfter next*) = [*next*] |
successors-of-FrameState:
successors-of (*FrameState monitorIds outerFrameState values virtualObjectMappings*) = [] |
successors-of-IfNode:
successors-of (*IfNode condition trueSuccessor falseSuccessor*) = [*trueSuccessor*, *falseSuccessor*] |
successors-of-IntegerBelowNode:

successors-of (*IntegerBelowNode x y*) = [] |
successors-of-IntegerEqualsNode:
successors-of (*IntegerEqualsNode x y*) = [] |
successors-of-IntegerLessThanNode:
successors-of (*IntegerLessThanNode x y*) = [] |
successors-of-InvokeNode:
successors-of (*InvokeNode nid0 callTarget classInit stateDuring stateAfter next*) = [*next*] |
successors-of-InvokeWithExceptionNode:
successors-of (*InvokeWithExceptionNode nid0 callTarget classInit stateDuring stateAfter next exceptionEdge*) = [*next*, *exceptionEdge*] |
successors-of-IsNullNode:
successors-of (*IsNullNode value*) = [] |
successors-of-KillingBeginNode:
successors-of (*KillingBeginNode next*) = [*next*] |
successors-of-LeftShiftNode:
successors-of (*LeftShiftNode x y*) = [] |
successors-of-LoadFieldNode:
successors-of (*LoadFieldNode nid0 field object next*) = [*next*] |
successors-of-LogicNegationNode:
successors-of (*LogicNegationNode value*) = [] |
successors-of-LoopBeginNode:
successors-of (*LoopBeginNode ends overflowGuard stateAfter next*) = [*next*] |
successors-of-LoopEndNode:
successors-of (*LoopEndNode loopBegin*) = [] |
successors-of-LoopExitNode:
successors-of (*LoopExitNode loopBegin stateAfter next*) = [*next*] |
successors-of-MergeNode:
successors-of (*MergeNode ends stateAfter next*) = [*next*] |
successors-of-MethodCallTargetNode:
successors-of (*MethodCallTargetNode targetMethod arguments*) = [] |
successors-of-MulNode:
successors-of (*MulNode x y*) = [] |
successors-of-NarrowNode:
successors-of (*NarrowNode inputBits resultBits value*) = [] |
successors-of-NegateNode:
successors-of (*NegateNode value*) = [] |
successors-of-NewArrayNode:
successors-of (*NewArrayNode length0 stateBefore next*) = [*next*] |
successors-of-NewInstanceNode:
successors-of (*NewInstanceNode nid0 instanceClass stateBefore next*) = [*next*] |
successors-of-NotNode:
successors-of (*NotNode value*) = [] |
successors-of-OrNode:
successors-of (*OrNode x y*) = [] |
successors-of-ParameterNode:
successors-of (*ParameterNode index*) = [] |
successors-of-PiNode:
successors-of (*PiNode object guard*) = [] |

successors-of-ReturnNode:
successors-of (*ReturnNode result memoryMap*) = [] |
successors-of-RightShiftNode:
successors-of (*RightShiftNode x y*) = [] |
successors-of-ShortCircuitOrNode:
successors-of (*ShortCircuitOrNode x y*) = [] |
successors-of-SignExtendNode:
successors-of (*SignExtendNode inputBits resultBits value*) = [] |
successors-of-SignedDivNode:
successors-of (*SignedDivNode nid0 x y zeroCheck stateBefore next*) = [*next*] |
successors-of-SignedRemNode:
successors-of (*SignedRemNode nid0 x y zeroCheck stateBefore next*) = [*next*] |
successors-of-StartNode:
successors-of (*StartNode stateAfter next*) = [*next*] |
successors-of-StoreFieldNode:
successors-of (*StoreFieldNode nid0 field value stateAfter object next*) = [*next*] |
successors-of-SubNode:
successors-of (*SubNode x y*) = [] |
successors-of-UnsignedRightShiftNode:
successors-of (*UnsignedRightShiftNode x y*) = [] |
successors-of-UnwindNode:
successors-of (*UnwindNode exception*) = [] |
successors-of-ValuePhiNode:
successors-of (*ValuePhiNode nid0 values merge*) = [] |
successors-of-ValueProxyNode:
successors-of (*ValueProxyNode value loopExit*) = [] |
successors-of-XorNode:
successors-of (*XorNode x y*) = [] |
successors-of-ZeroExtendNode:
successors-of (*ZeroExtendNode inputBits resultBits value*) = [] |
successors-of-NoNode: *successors-of* (*NoNode*) = [] |

successors-of-RefNode: *successors-of* (*RefNode ref*) = [*ref*]

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

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

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

end

3.2 IR Graph Node Hierarchy

theory *IRNodeHierarchy*
imports *IRNodes*
begin

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

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

These functions have been automatically generated from the compiler.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

is-FixedBinaryNode n = ((*is-IntegerDivRemNode n*))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

n))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

is-GuardingNode n = ((*is-AbstractBeginNode n*))

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

fun *is-same-ir-node-type* :: *IRNode* ⇒ *IRNode* ⇒ *bool* **where**
is-same-ir-node-type n1 n2 = (
((*is-AbsNode n1*) ∧ (*is-AbsNode n2*)) ∨
((*is-AddNode n1*) ∧ (*is-AddNode n2*)) ∨

((*is-AndNode n1*) ∧ (*is-AndNode n2*)) ∨
((*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-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-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-ShortCircuitOrNode n1*) ∧ (*is-ShortCircuitOrNode n2*)) ∨
((*is-SignedDivNode n1*) ∧ (*is-SignedDivNode n2*)) ∨
((*is-StartNode n1*) ∧ (*is-StartNode n2*)) ∨
((*is-StoreFieldNode n1*) ∧ (*is-StoreFieldNode n2*)) ∨
((*is-SubNode n1*) ∧ (*is-SubNode n2*)) ∨
((*is-UnwindNode n1*) ∧ (*is-UnwindNode n2*)) ∨
((*is-ValuePhiNode n1*) ∧ (*is-ValuePhiNode n2*)) ∨
((*is-ValueProxyNode n1*) ∧ (*is-ValueProxyNode n2*)) ∨
((*is-XorNode n1*) ∧ (*is-XorNode n2*)))

end

3.3 IR Graph Type

theory *IRGraph*
imports

```
  IRNodeHierarchy
  Stamp
  HOL−Library.FSet
  HOL.Relation
begin
```

This theory defines the main Graal data structure - an entire IR Graph.

IRGraph is defined as a partial map with a finite domain. The finite domain is required to be able to generate code and produce an interpreter.

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

setup-lifting type-definition-IRGraph

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

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

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

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

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

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

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

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

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

lift-definition irgraph :: (ID × (IRNode × Stamp)) list ⇒ IRGraph
```

is *map-of* ∘ *no-node*
by (*simp add*: *finite-dom-map-of*)

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

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

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

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

code-datatype *irgraph*

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

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

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

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

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

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

— Get the inputs set of a given node ID
fun *inputs* :: *IRGraph* ⇒ *ID* ⇒ *ID set* **where**

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

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

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

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

lemma *ids-some*[*simp*]: *x* ∈ *ids g* ⟷ *kind g x* ≠ *NoNode*
proof −
have *that*: *x* ∈ *ids g* ⟶ *kind g x* ≠ *NoNode*
using *ids.rep-eq kind.rep-eq* **by** *force*
have *kind g x* ≠ *NoNode* ⟶ *x* ∈ *ids g*
unfolding *with-default.simps kind-def ids-def*
by (*cases Rep-IRGraph g x* = *None*; *auto*)
from *this that* **show** *?thesis* **by** *auto*
qed

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

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

using *Abs-IRGraph-inverse* **by** (*metis Rep-IRGraph mem-Collect-eq*)

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

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

lemma [*simp*]: *finite* (*dom g*) $\longrightarrow$ *ids* (*Abs-IRGraph g*) = {*nid* $\in$ *dom g* . $\nexists$ *s. g nid* = *Some* (*NoNode*, *s*)}
  using *ids.rep-eq* **by** *simp*

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

lemma [*simp*]: *finite* (*dom g*) $\longrightarrow$ *stamp* (*Abs-IRGraph g*) = (λ*x* . (*case g x of None* $\Rightarrow$ *IllegalStamp* | *Some n* $\Rightarrow$ *snd n*))
  using *stamp.abs-eq stamp.rep-eq* **by** *auto*

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

lemma [*simp*]: *kind* (*irgraph g*) = (λ*nid.* (*case* (*map-of* (*no-node g*)) *nid of None* $\Rightarrow$ *NoNode* | *Some n* $\Rightarrow$ *fst n*))
  using *irgraph.rep-eq kind.transfer kind.rep-eq* **by** *auto*

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

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

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

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

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

lemma *remove-node-lookup*:
  gup = *remove-node nid g* ⟶ *kind gup nid* = *NoNode* ∧ *stamp gup nid* = *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*)

3.3.1 Example Graphs

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

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

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

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

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

]

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

end

4 java.lang.Long

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

theory *Long*
  imports *ValueThms*
begin

lemma *negative-all-set-32*:
  $n < 32 \Longrightarrow bit\ (-1::int32)\ n$
  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**
  $MinOrHighest\ s\ m = (if\ s = \{\}\ then\ m\ else\ Min\ s)$

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

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

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

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

4.1 Long.numberOfLeadingZeros

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

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

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

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

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

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

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

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

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

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

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

4.2 Long.numberOfTrailingZeros

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

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

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

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

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

4.3 Long.bitCount

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

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

4.4 Long.zeroCount

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

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

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

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

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

lemma *card-of-range*:
  $x = card\ \{n . 0 \leq n \wedge n < x\}$
  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 ∧ n ≤ x}
proof −
  have finite: finite {n . y ≤ n ∧ n < x}
    by auto
  have nonempty: {n . y ≤ n ∧ n < x} ≠ {}
    using assms by blast
  have simprep: {n . y < n ∧ n ≤ x} = {y<..x}
    by auto
  have x − y = card {y<..x}
    by auto
  then show ?thesis
    unfolding simprep by blast
qed

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

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

lemma zerosAboveHighestOne:
  n > highestOneBit a ⟹ ¬(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)) ⟹ ¬ bit a n
```

```
    using False by auto
  then show ?thesis
    by (metis False MinOrHighest-def assms lowestOneBit-def)
qed

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

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

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

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

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


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

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

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

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

hide-fact intersect-bitCount-helper

end
```

5 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* 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* 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 = (*λx. UndefVal*)

5.1 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*)

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

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*)
| *VariableExpr* (*ir-name*: *string*) (*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*

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

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

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

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

stamp-unary op (*IntegerStamp b lo hi*) =
unrestricted-stamp (*IntegerStamp* (*if op* ∈ *normal-unary then b 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*

5.3 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
```

fun *bin-eval* :: *IRBinaryOp* ⇒ *Value* ⇒ *Value* ⇒ *Value* **where**

```
  bin-eval BinAdd v1 v2 = intval-add v1 v2 |
  bin-eval BinMul v1 v2 = intval-mul v1 v2 |
  bin-eval BinSub v1 v2 = intval-sub 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*

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*;
branch = (*if val-to-bool cond then te else fe*);
[*m*,*p*] ⊢ *branch* ↦ *result*;
result ≠ *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* ([-,-] ⊢ - $\mapsto_L$ - *55*)
for *m p* **where**

EvalNil:
[*m*,*p*] ⊢ [] $\mapsto_L$ [] |

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

5.4 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**
  $(e1 \doteq e2) = (\forall\ m\ p\ v.\ (([m,p] \vdash e1 \mapsto v) \longleftrightarrow ([m,p] \vdash e2 \mapsto v)))$

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

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

instance proof
  fix $x\ y\ z$ :: *IRExpr*
  show $x < y \longleftrightarrow x \leq y \wedge \neg\ (y \leq x)$ **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 $e_1 \sqsupseteq e_2 \equiv (e_2 \leq e_1)$

5.5 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
  using up-spec
  using bit-and-iff bit-eq-iff bit-not-iff bit-unsigned-iff down-spec
  by (smt (verit, best) bit.double-compl)

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

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
  using assms
 by (smt (z3) and.commute and.right-neutral and-zero-eq bit.compl-zero bit.conj-cancel-right
bit.conj-disj-distribs(1) ucast-id up-spec word-bw-assocs(1) word-not-dist(2))

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*
using *assms*
by (*smt* (*z3*) *and-zero-eq bit.conj-cancel-left bit.conj-disj-distribs*(*1*) *bit.conj-disj-distribs*(*2*) *bit.de-Morgan-disj down-spec or-eq-not-not-and ucast-id up-spec word-ao-absorbs*(*2*) *word-ao-absorbs*(*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*
unfolding *IRExpr-up-def IRExpr-down-def*
apply *unfold-locales*
by (*simp add*: *ucast-minus-one*)+

end

5.6 Data-flow Tree Theorems

theory *IRTreeEvalThms*
imports
Graph.ValueThms
IRTreeEval
begin

5.6.1 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*:
$[m,p] \vdash e \mapsto_L v1 \Longrightarrow$
$[m,p] \vdash e \mapsto_L v2 \Longrightarrow$

```
  v1 = v2
  apply (induction arbitrary: v2 rule: evaltrees.induct)
   apply (elim EvalTreeE; auto)
  using evalDet by force
```

5.6.2 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-int:
  assumes def: unary-eval op x ≠ UndefVal
  shows is-IntVal (unary-eval op x)
  unfolding is-IntVal-def using def
  apply (cases unary-eval op x; auto)
  using unary-eval-not-obj-ref unary-eval-not-obj-str by simp+

lemma bin-eval-int:
  assumes def: bin-eval op x y ≠ UndefVal
  shows is-IntVal (bin-eval op x y)
  apply (cases op; cases x; cases y)
  unfolding is-IntVal-def using def apply auto
                apply presburger+
           apply (meson bool-to-val.elims)
          apply (meson bool-to-val.elims)
         apply (smt (verit) new-int.simps)+
  by (meson bool-to-val.elims)+

lemma IntVal0:
  (IntVal 32 0) = (new-int 32 0)
  unfolding new-int.simps
  by auto

lemma IntVal1:
  (IntVal 32 1) = (new-int 32 1)
  unfolding new-int.simps
```

by *auto*

lemma *bin-eval-new-int*:
  assumes *def*: *bin-eval op x y* $\neq$ *UndefVal*
  shows $\exists$ *b v*. (*bin-eval op x y*) = *new-int b v* $\wedge$
        b = (*if op* $\in$ *binary-fixed-32-ops then 32 else intval-bits x*)
  apply (*cases op*; *cases x*; *cases y*)
  unfolding *is-IntVal-def* **using** *def* **apply** *auto*
  apply *presburger+*
  apply (*metis take-bit-and*)
  apply *presburger*
  apply (*metis take-bit-or*)
  apply *presburger*
  apply (*metis take-bit-xor*)
  apply *presburger*
  using *IntVal0 IntVal1*
  apply (*metis bool-to-val.elims new-int.simps*)
  apply *presburger*
  apply (*smt* (*verit*) *new-int.elims*)
  apply (*smt* (*verit*, *best*) *new-int.elims*)
  apply (*metis IntVal0 IntVal1 bool-to-val.elims new-int.simps*)
  apply *presburger*
  apply (*metis IntVal0 IntVal1 bool-to-val.elims new-int.simps*)
  apply *presburger*
  apply (*metis IntVal0 IntVal1 bool-to-val.elims new-int.simps*)
  by *meson*

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

lemma *validStampIntConst*:
  assumes *v* = *IntVal b ival*
  assumes *0* < *b* $\wedge$ *b* $\leq$ *64*
  shows *valid-stamp* (*constantAsStamp v*)
proof –
  have *bnds*: *fst* (*bit-bounds b*) $\leq$ *int-signed-value b ival* $\wedge$ *int-signed-value b ival*
$\leq$ *snd* (*bit-bounds b*)
    using *assms int-signed-value-bounds*
    by *presburger*
  have *s*: *constantAsStamp v* = *IntegerStamp b* (*int-signed-value b ival*) (*int-signed-value*
b ival)
    using *assms*(*1*) *constantAsStamp.simps*(*1*) **by** *blast*
  then show *?thesis*
    unfolding *s valid-stamp.simps*

```
    using assms(2) assms 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 int-signed-value-bounds
    by presburger
  have s: constantAsStamp v = IntegerStamp b (int-signed-value b ival) (int-signed-value
b ival)
    using assms(1) constantAsStamp.simps(1) by blast
  then show ?thesis
    unfolding s unfolding v unfolding valid-value.simps
    using assms validStampIntConst
    by simp
qed
```

5.6.3 Evaluation Results are Valid

A valid value cannot be $UndefVal$.

```
lemma valid-not-undef:
  assumes a1: valid-value val s
  assumes a2: s ≠ VoidStamp
  shows val ≠ UndefVal
  apply (rule valid-value.elims(1)[of val s True])
  using a1 a2 by auto


lemma valid-VoidStamp[elim]:
  shows valid-value val VoidStamp ⟹
      val = UndefVal
  using valid-value.simps by metis

lemma valid-ObjStamp[elim]:
  shows valid-value val (ObjectStamp klass exact nonNull alwaysNull) ⟹
      (∃ v. val = ObjRef v)
  using valid-value.simps by (metis val-to-bool.cases)

lemma valid-int[elim]:
  shows valid-value val (IntegerStamp b lo hi) ⟹
      (∃ 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] ⊢ e ↦ v) ⟹ v ≠ UndefVal
  apply (induction rule: evaltree.induct)
  using valid-not-undef wf-value-def by auto
```

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

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

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

lemma valid-value-signed-int-range [simp]:
  assumes valid-value val (IntegerStamp b lo hi)
  assumes lo < 0
  shows ∃ v. (val = IntVal b v ∧
             lo ≤ int-signed-value b v ∧
             int-signed-value b v ≤ hi)
  using assms valid-int
  by (metis valid-value.simps(1))
```

5.6.4 Example Data-flow Optimisations

5.6.5 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 $e \geq e'$
shows (*UnaryExpr op e*) $\geq$ (*UnaryExpr op e'*)
using *UnaryExpr assms* **by** *auto*

lemma *mono-binary*:
assumes $x \geq x'$
assumes $y \geq y'$
shows (*BinaryExpr op x y*) $\geq$ (*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 *valid-value.simps*
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*; *simp del*: *valid-stamp.simps*)

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

lemma *mono-conditional*:
assumes $ce \geq ce'$
assumes $te \geq te'$
assumes $fe \geq fe'$
shows (*ConditionalExpr ce te fe*) $\geq$ (*ConditionalExpr ce' te' fe'*)
proof (*simp only*: *le-expr-def*; (*rule allI*)+; *rule impI*)
fix *m p v*
assume *a*: $[m,p] \vdash$ *ConditionalExpr ce te fe* $\mapsto v$
then obtain *cond* **where** *ce*: $[m,p] \vdash ce \mapsto cond$ **by** *auto*
then have *ce'*: $[m,p] \vdash ce' \mapsto cond$ **using** *assms* **by** *auto*

define *branch* **where** *b*: *branch* = (*if val-to-bool cond then te else fe*)
define *branch'* **where** *b'*: *branch'* = (*if val-to-bool cond then te' else fe'*)
then have *beval*: $[m,p] \vdash branch \mapsto v$ **using** *a b ce evalDet* **by** *blast*

from *beval* **have** $[m,p] \vdash branch' \mapsto v$ **using** *assms b b'* **by** *auto*
then show $[m,p] \vdash$ *ConditionalExpr ce' te' fe'* $\mapsto v$
using *ConditionalExpr ce' b'*

```
  using a by blast
qed
```

5.7 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:
  shows ([m,p] ⊢ ConstantExpr c ↦ v) = (wf-value v ∧ v = c)
  by blast
```

```
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] ⊢ xe ↦ x
        and [m,p] ⊢ ye ↦ 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] ⊢ UnaryExpr op xe ↦ val)
         = (∃ x.
             (([m,p] ⊢ xe ↦ x) ∧
              (val = unary-eval op x) ∧
              (val ≠ UndefVal)
             )) (is ?L = ?R)
  by auto
```

```
lemmas unfold-evaltree =
```

unfold-binary
unfold-unary

5.8 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 ir-resultBits op)
proof (cases op ∈ normal-unary)
  case True
  then show ?thesis
   by (metis def empty-iff insert-iff intval-abs.elims intval-bits.simps intval-logic-negation.elims
intval-negate.elims intval-not.elims unary-eval.simps(1) unary-eval.simps(2) unary-eval.simps(3)
unary-eval.simps(4))
next
  case False
  consider ib ob where op = UnaryNarrow ib ob |
         ib ob where op = UnaryZeroExtend ib ob |
         ib ob where op = UnarySignExtend ib ob
    by (metis False IRUnaryOp.exhaust insert-iff)
  then show ?thesis
  proof (cases)
    case 1
    then show ?thesis
     by (metis False IRUnaryOp.sel(4) def intval-narrow.elims unary-eval.simps(5))
  next
    case 2
    then show ?thesis
    by (metis False IRUnaryOp.sel(6) def intval-zero-extend.elims unary-eval.simps(7))
  next
    case 3
    then show ?thesis
    by (metis False IRUnaryOp.sel(5) def intval-sign-extend.elims unary-eval.simps(6))
  qed
qed

lemma new-int-unused-bits-zero:
  assumes IntVal b ival = new-int b ival0
  shows take-bit b ival = ival
  using assms(1) new-int-take-bits by blast

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

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

lemma eval-unused-bits-zero:
  [m,p] ⊢ xe ↦ (IntVal b ix) ⟹ take-bit b ix = ix
proof (induction xe)
  case (UnaryExpr x1 xe)
  then show ?case
    using unary-eval-unused-bits-zero by force
next
  case (BinaryExpr x1 xe1 xe2)
  then show ?case
    using bin-eval-unused-bits-zero by force
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 ParameterExprE Value.distinct(7) intval-bits.simps intval-word.simps
local.ParameterExpr valid-value.elims(2))
next
  case (LeafExpr x1 x2)
  then show ?case
   by (smt (z3) EvalTreeE(6) Value.simps(11) valid-value.elims(1) valid-value.simps(1))

next
  case (ConstantExpr x)
  then show ?case using wf-value-def
    by (metis EvalTreeE(1) constantAsStamp.simps(1) valid-value.simps(1))
next
  case (ConstantVar x)
  then show ?case
    by fastforce
next
  case (VariableExpr x1 x2)
  then show ?case
    by fastforce
qed


lemma unary-normal-bitsize:
  assumes unary-eval op x = IntVal b ival
  assumes op ∈ normal-unary
  shows ∃ ix. x = IntVal b ix
  apply (cases op)
        prefer 7 using assms apply blast
       prefer 6 using assms apply blast
      prefer 5 using assms apply blast
  using Value.distinct(1) Value.sel(1) assms(1) new-int.simps unary-eval.simps
      intval-abs.elims intval-negate.elims intval-not.elims intval-logic-negation.elims
     apply metis+
  done

lemma unary-not-normal-bitsize:
  assumes unary-eval op x = IntVal b ival
  assumes op ∉ normal-unary
  shows b = ir-resultBits op ∧ 0 < b ∧ b ≤ 64
  apply (cases op)
  using assms apply blast+
 apply (metis IRUnaryOp.sel(4) Value.distinct(1) Value.sel(1) assms(1) intval-narrow.elims
intval-narrow-ok new-int.simps unary-eval.simps(5))
   apply (smt (verit) IRUnaryOp.sel(5) Value.distinct(1) Value.sel(1) assms(1)
intval-sign-extend.elims new-int.simps order-less-le-trans unary-eval.simps(6))
  apply (metis IRUnaryOp.sel(6) Value.distinct(1) assms(1) intval-bits.simps int-
val-zero-extend.elims linorder-not-less neq0-conv new-int.simps unary-eval.simps(7))
  done


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
proof (cases op ∈ normal-unary)
  case True
  then obtain tmp where unary-eval op x = new-int bx tmp
    by (cases op; simp; auto simp: 2)
  then show ?thesis
    using assms by simp
next
  case False
  then obtain tmp where unary-eval op x = new-int b tmp ∧ 0 < b ∧ b ≤ 64
    apply (cases op; simp; auto simp: 2)
    apply (metis 2 Value.inject(1) Value.simps(5) assms(1) intval-narrow.simps(1)
intval-narrow-ok new-int.simps unary-eval.simps(5))
   apply (metis 2 Value.distinct(1) Value.inject(1) assms(1) bot-nat-0.not-eq-extremum
diff-is-0-eq intval-sign-extend.elims new-int.simps unary-eval.simps(6) zero-less-diff)
      by (smt (verit, del-insts) 2 Value.simps(5) assms(1) intval-bits.simps int-
val-zero-extend.simps(1) new-int.simps order-less-le-trans unary-eval.simps(7))
  then show ?thesis
    by blast
qed
```

```
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 apply (cases op; cases x; cases y; simp)
    using that by blast+
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
    using unfold-binary by auto
  then have b = (if op ∈ normal-unary then intval-bits xv else ir-resultBits op)
    using unary-eval-new-int
    by (metis Value.disc(1) Value.discI(1) Value.sel(1) new-int.simps)
  then show ?case
    by (metis xv UnaryExpr.IH unary-normal-bitsize unary-not-normal-bitsize)
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
    using unfold-binary by auto
  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 (metis BinaryExpr.IH(1) Value.distinct(7) Value.distinct(9) xv bin-eval-inputs-are-ints
intval-bits.elims le-add-same-cancel1 less-or-eq-imp-le numeral-Bit0 xy 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
   using ParameterExprE intval-bits.simps valid-stamp.simps(1) valid-value.elims(2)
valid-value.simps(17)
    by (metis (no-types, lifting))
next
  case (LeafExpr x1 x2)
  then show ?case
   by (smt (z3) EvalTreeE(6) Value.distinct(7) Value.inject(1) valid-stamp.simps(1)
valid-value.elims(1))
next
  case (ConstantExpr x)
  then show ?case using wf-value-def
   by (metis EvalTreeE(1) constantAsStamp.simps(1) valid-stamp.simps(1) valid-value.simps(1))
next
  case (ConstantVar x)
  then show ?case
    by blast
next
  case (VariableExpr x1 x2)
  then show ?case
    by blast
qed


lemma unfold-binary-width:
  assumes op ∉ binary-fixed-32-ops ∧ op ∉ binary-shift-ops
  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 force+ apply auto[1]
    using assms apply (cases op; auto)
          apply (smt (verit) intval-add.elims Value.inject(1))
    using intval-mul.elims Value.inject(1)
         apply (smt (verit) new-int.simps new-int-bin.simps)
    using intval-sub.elims Value.inject(1)
        apply (smt (verit) new-int.simps new-int-bin.simps)
    using intval-and.elims Value.inject(1)
       apply (smt (verit) new-int.simps new-int-bin.simps take-bit-and)
    using intval-or.elims Value.inject(1)
      apply (smt (verit) new-int.simps new-int-bin.simps take-bit-or)
    using intval-xor.elims Value.inject(1)
     apply (smt (verit) new-int.simps new-int-bin.simps take-bit-xor)
  by blast
```

```
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
```

6 Tree to Graph

```
theory TreeToGraph
  imports
    Semantics.IRTreeEval
    Graph.IRGraph
begin
```

6.1 Subgraph to Data-flow Tree

```
fun find-node-and-stamp :: IRGraph ⇒ (IRNode × Stamp) ⇒ ID option where
  find-node-and-stamp g (n,s) =
     find (λi. 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 - = False


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

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

$ParameterNode$:
$[\![kind\ g\ n = ParameterNode\ i;$
$\ \ stamp\ g\ n = s]\!]$
$\ \ \Longrightarrow g\ \vdash n \simeq (ParameterExpr\ i\ s)\ |$

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

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

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

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

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

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

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

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

$g \vdash x \simeq xe;$
$g \vdash y \simeq ye]\!]$
$\Longrightarrow g \vdash n \simeq (BinaryExpr\ BinSub\ xe\ ye)\ |$

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

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

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

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

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

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

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

$IntegerBelowNode$:
$[\![kind\ g\ n = IntegerBelowNode\ x\ y;$
$g \vdash x \simeq xe;$

$g \vdash y \simeq ye\rrbracket$
$\Longrightarrow g \vdash n \simeq (\mathit{BinaryExpr\ BinIntegerBelow\ xe\ ye})$ |

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

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

NarrowNode:
$\llbracket \mathit{kind}\ g\ n = \mathit{NarrowNode\ inputBits\ resultBits}\ x;$
$g \vdash x \simeq xe\rrbracket$
$\Longrightarrow g \vdash n \simeq (\mathit{UnaryExpr}\ (\mathit{UnaryNarrow\ inputBits\ resultBits})\ xe)$ |

SignExtendNode:
$\llbracket \mathit{kind}\ g\ n = \mathit{SignExtendNode\ inputBits\ resultBits}\ x;$
$g \vdash x \simeq xe\rrbracket$
$\Longrightarrow g \vdash n \simeq (\mathit{UnaryExpr}\ (\mathit{UnarySignExtend\ inputBits\ resultBits})\ xe)$ |

ZeroExtendNode:
$\llbracket \mathit{kind}\ g\ n = \mathit{ZeroExtendNode\ inputBits\ resultBits}\ x;$
$g \vdash x \simeq xe\rrbracket$
$\Longrightarrow g \vdash n \simeq (\mathit{UnaryExpr}\ (\mathit{UnaryZeroExtend\ inputBits\ resultBits})\ xe)$ |

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

RefNode:
$\llbracket \mathit{kind}\ g\ n = \mathit{RefNode}\ n';$
$g \vdash n' \simeq e\rrbracket$
$\Longrightarrow g \vdash n \simeq e$

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

inductive
replist :: *IRGraph* $\Rightarrow$ *ID list* $\Rightarrow$ *IRExpr list* $\Rightarrow$ *bool* (- $\vdash$ - $\simeq_L$ - *55*)
for *g* **where**

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

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

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

definition *wf-term-graph* :: *MapState* $\Rightarrow$ *Params* $\Rightarrow$ *IRGraph* $\Rightarrow$ *ID* $\Rightarrow$ *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 $\{t.\ eg2\text{-}sq \vdash 4 \simeq t\}$

6.2 Data-flow Tree to Subgraph

fun *unary-node* :: *IRUnaryOp* $\Rightarrow$ *ID* $\Rightarrow$ *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*

fun *bin-node* :: *IRBinaryOp* $\Rightarrow$ *ID* $\Rightarrow$ *ID* $\Rightarrow$ *IRNode* **where**
bin-node BinAdd x y = AddNode x y |
bin-node BinMul x y = MulNode 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

inductive *fresh-id* :: *IRGraph* $\Rightarrow$ *ID* $\Rightarrow$ *bool* **where**
$n \notin$ *ids g* $\Longrightarrow$ *fresh-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
unrep :: *IRGraph* ⇒ *IRExpr* ⇒ (*IRGraph* × *ID*) ⇒ *bool* (- ⊕ - ⇝ - *55*)
where

ConstantNodeSame:
⟦*find-node-and-stamp g* (*ConstantNode c*, *constantAsStamp c*) = *Some n*⟧
⟹ *g* ⊕ (*ConstantExpr c*) ⇝ (*g*, *n*) |

ConstantNodeNew:
⟦*find-node-and-stamp g* (*ConstantNode c*, *constantAsStamp c*) = *None*;
n = *get-fresh-id g*;
g′ = *add-node n* (*ConstantNode c*, *constantAsStamp c*) *g* ⟧
⟹ *g* ⊕ (*ConstantExpr c*) ⇝ (*g*′, *n*) |

ParameterNodeSame:
⟦*find-node-and-stamp g* (*ParameterNode i*, *s*) = *Some n*⟧
⟹ *g* ⊕ (*ParameterExpr i s*) ⇝ (*g*, *n*) |

ParameterNodeNew:
⟦*find-node-and-stamp g* (*ParameterNode i*, *s*) = *None*;
n = *get-fresh-id g*;
g′ = *add-node n* (*ParameterNode i*, *s*) *g*⟧
⟹ *g* ⊕ (*ParameterExpr i s*) ⇝ (*g*′, *n*) |

ConditionalNodeSame:
⟦*g* ⊕ *ce* ⇝ (*g2*, *c*);
g2 ⊕ *te* ⇝ (*g3*, *t*);
g3 ⊕ *fe* ⇝ (*g4*, *f*);
s′ = *meet* (*stamp g4 t*) (*stamp g4 f*);
find-node-and-stamp g4 (*ConditionalNode c t f*, *s*′) = *Some n*⟧
⟹ *g* ⊕ (*ConditionalExpr ce te fe*) ⇝ (*g4*, *n*) |

ConditionalNodeNew:
⟦*g* ⊕ *ce* ⇝ (*g2*, *c*);
g2 ⊕ *te* ⇝ (*g3*, *t*);

$g3 \oplus fe \leadsto (g4, f)$;
$s' = meet\ (stamp\ g4\ t)\ (stamp\ g4\ f)$;
$find\text{-}node\text{-}and\text{-}stamp\ g4\ (ConditionalNode\ c\ t\ f,\ s') = None$;
$n = get\text{-}fresh\text{-}id\ g4$;
$g' = add\text{-}node\ n\ (ConditionalNode\ c\ t\ f,\ s')\ g4]\!]$
$\Longrightarrow g \oplus (ConditionalExpr\ ce\ te\ fe) \leadsto (g', n)$ |

UnaryNodeSame:
$[\![g \oplus xe \leadsto (g2, x)$;
$s' = stamp\text{-}unary\ op\ (stamp\ g2\ x)$;
$find\text{-}node\text{-}and\text{-}stamp\ g2\ (unary\text{-}node\ op\ x,\ s') = Some\ n]\!]$
$\Longrightarrow g \oplus (UnaryExpr\ op\ xe) \leadsto (g2, n)$ |

UnaryNodeNew:
$[\![g \oplus xe \leadsto (g2, x)$;
$s' = stamp\text{-}unary\ op\ (stamp\ g2\ x)$;
$find\text{-}node\text{-}and\text{-}stamp\ g2\ (unary\text{-}node\ op\ x,\ s') = None$;
$n = get\text{-}fresh\text{-}id\ g2$;
$g' = add\text{-}node\ n\ (unary\text{-}node\ op\ x,\ s')\ g2]\!]$
$\Longrightarrow g \oplus (UnaryExpr\ op\ xe) \leadsto (g', n)$ |

BinaryNodeSame:
$[\![g \oplus xe \leadsto (g2, x)$;
$g2 \oplus ye \leadsto (g3, y)$;
$s' = stamp\text{-}binary\ op\ (stamp\ g3\ x)\ (stamp\ g3\ y)$;
$find\text{-}node\text{-}and\text{-}stamp\ g3\ (bin\text{-}node\ op\ x\ y,\ s') = Some\ n]\!]$
$\Longrightarrow g \oplus (BinaryExpr\ op\ xe\ ye) \leadsto (g3, n)$ |

BinaryNodeNew:
$[\![g \oplus xe \leadsto (g2, x)$;
$g2 \oplus ye \leadsto (g3, y)$;
$s' = stamp\text{-}binary\ op\ (stamp\ g3\ x)\ (stamp\ g3\ y)$;
$find\text{-}node\text{-}and\text{-}stamp\ g3\ (bin\text{-}node\ op\ x\ y,\ s') = None$;
$n = get\text{-}fresh\text{-}id\ g3$;
$g' = add\text{-}node\ n\ (bin\text{-}node\ op\ x\ y,\ s')\ g3]\!]$
$\Longrightarrow g \oplus (BinaryExpr\ op\ xe\ ye) \leadsto (g', n)$ |

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 **.**

unrepRules

$$\frac{\textit{find-node-and-stamp}\ g\ (\textit{ConstantNode}\ c,\ \textit{constantAsStamp}\ c) = \textit{Some}\ n}{g \oplus \textit{ConstantExpr}\ c \rightsquigarrow (g,\ n)}$$

$$\frac{\begin{array}{c}\textit{find-node-and-stamp}\ g\ (\textit{ConstantNode}\ c,\ \textit{constantAsStamp}\ c) = \textit{None} \\ n = \textit{get-fresh-id}\ g \\ g' = \textit{add-node}\ n\ (\textit{ConstantNode}\ c,\ \textit{constantAsStamp}\ c)\ g\end{array}}{g \oplus \textit{ConstantExpr}\ c \rightsquigarrow (g',\ n)}$$

$$\frac{\textit{find-node-and-stamp}\ g\ (\textit{ParameterNode}\ i,\ s) = \textit{Some}\ n}{g \oplus \textit{ParameterExpr}\ i\ s \rightsquigarrow (g,\ n)}$$

$$\frac{\begin{array}{c}\textit{find-node-and-stamp}\ g\ (\textit{ParameterNode}\ i,\ s) = \textit{None} \\ n = \textit{get-fresh-id}\ g \qquad g' = \textit{add-node}\ n\ (\textit{ParameterNode}\ i,\ s)\ g\end{array}}{g \oplus \textit{ParameterExpr}\ i\ s \rightsquigarrow (g',\ n)}$$

$$\frac{\begin{array}{c}g \oplus \textit{ce} \rightsquigarrow (\textit{g2},\ c) \qquad \textit{g2} \oplus \textit{te} \rightsquigarrow (\textit{g3},\ t) \\ \textit{g3} \oplus \textit{fe} \rightsquigarrow (\textit{g4},\ f) \qquad s' = \textit{meet}\ (\textit{stamp}\ \textit{g4}\ t)\ (\textit{stamp}\ \textit{g4}\ f) \\ \textit{find-node-and-stamp}\ \textit{g4}\ (\textit{ConditionalNode}\ c\ t\ f,\ s') = \textit{Some}\ n\end{array}}{g \oplus \textit{ConditionalExpr}\ \textit{ce}\ \textit{te}\ \textit{fe} \rightsquigarrow (\textit{g4},\ n)}$$

$$\frac{\begin{array}{c}g \oplus \textit{ce} \rightsquigarrow (\textit{g2},\ c) \qquad \textit{g2} \oplus \textit{te} \rightsquigarrow (\textit{g3},\ t) \\ \textit{g3} \oplus \textit{fe} \rightsquigarrow (\textit{g4},\ f) \qquad s' = \textit{meet}\ (\textit{stamp}\ \textit{g4}\ t)\ (\textit{stamp}\ \textit{g4}\ f) \\ \textit{find-node-and-stamp}\ \textit{g4}\ (\textit{ConditionalNode}\ c\ t\ f,\ s') = \textit{None} \\ n = \textit{get-fresh-id}\ \textit{g4} \qquad g' = \textit{add-node}\ n\ (\textit{ConditionalNode}\ c\ t\ f,\ s')\ \textit{g4}\end{array}}{g \oplus \textit{ConditionalExpr}\ \textit{ce}\ \textit{te}\ \textit{fe} \rightsquigarrow (g',\ n)}$$

$$\frac{\begin{array}{c}g \oplus \textit{xe} \rightsquigarrow (\textit{g2},\ x) \\ \textit{g2} \oplus \textit{ye} \rightsquigarrow (\textit{g3},\ y) \qquad s' = \textit{stamp-binary}\ \textit{op}\ (\textit{stamp}\ \textit{g3}\ x)\ (\textit{stamp}\ \textit{g3}\ y) \\ \textit{find-node-and-stamp}\ \textit{g3}\ (\textit{bin-node}\ \textit{op}\ x\ y,\ s') = \textit{Some}\ n\end{array}}{g \oplus \textit{BinaryExpr}\ \textit{op}\ \textit{xe}\ \textit{ye} \rightsquigarrow (\textit{g3},\ n)}$$

$$\frac{\begin{array}{c}g \oplus \textit{xe} \rightsquigarrow (\textit{g2},\ x) \\ \textit{g2} \oplus \textit{ye} \rightsquigarrow (\textit{g3},\ y) \qquad s' = \textit{stamp-binary}\ \textit{op}\ (\textit{stamp}\ \textit{g3}\ x)\ (\textit{stamp}\ \textit{g3}\ y) \\ \textit{find-node-and-stamp}\ \textit{g3}\ (\textit{bin-node}\ \textit{op}\ x\ y,\ s') = \textit{None} \\ n = \textit{get-fresh-id}\ \textit{g3} \qquad g' = \textit{add-node}\ n\ (\textit{bin-node}\ \textit{op}\ x\ y,\ s')\ \textit{g3}\end{array}}{g \oplus \textit{BinaryExpr}\ \textit{op}\ \textit{xe}\ \textit{ye} \rightsquigarrow (g',\ n)}$$

$$\frac{\begin{array}{c}g \oplus \textit{xe} \rightsquigarrow (\textit{g2},\ x) \qquad s' = \textit{stamp-unary}\ \textit{op}\ (\textit{stamp}\ \textit{g2}\ x) \\ \textit{find-node-and-stamp}\ \textit{g2}\ (\textit{unary-node}\ \textit{op}\ x,\ s') = \textit{Some}\ n\end{array}}{g \oplus \textit{UnaryExpr}\ \textit{op}\ \textit{xe} \rightsquigarrow (\textit{g2},\ n)}$$

$$\frac{\begin{array}{c}g \oplus \textit{xe} \rightsquigarrow (\textit{g2},\ x) \qquad s' = \textit{stamp-unary}\ \textit{op}\ (\textit{stamp}\ \textit{g2}\ x) \\ \textit{find-node-and-stamp}\ \textit{g2}\ (\textit{unary-node}\ \textit{op}\ x,\ s') = \textit{None} \\ n = \textit{get-fresh-id}\ \textit{g2} \qquad g' = \textit{add-node}\ n\ (\textit{unary-node}\ \textit{op}\ x,\ s')\ \textit{g2}\end{array}}{g \oplus \textit{UnaryExpr}\ \textit{op}\ \textit{xe} \rightsquigarrow (g',\ n)}$$

$$\frac{\textit{stamp}\ g\ n = s \qquad \textit{is-preevaluated}\ (\textit{kind}\ g\ n)}{g \oplus \textit{LeafExpr}\ n\ s \rightsquigarrow (g,\ n)}$$

values $\{(n,\ g)\ .\ (eg2\text{-}sq \oplus sq\text{-}param0 \leadsto (g,\ n))\}$

6.3 Lift Data-flow Tree Semantics

definition *encodeeval* :: *IRGraph* ⇒ *MapState* ⇒ *Params* ⇒ *ID* ⇒ *Value* ⇒ *bool*
  $([-,-,-] \vdash - \mapsto -\ 50)$
  where
  encodeeval g m p n v $= (\exists\ e.\ (g \vdash n \simeq e) \land ([m,p] \vdash e \mapsto v))$

6.4 Graph Refinement

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

definition *graph-refinement* :: *IRGraph* ⇒ *IRGraph* ⇒ *bool* **where**
  graph-refinement $g_1\ g_2 =$
    $((ids\ g_1 \subseteq ids\ g_2) \land$
    $(\forall\ n\ .\ n \in ids\ g_1 \longrightarrow (\forall\, e.\ (g_1 \vdash n \simeq e) \longrightarrow (g_2 \vdash n \trianglelefteq e))))$

lemma *graph-refinement*:
  graph-refinement g1 g2 $\Longrightarrow (\forall\, n\ m\ p\ v.\ n \in ids\ g1 \longrightarrow ([g1,\ m,\ p] \vdash n \mapsto v) \longrightarrow ([g2,\ m,\ p] \vdash n \mapsto v))$
  by (*meson encodeeval-def graph-refinement-def graph-represents-expression-def le-expr-def*)

6.5 Maximal Sharing

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

end

6.6 Formedness Properties

theory *Form*
imports
  Semantics.TreeToGraph
begin

definition *wf-start* **where**
  wf-start g $= (0 \in ids\ g \land$
    is-StartNode $(kind\ g\ 0))$

definition *wf-closed* **where**
  wf-closed g $=$
    $(\forall\ n \in ids\ g\ .$

```
    inputs g n ⊆ ids g ∧
    succ g n ⊆ ids g ∧
    kind g n ≠ NoNode)

definition wf-phis where
  wf-phis g =
   (∀ n ∈ ids g.
     is-PhiNode (kind g n) ⟶
     length (ir-values (kind g n))
      = length (ir-ends
         (kind g (ir-merge (kind g n)))))

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

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

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

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

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

lemma wf-empty: wf-graph start-end-graph
  unfolding start-end-graph-def wf-folds by simp

lemma wf-eg2-sq: wf-graph eg2-sq
  unfolding eg2-sq-def wf-folds by simp

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

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

end

6.7 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 auto

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
  using changeonly.simps by blast
```

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 eval-usages.simps eval-uses.intros(1)
  by (simp add: ids.rep-eq)

lemma not-in-g-inputs:
  assumes nid ∉ ids g
  shows inputs g nid = {}
proof –
  have k: kind g nid = NoNode using assms not-in-g by blast
  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
  unfolding inputs.simps using assms
  by (metis in-set-member)

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



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

lemma inp-in-g-wf:
  assumes wf-graph g
  assumes n ∈ inputs g nid
  shows n ∈ ids g
  using assms unfolding wf-folds
  using 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
    using unchanged.simps by blast
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(1) assms(2) 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(1) assms(2) 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 eval-usages.simps
  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(1) assms(2) eval-usages inputs-are-uses by blast

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(1) assms(2) 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 eval-usages
  using assms(1) assms(2) assms(3) 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 auto

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

lemma eval-in-ids:
  assumes [g, m, p] ⊢ nid ↦ v
  shows nid ∈ ids g
  using assms using encodeeval-def encode-in-ids
  by auto

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

```
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
    using dom nc kind-unchanged
    by metis
  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) ParameterNode.prems(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
    using inputs-of-ConditionalNode
      by (metis ConditionalNode.hyps(1) ConditionalNode.hyps(2) ConditionalN-
ode.hyps(3) ConditionalNode.hyps(4) encode-in-ids inputs.simps inputs-are-usages
list.set-intros(1) set-subset-Cons subset-code(1))
  then show ?case using transitive-kind-same
   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) local.ConditionalNode(6) local.ConditionalNode(7) local.ConditionalNode(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
    using kind-unchanged
    by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-AbsNode
     by (metis AbsNode.hyps(1) AbsNode.hyps(2) encode-in-ids inputs.simps in-
puts-are-usages list.set-intros(1))
  then show ?case
    by (metis AbsNode.IH AbsNode.hyps(1) AbsNode.prems(1) AbsNode.prems(3)
IRNodes.inputs-of-AbsNode ‹kind g2 n = AbsNode x› child-member-in child-unchanged
local.wf member-rec(1) rep.AbsNode unchanged.simps)
next
  case (NotNode n x xe)
  then have kind g2 n = NotNode x
    using kind-unchanged
    by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-NotNode
      by (metis NotNode.hyps(1) NotNode.hyps(2) encode-in-ids inputs.simps in-
puts-are-usages list.set-intros(1))
  then show ?case
     by (metis NotNode.IH NotNode.hyps(1) NotNode.prems(1) NotNode.prems(3)
IRNodes.inputs-of-NotNode ‹kind g2 n = NotNode x› child-member-in child-unchanged
local.wf member-rec(1) rep.NotNode unchanged.simps)
next
  case (NegateNode n x xe)
  then have kind g2 n = NegateNode x
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-NegateNode
     by (metis NegateNode.hyps(1) NegateNode.hyps(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) NegateNode.prems(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
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-LogicNegationNode inputs-of-are-usages
    by (metis LogicNegationNode.hyps(1) LogicNegationNode.hyps(2) encode-in-ids
member-rec(1))
  then show ?case
     by (metis IRNodes.inputs-of-LogicNegationNode LogicNegationNode.IH Logic-
NegationNode.hyps(1) LogicNegationNode.hyps(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
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n ∧ y ∈ eval-usages g1 n
    using inputs-of-LogicNegationNode inputs-of-are-usages
   by (metis AddNode.hyps(1) AddNode.hyps(2) AddNode.hyps(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*) *AddNode.IH*(*2*) *AddNode.hyps*(*1*) *AddNode.hyps*(*2*) *AddNode.hyps*(*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*
using *kind-unchanged* **by** *metis*
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
using *inputs-of-LogicNegationNode inputs-of-are-usages*
by (*metis MulNode.hyps*(*1*) *MulNode.hyps*(*2*) *MulNode.hyps*(*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* **using** *MulNode inputs-of-MulNode*
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*))
next
case (*SubNode n x y xe ye*)
then have *kind g2 n* = *SubNode x y*
using *kind-unchanged* **by** *metis*
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
using *inputs-of-LogicNegationNode inputs-of-are-usages*
by (*metis SubNode.hyps*(*1*) *SubNode.hyps*(*2*) *SubNode.hyps*(*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* **using** *SubNode inputs-of-SubNode*
by (*metis* ‹*kind g2 n* = *SubNode x y*› *child-member child-unchanged encode-in-ids ids-some member-rec*(*1*) *rep.SubNode*)
next
case (*AndNode n x y xe ye*)
then have *kind g2 n* = *AndNode x y*
using *kind-unchanged* **by** *metis*
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
using *inputs-of-LogicNegationNode inputs-of-are-usages*
by (*metis AndNode.hyps*(*1*) *AndNode.hyps*(*2*) *AndNode.hyps*(*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* **using** *AndNode inputs-of-AndNode*
by (*metis* ‹*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*
using *kind-unchanged* **by** *metis*
then have *x* ∈ *eval-usages g1 n* ∧ *y* ∈ *eval-usages g1 n*
using *inputs-of-OrNode inputs-of-are-usages*
by (*metis OrNode.hyps*(*1*) *OrNode.hyps*(*2*) *OrNode.hyps*(*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* **using** *OrNode inputs-of-OrNode*
by (*metis* ‹*kind g2 n* = *OrNode x y*› *child-member child-unchanged encode-in-ids ids-some member-rec*(*1*) *rep.OrNode*)
next

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

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

```
  then have x ∈ eval-usages g1 n
    using inputs-of-NarrowNode inputs-of-are-usages
   by (metis NarrowNode.hyps(1) NarrowNode.hyps(2) IRNodes.inputs-of-NarrowNode
encode-in-ids inputs.simps inputs-are-usages list.set-intros(1))
  then show ?case using NarrowNode inputs-of-NarrowNode
      by (metis ‹kind g2 n = NarrowNode ib rb x› child-unchanged inputs.elims
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
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-SignExtendNode inputs-of-are-usages
     by (metis SignExtendNode.hyps(1) SignExtendNode.hyps(2) encode-in-ids in-
puts.simps inputs-are-usages list.set-intros(1))
  then show ?case using SignExtendNode inputs-of-SignExtendNode
   by (metis ‹kind g2 n = SignExtendNode ib rb x› child-member-in child-unchanged
in-set-member list.set-intros(1) rep.SignExtendNode unchanged.elims(2))
next
  case (ZeroExtendNode n ib rb x xe)
  then have kind g2 n = ZeroExtendNode ib rb x
    using kind-unchanged by metis
  then have x ∈ eval-usages g1 n
    using inputs-of-ZeroExtendNode inputs-of-are-usages
   by (metis ZeroExtendNode.hyps(1) ZeroExtendNode.hyps(2) IRNodes.inputs-of-ZeroExtendNode
encode-in-ids inputs.simps inputs-are-usages list.set-intros(1))
  then show ?case using ZeroExtendNode inputs-of-ZeroExtendNode
   by (metis ‹kind g2 n = ZeroExtendNode ib rb x› child-member-in child-unchanged
member-rec(1) rep.ZeroExtendNode unchanged.simps)
next
  case (LeafNode n s)
  then show ?case
    by (metis kind-unchanged rep.LeafNode stamp-unchanged)
next
  case (RefNode n n')
  then have kind g2 n = RefNode n'
    using kind-unchanged by metis
  then have n' ∈ eval-usages g1 n
      by (metis IRNodes.inputs-of-RefNode RefNode.hyps(1) RefNode.hyps(2) en-
code-in-ids inputs.elims inputs-are-usages list.set-intros(1))
  then show ?case
   by (metis IRNodes.inputs-of-RefNode RefNode.IH RefNode.hyps(1) RefNode.hyps(2)
RefNode.prems(1) ‹kind g2 n = RefNode n'› child-unchanged encode-in-ids in-
puts.elims list.set-intros(1) local.wf rep.RefNode)
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 blast
  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 encodeeval-def g1
    by auto
  then have val: [m,p] ⊢ e ↦ v1
    using g1 encodeeval-def
    by simp
  then show ?thesis using e nid nc
    unfolding encodeeval-def
  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
      using stay-same-encoding ParameterExpr
      by (meson local.wf)
    then show ?case using evaltree.ParameterExpr
      by (meson ParameterExpr.hyps)
  next
    case (ConditionalExpr ce cond branch te fe v)
    then have g2 ⊢ nid ≃ ConditionalExpr ce te fe
    using ConditionalExpr.prems(1) ConditionalExpr.prems(3) local.wf stay-same-encoding
      by presburger
    then show ?case
        by (meson ConditionalExpr.prems(1) ConditionalExpr.prems(3) local.wf
stay-same-encoding)
  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
  using assms unfolding add-node-def changeonly.simps
  using add-node.rep-eq add-node-def kind.rep-eq stamp.rep-eq by simp

lemma disjoint-change:
  assumes changeonly change g gup
  assumes nochange = ids g − change
  shows unchanged nochange g gup
  using assms unfolding changeonly.simps unchanged.simps
  by blast

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
    using eval-usages.simps by blast
  then have changeonly {new} g gup
    using assms add-changed by blast
  then show ?thesis using assms add-node-def disjoint-change
    using Diff-insert-absorb 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'
  using use0 use-inp use-trans
  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)
proof (induction rule: eval-uses.induct)
```

```
  case use0
  then show ?case by simp
next
  case use-inp
  then show ?case
    using assms(1) inp-in-g-wf by blast
next
  case use-trans
  then show ?case by blast
qed

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 blast
  have inp: nid' ∉ inputs g nid
    using assms
    using inp-in-g-wf by blast
  have rec-0: ∄n . n ∈ ids g ∧ n = nid'
    using assms by blast
  have rec-inp: ∄n . n ∈ ids g ∧ n ∈ inputs g nid'
    using assms(2) inp-in-g by blast
  have rec: ∄nid'' . eval-uses g nid nid'' ∧ eval-uses g nid'' nid'
    using wf-use-ids assms(1) assms(2) assms(3) by blast
  from inp 0 rec show ?thesis
    using eval-uses-imp by blast
qed

end
```

6.8 Tree to Graph Theorems

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

6.8.1 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-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-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-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-ref [rep]:
  g ⊢ n ≃ e ⟹
   kind g n = RefNode n' ⟹
   g ⊢ n' ≃ e
  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 - ⇒
            ‹metis i e r d›››› |
   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 - ⇒
            ‹metis i e r d›››› |
   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 - ⇒
            ‹metis i e r d›››› |
  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 - ⇒
            ‹metis i e r d›››››)
```

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

```
lemma repDet:
  shows (g ⊢ n ≃ e₁) ⟹ (g ⊢ n ≃ e₂) ⟹ e₁ = e₂
proof (induction arbitrary: e₂ rule: rep.induct)
  case (ConstantNode n c)
  then show ?case using rep-constant by auto
next
  case (ParameterNode n i s)
  then show ?case
    by (metis IRNode.disc(2685) ParameterNodeE is-RefNode-def rep-parameter)
next
  case (ConditionalNode n c t f ce te fe)
  then show ?case
    using IRNode.distinct(593)
    using IRNode.inject(6) ConditionalNodeE rep-conditional
    by metis
next
  case (AbsNode n x xe)
  then show ?case
    by (solve-det node: AbsNode)
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 (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 (NarrowNode n x xe)
  then show ?case
    by (metis IRNode.distinct(2203) IRNode.inject(28) NarrowNodeE rep-narrow)
next
  case (SignExtendNode n x xe)
  then show ?case
   by (metis IRNode.distinct(2599) IRNode.inject(39) SignExtendNodeE rep-sign-extend)
next
  case (ZeroExtendNode n x xe)
  then show ?case
   by (metis IRNode.distinct(2753) IRNode.inject(50) ZeroExtendNodeE rep-zero-extend)
next
  case (LeafNode n s)
  then show ?case using rep-load-field LeafNodeE
    by (metis is-preevaluated.simps(53))
next
  case (RefNode n')
  then show ?case
    using rep-ref by blast
qed

lemma repAllDet:
  g ⊢ xs ≃L e1 ⟹
   g ⊢ xs ≃L e2 ⟹
   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) list.sel(3) repDet replist.cases)
qed

lemma encodeEvalDet:
  [g,m,p] ⊢ e ↦ v1 ⟹
   [g,m,p] ⊢ e ↦ v2 ⟹
   v1 = v2
  by (metis encodeeval-def evalDet repDet)

lemma graphDet: ([g,m,p] ⊢ n ↦ v₁) ∧ ([g,m,p] ⊢ n ↦ v₂) ⟹ v₁ = v₂
  using encodeEvalDet by blast
```

6.8.2 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 ∧ kind g2 n = AbsNode x
  assumes (g1 ⊢ x ≃ xe1) ∧ (g2 ⊢ x ≃ xe2)
  assumes xe1 ≥ xe2
  assumes (g1 ⊢ n ≃ e1) ∧ (g2 ⊢ n ≃ e2)
  shows e1 ≥ e2
  by (metis AbsNode assms(1) assms(2) assms(3) assms(4) mono-unary repDet)

lemma mono-not:
  assumes kind g1 n = NotNode x ∧ kind g2 n = NotNode x
  assumes (g1 ⊢ x ≃ xe1) ∧ (g2 ⊢ x ≃ xe2)
  assumes xe1 ≥ xe2
  assumes (g1 ⊢ n ≃ e1) ∧ (g2 ⊢ n ≃ e2)
  shows e1 ≥ e2
  by (metis NotNode assms(1) assms(2) assms(3) assms(4) mono-unary repDet)

lemma mono-negate:
  assumes kind g1 n = NegateNode x ∧ kind g2 n = NegateNode x
  assumes (g1 ⊢ x ≃ xe1) ∧ (g2 ⊢ x ≃ xe2)
  assumes xe1 ≥ xe2
  assumes (g1 ⊢ n ≃ e1) ∧ (g2 ⊢ n ≃ e2)
  shows e1 ≥ e2
  by (metis NegateNode assms(1) assms(2) assms(3) assms(4) mono-unary repDet)

lemma mono-logic-negation:
  assumes kind g1 n = LogicNegationNode x ∧ kind g2 n = LogicNegationNode x
  assumes (g1 ⊢ x ≃ xe1) ∧ (g2 ⊢ x ≃ xe2)
  assumes xe1 ≥ xe2
  assumes (g1 ⊢ n ≃ e1) ∧ (g2 ⊢ n ≃ e2)
  shows e1 ≥ e2
  by (metis LogicNegationNode assms(1) assms(2) assms(3) assms(4) mono-unary repDet)

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

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(1) assms(2) assms(3) assms(4) 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
  using assms mono-unary repDet ZeroExtendNode
  by metis

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
  using ConditionalNodeE IRNode.inject(6) assms(1) assms(2) assms(3) assms(4) assms(5) assms(6) mono-conditional repDet rep-conditional
  by (smt (verit, best) ConditionalNode)

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* $\geq$ *xe2* $\wedge$ *ye1* $\geq$ *ye2*
assumes (*g1* $\vdash$ *n* $\simeq$ *e1*) $\wedge$ (*g2* $\vdash$ *n* $\simeq$ *e2*)
shows *e1* $\geq$ *e2*
using *mono-binary assms AddNodeE IRNode.inject*(*2*) *repDet rep-add*
by (*metis IRNode.distinct*(*205*))

lemma *mono-mul*:
assumes *kind g1 n* = *MulNode x y* $\wedge$ *kind g2 n* = *MulNode x y*
assumes (*g1* $\vdash$ *x* $\simeq$ *xe1*) $\wedge$ (*g2* $\vdash$ *x* $\simeq$ *xe2*)
assumes (*g1* $\vdash$ *y* $\simeq$ *ye1*) $\wedge$ (*g2* $\vdash$ *y* $\simeq$ *ye2*)
assumes *xe1* $\geq$ *xe2* $\wedge$ *ye1* $\geq$ *ye2*
assumes (*g1* $\vdash$ *n* $\simeq$ *e1*) $\wedge$ (*g2* $\vdash$ *n* $\simeq$ *e2*)
shows *e1* $\geq$ *e2*
using *mono-binary assms IRNode.inject*(*27*) *MulNodeE repDet rep-mul*
by (*smt* (*verit*, *best*) *MulNode*)

lemma *term-graph-evaluation*:
(*g* $\vdash$ *n* $\trianglelefteq$ *e*) $\Longrightarrow$ ($\forall$ *m p v* . ([*m*,*p*] $\vdash$ *e* $\mapsto$ *v*) $\longrightarrow$ ([*g*,*m*,*p*] $\vdash$ *n* $\mapsto$ *v*))
unfolding *graph-represents-expression-def* **apply** *auto*
by (*meson encodeeval-def*)

lemma *encodes-contains*:
g $\vdash$ *n* $\simeq$ *e* $\Longrightarrow$
kind g n $\neq$ *NoNode*
apply (*induction rule*: *rep.induct*)
apply (*match IRNode.distinct* **in** *e*: *?n* $\neq$ *NoNode* $\Rightarrow$
‹*presburger add*: *e*›)+
apply *force*
by *fastforce*

lemma *no-encoding*:
assumes *n* $\notin$ *ids g*
shows $\neg$(*g* $\vdash$ *n* $\simeq$ *e*)
using *assms* **apply** *simp* **apply** (*rule notI*) **by** (*induction e*; *simp add*: *encodes-contains*)

lemma *not-excluded-keep-type*:
assumes *n* $\in$ *ids g1*
assumes *n* $\notin$ *excluded*
assumes (*excluded* $\trianglelefteq$ *as-set g1*) $\subseteq$ *as-set g2*
shows *kind g1 n* = *kind g2 n* $\wedge$ *stamp g1 n* = *stamp g2 n*
using *assms* **unfolding** *as-set-def domain-subtraction-def* **by** *blast*

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

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

6.8.3 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 c f True repDet by simp
    have h: (g2 ⊢ n ≃ e2') ∧ e1' ≥ e2'
      using True a d by blast
    then show ?thesis
      using g by blast
  next
    case False
    have n ∉ {n'}
      using False by simp
    then have i: kind g1 n = kind g2 n ∧ stamp g1 n = stamp g2 n
      using not-excluded-keep-type
      using 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 f ConditionalNode
        by (simp add: ConditionalNode.hyps(2) rep.ConditionalNode)
      obtain cn tn fn where l: kind g1 n = ConditionalNode cn tn fn
        using ConditionalNode.hyps(1) by blast
      then have mc: g1 ⊢ cn ≃ ce1
        using ConditionalNode.hyps(1) ConditionalNode.hyps(2) by fastforce
      from l have mt: g1 ⊢ tn ≃ te1
        using ConditionalNode.hyps(1) ConditionalNode.hyps(3) by fastforce
      from l have mf: g1 ⊢ fn ≃ fe1
        using ConditionalNode.hyps(1) ConditionalNode.hyps(4) by fastforce
      then show ?case
      proof -
        have g1 ⊢ cn ≃ ce1 using mc by simp
        have g1 ⊢ tn ≃ te1 using mt by simp
        have g1 ⊢ fn ≃ fe1 using mf by simp
        have cer: ∃ ce2. (g2 ⊢ cn ≃ ce2) ∧ ce1 ≥ ce2
          using ConditionalNode
          using 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
          using ConditionalNode.prems l rep.ConditionalNode cer ter
          by (smt (verit) mono-conditional)
        then show ?thesis
          by meson
      qed
    next
      case (AbsNode n x xe1)
      have k: g1 ⊢ n ≃ UnaryExpr UnaryAbs xe1 using f AbsNode
        by (simp add: AbsNode.hyps(2) rep.AbsNode)
      obtain xn where l: kind g1 n = AbsNode xn
        using AbsNode.hyps(1) by blast
      then have m: g1 ⊢ xn ≃ xe1
        using AbsNode.hyps(1) AbsNode.hyps(2) by fastforce
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1' using c m repDet by simp
        then have ev: g2 ⊢ n ≃ UnaryExpr UnaryAbs e2' using AbsNode.hyps(1)
l m n
          using AbsNode.prems True d rep.AbsNode by simp
        then have r: UnaryExpr UnaryAbs e1′ ≥ UnaryExpr UnaryAbs e2′
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using AbsNode
         using 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 (NotNode n x xe1)
      have k: g1 ⊢ n ≃ UnaryExpr UnaryNot xe1 using f NotNode
        by (simp add: NotNode.hyps(2) rep.NotNode)
      obtain xn where l: kind g1 n = NotNode xn
        using NotNode.hyps(1) by blast
      then have m: g1 ⊢ xn ≃ xe1
        using NotNode.hyps(1) NotNode.hyps(2) by fastforce
      then show ?case
      proof (cases xn = n′)
        case True
        then have n: xe1 = e1′ using c m repDet by simp
        then have ev: g2 ⊢ n ≃ UnaryExpr UnaryNot e2′ using NotNode.hyps(1)
l m n
          using NotNode.prems True d rep.NotNode by simp
        then have r: UnaryExpr UnaryNot e1′ ≥ UnaryExpr UnaryNot e2′
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using NotNode
          using False i 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 f NegateNode
      by (simp add: NegateNode.hyps(2) rep.NegateNode)
    obtain xn where l: kind g1 n = NegateNode xn
      using NegateNode.hyps(1) by blast
    then have m: g1 ⊢ xn ≃ xe1
      using NegateNode.hyps(1) NegateNode.hyps(2) by fastforce
    then show ?case
    proof (cases xn = n')
      case True
      then have n: xe1 = e1' using c m repDet by simp
      then have ev: g2 ⊢ n ≃ UnaryExpr UnaryNeg e2' using NegateNode.hyps(1)
l m n
        using NegateNode.prems True d rep.NegateNode by simp
      then have r: UnaryExpr UnaryNeg e1' ≥ UnaryExpr UnaryNeg e2'
        by (meson a mono-unary)
      then show ?thesis using ev r
        by (metis n)
    next
      case False
      have g1 ⊢ xn ≃ xe1 using m by simp
      have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
        using NegateNode
        using False i 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 f LogicNega-
tionNode
      by (simp add: LogicNegationNode.hyps(2) rep.LogicNegationNode)
    obtain xn where l: kind g1 n = LogicNegationNode xn
      using LogicNegationNode.hyps(1) by blast
    then have m: g1 ⊢ xn ≃ xe1
      using LogicNegationNode.hyps(1) LogicNegationNode.hyps(2) by fastforce
    then show ?case
    proof (cases xn = n')
      case True
      then have n: xe1 = e1' using c m repDet by simp
        then have ev: g2 ⊢ n ≃ UnaryExpr UnaryLogicNegation e2' using
LogicNegationNode.hyps(1) l m n
        using LogicNegationNode.prems True d rep.LogicNegationNode by simp
      then have r: UnaryExpr UnaryLogicNegation e1' ≥ UnaryExpr UnaryLog-
```

```
icNegation e2′
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using LogicNegationNode
          using False i 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 f AddNode
        by (simp add: AddNode.hyps(2) rep.AddNode)
      obtain xn yn where l: kind g1 n = AddNode xn yn
        using AddNode.hyps(1) by blast
      then have mx: g1 ⊢ xn ≃ xe1
        using AddNode.hyps(1) AddNode.hyps(2) by fastforce
      from l have my: g1 ⊢ yn ≃ ye1
        using AddNode.hyps(1) AddNode.hyps(3) by fastforce
      then show ?case
      proof −
        have g1 ⊢ xn ≃ xe1 using mx by simp
        have g1 ⊢ yn ≃ ye1 using my by simp
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using AddNode
          using 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
          using 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 f MulNode
        by (simp add: MulNode.hyps(2) rep.MulNode)
      obtain xn yn where l: kind g1 n = MulNode xn yn
        using MulNode.hyps(1) by blast
      then have mx: g1 ⊢ xn ≃ xe1
        using MulNode.hyps(1) MulNode.hyps(2) by fastforce
      from l have my: g1 ⊢ yn ≃ ye1
        using MulNode.hyps(1) MulNode.hyps(3) by fastforce
      then show ?case
      proof –
        have g1 ⊢ xn ≃ xe1 using mx by simp
        have g1 ⊢ yn ≃ ye1 using my by simp
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using MulNode
          using 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
          using 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 (SubNode n x y xe1 ye1)
      have k: g1 ⊢ n ≃ BinaryExpr BinSub xe1 ye1 using f SubNode
        by (simp add: SubNode.hyps(2) rep.SubNode)
      obtain xn yn where l: kind g1 n = SubNode xn yn
        using SubNode.hyps(1) by blast
      then have mx: g1 ⊢ xn ≃ xe1
        using SubNode.hyps(1) SubNode.hyps(2) by fastforce
      from l have my: g1 ⊢ yn ≃ ye1
        using SubNode.hyps(1) SubNode.hyps(3) by fastforce
      then show ?case
      proof –
        have g1 ⊢ xn ≃ xe1 using mx by simp
        have g1 ⊢ yn ≃ ye1 using my by simp
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using SubNode
          using 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 f AndNode
        by (simp add: AndNode.hyps(2) rep.AndNode)
      obtain xn yn where l: kind g1 n = AndNode xn yn
        using AndNode.hyps(1) by blast
      then have mx: g1 ⊢ xn ≃ xe1
        using AndNode.hyps(1) AndNode.hyps(2) by fastforce
      from l have my: g1 ⊢ yn ≃ ye1
        using AndNode.hyps(1) AndNode.hyps(3) by fastforce
      then show ?case
      proof -
        have g1 ⊢ xn ≃ xe1 using mx by simp
        have g1 ⊢ yn ≃ ye1 using my by simp
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using AndNode
          using 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 f OrNode
        by (simp add: OrNode.hyps(2) rep.OrNode)
      obtain xn yn where l: kind g1 n = OrNode xn yn
        using OrNode.hyps(1) by blast
      then have mx: g1 ⊢ xn ≃ xe1
        using OrNode.hyps(1) OrNode.hyps(2) by fastforce
      from l have my: g1 ⊢ yn ≃ ye1
        using OrNode.hyps(1) OrNode.hyps(3) by fastforce
      then show ?case
      proof -
        have g1 ⊢ xn ≃ xe1 using mx by simp
        have g1 ⊢ yn ≃ ye1 using my by simp
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using OrNode
          using 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 f XorNode
        by (simp add: XorNode.hyps(2) rep.XorNode)
      obtain xn yn where l: kind g1 n = XorNode xn yn
        using XorNode.hyps(1) by blast
      then have mx: g1 ⊢ xn ≃ xe1
        using XorNode.hyps(1) XorNode.hyps(2) by fastforce
      from l have my: g1 ⊢ yn ≃ ye1
        using XorNode.hyps(1) XorNode.hyps(3) by fastforce
      then show ?case
      proof -
        have g1 ⊢ xn ≃ xe1 using mx by simp
        have g1 ⊢ yn ≃ ye1 using my by simp
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using XorNode
          using 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 f ShortCir-
cuitOrNode
        by (simp add: ShortCircuitOrNode.hyps(2) rep.ShortCircuitOrNode)
      obtain xn yn where l: kind g1 n = ShortCircuitOrNode xn yn
        using ShortCircuitOrNode.hyps(1) by blast
      then have mx: g1 ⊢ xn ≃ xe1
       using ShortCircuitOrNode.hyps(1) ShortCircuitOrNode.hyps(2) by fastforce
      from l have my: g1 ⊢ yn ≃ ye1
       using ShortCircuitOrNode.hyps(1) ShortCircuitOrNode.hyps(3) by fastforce
      then show ?case
      proof -
        have g1 ⊢ xn ≃ xe1 using mx by simp
        have g1 ⊢ yn ≃ ye1 using my by simp
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using ShortCircuitOrNode
          using 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 f LeftShiftNode
        by (simp add: LeftShiftNode.hyps(2) rep.LeftShiftNode)
      obtain xn yn where l: kind g1 n = LeftShiftNode xn yn
        using LeftShiftNode.hyps(1) by blast
      then have mx: g1 ⊢ xn ≃ xe1
        using LeftShiftNode.hyps(1) LeftShiftNode.hyps(2) by fastforce
      from l have my: g1 ⊢ yn ≃ ye1
        using LeftShiftNode.hyps(1) LeftShiftNode.hyps(3) by fastforce
      then show ?case
      proof −
        have g1 ⊢ xn ≃ xe1 using mx by simp
        have g1 ⊢ yn ≃ ye1 using my by simp
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using LeftShiftNode
          using 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 f 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 blast
      then have mx: g1 ⊢ xn ≃ xe1
        using RightShiftNode.hyps(1) RightShiftNode.hyps(2) by fastforce
      from l have my: g1 ⊢ yn ≃ ye1
        using RightShiftNode.hyps(1) RightShiftNode.hyps(3) by fastforce
      then show ?case
      proof -
        have g1 ⊢ xn ≃ xe1 using mx by simp
        have g1 ⊢ yn ≃ ye1 using my by simp
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using RightShiftNode
          using 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 f UnsignedRight-
ShiftNode
       by (simp add: UnsignedRightShiftNode.hyps(2) rep.UnsignedRightShiftNode)
      obtain xn yn where l: kind g1 n = UnsignedRightShiftNode xn yn
        using UnsignedRightShiftNode.hyps(1) by blast
      then have mx: g1 ⊢ xn ≃ xe1
        using UnsignedRightShiftNode.hyps(1) UnsignedRightShiftNode.hyps(2) by
fastforce
      from l have my: g1 ⊢ yn ≃ ye1
        using UnsignedRightShiftNode.hyps(1) UnsignedRightShiftNode.hyps(3) by
fastforce
      then show ?case
      proof -
        have g1 ⊢ xn ≃ xe1 using mx by simp
        have g1 ⊢ yn ≃ ye1 using my by simp
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using UnsignedRightShiftNode
          using a b c d l no-encoding not-excluded-keep-type repDet singletonD
          by (metis-node-eq-binary UnsignedRightShiftNode)
        have ∃ ye2. (g2 ⊢ yn ≃ ye2) ∧ ye1 ≥ ye2
          using UnsignedRightShiftNode a b c d l no-encoding not-excluded-keep-type
repDet singletonD
          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 f IntegerBe-
lowNode
        by (simp add: IntegerBelowNode.hyps(2) rep.IntegerBelowNode)
      obtain xn yn where l: kind g1 n = IntegerBelowNode xn yn
        using IntegerBelowNode.hyps(1) by blast
      then have mx: g1 ⊢ xn ≃ xe1
        using IntegerBelowNode.hyps(1) IntegerBelowNode.hyps(2) by fastforce
      from l have my: g1 ⊢ yn ≃ ye1
        using IntegerBelowNode.hyps(1) IntegerBelowNode.hyps(3) by fastforce
      then show ?case
      proof -
        have g1 ⊢ xn ≃ xe1 using mx by simp
        have g1 ⊢ yn ≃ ye1 using my by simp
        have xer: ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using IntegerBelowNode
          using 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 f IntegerEqual-
sNode
        by (simp add: IntegerEqualsNode.hyps(2) rep.IntegerEqualsNode)
      obtain xn yn where l: kind g1 n = IntegerEqualsNode xn yn
        using IntegerEqualsNode.hyps(1) by blast
      then have mx: g1 ⊢ xn ≃ xe1
        using IntegerEqualsNode.hyps(1) IntegerEqualsNode.hyps(2) by fastforce
      from l have my: g1 ⊢ yn ≃ ye1
        using IntegerEqualsNode.hyps(1) IntegerEqualsNode.hyps(3) by fastforce
      then show ?case
```

proof –
have *g1* ⊢ *xn* ≃ *xe1* **using** *mx* **by** *simp*
have *g1* ⊢ *yn* ≃ *ye1* **using** *my* **by** *simp*
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *IntegerEqualsNode*
using *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** *f 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** *blast*
then have *mx*: *g1* ⊢ *xn* ≃ *xe1*
using *IntegerLessThanNode.hyps*(*1*) *IntegerLessThanNode.hyps*(*2*) **by** *fastforce*
from *l* **have** *my*: *g1* ⊢ *yn* ≃ *ye1*
using *IntegerLessThanNode.hyps*(*1*) *IntegerLessThanNode.hyps*(*3*) **by** *fastforce*
then show *?case*
proof –
have *g1* ⊢ *xn* ≃ *xe1* **using** *mx* **by** *simp*
have *g1* ⊢ *yn* ≃ *ye1* **using** *my* **by** *simp*
have *xer*: ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
using *IntegerLessThanNode*
using *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 (NarrowNode n inputBits resultBits x xe1)
      have k: g1 ⊢ n ≃ UnaryExpr (UnaryNarrow inputBits resultBits) xe1 using
f 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 blast
      then have m: g1 ⊢ xn ≃ xe1
        using NarrowNode.hyps(1) NarrowNode.hyps(2)
        by auto
      then show ?case
      proof (cases xn = n')
        case True
        then have n: xe1 = e1' using c m repDet by simp
        then have ev: g2 ⊢ n ≃ UnaryExpr (UnaryNarrow inputBits resultBits) e2'
using NarrowNode.hyps(1) l m n
          using NarrowNode.prems True d rep.NarrowNode by simp
       then have r: UnaryExpr (UnaryNarrow inputBits resultBits) e1' ≥ UnaryExpr
(UnaryNarrow inputBits resultBits) e2'
          by (meson a mono-unary)
        then show ?thesis using ev r
          by (metis n)
      next
        case False
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using NarrowNode
        using 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 re-
sultBits) 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 f 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 blast
      then have m: g1 ⊢ xn ≃ xe1
        using SignExtendNode.hyps(1) SignExtendNode.hyps(2)
        by auto
      then show ?case
```

proof (*cases xn* = *n*′)
  case *True*
  then have *n*: *xe1* = *e1*′ **using** *c m repDet* **by** *simp*
  then have *ev*: *g2* ⊢ *n* ≃ *UnaryExpr* (*UnarySignExtend inputBits resultBits*) *e2*′ **using** *SignExtendNode.hyps*(*1*) *l m n*
    using *SignExtendNode.prems True d rep.SignExtendNode* **by** *simp*
    then have *r*: *UnaryExpr* (*UnarySignExtend inputBits resultBits*) *e1*′ ≥ *UnaryExpr* (*UnarySignExtend inputBits resultBits*) *e2*′
    by (*meson a mono-unary*)
  then show *?thesis* **using** *ev r*
    by (*metis n*)
next
  case *False*
  have *g1* ⊢ *xn* ≃ *xe1* **using** *m* **by** *simp*
  have ∃ *xe2*. (*g2* ⊢ *xn* ≃ *xe2*) ∧ *xe1* ≥ *xe2*
    using *SignExtendNode*
  using *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** *f 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** *blast*
  then have *m*: *g1* ⊢ *xn* ≃ *xe1*
    using *ZeroExtendNode.hyps*(*1*) *ZeroExtendNode.hyps*(*2*)
    by *auto*
  then show *?case*
  proof (*cases xn* = *n*′)
    case *True*
    then have *n*: *xe1* = *e1*′ **using** *c m repDet* **by** *simp*
    then have *ev*: *g2* ⊢ *n* ≃ *UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *e2*′ **using** *ZeroExtendNode.hyps*(*1*) *l m n*
      using *ZeroExtendNode.prems True d rep.ZeroExtendNode* **by** *simp*
      then have *r*: *UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *e1*′ ≥ *UnaryExpr* (*UnaryZeroExtend inputBits resultBits*) *e2*′
      by (*meson a mono-unary*)
    then show *?thesis* **using** *ev r*
      by (*metis n*)
  next
    case *False*

```
        have g1 ⊢ xn ≃ xe1 using m by simp
        have ∃ xe2. (g2 ⊢ xn ≃ xe2) ∧ xe1 ≥ xe2
          using ZeroExtendNode
        using False b encodes-contains l not-excluded-keep-type not-in-g singleton-iff
          by (metis-node-eq-ternary ZeroExtendNode)
      then have ∃ xe2. (g2 ⊢ n ≃ UnaryExpr (UnaryZeroExtend inputBits result-
Bits) 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 (RefNode n')
      then show ?case
         by (metis a b c d no-encoding not-excluded-keep-type rep.RefNode repDet
singletonD)
    qed
  qed
qed
```

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

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

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** *blast*

lemma *subset-kind*[*simp*]: *as-set g1* ⊆ *as-set g2* ⟹ *g1* ⊢ *n* ≃ *e* ⟹ *kind g1 n* = *kind g2 n*
  using *eval-contains-id* **unfolding** *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* **unfolding** *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*: *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*: *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: 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)
by (solve-subset-eval as-set: assms(1) eval: RefNode)
```

```
lemma subset-refines:
  assumes as-set g1 ⊆ as-set g2
  shows graph-refinement g1 g2
proof –
  have ids g1 ⊆ ids g2 using assms unfolding 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 ∈ ids g1
      assume 2:g1 ⊢ n ≃ e1

      show ∃ e2. (g2 ⊢ n ≃ e2) ∧ e1 ≥ e2
        using assms 1 2 using subset-implies-evals
        by (meson equal-refines)
    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 $g_1$ $g_2$
  using subset-refines
  by (meson encodeeval-def graph-represents-expression-def le-expr-def)
```

6.8.4 Term Graph Reconstruction

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

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

lemma find-new-kind:
  assumes g′ = add-node nid (node, s) g
  assumes node ≠ NoNode
  shows kind g′ nid = node
  using assms
  using add-node-lookup by presburger

lemma find-new-stamp:
  assumes g′ = add-node nid (node, s) g
  assumes node ≠ NoNode
  shows stamp g′ nid = s
  using assms
  using add-node-lookup by presburger

lemma sorted-bottom:
  assumes finite xs
  assumes x ∈ xs
  shows x ≤ last(sorted-list-of-set(xs::nat set))
  using assms
  using sorted2-simps(2) sorted-list-of-set(2)
 by (smt (verit, del-insts) Diff-iff Max-ge Max-in empty-iff list.set(1) snoc-eq-iff-butlast
sorted-insort-is-snoc sorted-list-of-set(1) sorted-list-of-set.fold-insort-key.infinite sorted-list-of-set.fold-insort-ke

lemma fresh: finite xs ⟹ last(sorted-list-of-set(xs::nat set)) + 1 ∉ xs
  using sorted-bottom
  using not-le by auto

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

lemma graph-unchanged-rep-unchanged:
  assumes ∀ n ∈ ids g. kind g n = kind g′ n
  assumes ∀ n ∈ ids g. stamp g n = stamp g′ n
  shows (g ⊢ n ≃ e) ⟶ (g′ ⊢ n ≃ 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.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.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.NarrowNode)
        apply (metis no-encoding rep.SignExtendNode)
       apply (metis no-encoding rep.ZeroExtendNode)
      apply (metis no-encoding rep.LeafNode)
     by (metis no-encoding rep.RefNode)
  done

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

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 (ConstantNodeSame g c n)
  then show ?case by blast
next
  case (ConstantNodeNew g c n g')
  then show ?case using fresh-ids fresh-node-subset
    by presburger
next
  case (ParameterNodeSame g i s n)
  then show ?case by blast
next
  case (ParameterNodeNew g i s n g')
  then show ?case using fresh-ids fresh-node-subset
    by presburger
next
  case (ConditionalNodeSame g ce g2 c te g3 t fe g4 f s' n)
  then show ?case by blast
next
```

case (*ConditionalNodeNew g ce g2 c te g3 t fe g4 f s′ n g′*)
then show *?case* **using** *fresh-ids fresh-node-subset*
by (*meson subset-trans*)
next
case (*UnaryNodeSame g xe g2 x s′ op n*)
then show *?case* **by** *blast*
next
case (*UnaryNodeNew g xe g2 x s′ op n g′*)
then show *?case* **using** *fresh-ids fresh-node-subset*
by (*meson subset-trans*)
next
case (*BinaryNodeSame g xe g2 x ye g3 y s′ op n*)
then show *?case* **by** *blast*
next
case (*BinaryNodeNew g xe g2 x ye g3 y s′ op n g′*)
then show *?case* **using** *fresh-ids fresh-node-subset*
by (*meson subset-trans*)
next
case (*AllLeafNodes g n s*)
then show *?case* **by** *blast*
qed

lemma *fresh-node-preserves-other-nodes*:
assumes $n' = \textit{get-fresh-id}\ g$
assumes $g' = \textit{add-node}\ n'\ (k, s)\ g$
shows $\forall\ n \in \textit{ids}\ g\ .\ (g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$
using *assms*
by (*smt* (*verit*, *ccfv-SIG*) *Diff-idemp Diff-insert-absorb add-changed disjoint-change fresh-ids graph-unchanged-rep-unchanged unchanged.elims*(*2*))

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

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

lemma *unrep-unchanged*:
assumes $g \oplus e \rightsquigarrow (g', n)$
shows $\forall\ n \in \textit{ids}\ g\ .\ \forall\ e.\ (g \vdash n \simeq e) \longrightarrow (g' \vdash n \simeq e)$
using *assms unrep-subset fresh-node-preserves-other-nodes*
by (*meson subset-implies-evals*)

theorem *term-graph-reconstruction*:

```
g ⊕ e ⇝ (g', n) ⟹ (g' ⊢ n ≃ e) ∧ 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 (ConstantNodeSame g' c n)
  then have kind g' n = ConstantNode c
    using find-exists-kind local.ConstantNodeSame by blast
  then show ?case using ConstantNode by blast
next
  case (ConstantNodeNew g c)
  then show ?case
    using ConstantNode IRNode.distinct(683) add-node-lookup by presburger
next
  case (ParameterNodeSame i s)
  then show ?case
    by (metis ParameterNode find-exists-kind find-exists-stamp)
next
  case (ParameterNodeNew g i s)
  then show ?case
    by (metis IRNode.distinct(2447) ParameterNode add-node-lookup)
next
  case (ConditionalNodeSame g ce g2 c te g3 t fe g4 f s' n)
  then have k: kind g4 n = ConditionalNode c t f
    using find-exists-kind by blast
  have c: g4 ⊢ c ≃ ce using local.ConditionalNodeSame unrep-unchanged
    using no-encoding by blast
  have t: g4 ⊢ t ≃ te using local.ConditionalNodeSame unrep-unchanged
    using no-encoding by blast
  have f: g4 ⊢ f ≃ fe using local.ConditionalNodeSame unrep-unchanged
    using no-encoding by blast
  then show ?case using c t f
    using ConditionalNode k by blast
next
  case (ConditionalNodeNew g ce g2 c te g3 t fe g4 f s' n g')
  moreover have ConditionalNode c t f ≠ NoNode
    using unary-node.elims by blast
  ultimately have k: kind g' n = ConditionalNode c t f
    using find-new-kind local.ConditionalNodeNew
    by presburger
  then have c: g' ⊢ c ≃ ce using local.ConditionalNodeNew unrep-unchanged
    using no-encoding
    by (metis ConditionalNodeNew.hyps(9) fresh-node-preserves-other-nodes)
  then have t: g' ⊢ t ≃ te using local.ConditionalNodeNew unrep-unchanged
    using no-encoding fresh-node-preserves-other-nodes
    by metis
  then have f: g' ⊢ f ≃ fe using local.ConditionalNodeNew unrep-unchanged
    using no-encoding fresh-node-preserves-other-nodes
    by metis
  then show ?case using c t f
```

```
    using ConditionalNode k by blast
next
  case (UnaryNodeSame g xe g′ x s′ op n)
  then have k: kind g′ n = unary-node op x
    using find-exists-kind local.UnaryNodeSame by blast
  then have g′ ⊢ x ≃ xe using local.UnaryNodeSame by blast
  then show ?case using k
    apply (cases op)
    using AbsNode unary-node.simps(1) apply presburger
    using NegateNode unary-node.simps(3) apply presburger
    using NotNode unary-node.simps(2) apply presburger
    using LogicNegationNode unary-node.simps(4) apply presburger
    using NarrowNode unary-node.simps(5) apply presburger
    using SignExtendNode unary-node.simps(6) apply presburger
    using ZeroExtendNode unary-node.simps(7) by presburger
next
  case (UnaryNodeNew g xe g2 x s′ op n g′)
  moreover have unary-node op x ≠ NoNode
    using unary-node.elims by blast
  ultimately have k: kind g′ n = unary-node op x
    using find-new-kind local.UnaryNodeNew
    by presburger
  have x ∈ ids g2 using local.UnaryNodeNew
    using eval-contains-id by blast
  then have x ≠ n using local.UnaryNodeNew(5) fresh-ids by blast
  have g′ ⊢ x ≃ xe using local.UnaryNodeNew fresh-node-preserves-other-nodes
    using ‹x ∈ ids g2› by blast
  then show ?case using k
    apply (cases op)
    using AbsNode unary-node.simps(1) apply presburger
    using NegateNode unary-node.simps(3) apply presburger
    using NotNode unary-node.simps(2) apply presburger
    using LogicNegationNode unary-node.simps(4) apply presburger
    using NarrowNode unary-node.simps(5) apply presburger
    using SignExtendNode unary-node.simps(6) apply presburger
    using ZeroExtendNode unary-node.simps(7) by presburger
next
  case (BinaryNodeSame g xe g2 x ye g3 y s′ op n)
  then have k: kind g3 n = bin-node op x y
    using find-exists-kind by blast
  have x: g3 ⊢ x ≃ xe using local.BinaryNodeSame unrep-unchanged
    using no-encoding by blast
  have y: g3 ⊢ y ≃ ye using local.BinaryNodeSame unrep-unchanged
    using no-encoding by blast
  then show ?case using x y k apply (cases op)
    using AddNode bin-node.simps(1) apply presburger
    using MulNode bin-node.simps(2) apply presburger
    using SubNode bin-node.simps(3) apply presburger
    using AndNode bin-node.simps(4) apply presburger
    using OrNode bin-node.simps(5) apply presburger
    using XorNode bin-node.simps(6) apply presburger
    using ShortCircuitOrNode bin-node.simps(7) apply presburger
    using LeftShiftNode bin-node.simps(8) apply presburger
    using RightShiftNode bin-node.simps(9) apply presburger
    using UnsignedRightShiftNode bin-node.simps(10) apply presburger
    using IntegerEqualsNode bin-node.simps(11) apply presburger
    using IntegerLessThanNode bin-node.simps(12) apply presburger
    using IntegerBelowNode bin-node.simps(13) by presburger
  next
   case (BinaryNodeNew g xe g2 x ye g3 y s' op n g')
   moreover have bin-node op x y ≠ NoNode
    using bin-node.elims by blast
   ultimately have k: kind g' n = bin-node op x y
    using find-new-kind local.BinaryNodeNew
    by presburger
   then have k: kind g' n = bin-node op x y
    using find-exists-kind by blast
   have x: g' ⊢ x ≃ xe using local.BinaryNodeNew unrep-unchanged
    using no-encoding
    by (meson fresh-node-preserves-other-nodes)
   have y: g' ⊢ y ≃ ye using local.BinaryNodeNew unrep-unchanged
    using no-encoding
    by (meson fresh-node-preserves-other-nodes)
   then show ?case using x y k apply (cases op)
    using AddNode bin-node.simps(1) apply presburger
    using MulNode bin-node.simps(2) apply presburger
    using SubNode bin-node.simps(3) apply presburger
    using AndNode bin-node.simps(4) apply presburger
    using OrNode bin-node.simps(5) apply presburger
    using XorNode bin-node.simps(6) apply presburger
    using ShortCircuitOrNode bin-node.simps(7) apply presburger
    using LeftShiftNode bin-node.simps(8) apply presburger
    using RightShiftNode bin-node.simps(9) apply presburger
    using UnsignedRightShiftNode bin-node.simps(10) apply presburger
    using IntegerEqualsNode bin-node.simps(11) apply presburger
    using IntegerLessThanNode bin-node.simps(12) apply presburger
    using IntegerBelowNode bin-node.simps(13) by presburger
  next
   case (AllLeafNodes g n s)
   then show ?case using rep.LeafNode by blast
  qed
  done

lemma ref-refinement:
  assumes g ⊢ n ≃ e1
  assumes kind g n' = RefNode n
  shows g ⊢ n' ⊴ e1
  using assms RefNode
```

by (*meson equal-refines graph-represents-expression-def*)

lemma *unrep-refines*:
assumes $g \oplus e \rightsquigarrow (g', n)$
shows *graph-refinement* $g\ g'$
using *assms*
using *graph-refinement-def subset-refines unrep-subset* **by** *blast*

lemma *add-new-node-refines*:
assumes $n \notin ids\ g$
assumes $g' = add\text{-}node\ n\ (k, s)\ g$
shows *graph-refinement* $g\ g'$
using *assms* **unfolding** *graph-refinement*
using *fresh-node-subset subset-refines* **by** *presburger*

lemma *add-node-as-set*:
assumes $g' = add\text{-}node\ n\ (k, s)\ g$
shows $(\{n\} \trianglelefteq as\text{-}set\ g) \subseteq as\text{-}set\ g'$
using *assms* **unfolding** *as-set-def domain-subtraction-def*
using *add-changed*
by (*smt* (*z3*) *case-prodE changeonly.simps mem-Collect-eq prod.sel*(*1*) *subsetI*)

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

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

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

lemma *find-none*:
assumes *find-node-and-stamp g* (*k*, *s*) = *None*
shows $\forall\ n \in ids\ g.\ kind\ g\ n \neq k \lor stamp\ g\ n \neq s$
proof –
have $(\nexists n.\ n \in ids\ g \land (kind\ g\ n = k \land stamp\ g\ n = s))$
using *assms* **unfolding** *find-node-and-stamp.simps* **using** *find-None-iff un-wrap-sorted*
by (*metis* (*mono-tags*, *lifting*))
then show *?thesis*
by *blast*
qed

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

6.8.5 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 ∀ e. (g ⊢ n ≃ e) ⟶ (g ⊢ n' ≃ 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 NotNode)
                  apply (metis NegateNode)
                 apply (metis LogicNegationNode)
                apply (metis AddNode)
               apply (metis MulNode)
              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 NarrowNode)
      apply (metis SignExtendNode)
      apply (metis ZeroExtendNode)
    defer
      apply (metis RefNode)
    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*
  using *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*)}
  unfolding *true-ids-def ids-def*
  using *ids-def is-RefNode-def* **by** *fastforce*

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*)))
  using *assms*
  by (*metis Rep-IRGraph-inverse add-node.rep-eq fst-conv*)

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*}
  using *assms ids.rep-eq add-node-some-node-def*
  by (*simp add*: *add-node.rep-eq*)

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

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* **unfolding** *add-node-def*
  apply (*cases fst node* = *NoNode*)
  using *ids.rep-eq replace-node.rep-eq replace-node-def* **apply** *auto*[*1*]

unfolding *ids-def*
by (*smt* (*verit*, *best*) *Collect-cong Un-insert-right dom-fun-upd fst-conv fun-upd-apply ids.rep-eq ids-def insert-absorb mem-Collect-eq option.inject option.simps*(*3*) *replace-node.rep-eq replace-node-def sup-bot.right-neutral*)

lemma *convert-maximal*:
assumes $\forall\, n\ n'.\ n \in \textit{true-ids}\ g \wedge n' \in \textit{true-ids}\ g \longrightarrow (\forall\, e\ e'.\ (g \vdash n \simeq e) \wedge (g \vdash n' \simeq e') \longrightarrow e \neq e')$
shows *maximal-sharing g*
using *assms*
using *maximal-sharing* **by** *blast*

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

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

lemma *true-ids*:
$\textit{true-ids}\ g = \textit{ids}\ g - \{n \in \textit{ids}\ g.\ \textit{is-RefNode}\ (\textit{kind}\ g\ n)\}$
unfolding *true-ids-def*
by *fastforce*

lemma *as-set-ids*:
assumes $\textit{as-set}\ g = \textit{as-set}\ g'$
shows $\textit{ids}\ g = \textit{ids}\ g'$
using *assms*
by (*metis antisym equalityD1 graph-refinement-def subset-refines*)

lemma *ids-add-update*:
assumes $k \neq \textit{NoNode}$
assumes $n \notin \textit{ids}\ g$
assumes $g' = \textit{add-node}\ n\ (k,\ s)\ g$
shows $\textit{ids}\ g' = \textit{ids}\ g \cup \{n\}$
using *assms* **apply** (*subst assms*(*3*)) **using** *add-node-set-eq as-set-ids*
by (*smt* (*verit*, *del-insts*) *Collect-cong Diff-idemp Diff-insert-absorb Un-commute add-node.rep-eq add-node-def ids.rep-eq ids-add-update-v1 ids-add-update-v2 insertE insert-Collect insert-is-Un map-upd-Some-unfold mem-Collect-eq replace-node-def*

replace-node-unchanged)

lemma *true-ids-add-update*:
  assumes $k \neq NoNode$
  assumes $n \notin ids\ g$
  assumes $g' = add\text{-}node\ n\ (k,\ s)\ g$
  assumes $\neg(is\text{-}RefNode\ k)$
  shows $true\text{-}ids\ g' = true\text{-}ids\ g \cup \{n\}$
  using *assms* **using** *true-ids ids-add-update*
  by (*smt* (*z3*) *Collect-cong Diff-iff Diff-insert-absorb Un-commute add-node-def find-new-kind insert-Diff-if insert-is-Un mem-Collect-eq replace-node-def replace-node-unchanged*)

lemma *new-def*:
  assumes $(new \trianglelefteq as\text{-}set\ g') = as\text{-}set\ g$
  shows $n \in ids\ g \longrightarrow n \notin new$
  using *assms*
  by (*smt* (*z3*) *as-set-def case-prodD domain-subtraction-def mem-Collect-eq*)

lemma *add-preserves-rep*:
  assumes *unchanged*: $(new \trianglelefteq as\text{-}set\ g') = as\text{-}set\ g$
  assumes *closed*: *wf-closed g*
  assumes *existed*: $n \in ids\ g$
  assumes $g' \vdash n \simeq e$
  shows $g \vdash n \simeq e$
proof (*cases* $n \in new$)
  case *True*
  have $n \notin ids\ g$
    using *unchanged True* **unfolding** *as-set-def domain-subtraction-def*
    by *blast*
  then show *?thesis* **using** *existed* **by** *simp*
next
  case *False*
  then have *kind-eq*: $\forall\ n'\ .\ n' \notin new \longrightarrow kind\ g\ n' = kind\ g'\ n'$
    — can be more general than *stamp_eq* because NoNode default is equal
    using *unchanged not-excluded-keep-type*
    by (*smt* (*z3*) *case-prodE domain-subtraction-def ids-some mem-Collect-eq subsetI*)
  from *False* **have** *stamp-eq*: $\forall\ n' \in ids\ g'\ .\ n' \notin new \longrightarrow stamp\ g\ n' = stamp\ g'\ n'$
    using *unchanged not-excluded-keep-type*
    by (*metis equalityE*)
  show *?thesis* **using** *assms*(*4*) *kind-eq stamp-eq False*
  proof (*induction n e rule*: *rep.induct*)
    case (*ConstantNode n c*)
    then show *?case*
      using *rep.ConstantNode kind-eq* **by** *presburger*
  next

```
    case (ParameterNode n i s)
    then show ?case
      using rep.ParameterNode
      by (metis no-encoding)
  next
    case (ConditionalNode n c t f ce te fe)
    have kind: kind g n = ConditionalNode c t f
       using ConditionalNode.hyps(1) ConditionalNode.prems(3) kind-eq by pres-
burger
    then have isin: n ∈ ids g
      by simp
    have inputs: {c, t, f} = inputs g n
     using kind unfolding inputs.simps using inputs-of-ConditionalNode by simp
    have c ∈ ids g ∧ t ∈ ids g ∧ f ∈ ids g
      using closed unfolding wf-closed-def
      using isin inputs by blast
    then have c ∉ new ∧ t ∉ new ∧ f ∉ new
      using new-def unchanged by blast
    then show ?case using ConditionalNode apply simp
      using rep.ConditionalNode by presburger
  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
      using kind unfolding inputs.simps by simp
    have x ∈ ids g
      using closed unfolding wf-closed-def
      using isin inputs by blast
    then have x ∉ new
      using new-def unchanged by blast
    then show ?case
      using AbsNode
      using rep.AbsNode by presburger
  next
    case (NotNode n x xe)
    then have kind: kind g n = NotNode x
      by simp
    then have isin: n ∈ ids g
      by simp
    have inputs: {x} = inputs g n
      using kind unfolding inputs.simps by simp
    have x ∈ ids g
      using closed unfolding wf-closed-def
      using isin inputs by blast
    then have x ∉ new
      using new-def unchanged by blast
  then show ?case using NotNode
    using rep.NotNode by presburger
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
    using kind unfolding inputs.simps by simp
  have x ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new
    using new-def unchanged by blast
  then show ?case using NegateNode
    using rep.NegateNode by presburger
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
    using kind unfolding inputs.simps by simp
  have x ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new
    using new-def unchanged by blast
  then show ?case using LogicNegationNode
    using rep.LogicNegationNode by presburger
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
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using AddNode
    using rep.AddNode by presburger
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
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using MulNode
    using rep.MulNode by presburger
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
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using SubNode
    using rep.SubNode by presburger
next
  case (AndNode n x y xe ye)
  then have kind: kind g n = AndNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using AndNode
    using rep.AndNode by presburger
next
  case (OrNode n x y xe ye)
  then have kind: kind g n = OrNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using OrNode
    using rep.OrNode by presburger
next
  case (XorNode n x y xe ye)
  then have kind: kind g n = XorNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using XorNode
    using rep.XorNode by presburger
next
  case (ShortCircuitOrNode n x y xe ye)
  then have kind: kind g n = ShortCircuitOrNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using ShortCircuitOrNode
    using rep.ShortCircuitOrNode by presburger
next
  case (LeftShiftNode n x y xe ye)
  then have kind: kind g n = LeftShiftNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using LeftShiftNode
    using rep.LeftShiftNode by presburger
next
  case (RightShiftNode n x y xe ye)
  then have kind: kind g n = RightShiftNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using RightShiftNode
    using rep.RightShiftNode by presburger
next
  case (UnsignedRightShiftNode n x y xe ye)
  then have kind: kind g n = UnsignedRightShiftNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using UnsignedRightShiftNode
    using rep.UnsignedRightShiftNode by presburger
next
  case (IntegerBelowNode n x y xe ye)
  then have kind: kind g n = IntegerBelowNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using IntegerBelowNode
```

```
    using rep.IntegerBelowNode by presburger
next
  case (IntegerEqualsNode n x y xe ye)
  then have kind: kind g n = IntegerEqualsNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using IntegerEqualsNode
    using rep.IntegerEqualsNode by presburger
next
  case (IntegerLessThanNode n x y xe ye)
  then have kind: kind g n = IntegerLessThanNode x y
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x, y} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g ∧ y ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new ∧ y ∉ new
    using new-def unchanged by blast
  then show ?case using IntegerLessThanNode
    using rep.IntegerLessThanNode by presburger
next
  case (NarrowNode n inputBits resultBits x xe)
  then have kind: kind g n = NarrowNode inputBits resultBits x
    by simp
  then have isin: n ∈ ids g
    by simp
  have inputs: {x} = inputs g n
    using kind unfolding inputs.simps by simp
  have x ∈ ids g
    using closed unfolding wf-closed-def
    using isin inputs by blast
  then have x ∉ new
    using new-def unchanged by blast
  then show ?case using NarrowNode
    using rep.NarrowNode by presburger
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
      using kind unfolding inputs.simps by simp
    have x ∈ ids g
      using closed unfolding wf-closed-def
      using isin inputs by blast
    then have x ∉ new
      using new-def unchanged by blast
    then show ?case using SignExtendNode
      using rep.SignExtendNode by presburger
  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
      using kind unfolding inputs.simps by simp
    have x ∈ ids g
      using closed unfolding wf-closed-def
      using isin inputs by blast
    then have x ∉ new
      using new-def unchanged by blast
    then show ?case using ZeroExtendNode
      using rep.ZeroExtendNode by presburger
  next
    case (LeafNode n s)
    then show ?case
      by (metis no-encoding rep.LeafNode)
  next
    case (RefNode n n' e)
    then have kind: kind g n = RefNode n'
      by simp
    then have isin: n ∈ ids g
      by simp
    have inputs: {n'} = inputs g n
      using kind unfolding inputs.simps by simp
    have n' ∈ ids g
      using closed unfolding wf-closed-def
      using isin inputs by blast
    then have n' ∉ new
      using new-def unchanged by blast
    then show ?case
      using RefNode
      using rep.RefNode by presburger
  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** *blast*

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

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

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

lemma *binary-succ*:
  assumes $kind\ g\ n = bin\text{-}node\ op\ x\ y$
  shows $succ\ g\ n = \{\}$
  using *assms* **by** (*cases op*; *auto*)

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

lemma *unrep-preserves-closure*:
  assumes *wf-closed g*
  assumes $g \oplus e \rightsquigarrow (g',\ n)$
  shows *wf-closed g'*
  using *assms(2,1)* **unfolding** *wf-closed-def*
  proof (*induction g e* (g', n) *arbitrary*: g' n)
    case (*ConstantNodeSame g c n*)
    then show *?case*
      by *blast*

```
  next
    case (ConstantNodeNew g c n g')
    then have dom: ids g' = ids g ∪ {n}
      by (meson IRNode.distinct(683) add-node-ids-subset ids-add-update)
    have k: kind g' n = ConstantNode c
      using ConstantNodeNew add-node-lookup by simp
    then have inp: {} = inputs g' n
      unfolding inputs.simps by simp
    from k have suc: {} = succ g' n
      unfolding succ.simps by simp
    have inputs g' n ⊆ ids g' ∧ succ g' n ⊆ ids g' ∧ kind g' n ≠ NoNode
      using inp suc k by simp
    then show ?case
    by (smt (verit) ConstantNodeNew.hyps(3) ConstantNodeNew.prems Un-insert-right
add-changed changeonly.elims(2) dom inputs.simps insert-iff singleton-iff subset-insertI
subset-trans succ.simps sup-bot-right)
  next
    case (ParameterNodeSame g i s n)
    then show ?case by blast
  next
    case (ParameterNodeNew g i s n g')
    then have dom: ids g' = ids g ∪ {n}
      using IRNode.distinct(2447) fresh-ids ids-add-update by presburger
    have k: kind g' n = ParameterNode i
      using ParameterNodeNew add-node-lookup by simp
    then have inp: {} = inputs g' n
      unfolding inputs.simps by simp
    from k have suc: {} = succ g' n
      unfolding succ.simps by simp
    have inputs g' n ⊆ ids g' ∧ succ g' n ⊆ ids g' ∧ kind g' n ≠ NoNode
      using k inp suc by simp
    then show ?case
    by (smt (verit) ParameterNodeNew.hyps(3) ParameterNodeNew.prems Un-insert-right
add-node-as-set dom inputs.elims insertE not-excluded-keep-type order-trans single-
tonD subset-insertI succ.elims sup-bot-right)
  next
    case (ConditionalNodeSame g ce g2 c te g3 t fe g4 f s' n)
    then show ?case by blast
  next
    case (ConditionalNodeNew g ce g2 c te g3 t fe g4 f s' n g')
    then have dom: ids g' = ids g4 ∪ {n}
      by (meson IRNode.distinct(591) add-node-ids-subset ids-add-update)
    have k: kind g' n = ConditionalNode c t f
      using ConditionalNodeNew add-node-lookup by simp
    then have inp: {c, t, f} = inputs g' n
      unfolding inputs.simps by simp
    from k have suc: {} = succ g' n
      unfolding succ.simps by simp
    have inputs g' n ⊆ ids g' ∧ succ g' n ⊆ ids g' ∧ kind g' n ≠ NoNode
```

```
      using k inp suc unrep-contains unrep-preserves-contains
      using ConditionalNodeNew(1,3,5,10)
        by (smt (verit) IRNode.simps(643) Un-insert-right bot.extremum dom in-
sert-absorb insert-subset subset-insertI sup-bot-right)
    then show ?case using dom
     by (smt (z3) ConditionalNodeNew.hyps(10) ConditionalNodeNew.hyps(2) Con-
ditionalNodeNew.hyps(4) ConditionalNodeNew.hyps(6) ConditionalNodeNew.prems
Diff-eq-empty-iff Diff-iff Un-insert-right Un-upper1 add-node-def inputs.simps in-
sertE replace-node-def replace-node-unchanged subset-trans succ.simps sup-bot-right)
  next
    case (UnaryNodeSame g xe g2 x s' op n)
    then show ?case by blast
  next
    case (UnaryNodeNew g xe g2 x s' op n g')
    then have dom: ids g' = ids g2 ∪ {n}
       by (metis add-node-ids-subset add-node-lookup ids-add-update ids-some un-
rep.UnaryNodeNew unrep-contains)
    have k: kind g' n = unary-node op x
      using UnaryNodeNew add-node-lookup
      by (metis fresh-ids ids-some)
    then have inp: {x} = inputs g' n
      using unary-inputs by simp
    from k have suc: {} = succ g' n
      using unary-succ by simp
    have inputs g' n ⊆ ids g' ∧ succ g' n ⊆ ids g' ∧ kind g' n ≠ NoNode
      using k inp suc unrep-contains unrep-preserves-contains
      using UnaryNodeNew(1,6)
         by (metis Un-upper1 dom empty-subsetI ids-some insert-not-empty in-
sert-subsetI not-in-g-inputs subset-iff)
    then show ?case
    by (smt (verit) Un-insert-right UnaryNodeNew.hyps(2) UnaryNodeNew.hyps(6)
UnaryNodeNew.prems add-changed changeonly.elims(2) dom inputs.simps insert-iff
singleton-iff subset-insertI subset-trans succ.simps sup-bot-right)
  next
    case (BinaryNodeSame g xe g2 x ye g3 y s' op n)
    then show ?case by blast
  next
    case (BinaryNodeNew g xe g2 x ye g3 y s' op n g')
    then have dom: ids g' = ids g3 ∪ {n}
       by (metis binary-inputs fresh-ids ids-add-update ids-some insert-not-empty
not-in-g-inputs)
    have k: kind g' n = bin-node op x y
      using BinaryNodeNew add-node-lookup
      by (metis fresh-ids ids-some)
    then have inp: {x, y} = inputs g' n
      using binary-inputs by simp
    from k have suc: {} = succ g' n
      using binary-succ by simp
    have inputs g' n ⊆ ids g' ∧ succ g' n ⊆ ids g' ∧ kind g' n ≠ NoNode
```

```
      using k inp suc unrep-contains unrep-preserves-contains
      using BinaryNodeNew(1,3,6)
         by (metis Un-upper1 dom empty-subsetI ids-some insert-not-empty in-
sert-subsetI not-in-g-inputs subset-iff)
    then show ?case using dom BinaryNodeNew
      by (smt (verit, del-insts) Diff-eq-empty-iff Diff-iff Un-insert-right Un-upper1
add-node-def inputs.simps insertE replace-node-def replace-node-unchanged subset-trans
succ.simps sup-bot-right)
  next
    case (AllLeafNodes g n s)
    then show ?case
      by blast
  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
```

7 Control-flow Semantics

```
theory IRStepObj
  imports
    TreeToGraph
begin
```

7.1 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 ⇒ 'b ⇒ Value
type-synonym Free = nat
type-synonym ('a, 'b) DynamicHeap = ('a, 'b) Heap × Free

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

fun h-store-field :: 'a ⇒ 'b ⇒ 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 :: ('a, 'b) DynamicHeap ⇒ ('a, 'b) DynamicHeap × Value
where
  h-new-inst (h, n) = ((h,n+1), (ObjRef (Some n)))

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

```
definition new-heap :: ('a, 'b) DynamicHeap where
  new-heap =  ((λf. λp. UndefVal), 0)
```

7.2 Intraprocedural Semantics

```
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)

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

fun input-index :: IRGraph ⇒ ID ⇒ ID ⇒ nat where
  input-index g n n' = find-index n' (inputs-of (kind g n))

fun phi-inputs :: IRGraph ⇒ nat ⇒ ID list ⇒ ID list where
  phi-inputs g i nodes = (map (λn. (inputs-of (kind g n))!(i + 1)) nodes)

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

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) |

  IfNode:
  ⟦kind g nid = (IfNode cond tb fb);
    g ⊢ cond ≃ condE;
    [m, p] ⊢ condE ↦ 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);

    i = find-index nid (inputs-of (kind g merge));
    phis = (phi-list g merge);
    inps = (phi-inputs g i phis);
    g ⊢ inps ≃_L inpsE;
    [m, p] ⊢ inpsE ↦_L vs;

    m′ = set-phis phis vs m⟧
    ⟹ g, p ⊢ (nid, m, h) → (merge, m′, h) |


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

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

  SignedDivNode:
    ⟦kind g nid = (SignedDivNode nid x y zero sb nxt);
      g ⊢ x ≃ xe;
      g ⊢ y ≃ ye;
      [m, p] ⊢ xe ↦ v1;
      [m, p] ⊢ ye ↦ v2;
      v = (intval-div v1 v2);
      m' =  m(nid := v)⟧
    ⟹ g, p ⊢ (nid, m, h) → (nxt, m', h) |

  SignedRemNode:
    ⟦kind g nid = (SignedRemNode nid x y zero sb nxt);
      g ⊢ x ≃ xe;
      g ⊢ y ≃ ye;
      [m, p] ⊢ xe ↦ v1;
      [m, p] ⊢ ye ↦ v2;
      v = (intval-mod v1 v2);
      m' =  m(nid := v)⟧
    ⟹ g, p ⊢ (nid, m, h) → (nxt, m', h) |

  StaticLoadFieldNode:
    ⟦kind g nid = (LoadFieldNode nid f None nid');
      h-load-field f None h = v;
      m' =  m(nid := v)⟧
    ⟹ g, p ⊢ (nid, m, h) → (nid', m', h) |

  StoreFieldNode:
    ⟦kind g nid = (StoreFieldNode nid f newval - (Some obj) nid');
      g ⊢ newval ≃ newvalE;
      g ⊢ obj ≃ objE;
      [m, p] ⊢ newvalE ↦ val;
      [m, p] ⊢ objE ↦ ObjRef ref;
      h' = h-store-field f ref val h;
      m' =  m(nid := val)⟧
    ⟹ g, p ⊢ (nid, m, h) → (nid', m', h') |

  StaticStoreFieldNode:
    ⟦kind g nid = (StoreFieldNode nid f newval - None nid');
      g ⊢ newval ≃ newvalE;
      [m, p] ⊢ newvalE ↦ val;
      h' = h-store-field f None val h;
      m' =  m(nid := val)⟧
    ⟹ g, p ⊢ (nid, m, h) → (nid', m', h')
```

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

7.3 Interprocedural Semantics

type-synonym *Signature* = *string*
type-synonym *Program* = *Signature* ⇀ *IRGraph*

inductive *step-top* :: *Program* ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap* ⇒ (*IRGraph* × *ID* × *MapState* × *Params*) *list* × *FieldRefHeap* ⇒

```
bool
  (- ⊢ - ⟶ - 55)
  for P where

  Lift:
  ⟦g, p ⊢ (nid, m, h) → (nid', m', h')⟧
    ⟹ P ⊢ ((g,nid,m,p)#stk, h) ⟶ ((g,nid',m',p)#stk, h') |

  InvokeNodeStep:
  ⟦is-Invoke (kind g nid);

    callTarget = ir-callTarget (kind g nid);
    kind g callTarget = (MethodCallTargetNode targetMethod arguments);
    Some targetGraph = P targetMethod;
    m' = new-map-state;
    g ⊢ arguments ≃L argsE;
    [m, p] ⊢ argsE ↦L p'⟧
    ⟹ P ⊢ ((g,nid,m,p)#stk, h) ⟶ ((targetGraph,0,m',p')#(g,nid,m,p)#stk, h)
|

  ReturnNode:
  ⟦kind g nid = (ReturnNode (Some expr) -);
    g ⊢ expr ≃ e;
    [m, p] ⊢ e ↦ v;

    cm' = cm(cnid := v);
    cnid' = (successors-of (kind cg cnid))!0⟧
   ⟹ P ⊢ ((g,nid,m,p)#(cg,cnid,cm,cp)#stk, h) ⟶ ((cg,cnid',cm',cp)#stk, h) |

  ReturnNodeVoid:
  ⟦kind g nid = (ReturnNode None -);
    cm' = cm(cnid := (ObjRef (Some (2048))));

    cnid' = (successors-of (kind cg cnid))!0⟧
   ⟹ P ⊢ ((g,nid,m,p)#(cg,cnid,cm,cp)#stk, h) ⟶ ((cg,cnid',cm',cp)#stk, h) |

  UnwindNode:
  ⟦kind g nid = (UnwindNode exception);

    g ⊢ exception ≃ exceptionE;
    [m, p] ⊢ exceptionE ↦ e;

    kind cg cnid = (InvokeWithExceptionNode - - - - - - exEdge);

    cm' = cm(cnid := e)⟧
  ⟹ P ⊢ ((g,nid,m,p)#(cg,cnid,cm,cp)#stk, h) ⟶ ((cg,exEdge,cm',cp)#stk, h)

code-pred (modes: i ⇒ i ⇒ o ⇒ bool) step-top .
```

7.4 Big-step Execution

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

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

inductive *exec* :: *Program*
      ⇒ (*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* :: *Program*
     ⇒ (*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* **.**

7.4.1 Heap Testing

definition *p3*:: *Params* **where**
  p3 = [*IntVal 32 3*]

values {(*prod.fst*(*prod.snd* (*prod.snd* (*hd* (*prod.fst res*))))) *0*
  | *res.* (λ*x* . *Some 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.* (λ*x. Some 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 True*),
    (*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.*
  (λ*x. Some eg4-sq*) ⊢ ([(*eg4-sq*, *0*, *new-map-state*, *p3*), (*eg4-sq*, *0*, *new-map-state*, *p3*)], *new-heap*) →*3* *res*}

end

7.5 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.

7.5.1 Control-flow Step is Deterministic

```
theorem stepDet:
  (g, p ⊢ (nid,m,h) → next) ⟹
  (∀ next'. ((g, p ⊢ (nid,m,h) → next') ⟶ next = next'))
proof (induction rule: step.induct)
  case (SequentialNode nid next m h)
  have notif: ¬(is-IfNode (kind g nid))
    using SequentialNode.hyps(1) is-sequential-node.simps
    by (metis is-IfNode-def)
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using SequentialNode.hyps(1) is-sequential-node.simps
   by (metis is-AbstractEndNode.simps is-EndNode.elims(2) is-LoopEndNode-def)
  have notnew: ¬(is-NewInstanceNode (kind g nid))
    using SequentialNode.hyps(1) is-sequential-node.simps
    by (metis is-NewInstanceNode-def)
  have notload: ¬(is-LoadFieldNode (kind g nid))
    using SequentialNode.hyps(1) is-sequential-node.simps
    by (metis is-LoadFieldNode-def)
  have notstore: ¬(is-StoreFieldNode (kind g nid))
    using SequentialNode.hyps(1) is-sequential-node.simps
    by (metis is-StoreFieldNode-def)
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
      using SequentialNode.hyps(1) is-sequential-node.simps is-SignedDivNode-def
is-SignedRemNode-def
    by (metis is-IntegerDivRemNode.simps)
  from notif notend notnew notload notstore notdivrem
  show ?case using SequentialNode step.cases
   by (smt (z3) IRNode.disc(1028) IRNode.disc(2270) IRNode.discI(31) Pair-inject
is-sequential-node.simps(18) is-sequential-node.simps(43) is-sequential-node.simps(44))
next
  case (IfNode nid cond tb fb m val next h)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: IfNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: IfNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: IfNode.hyps(1))
  from notseq notend notdivrem show ?case using IfNode repDet evalDet IRN-
```

```
ode.distinct IRNode.inject(11) Pair-inject step.simps
    by (smt (z3) IRNode.distinct IRNode.inject(12) Pair-inject step.simps)
next
  case (EndNodes nid merge i phis inputs m vs m' h)
  have notseq: ¬(is-sequential-node (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps is-sequential-node.simps
    by (metis is-EndNode.elims(2) is-LoopEndNode-def)
  have notif: ¬(is-IfNode (kind g nid))
    using EndNodes.hyps(1) is-IfNode-def is-AbstractEndNode.elims
    by (metis IRNode.distinct-disc(1058) is-EndNode.simps(12))
  have notref: ¬(is-RefNode (kind g nid))
    using EndNodes.hyps(1) is-sequential-node.simps
      using IRNode.disc(1899) IRNode.distinct(1473) is-AbstractEndNode.simps
is-EndNode.elims(2) is-LoopEndNode-def is-RefNode-def
    by metis
  have notnew: ¬(is-NewInstanceNode (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps
   using IRNode.distinct-disc(1442) is-EndNode.simps(29) is-NewInstanceNode-def
    by (metis IRNode.distinct-disc(1901) is-EndNode.simps(32))
  have notload: ¬(is-LoadFieldNode (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps
    using is-LoadFieldNode-def
    by (metis IRNode.distinct-disc(1706) is-EndNode.simps(21))
  have notstore: ¬(is-StoreFieldNode (kind g nid))
    using EndNodes.hyps(1) is-AbstractEndNode.simps is-StoreFieldNode-def
    by (metis IRNode.distinct-disc(1926) is-EndNode.simps(44))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
   using EndNodes.hyps(1) is-AbstractEndNode.simps is-SignedDivNode-def is-SignedRemNode-def
   using IRNode.distinct-disc(1498) IRNode.distinct-disc(1500) is-IntegerDivRemNode.simps
is-EndNode.simps(36) is-EndNode.simps(37)
    by auto
  from notseq notif notref notnew notload notstore notdivrem
  show ?case using EndNodes repAllDet evalAllDet
   by (smt (z3) is-IfNode-def is-LoadFieldNode-def is-NewInstanceNode-def is-RefNode-def
is-StoreFieldNode-def is-SignedDivNode-def is-SignedRemNode-def Pair-inject is-IntegerDivRemNode.elims(3)
step.cases)
next
  case (NewInstanceNode nid f obj nxt h' ref h m' m)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notif: ¬(is-IfNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notref: ¬(is-RefNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notload: ¬(is-LoadFieldNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notstore: ¬(is-StoreFieldNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  have notdivrem:  ¬(is-IntegerDivRemNode (kind g nid))
    using is-AbstractMergeNode.simps
    by (simp add: NewInstanceNode.hyps(1))
  from notseq notend notif notref notload notstore notdivrem
  show ?case using NewInstanceNode step.cases
     by (smt (z3) IRNode.disc(1028) IRNode.disc(2270) IRNode.discI(11) IRN-
ode.distinct(2311) IRNode.distinct(2313) IRNode.inject(31) Pair-inject)
next
  case (LoadFieldNode nid f obj nxt m ref h v m')
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  have notdivrem:  ¬(is-IntegerDivRemNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: LoadFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using LoadFieldNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject Value.inject(2)
option.distinct(1) option.inject)
next
  case (StaticLoadFieldNode nid f nxt h v m' m)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StaticLoadFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StaticLoadFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StaticLoadFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StaticLoadFieldNode step.cases
   by (smt (z3) IRNode.distinct(1051) IRNode.distinct(1721) IRNode.distinct(1739)
IRNode.distinct(1741) IRNode.distinct(1745) IRNode.inject(20) Pair-inject option.distinct(1))
next
  case (StoreFieldNode nid f newval uu obj nxt m val ref h' h m')
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StoreFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StoreFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StoreFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StoreFieldNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject Value.inject(2)
option.distinct(1) option.inject)
next
  case (StaticStoreFieldNode nid f newval uv nxt m val h' h m')
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: StaticStoreFieldNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: StaticStoreFieldNode.hyps(1))
  have notdivrem: ¬(is-IntegerDivRemNode (kind g nid))
    by (simp add: StaticStoreFieldNode.hyps(1))
  from notseq notend notdivrem
  show ?case using StoreFieldNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1097) IRNode.distinct(1745) IRNode.distinct(2317)
IRNode.distinct(2605) IRNode.distinct(2627) IRNode.inject(43) Pair-inject Static-
StoreFieldNode.hyps(1) StaticStoreFieldNode.hyps(2) StaticStoreFieldNode.hyps(3)
StaticStoreFieldNode.hyps(4) StaticStoreFieldNode.hyps(5) option.distinct(1))
next
  case (SignedDivNode nid x y zero sb nxt m v1 v2 v m' h)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: SignedDivNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: SignedDivNode.hyps(1))
  from notseq notend
  show ?case using SignedDivNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1091) IRNode.distinct(1739) IRNode.distinct(2311)
IRNode.distinct(2601) IRNode.distinct(2605) IRNode.inject(40) Pair-inject)
next
  case (SignedRemNode nid x y zero sb nxt m v1 v2 v m' h)
  then have notseq: ¬(is-sequential-node (kind g nid))
    using is-sequential-node.simps is-AbstractMergeNode.simps
    by (simp add: SignedRemNode.hyps(1))
  have notend: ¬(is-AbstractEndNode (kind g nid))
    using is-AbstractEndNode.simps
    by (simp add: SignedRemNode.hyps(1))
  from notseq notend
  show ?case using SignedRemNode step.cases repDet evalDet
   by (smt (z3) IRNode.distinct(1093) IRNode.distinct(1741) IRNode.distinct(2313)
```

IRNode.distinct(*2601*) *IRNode.distinct*(*2627*) *IRNode.inject*(*41*) *Pair-inject*)
qed

lemma *stepRefNode*:
  ⟦*kind g nid* = *RefNode nid'*⟧ ⟹ *g*, *p* ⊢ (*nid*,*m*,*h*) → (*nid'*,*m*,*h*)
  using *SequentialNode*
 by (*metis IRNodes.successors-of-RefNode is-sequential-node.simps*(*7*) *nth-Cons-0*)

lemma *IfNodeStepCases*:
  assumes *kind g nid* = *IfNode cond tb fb*
  assumes *g* ⊢ *cond* ≃ *condE*
  assumes [*m*, *p*] ⊢ *condE* ↦ *v*
  assumes *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid'*, *m*, *h*)
  shows *nid'* ∈ {*tb*, *fb*}
  using *step.IfNode repDet stepDet assms*
  by (*metis insert-iff old.prod.inject*)

lemma *IfNodeSeq*:
  shows *kind g nid* = *IfNode cond tb fb* ⟶ ¬(*is-sequential-node* (*kind g nid*))
  unfolding *is-sequential-node.simps*
  using *is-sequential-node.simps*(*18*) **by** *presburger*

lemma *IfNodeCond*:
  assumes *kind g nid* = *IfNode cond tb fb*
  assumes *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid'*, *m*, *h*)
  shows ∃ *condE v*. ((*g* ⊢ *cond* ≃ *condE*) ∧ ([*m*, *p*] ⊢ *condE* ↦ *v*))
  using *assms*(*2*,*1*) **by** (*induct* (*nid*,*m*,*h*) (*nid'*,*m*,*h*) *rule*: *step.induct*; *auto*)

lemma *step-in-ids*:
  assumes *g*, *p* ⊢ (*nid*, *m*, *h*) → (*nid'*, *m'*, *h'*)
  shows *nid* ∈ *ids g*
  using *assms* **apply** (*induct* (*nid*, *m*, *h*) (*nid'*, *m'*, *h'*) *rule*: *step.induct*)
  using *is-sequential-node.simps*(*45*) *not-in-g*
  apply *simp*
  apply (*metis is-sequential-node.simps*(*53*))
  using *ids-some*
  using *IRNode.distinct*(*1113*) **apply** *presburger*
  using *EndNodes*(*1*) *is-AbstractEndNode.simps is-EndNode.simps*(*45*) *ids-some*
  apply (*metis IRNode.disc*(*1218*) *is-EndNode.simps*(*52*))
  by *simp*+

end

7.6 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*
  using *assms*
  by (*metis*(*full-types*) *eq-imp-le signed-take-bit-take-bit*)

lemma *unwrap-signed-take-bit*:
  fixes *v* :: *int64*
  assumes $0 < b \wedge b \leq 64$
  assumes *signed-take-bit* (*b* − *1*) *v = v*
  shows *signed-take-bit 63* (*Word.rep* (*signed-take-bit* (*b* − *Suc 0*) *v*)) = *sint v*
  using *assms* **using** *size64* **unfolding** *signed-def* **by** *auto*

lemma *unrestricted-new-int-always-valid* [*simp*]:
  assumes $0 < b \wedge b \leq 64$
  shows *valid-value* (*new-int b v*) (*unrestricted-stamp* (*IntegerStamp b lo hi*))
  unfolding *unrestricted-stamp.simps new-int.simps valid-value.simps*
   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* ⟹ *unary-eval op val = UndefVal*
  by (*cases op*; *auto*)

lemma *unrestricted-stamp-valid*:
  assumes *s = unrestricted-stamp* (*IntegerStamp b lo hi*)
  assumes $0 < b \wedge b \leq 64$
  shows *valid-stamp s*
  using *assms*
   by (*smt* (*z3*) *Stamp.inject*(*1*) *bit-bounds.simps not-exp-less-eq-0-int prod.sel*(*1*) *prod.sel*(*2*) *unrestricted-stamp.simps*(*2*) *upper-bounds-equiv valid-stamp.elims*(*1*))

lemma *unrestricted-stamp-valid-value* [*simp*]:
  assumes *1*: *result = IntVal b ival*
  assumes *take-bit b ival = ival*
  assumes $0 < b \wedge b \leq 64$
  shows *valid-value result* (*unrestricted-stamp* (*IntegerStamp b lo hi*))
proof −
  have *valid-stamp* (*unrestricted-stamp* (*IntegerStamp b lo hi*))
    using *assms unrestricted-stamp-valid* **by** *blast*
  then show *?thesis*
    unfolding *1 unrestricted-stamp.simps valid-value.simps*
    using *assms int-signed-value-bounds* **by** *presburger*
qed

7.6.1 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*
  by (*smt* (*z3*) *Value.distinct*(*7*) *Value.inject*(*1*) *valid-value.elims*(*1*))

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 power-increasing signed-take-bit-int-less-exp-word sint-lt)

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*
by (*smt* (*verit*) *diff-le-self int-signed-value.simps linorder-not-less power-increasing-iff signed-take-bit-int-greater-eq-minus-exp-word sint-greater-eq*)

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** *force*
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** *force*
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*))

7.6.2 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 *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 assms*(*1*) *assms*(*2*) *assms*(*4*) *assms*(*5*) *evaltree-not-undef unary-obj valid-value.simps*(*11*))
  then obtain *b2 v2* **where** *v2*: *result* = *IntVal b2 v2*
   using *assms*(*2*) *assms*(*4*) *is-IntVal-def unary-eval-int* **by** *presburger*
  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*) *v2* **by** *auto*
  obtain *b′ lo′ hi′* **where** *stamp-expr expr* = *IntegerStamp b′ lo′ hi′*
   by (*metis assms*(*5*) *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 *stamp-unary.simps*(*1*) **by** *presburger*
  then obtain *lo2 hi2* **where** *s*: (*stamp-expr* (*UnaryExpr op expr*)) = *unrestricted-stamp* (*IntegerStamp b2 lo2 hi2*)
   unfolding *stamp-expr.simps*
   using *vtmp op*
  by (*smt* (*verit*, *best*) *Value.inject*(*1*) ‹(*result*:: *Value*) = *unary-eval* (*op*::*IRUnaryOp*) (*IntVal* (*b1*::*nat*) (*v1*::*64 word*))› ‹*stamp-expr* (*expr*::*IRExpr*) = *IntegerStamp* (*b′*::*nat*) (*lo′*::*int*) (*hi′*::*int*)› *assms*(*5*) *insertE intval-abs.simps*(*1*) *intval-logic-negation.simps*(*1*) *intval-negate.simps*(*1*) *intval-not.simps*(*1*) *new-int.elims singleton-iff unary-eval.simps*(*1*) *unary-eval.simps*(*2*) *unary-eval.simps*(*3*) *unary-eval.simps*(*4*) *v1 valid-int-same-bits*)
  then have *0* < *b1* ∧ *b1* ≤ *64*
   using *valid-int-gives*
   by (*metis assms*(*5*) *v1 valid-stamp.simps*(*1*))
  then have *fst* (*bit-bounds b2*) ≤ *int-signed-value b2 v2* ∧
      int-signed-value b2 v2 ≤ *snd* (*bit-bounds b2*)
  by (*smt* (*verit*, *del-insts*) *Stamp.inject*(*1*) *assms*(*3*) *assms*(*5*) *int-signed-value-bounds s stamp-expr.simps*(*1*) *stamp-unary.simps*(*1*) *unrestricted-stamp.simps*(*2*) *v1 valid-int-gives*)

```
  then show ?thesis
    unfolding s v2 unrestricted-stamp.simps valid-value.simps
     by (smt (z3) assms(3) assms(5) is-stamp-empty.simps(1) new-int-take-bits s
stamp-expr.simps(1) stamp-unary.simps(1) unrestricted-stamp.simps(2) v1 v2 valid-int-gives
valid-stamp.simps(1) vtmp)
qed

lemma narrow-widen-output-bits:
  assumes unary-eval op val ≠ UndefVal
  assumes op ∉ normal-unary
  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) 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
  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 assms(1) assms(2) assms(4) assms(5) evaltree-not-undef
unary-obj valid-value.simps(11))
  then have result = unary-eval op (IntVal b1 v1)
    using assms(2) v1 by blast
  then obtain v2 where v2: result = new-int (ir-resultBits op) v2
    using assms by (cases op; simp; (meson new-int.simps)+)
  then obtain v3 where v3: result = IntVal (ir-resultBits op) v3
    using assms by (cases op; simp; (meson new-int.simps)+)
   then obtain lo2 hi2 where s: (stamp-expr (UnaryExpr op expr)) = unre-
stricted-stamp (IntegerStamp (ir-resultBits op) lo2 hi2)
```

```
    unfolding stamp-expr.simps stamp-unary.simps
    using assms(3) assms(5) v1 valid-int-gives by fastforce
  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 int-signed-value-bounds
   by (smt (verit, del-insts) Stamp.inject(1) assms(3) assms(5) int-signed-value-bounds
s stamp-expr.simps(1) stamp-unary.simps(1) unrestricted-stamp.simps(2) v1 valid-int-gives)
  then show ?thesis
    unfolding s v3 unrestricted-stamp.simps valid-value.simps
    using outBits v2 v3 by auto
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 by (metis assms eval-normal-unary-implies-valid-value)
  next
    case False
   then show ?thesis by (metis assms eval-widen-narrow-unary-implies-valid-value)
  qed
```

7.6.3 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*
  using *assms*
  apply (*cases op*; *simp*)
  using *assms bool-to-val.simps bin-eval-new-int new-int.simps bin-eval-unused-bits-zero*
  by *metis*+

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*)
  using *assms* **apply** (*metis* (*mono-tags*))+
  using *take-bit-and* **apply** *metis*
  using *take-bit-or* **apply** *metis*
  using *take-bit-xor* **by** *metis*

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 *presburger*+

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*) *assms*(*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*) *assms*(*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 blast
  then obtain bres vtmp where vtmp: result = new-int bres vtmp
    using assms bin-eval-bits-binary-shift-ops
    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 s1 s2 stamp-binary.simps stamp-expr.simps
        using assms bin-eval-bits-binary-shift-ops
        by (metis Value.inject(1) eval-bits-1-64 new-int.simps r v1 vres)
    next
      case False
      then have op ∉ binary-shift-ops
        by simp
      then have beq: b1 = b2
        using v1 v2 assms bin-eval-input-bits-equal by simp
      then show ?thesis
      proof (cases op ∈ binary-fixed-32-ops)
        case True
        then show ?thesis
        unfolding s1 s2 stamp-binary.simps stamp-expr.simps
        using assms bin-eval-bits-fixed-32-ops
          by (metis False Value.inject(1) beq bin-eval-new-int le-add-same-cancel1
new-int.simps numeral-Bit0 vres zero-le-numeral zero-less-numeral)
      next
        case False
        then show ?thesis
        unfolding s1 s2 stamp-binary.simps stamp-expr.simps
        using assms
     by (metis beq bin-eval-new-int eval-bits-1-64 intval-bits.simps unrestricted-new-int-always-valid
unrestricted-stamp.simps(2) v1 valid-int-same-bits vres)
    qed
  qed
  then show ?thesis
    unfolding vres
    using unrestricted-new-int-always-valid vres vtmp by presburger
qed
```

7.6.4 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 unfolding is-IntegerStamp-def valid-stamp.simps meet.simps
 by (smt (verit, del-insts) meet.simps(2) valid-stamp.simps(1) valid-stamp.simps(8))

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)
    using assms by (metis 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
    using assms by (metis 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))
    using assms v stamp-meet-is-valid-stamp
    by (metis meet.simps(2))
  then show ?thesis
    unfolding m val valid-value.simps
    using mm v 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*)
using *assms stamp-meet-commutes stamp-meet-is-valid-value1*
by *metis*

7.6.5 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*
by (*metis Stamp.distinct*(*13*) *Stamp.distinct*(*25*) *compatible.elims*(*2*) *meet.simps*(*1*) *meet.simps*(*2*))
then have *valid-stamp* (*meet* (*stamp-expr expr1*) (*stamp-expr expr2*))
using *assms*
by (*smt* (*verit*, *best*) *compatible.elims*(*2*) *stamp-meet-is-valid-stamp valid-stamp.simps*(*2*))

then show *?thesis* **using** *stamp-meet-is-valid-value*
using *assms def*
by (*smt* (*verit*, *best*) *compatible.elims*(*2*) *never-void stamp-expr.simps*(*6*) *stamp-meet-commutes*)

qed

7.6.6 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*)
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*

```
    using assms(1)
    by (metis stamp-under.elims(2))
  then obtain lo hy where ystamp: stamp-expr y = IntegerStamp b lo hy
    using assms(1)
    by (metis Stamp.sel(1) stamp-under.elims(2))
  obtain xvv where xvv: xv = IntVal b xvv
    by (metis assms(2) assms(3) valid-int wf-stamp-def xstamp)
  then have xval: valid-value (IntVal b xvv) (stamp-expr x)
    using assms(2) assms(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 auto
  have xunder: int-signed-value b xvv ≤ hi
    using xvv xval valid-value.simps
    by (metis assms(2) assms(3) wf-stamp-def xstamp)
  have yunder: lo ≤ int-signed-value b yvv
    using yvv yval valid-value.simps
    by (metis ystamp)
  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 auto
  then have (intval-less-than xv yv) = IntVal 32 1
    using xvv yvv
    using intval-less-than.simps(1) unwrap
    using bool-to-val.simps(1) by presburger
  then show ?thesis
    by simp
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))
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
    using assms(1)
    by (metis stamp-under.elims(2))
  then obtain ly hi where ystamp: stamp-expr y = IntegerStamp b ly hi
    using assms(1)
    by (metis Stamp.sel(1) stamp-under.elims(2))
  obtain xvv where xvv: xv = IntVal b xvv
    by (metis assms(2) assms(3) valid-int wf-stamp-def xstamp)
  then have xval: valid-value (IntVal b xvv) (stamp-expr x)
    using assms(2) assms(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 auto
  have yunder: int-signed-value b yvv ≤ hi
    using yvv yval valid-value.simps
    by (metis ystamp)
  have xover: lo ≤ int-signed-value b xvv
    using xvv xval valid-value.simps
    by (metis assms(2) assms(3) wf-stamp-def xstamp)
  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 auto
  then have (intval-less-than xv yv) = IntVal 32 0
    using xvv yvv
    using intval-less-than.simps(1) unwrap
    by force
  then show ?thesis
    by simp
qed

end
```

8 Optization DSL

8.1 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 (- || -)

definition word :: ('a::len) word ⇒ 'a word where
```

word x = x

ML-file ‹*markup.ML*›

8.1.1 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.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 e_2) e_1 e_2

8.1.2 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.And = @{term intval-and}
  | markup DSL-Tokens.Or = @{term intval-or}
  | markup DSL-Tokens.ShortCircuitOr = @{term intval-short-circuit-or}
  | markup DSL-Tokens.Xor = @{term intval-xor}
  | markup DSL-Tokens.Abs = @{term intval-abs}
  | markup DSL-Tokens.Less = @{term intval-less-than}
  | markup DSL-Tokens.Equals = @{term intval-equals}
  | markup DSL-Tokens.Not = @{term intval-not}
  | markup DSL-Tokens.Negate = @{term intval-negate}
  | markup DSL-Tokens.LogicNegate = @{term intval-logic-negation}
  | markup DSL-Tokens.LeftShift = @{term intval-left-shift}
  | markup DSL-Tokens.RightShift = @{term intval-right-shift}
  | markup DSL-Tokens.UnsignedRightShift = @{term intval-uright-shift}
  | markup DSL-Tokens.Conditional = @{term intval-conditional}
  | markup DSL-Tokens.Constant = @{term IntVal 32}
  | markup DSL-Tokens.TrueConstant = @{term IntVal 32 1}
  | markup DSL-Tokens.FalseConstant = @{term IntVal 32 0}
end
structure IntValMarkup = DSL-Markup(IntValTranslator);
```
›

value expression translation

syntax *-expandIntVal* :: *term* ⇒ *term* (*val*[-])
parse-translation ‹ [(@{*syntax-const -expandIntVal*} , *IntValMarkup.markup-expr* [])] ›

value expression example

value *val*[$(e_1 < e_2)$ *?* e_1 : e_2]

intval-conditional (*intval-less-than* e_1 e_2) e_1 e_2

8.1.3 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.And* = @{*term Bit-Operations.semiring-bit-operations-class.and*}
| *markup DSL-Tokens.Or* = @{*term or*}
| *markup DSL-Tokens.Xor* = @{*term xor*}
| *markup DSL-Tokens.Abs* = @{*term abs*}
| *markup DSL-Tokens.Less* = @{*term less*}
| *markup DSL-Tokens.Equals* = @{*term HOL.eq*}
| *markup DSL-Tokens.Not* = @{*term not*}
| *markup DSL-Tokens.Negate* = @{*term uminus*}
| *markup DSL-Tokens.LogicNegate* = @{*term logic-negate*}
| *markup DSL-Tokens.LeftShift* = @{*term shiftl*}
| *markup DSL-Tokens.RightShift* = @{*term signed-shiftr*}
| *markup DSL-Tokens.UnsignedRightShift* = @{*term shiftr*}
| *markup DSL-Tokens.Constant* = @{*term word*}
| *markup DSL-Tokens.TrueConstant* = @{*term 1*}
| *markup DSL-Tokens.FalseConstant* = @{*term 0*}
end
structure WordMarkup = *DSL-Markup*(*WordTranslator*);
›

word expression translation

syntax *-expandWord* :: *term* ⇒ *term* (*bin*[-])
parse-translation ‹ [(@{*syntax-const -expandWord*} , *WordMarkup.markup-expr* [])] ›

word expression example

value *bin*[*x* & *y* | *z*]

intval-conditional (*intval-less-than* e_1 e_2) 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

8.2 Optimization Phases

theory *Phase*
  imports *Main*
begin

ML-file *map.ML*
ML-file *phase.ML*

end

8.3 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.theory-toplevel 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, 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.text >>
      (fn name => Toplevel.theory (fn state => export-phases state name)))
```

›

ML-file *rewrites.ML*

8.3.1 Semantic Preservation Obligation

fun *rewrite-preservation* :: *IRExpr Rewrite* ⇒ *bool* **where**
  rewrite-preservation (*Transform x y*) = ($y \leq x$) |
  rewrite-preservation (*Conditional x y cond*) = (*cond* ⟶ ($y \leq x$)) |
 rewrite-preservation (*Sequential x y*) = (*rewrite-preservation x* ∧ *rewrite-preservation y*) |
  rewrite-preservation (*Transitive x*) = *rewrite-preservation x*

8.3.2 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* ≠ *UndefVal* ∧ *y* ≠ *UndefVal* ⟶ *x* = *y*) |
  intval (*Conditional x y cond*) = (*cond* ⟶ (*x* = *y*)) |
  intval (*Sequential x y*) = (*intval x* ∧ *intval y*) |
  intval (*Transitive x*) = *intval x*

8.3.3 Standard Termination Measure

fun *size* :: *IRExpr* ⇒ *nat* **where**
  unary-size:
  size (*UnaryExpr op e*) = (*size e*) + *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 cond t f*) = (*size cond*) + (*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*

8.3.4 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);
›

end
```

9 Canonicalization Optimizations

```
theory Common
  imports
    OptimizationDSL.Canonicalization
    Semantics.IRTreeEvalThms
begin

lemma size-pos[size-simps]: 0 < size y
  apply (induction y; auto?)
```

by (*smt* (*z3*) *add-2-eq-Suc′ add-is-0 not-gr0 size.elims size.simps*(*12*) *size.simps*(*13*) *size.simps*(*14*) *size.simps*(*15*) *zero-neq-numeral zero-neq-one*)

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*(*11*) *size.simps*(*2*) *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*
by (*smt* (*verit*, *best*) *less-Suc-eq less-add-Suc2 less-add-same-cancel1 linorder-neqE-nat not-add-less1 order-less-trans pos2 size.simps*(*4*) *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*(*11*) *size.simps*(*4*) *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

9.1 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*
  using *assms intval-add-sym* **by** *simp*

optimization *AddShiftConstantRight*: ((*const v*) + *y*) ⟼ *y* + (*const v*) *when* ¬(*is-ConstantExpr y*)
  using *size-non-const*
 apply (*metis add-2-eq-Suc′ less-Suc-eq plus-1-eq-Suc size.simps*(*11*) *size-non-add*)

```
  unfolding le-expr-def
  apply (rule impI)
  subgoal premises 1
    apply (rule allI impI)+

    subgoal premises 2 for m p va
      apply (rule BinaryExprE[OF 2])
      subgoal premises 3 for x ya
        apply (rule BinaryExpr)
        using 3 apply simp
        using 3 apply simp
        using 3 binadd-commute apply auto
        done
      done
    done
  done
```

```
optimization AddShiftConstantRight2: ((const v) + y) ⟼ y + (const v) when
¬(is-ConstantExpr y)
  unfolding le-expr-def
   apply (auto simp: intval-add-sym)

  using size-non-const
  by (metis add-2-eq-Suc' lessI plus-1-eq-Suc size.simps(11) size-non-add)
```

```
lemma is-neutral-0 [simp]:
  assumes 1: intval-add (IntVal b x) (IntVal b 0) ≠ UndefVal
  shows intval-add (IntVal b x) (IntVal b 0) = (new-int b x)
  using 1 by auto

optimization AddNeutral: (e + (const (IntVal 32 0))) ⟼ e
  unfolding le-expr-def apply auto
  using is-neutral-0 eval-unused-bits-zero
  by (smt (verit) add-cancel-left-right intval-add.elims val-to-bool.simps(1))
```

```
ML-val ‹@{term ‹x = y›}›

lemma NeutralLeftSubVal:
  assumes e1 = new-int b ival
  shows val[(e1 − e2) + e2] ≈ e1
  apply simp using assms by (cases e1; cases e2; auto)
```

optimization *RedundantSubAdd*: $((e_1 - e_2) + e_2) \longmapsto e_1$
  apply *auto* **using** *eval-unused-bits-zero NeutralLeftSubVal*
  unfolding *well-formed-equal-defn*
  by (*smt* (*verit*) *evalDet intval-sub.elims new-int.elims*)

lemma *allE2*: $(\forall x\ y.\ P\ x\ y) \Longrightarrow (P\ a\ b \Longrightarrow R) \Longrightarrow R$
  by *simp*

lemma *just-goal2*:
  assumes *1*: $(\forall\ a\ b.$ (*intval-add* (*intval-sub* *a* *b*) *b* $\neq$ *UndefVal* $\wedge$ *a* $\neq$ *UndefVal*
$\longrightarrow$
    intval-add (*intval-sub* *a* *b*) *b* = *a*))
  shows (*BinaryExpr BinAdd* (*BinaryExpr BinSub* e_1 e_2) e_2) $\geq$ e_1
  unfolding *le-expr-def unfold-binary bin-eval.simps*
  by (*metis 1 evalDet evaltree-not-undef*)

optimization *RedundantSubAdd2*: $e_2 + (e_1 - e_2) \longmapsto e_1$
 apply (*metis add.commute add-less-cancel-right less-add-Suc2 plus-1-eq-Suc size-binary-const size-non-add trans-less-add2*)
   by (*smt* (*verit*, *del-insts*) *BinaryExpr BinaryExprE RedundantSubAdd*(*1*) *binadd-commute le-expr-def rewrite-preservation.simps*(*1*))

lemma *AddToSubHelperLowLevel*:
  shows *intval-add* (*intval-negate* *e*) *y* = *intval-sub* *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*] $\neq$ *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 $val[x + e] \neq UndefVal$
shows $val[x + (-e)] = val[x - e]$
using *assms* **by** (*cases x*; *cases e*; *auto*)

lemma *exp-add-left-negate-to-sub*:
$exp[-e + y] \geq exp[y - e]$
apply (*cases e*; *cases y*; *auto*)
using *AddToSubHelperLowLevel* **by** *auto+*

Optimisations

optimization *RedundantAddSub*: $(b + a) - b \longmapsto a$
apply *auto* **using** *val-redundant-add-sub eval-unused-bits-zero*
by (*smt* (*verit*) *evalDet intval-add.elims new-int.elims*)

optimization *AddRightNegateToSub*: $x + -e \longmapsto x - e$
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*: $-e + y \longmapsto y - e$
defer
using *exp-add-left-negate-to-sub* **apply** *blast*
by (*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*)

end

end

9.2 AndNode Phase

theory *AndPhase*
imports
Common
Proofs.StampEvalThms
begin

context *stamp-mask*
begin

lemma *AndRightFallthrough*: $(((and\ (not\ (\downarrow x))\ (\uparrow y)) = 0)) \longrightarrow exp[x\ \&\ y] \geq exp[y]$
apply *simp* **apply** (*rule impI*; (*rule allI*)+)

```
  apply (rule impI)
  subgoal premises p for m p v
  proof -
    obtain xv where xv: [m, p] ⊢ x ↦ xv
      using p(2) by blast
    obtain yv where yv: [m, p] ⊢ y ↦ yv
      using p(2) by blast
    have v = val[xv & yv]
      using p(2) xv yv
      by (metis BinaryExprE bin-eval.simps(4) evalDet)
    then have v = yv
      using p(1) not-down-up-mask-and-zero-implies-zero
    by (smt (verit) eval-unused-bits-zero intval-and.elims new-int.elims new-int-bin.elims
p(2) unfold-binary xv yv)
    then show ?thesis using yv by simp
  qed
  done

lemma AndLeftFallthrough: (((and (not (↓ y)) (↑ x)) = 0)) ⟶ exp[x & y] ≥
exp[x]
  apply simp apply (rule impI; (rule allI)+)
  apply (rule impI)
  subgoal premises p for m p v
  proof -
    obtain xv where xv: [m, p] ⊢ x ↦ xv
      using p(2) by blast
    obtain yv where yv: [m, p] ⊢ y ↦ yv
      using p(2) by blast
    have v = val[xv & yv]
      using p(2) xv yv
      by (metis BinaryExprE bin-eval.simps(4) evalDet)
    then have v = xv
      using p(1) not-down-up-mask-and-zero-implies-zero
    by (smt (verit) and.commute eval-unused-bits-zero intval-and.elims new-int.simps
new-int-bin.simps p(2) unfold-binary xv yv)
    then show ?thesis using xv by simp
  qed
  done
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
using *assms* **by** (*cases x*; *auto*)

lemma *val-and-nots*:
val[~x & ~y] = *val*[~(x | y)]
apply (*cases x*; *cases y*; *auto*) **by** (*simp add*: *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** (*cases x*; *auto*) **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*
using *assms* **by** (*cases x*; *auto*)

lemma *exp-and-equal*:
exp[x & x] ≥ *exp*[x]
apply *auto* **using** *val-and-equal eval-unused-bits-zero*
by (*smt* (*verit*) *evalDet intval-and.elims new-int.elims*)

lemma *exp-and-nots*:
exp[~x & ~y] ≥ *exp*[~(x | y)]
apply (*cases x*; *cases y*; *auto*) **using** *val-and-nots*
by *fastforce+*

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 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: ‹(va::Value) ≠ UndefVal›) defer
       subgoal premises p for x3
        proof -
          have va = ObjRef x3
            using p(1) by auto
             then have sint (signed-take-bit (b - Suc (0::nat)) (take-bit b e)) <
(2::int) ^ b div (2::int)
            by (simp add: 5)
          then show ?thesis
            using 2 intval-sign-extend.simps(3) p(1) by blast
        qed

       subgoal premises p for x4
        proof -
          have sg1: va = ObjStr x4
            using 2 p(1) by auto
             then have sint (signed-take-bit (b - Suc (0::nat)) (take-bit b e)) <
(2::int) ^ b div (2::int)
            by (simp add: 5)
          then show ?thesis
            using 1 sg1 by auto
        qed


        subgoal premises p for x21 x22
          proof -
```

```
              have sgg1: va = IntVal x21 x22
                by (simp add: p(1))
              then have sgg2: sint (signed-take-bit (b − Suc (0::nat)) (take-bit b e))
< (2::int) ^ b div (2::int)
                by (simp add: 5)
              then show ?thesis
                sorry
              qed
            done
      then show ?thesis
        by (metis evalDet p(2) va)
    qed
  done
```

```
lemma val-and-commute[simp]:
   val[x & y] = val[y & x]
   apply (cases x; cases y; auto)
  by (simp add: 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)
    defer using exp-and-nots
   apply presburger
 by (metis add-2-eq-Suc' less-SucI less-add-Suc1 not-less-eq size-binary-const size-non-add)


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)
   apply auto using val-and-neutral
```

by (*smt* (*verit*) *Value.sel*(*1*) *eval-unused-bits-zero intval-and.elims intval-word.simps*

new-int.simps new-int-bin.simps take-bit-eq-mask)

optimization *AndRightFallThrough*: $(x \ \& \ y) \longmapsto y$
when $(((and\ (not\ (IRExpr\text{-}down\ x))\ (IRExpr\text{-}up\ y)) = 0))$
by (*simp add*: *IRExpr-down-def IRExpr-up-def*)

optimization *AndLeftFallThrough*: $(x \ \& \ y) \longmapsto x$
when $(((and\ (not\ (IRExpr\text{-}down\ y))\ (IRExpr\text{-}up\ x)) = 0))$
by (*simp add*: *IRExpr-down-def IRExpr-up-def*)

end

end

9.3 Experimental AndNode Phase

theory *NewAnd*
imports
Common
Graph.Long
begin

lemma *bin-distribute-and-over-or*:
bin[*z* & (*x* | *y*)] = *bin*[(*z* & *x*) | (*z* & *y*)]
by (*smt* (*verit*, *best*) *bit-and-iff bit-eqI bit-or-iff*)

lemma *intval-distribute-and-over-or*:
val[*z* & (*x* | *y*)] = *val*[(*z* & *x*) | (*z* & *y*)]
apply (*cases x*; *cases y*; *cases z*; *auto*)
using *bin-distribute-and-over-or* **by** *blast*+

lemma *exp-distribute-and-over-or*:
exp[*z* & (*x* | *y*)] ≥ *exp*[(*z* & *x*) | (*z* & *y*)]
apply *simp* **using** *intval-distribute-and-over-or*
using *BinaryExpr bin-eval.simps*(*4*,*5*)
using *intval-or.simps*(*1*) **unfolding** *new-int-bin.simps new-int.simps* **apply** *auto*
by (*metis bin-eval.simps*(*4*) *bin-eval.simps*(*5*) *intval-or.simps*(*2*) *intval-or.simps*(*5*))

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[$x \oplus y$] = *val*[$y \oplus x$]
by (*cases x*; *cases y*; *auto simp*: *xor.commute*)

lemma *exp-and-commute*:
exp[x & z] $\geq$ *exp*[z & x]
apply *simp* **using** *intval-and-commute* **by** *auto*

lemma *exp-or-commute*:
exp[$x \mid y$] $\geq$ *exp*[$y \mid x$]
apply *simp* **using** *intval-or-commute* **by** *auto*

lemma *exp-xor-commute*:
exp[$x \oplus y$] $\geq$ *exp*[$y \oplus x$]
apply *simp* **using** *intval-xor-commute* **by** *auto*

lemma *bin-eliminate-y*:
assumes *bin*[y & z] = *0*
shows *bin*[($x \mid y$) & z] = *bin*[x & z]
using *assms*
by (*simp add*: *and.commute bin-distribute-and-over-or*)

lemma *intval-eliminate-y*:
assumes *val*[y & z] = *IntVal b 0*
shows *val*[($x \mid y$) & z] = *val*[x & z]
using *assms bin-eliminate-y* **by** (*cases x*; *cases y*; *cases z*; *auto*)

lemma *intval-and-associative*:
val[(x & y) & z] = *val*[x & (y & z)]
apply (*cases x*; *cases y*; *cases z*; *auto*)
by (*simp add*: *and.assoc*)+

lemma *intval-or-associative*:
val[($x \mid y$) $\mid z$] = *val*[$x \mid$ ($y \mid z$)]
apply (*cases x*; *cases y*; *cases z*; *auto*)
by (*simp add*: *or.assoc*)+

lemma *intval-xor-associative*:
val[($x \oplus y$) $\oplus z$] = *val*[$x \oplus$ ($y \oplus z$)]
apply (*cases x*; *cases y*; *cases z*; *auto*)
by (*simp add*: *xor.assoc*)+

lemma *exp-and-associative*:
exp[(x & y) & z] $\geq$ *exp*[x & (y & z)]
apply *simp* **using** *intval-and-associative* **by** *fastforce*

lemma *exp-or-associative*:
  $exp[(x \mid y) \mid z] \geq exp[x \mid (y \mid z)]$
  apply *simp* **using** *intval-or-associative* **by** *fastforce*

lemma *exp-xor-associative*:
  $exp[(x \oplus y) \oplus z] \geq exp[x \oplus (y \oplus z)]$
  apply *simp* **using** *intval-xor-associative* **by** *fastforce*

lemma *intval-and-absorb-or*:
  assumes $\exists\, b\ v\ .\ x = new\text{-}int\ b\ v$
  assumes $val[x\ \&\ (x \mid y)] \neq UndefVal$
  shows $val[x\ \&\ (x \mid y)] = val[x]$
  using *assms* **apply** (*cases x*; *cases y*; *auto*)
  by (*metis* (*mono-tags*, *lifting*) *intval-and.simps*(*5*))

lemma *intval-or-absorb-and*:
  assumes $\exists\, b\ v\ .\ x = new\text{-}int\ b\ v$
  assumes $val[x \mid (x\ \&\ y)] \neq UndefVal$
  shows $val[x \mid (x\ \&\ y)] = val[x]$
  using *assms* **apply** (*cases x*; *cases y*; *auto*)
  by (*metis* (*mono-tags*, *lifting*) *intval-or.simps*(*5*))

lemma *exp-and-absorb-or*:
  $exp[x\ \&\ (x \mid y)] \geq exp[x]$
  apply *auto* **using** *intval-and-absorb-or eval-unused-bits-zero*
  by (*smt* (*verit*) *evalDet intval-or.elims new-int.elims*)

lemma *exp-or-absorb-and*:
  $exp[x \mid (x\ \&\ y)] \geq exp[x]$
  apply *auto* **using** *intval-or-absorb-and eval-unused-bits-zero*
  by (*smt* (*verit*) *evalDet intval-or.elims new-int.elims*)

lemma
  assumes $y = 0$
  shows $x + y = or\ x\ y$
  using *assms*
  by *simp*

lemma *no-overlap-or*:
  assumes $and\ x\ y = 0$
  shows $x + y = or\ x\ y$
  using *assms*
  by (*metis bit-and-iff bit-xor-iff disjunctive-add xor-self-eq*)

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*)
  using *up-mask-and-zero-implies-zero*
  apply (*smt* (*verit*, *best*) *take-bit-and take-bit-of-0*)
  by *presburger*

lemma *exp-eliminate-y*:
  and (↑*y*) (↑*z*) = *0* ⟶ *BinaryExpr BinAnd* (*BinaryExpr BinOr x y*) *z* ≥ *BinaryExpr BinAnd 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*]
      using *xv yv zv*
        by (*smt* (*verit*, *best*) *BinaryExprE bin-eval.simps*(*4*) *bin-eval.simps*(*5*) *e evalDet*)
    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*
      using *intval-up-and-zero-implies-zero*
      by (*metis calculation e intval-or.simps*(*5*) *p unfold-binary yv zv*)
    ultimately have *rhs*: *v* = *val*[*xv* & *zv*]
      using *intval-eliminate-y lhs* **by** *force*
    from *lhs rhs* **show** *?thesis*
      by (*metis BinaryExpr BinaryExprE bin-eval.simps*(*4*) *e xv zv*)
  qed
  done
  done

lemma *leadingZeroBounds*:

fixes x :: *'a::len word*
assumes n = *numberOfLeadingZeros x*
shows $0 \leq n \wedge n \leq$ *Nat.size x*
using *assms* **unfolding** *numberOfLeadingZeros-def*
by (*simp add*: *MaxOrNeg-def highestOneBit-def nat-le-iff*)

lemma *above-nth-not-set*:
fixes x :: *int64*
assumes $n = 64 -$ *numberOfLeadingZeros x*
shows $j > n \longrightarrow \neg$(*bit x j*)
using *assms* **unfolding** *numberOfLeadingZeros-def*
by (*smt* (*verit*, *ccfv-SIG*) *highestOneBit-def int-nat-eq int-ops*(*6*) *less-imp-of-nat-less max-set-bit size64 zerosAboveHighestOne*)

no-notation *LogicNegationNotation* (!-)

lemma *zero-horner*:
horner-sum of-bool 2 (*map* (λx. *False*) *xs*) = *0*
apply (*induction xs*) **apply** *simp*
by *force*

lemma *zero-map*:
assumes $j \leq n$
assumes $\forall i.\ j \leq i \longrightarrow \neg(f\ i)$
shows *map f* [*0*..<*n*] = *map f* [*0*..<*j*] @ *map* (λx. *False*) [*j*..<*n*]
apply (*insert assms*)
by (*smt* (*verit*, *del-insts*) *add-diff-inverse-nat atLeastLessThan-iff bot-nat-0.extremum leD 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* (λx. *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 *horner-sum-append*
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*)
also have ... = *horner-sum of-bool 2* (*map f* [*0*..<*j*]) + *2* ^ *length* [*0*..<*j*] ∗ *horner-sum of-bool 2* (*map* (λx. *False*) [*j*..<*n*])
using *assms*
by (*metis calculation horner-sum-append length-map*)
also have ... = *horner-sum of-bool 2* (*map f* [*0*..<*j*])
using *zero-horner*
using *mult-not-zero* **by** *auto*
finally show *?thesis* **by** *simp*
qed

```
lemma split-horner:
  assumes j ≤ n
  assumes ∀ i. j ≤ i ⟶ ¬(f i)
  shows horner-sum of-bool (2::'a::len word) (map f [0..<n]) = horner-sum of-bool
2 (map f [0..<j])
  apply (rule map-join-horner)
  apply (rule zero-map)
  using assms by auto

lemma transfer-map:
  assumes ∀ i. i < n ⟶ f i = f' i
  shows (map f [0..<n]) = (map f' [0..<n])
  using assms by simp

lemma transfer-horner:
  assumes ∀ i. i < n ⟶ 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])
  using assms using transfer-map
  by (smt (verit, best))

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 ≤ 64
    using assms
    using diff-le-self by blast
  also have and v zv = horner-sum of-bool 2 (map (bit (and v zv)) [0..<64])
    using horner-sum-bit-eq-take-bit size64
    by (metis size-word.rep-eq take-bit-length-eq)
  also have ... = horner-sum of-bool 2 (map (λi. bit (and v zv) i) [0..<64])
    by blast
  also have ... = horner-sum of-bool 2 (map (λi. ((bit v i) ∧ (bit zv i))) [0..<64])
    using bit-and-iff by metis
  also have ... = horner-sum of-bool 2 (map (λi. ((bit v i) ∧ (bit zv i))) [0..<n])
  proof −
    have ∀ i. i ≥ n ⟶ ¬(bit zv i)
      using above-nth-not-set assms(1)
      using assms(2) not-may-implies-false
     by (smt (verit, ccfv-SIG) One-nat-def diff-less int-ops(6) leadingZerosAddHigh-
estOne linorder-not-le nat-int-comparison(2) not-numeral-le-zero size64 zero-less-Suc
zerosAboveHighestOne)
    then have ∀ i. i ≥ n ⟶ ¬((bit v i) ∧ (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*)
using *above-nth-not-set assms*(*1*)
using *assms*(*2*) *not-may-implies-false*
by (*smt* (*verit*, *ccfv-SIG*) *One-nat-def diff-less int-ops*(*6*) *leadingZerosAddHighestOne linorder-not-le nat-int-comparison*(*2*) *not-numeral-le-zero size64 zero-less-Suc zerosAboveHighestOne*)
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*
using *horner-sum-bit-eq-take-bit size64*
by (*metis size-word.rep-eq take-bit-length-eq*)
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 $[m,\ p] \vdash x \mapsto$ *IntVal b xv*
  shows $xv \le (\uparrow x)$
  using *assms*
 by (*metis* (*no-types*, *lifting*) *and.idem 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*))

lemma *L2*:
  assumes *numberOfLeadingZeros* $(\uparrow z)$ + *numberOfTrailingZeros* $(\uparrow y) \ge 64$
  assumes $n = 64 -$ *numberOfLeadingZeros* $(\uparrow z)$
  assumes $[m,\ p] \vdash z \mapsto$ *IntVal b zv*
  assumes $[m,\ p] \vdash y \mapsto$ *IntVal b yv*
  shows *yv mod* $2\hat{\ }n = 0$
proof –
  have *yv mod* $2\hat{\ }n$ = *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 ... $\le$ *horner-sum of-bool 2* (*map* (*bit* $(\uparrow y)$) [*0..<n*])
    using *up-mask-upper-bound assms*(*4*)
   by (*metis* (*no-types*, *opaque-lifting*) *and.right-neutral bit.conj-cancel-right bit.conj-disj-distribs*(*1*) *bit.double-compl horner-sum-bit-eq-take-bit take-bit-and ucast-id up-spec word-and-le1 word-not-dist*(*2*))
  also have *horner-sum of-bool 2* (*map* (*bit* $(\uparrow y)$) [*0..<n*]) = *horner-sum of-bool 2* (*map* (λx. *False*) [*0..<n*])
  proof –
    have $\forall\, i < n.\ \neg$(*bit* $(\uparrow y)$ *i*)
      using *assms*(*1*,*2*) *zerosBelowLowestOne*
      by (*metis add.commute add-diff-inverse-nat add-lessD1 leD le-diff-conv numberOfTrailingZeros-def*)
    then show *?thesis*
      by (*metis* (*full-types*) *transfer-map*)
  qed
  also have *horner-sum of-bool 2* (*map* (λx. *False*) [*0..<n*]) = *0*
    using *zero-horner*
    by *blast*
  finally show *?thesis*
    by *auto*
qed

thm-oracles *L1 L2*

lemma *unfold-binary-width-add*:
  shows ([*m*,*p*] $\vdash$ *BinaryExpr BinAdd xe ye* $\mapsto$ *IntVal b val*) = ($\exists$ *x y*.
          (([*m*,*p*] $\vdash$ *xe* $\mapsto$ *IntVal b x*) $\wedge$
           ([*m*,*p*] $\vdash$ *ye* $\mapsto$ *IntVal b y*) $\wedge$
           (*IntVal b val* = *bin-eval BinAdd* (*IntVal b x*) (*IntVal b y*)) $\wedge$
           (*IntVal b val* $\neq$ *UndefVal*)
        )) (**is** *?L* = *?R*)
proof (*intro iffI*)
  assume *3*: *?L*
  show *?R* **apply** (*rule evaltree.cases*[*OF 3*])

```
    apply force+ apply auto[1]
    apply (smt (verit) intval-add.elims intval-bits.simps)
    by blast
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 BinAdd (IntVal b x) (IntVal b y)
        and new-int b val ≠ UndefVal
    by auto
  then show ?L
    using R by blast
 qed

lemma unfold-binary-width-and:
  shows ([m,p] ⊢ BinaryExpr BinAnd xe ye ↦ IntVal b val) = (∃ x y.
          (([m,p] ⊢ xe ↦ IntVal b x) ∧
           ([m,p] ⊢ ye ↦ IntVal b y) ∧
           (IntVal b val = bin-eval BinAnd (IntVal b x) (IntVal b y)) ∧
           (IntVal b val ≠ UndefVal)
        )) (is ?L = ?R)
proof (intro iffI)
  assume 3: ?L
  show ?R apply (rule evaltree.cases[OF 3])
    apply force+ apply auto[1] using intval-and.elims intval-bits.simps
    apply (smt (verit) new-int.simps new-int-bin.simps take-bit-and)
    by blast
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 BinAnd (IntVal b x) (IntVal b y)
        and new-int b val ≠ UndefVal
    by auto
  then show ?L
    using R by blast
qed

lemma mod-dist-over-add-right:
  fixes a b c :: int64
  fixes n :: nat
  assumes 1: 0 < n
  assumes 2: n < 64
  shows (a + b mod 2^n) mod 2^n = (a + b) mod 2^n
  using mod-dist-over-add
  by (simp add: 1 2 add.commute)

lemma numberOfLeadingZeros-range:
  0 ≤ numberOfLeadingZeros n ∧ numberOfLeadingZeros n ≤ Nat.size n
  unfolding numberOfLeadingZeros-def highestOneBit-def using max-set-bit
  by (simp add: highestOneBit-def leadingZeroBounds numberOfLeadingZeros-def)

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)
    apply (rule evaltree.BinaryExpr)
    using xv apply simp
    using yv apply simp
    by simp+
  have lhs: [m, p] ⊢ exp[(x + y) & z] ↦ new-int b (and (xv + yv) zv)
    apply (rule evaltree.BinaryExpr)
    using addv apply simp
    using zv apply simp
    using addv apply auto[1]
    by simp
  have rhs: [m, p] ⊢ exp[x & z] ↦ new-int b (and xv zv)
    apply (rule evaltree.BinaryExpr)
    using xv apply simp
    using zv apply simp
     apply force
    by simp
  then show ?thesis
  proof (cases numberOfLeadingZeros (↑z) > 0)
    case True
    have n-bounds: 0 ≤ n ∧ n < 64
      using diff-le-self n numberOfLeadingZeros-range
      by (simp add: True)
    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
      using mod-dist-over-add-right n-bounds
      by (metis take-bit-0 take-bit-eq-mod zero-less-iff-neq-zero)
    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
      using assms by auto
    also have ... = and (xv mod 2^n) zv
      using mod-mod-trivial
    by (smt (verit, best) and.idem take-bit-eq-mask take-bit-eq-mod word-bw-assocs(1))
    also have ... = and xv zv
      using L1 n zv by metis
    finally show ?thesis
      using eval lhs rhs
      by (metis evalDet)
  next
    case False
    then have numberOfLeadingZeros (↑z) = 0
      by simp
    then have numberOfTrailingZeros (↑y) ≥ 64
      using assms(1)
      by fastforce
    then have yv = 0
      using yv
        by (metis (no-types, lifting) L1 L2 add-diff-cancel-left′ and.comm-neutral
and.idem bit.compl-zero bit.conj-cancel-right bit.conj-disj-distribs(1) bit.double-compl
less-imp-diff-less linorder-not-le 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))
  apply (simp add: IRExpr-up-def)
  using simple-mask.exp-eliminate-y by blast
```

optimization *redundant-lhs-x-or*: $((x \mid y)\ \&\ z) \longmapsto y\ \&\ z$
when $(((and\ (IRExpr\text{-}up\ x)\ (IRExpr\text{-}up\ z)) = 0))$
apply (*simp add*: *IRExpr-up-def*)
using *simple-mask.exp-eliminate-y*
by (*meson exp-or-commute mono-binary order-refl order-trans*)

optimization *redundant-rhs-y-or*: $(z\ \&\ (x \mid y)) \longmapsto z\ \&\ x$
when $(((and\ (IRExpr\text{-}up\ y)\ (IRExpr\text{-}up\ z)) = 0))$
apply (*simp add*: *IRExpr-up-def*)
using *simple-mask.exp-eliminate-y*
by (*meson exp-and-commute order.trans*)

optimization *redundant-rhs-x-or*: $(z\ \&\ (x \mid y)) \longmapsto z\ \&\ y$
when $(((and\ (IRExpr\text{-}up\ x)\ (IRExpr\text{-}up\ z)) = 0))$
apply (*simp add*: *IRExpr-up-def*)
using *simple-mask.exp-eliminate-y*
by (*meson dual-order.trans exp-and-commute exp-or-commute mono-binary order-refl*)

end

end

9.4 ConditionalNode Phase

theory *ConditionalPhase*
imports
Common
Proofs.StampEvalThms
begin

phase *ConditionalNode*
terminating *size*
begin

lemma *negates*: $\exists\, v\ b.\ e = IntVal\ b\ v \land b > 0 \Longrightarrow val\text{-}to\text{-}bool\ (val[e]) \longleftrightarrow \neg(val\text{-}to\text{-}bool\ (val[!e]))$
unfolding *intval-logic-negation.simps*
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 $val[(!e)\ ?\ x : y] = val[e\ ?\ y : x]$
using *assms* **by** (*cases e*; *auto simp*: *negates logic-negate-def*)

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 $\exists v'\ b\ vv. ([m, p] \vdash exp[e] \mapsto v') \wedge v' = IntVal\ b\ vv \wedge b > 0$
  using *assms*
   by (*metis eval-bits-1-64 intval-logic-negation.elims negation-preserve-eval unfold-unary*)

optimization *NegateConditionFlipBranches*: $((!e)\ ?\ x : y) \longmapsto (e\ ?\ y : x)$
  apply *simp* **using** *negation-condition-intval negation-preserve-eval-intval*
 by (*smt* (*z3*) *ConditionalExpr ConditionalExprE evalDet negates negation-preserve-eval*)

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*) $\wedge$ *wf-stamp u* $\wedge$ *wf-stamp v*)
  using *stamp-under-defn* **by** *auto*

optimization *condition-bounds-y*: $((u < v)\ ?\ x : y) \longmapsto y$
   when (*stamp-under* (*stamp-expr v*) (*stamp-expr u*) $\wedge$ *wf-stamp u* $\wedge$ *wf-stamp v*)
  using *stamp-under-defn-inverse* **by** *auto*

lemma *val-optimise-integer-test*:
  assumes $\exists v.\ x = IntVal\ 32\ v$
  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*) *and-one-eq bool-to-val.simps*(*1*) *even-iff-mod-2-eq-zero odd-iff-mod-2-eq-one val-to-bool.simps*(*1*))

optimization *ConditionalEliminateKnownLess*: $((x < y)\ ?\ x : y) \longmapsto x$
        when (*stamp-under* (*stamp-expr x*) (*stamp-expr y*)
          $\wedge$ *wf-stamp x* $\wedge$ *wf-stamp y*)
   using *stamp-under-defn* **by** *auto*

optimization *ConditionalEqualIsRHS*: $((x\ eq\ y)\ ?\ x : y) \longmapsto y$
  apply *auto*
  by (*smt* (*verit*) *Value.inject*(*1*) *bool-to-val.simps*(*2*) *bool-to-val-bin.simps evalDet intval-equals.elims val-to-bool.elims*(*1*))

optimization *normalizeX*: $((x\ eq\ const\ (IntVal\ 32\ 0))\ ?$
$(const\ (IntVal\ 32\ 0)) : (const\ (IntVal\ 32\ 1))) \longmapsto x$
when $(x = ConstantExpr\ (IntVal\ 32\ 0) \mid (x = ConstantExpr\ (IntVal\ 32\ 1)))$ **.**

optimization *normalizeX2*: $((x\ eq\ (const\ (IntVal\ 32\ 1)))\ ?$
$(const\ (IntVal\ 32\ 1)) : (const\ (IntVal\ 32\ 0))) \longmapsto x$
when $(x = ConstantExpr\ (IntVal\ 32\ 0) \mid (x = ConstantExpr\ (IntVal\ 32\ 1)))$ **.**

optimization *flipX*: $((x\ eq\ (const\ (IntVal\ 32\ 0)))\ ?$
$(const\ (IntVal\ 32\ 1)) : (const\ (IntVal\ 32\ 0))) \longmapsto$
$x \oplus (const\ (IntVal\ 32\ 1))$
when $(x = ConstantExpr\ (IntVal\ 32\ 0) \mid (x = ConstantExpr\ (IntVal\ 32\ 1)))$ **.**

optimization *flipX2*: $((x\ eq\ (const\ (IntVal\ 32\ 1)))\ ?$
$(const\ (IntVal\ 32\ 0)) : (const\ (IntVal\ 32\ 1))) \longmapsto$
$x \oplus (const\ (IntVal\ 32\ 1))$
when $(x = ConstantExpr\ (IntVal\ 32\ 0) \mid (x = ConstantExpr\ (IntVal\ 32\ 1)))$ **.**

lemma *stamp-of-default*:
  assumes *stamp-expr x* = *default-stamp*
  assumes *wf-stamp x*
  shows $([m, p] \vdash x \mapsto v) \longrightarrow (\exists vv.\ v = IntVal\ 32\ vv)$
  using *assms*
  by (*metis default-stamp valid-value-elims*(*3*) *wf-stamp-def*)

optimization *OptimiseIntegerTest*:
   $(((x\ \&\ (const\ (IntVal\ 32\ 1)))\ eq\ (const\ (IntVal\ 32\ 0)))\ ?$
   $(const\ (IntVal\ 32\ 0)) : (const\ (IntVal\ 32\ 1))) \longmapsto$
   $x\ \&\ (const\ (IntVal\ 32\ 1))$
   when (*stamp-expr x* = *default-stamp* ∧ *wf-stamp x*)
  apply *simp* **apply** (*rule impI*; (*rule allI*)+; *rule impI*)
  subgoal premises *eval* **for** *m p v*
proof –
  obtain *xv* **where** *xv*: $[m, p] \vdash x \mapsto xv$
    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 *xv evaltree.BinaryExpr evaltree.ConstantExpr evaltree.ConditionalExpr*
  by (*smt* (*verit*) *ConditionalExprE ConstantExprE bin-eval.simps*(*11*) *bin-eval.simps*(*4*) *evalDet intval-conditional.simps unfold-binary*)
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*(*4*) *evalDet xv*)
have *lhs* = *rhs* **using** *val-optimise-integer-test x32*
  using *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

9.5 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 testt:
  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]
  using assms apply (cases x; cases y; auto)
  using testt by auto

lemma val-multiply-neutral:
  assumes x = new-int b v
  shows val[x * (IntVal b 1)] = val[x]
  using assms by force

lemma val-multiply-zero:
  assumes x = new-int b v
  shows val[x * (IntVal b 0)] = IntVal b 0
  using assms by simp

lemma val-multiply-negative:
  assumes x = new-int b v
  shows val[x * intval-negate (IntVal b 1)] = intval-negate x
  using assms
 by (smt (verit) Value.disc(1) Value.inject(1) add.inverse-neutral intval-negate.simps(1)

      is-IntVal-def mask-0 mask-eq-take-bit-minus-one new-int.elims of-bool-eq(2)
take-bit-dist-neg
    take-bit-of-1 val-eliminate-redundant-negative val-multiply-neutral val-multiply-zero

      verit-minus-simplify(4) zero-neq-one)


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) size64 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))
        unfolding 63
        by force
    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
      using mask-eq-iff by (simp add: less-mask-eq p(6))
    then have x2 * ((2::64 word) ^ unat i + (1::64 word)) = (x2 * ((2::64 word)
^ unat i)) + x2
      by (simp add: distrib-left)
    then show x2 * ((2::64 word) ^ unat i + (1::64 word)) = x2 << unat (and i
(63::64 word)) + x2
      by (simp add: 63 ‹and (i::64 word) (mask (6::nat)) = 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
      using mask-eq-iff by (simp add: less-mask-eq p(6))
    then have x2 ∗ ((2::64 word) ^ unat i − (1::64 word)) = (x2 ∗ ((2::64 word)
^ unat i)) − x2
      by (simp add: right-diff-distrib′)
    then show x2 ∗ ((2::64 word) ^ unat i − (1::64 word)) = x2 << unat (and i
(63::64 word)) − x2
      by (simp add: 63 ‹and (i::64 word) (mask (6::nat)) = i›)
    qed
    using val-to-bool.simps(2) by presburger


lemma val-distribute-multiplication:
  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)]
  apply (cases x; cases q; cases a; auto) using distrib-left assms by auto


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)]
  using assms
  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)))]

      using assms by (cases i; cases j; 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-multiplication val-MulPower2* **by** *simp*
then have *2*: $val[(x * IntVal\ 64\ (2\ \hat{}\ unat(i)))] = val[x << IntVal\ 64\ i]$
using *assms val-MulPower2*
using *Value.distinct(1) intval-mul.simps(1) new-int.simps new-int-bin.simps*
by (*smt* (*verit*))
then show *?thesis*
using *1 Value.distinct(1) assms(1) assms(3) assms(5) assms(6) intval-mul.simps(1)*
n
new-int.simps new-int-bin.elims val-MulPower2
by (*smt* (*verit*, *del-insts*))
qed

thm-oracles *val-MulPower2AddPower2*

lemma *exp-multiply-zero-64*:
$exp[x * (const\ (IntVal\ 64\ 0))] \geq ConstantExpr\ (IntVal\ 64\ 0)$
using *val-multiply-zero* **apply** *auto*
using *Value.inject(1) constantAsStamp.simps(1) int-signed-value-bounds intval-mul.elims*

mult-zero-right new-int.simps new-int-bin.simps nle-le numeral-eq-Suc
take-bit-of-0
unfold-const valid-stamp.simps(1) valid-value.simps(1) zero-less-Suc wf-value-def
by (*smt* (*verit*))

lemma *exp-multiply-neutral*:
$exp[x * (const\ (IntVal\ b\ 1))] \geq x$
using *val-multiply-neutral* **apply** *auto*
by (*smt* (*verit*) *Value.inject(1) eval-unused-bits-zero intval-mul.elims mult.right-neutral*

new-int.elims new-int-bin.elims)

thm-oracles *exp-multiply-neutral*

lemma *exp-MulPower2*:
fixes $i :: 64\ word$
assumes $y = ConstantExpr\ (IntVal\ 64\ (2\ \hat{}\ unat(i)))$
and $0 < i$
and $i < 64$
and $exp[x > (const\ IntVal\ b\ 0)]$
and $exp[y > (const\ IntVal\ b\ 0)]$
shows $exp[x * y] \geq exp[x << ConstantExpr\ (IntVal\ 64\ i)]$
using *assms* **apply** *simp* **using** *val-MulPower2*
by (*metis ConstantExprE equiv-exprs-def unfold-binary*)

lemma *exp-MulPower2Add1*:
fixes $i :: 64\ word$
assumes $y = ConstantExpr\ (IntVal\ 64\ ((2\ \hat{}\ unat(i)) + 1))$
and $0 < i$

  and *i* < *64*
  and *exp*[*x* > (*const IntVal b 0*)]
  and *exp*[*y* > (*const IntVal b 0*)]
shows *exp*[*x* ∗ *y*] = *exp*[(*x* << *ConstantExpr* (*IntVal 64 i*)) + *x*]
  sorry

lemma *greaterConstant*:
  assumes *a* > *b*
  and *y* = *ConstantExpr* (*IntVal 64 a*)
  and *x* = *ConstantExpr* (*IntVal 64 b*)
  shows *y* > *x*
  apply *auto*
  sorry

Optimisations

optimization *EliminateRedundantNegative*: −*x* ∗ −*y* ⟼ *x* ∗ *y*
  using *mul-size.simps* **apply** *auto*[*1*]
  using *val-eliminate-redundant-negative bin-eval.simps*(*2*)
  by (*metis BinaryExpr*)

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)
  apply *auto* **using** *val-multiply-zero*
  using *Value.inject*(*1*) *constantAsStamp.simps*(*1*) *int-signed-value-bounds intval-mul.elims*

    mult-zero-right new-int.simps new-int-bin.simps take-bit-of-0 unfold-const
    valid-stamp.simps(*1*) *valid-value.simps*(*1*) *wf-value-def*
  by (*smt* (*verit*))

optimization *MulNegate*: *x* ∗ −(*const* (*IntVal b 1*)) ⟼ −*x*
  apply *auto* **using** *val-multiply-negative wf-value-def*
  by (*smt* (*verit*) *Value.distinct*(*1*) *Value.sel*(*1*) *add.inverse-inverse intval-mul.elims*

   intval-negate.simps(*1*) *mask-eq-take-bit-minus-one new-int.simps new-int-bin.simps*

    take-bit-dist-neg unary-eval.simps(*2*) *unfold-unary*
    val-eliminate-redundant-negative)

fun *isNonZero* :: *Stamp* ⇒ *bool* **where**
  isNonZero (*IntegerStamp b lo hi*) = (*lo* > *0*) |
  isNonZero - = *False*

lemma *isNonZero-defn*:

```
  assumes isNonZero (stamp-expr x)
  assumes wf-stamp x
  shows ([m, p] ⊢ x ↦ v) ⟶ (∃ vv b. (v = IntVal b vv ∧ val-to-bool val[(IntVal b
0) < v]))
  apply (rule impI) subgoal premises eval
proof –
  obtain b lo hi where xstamp: stamp-expr x = IntegerStamp b lo hi
    using assms
    by (meson isNonZero.elims(2))
  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 unfolding wf-stamp-def
    using eval vdef xstamp by fastforce
  have take-bit b vv = vv
    using eval eval-unused-bits-zero vdef by auto
  then have vv > 0
    using signed-above
      by (metis bit-take-bit-iff int-signed-value.simps not-less-zero signed-eq-0-iff
signed-take-bit-eq-if-positive take-bit-0 take-bit-of-0 verit-comp-simplify1(1) word-gt-0)
  then show ?thesis
    using vdef using signed-above
    by simp
qed
  done


optimization MulPower2: x * y ⟼ x << const (IntVal 64 i)
                              when (i > 0 ∧
                                    64 > i ∧
                                    y = exp[const (IntVal 64 (2 ^ unat(i)))])
   defer
   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 obtain xvv where xvv: xv = IntVal 64 xvv
    using eval
   using ConstantExprE bin-eval.simps(2) evalDet intval-bits.simps intval-mul.elims
new-int-bin.simps unfold-binary
    by (smt (verit))
  obtain yv where yv: [m, p] ⊢ y ↦ yv
    using eval(1) eval(2) by blast
  then have lhs: [m, p] ⊢ exp[x * y] ↦ val[xv * yv]
    by (metis bin-eval.simps(2) eval(1) eval(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 validStampIntConst wf-value-def valid-value.simps(1) xv xvv)
  then have rhs: [m, p] ⊢ exp[x << const (IntVal 64 i)] ↦ val[xv << (IntVal 64
i)]
    using xv xvv using evaltree.BinaryExpr
   by (metis Value.simps(5) bin-eval.simps(8) intval-left-shift.simps(1) new-int.simps)
  have val[xv * yv] = val[xv << (IntVal 64 i)]
    using val-MulPower2
    by (metis ConstantExprE eval(1) evaltree-not-undef lhs yv)
  then show ?thesis
    by (metis eval(1) eval(2) evalDet lhs rhs)
qed
  done



optimization MulPower2Add1: x * y ⟼ (x << const (IntVal 64 i)) + x
                              when (i > 0 ∧
                                    64 > i ∧
                                    y = ConstantExpr (IntVal 64 ((2 ^ unat(i)) + 1)) )
   defer
   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
      by (smt (verit) p ConstantExprE bin-eval.simps(2) evalDet intval-bits.simps
intval-mul.elims
          new-int-bin.simps unfold-binary)
    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 by fastforce
    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(2) evalDet p(1) 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
constantAsStamp.simps(1)
          take-bit64 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(8) intval-left-shift.simps(1) new-int.simps
xv xvv
          evaltree.BinaryExpr)
```

then have *rhs*: $[m, p] \vdash$ *exp*$[(x << const\ (IntVal\ 64\ i)) + x] \mapsto$ *val*$[(xv << (IntVal\ 64\ i)) + xv]$
by (*metis* (*no-types*, *lifting*) *intval-add.simps*(*1*) *rhs2* *bin-eval.simps*(*1*) *Value.simps*(*5*)
evaltree.BinaryExpr *intval-left-shift.simps*(*1*) *new-int.simps* *xv* *xvv*)
then have *val*$[xv * yv]$ = *val*$[(xv << (IntVal\ 64\ i)) + xv]$
using *1* *exp-MulPower2Add1* *ygezero* **by** *auto*
then show *?thesis*
by (*metis* *evalDet* *lhs* *p*(*1*) *p*(*2*) *rhs*)
qed
done

end

end

9.6 NotNode Phase

theory *NotPhase*
imports
Common
begin

phase *NotNode*
terminating *size*
begin

lemma *bin-not-cancel*:
bin$[\neg(\neg(e))]$ = *bin*$[e]$
by *auto*

lemma *val-not-cancel*:
assumes *val*$[\sim(new\text{-}int\ b\ v)] \neq$ *UndefVal*
shows *val*$[\sim(\sim(new\text{-}int\ b\ v))] = (new\text{-}int\ b\ v)$
using *bin-not-cancel*
by (*simp* *add*: *take-bit-not-take-bit*)

lemma *exp-not-cancel*:
shows *exp*$[\sim(\sim a)] \geq$ *exp*$[a]$
using *val-not-cancel* **apply** *auto*
by (*metis* *eval-unused-bits-zero* *intval-logic-negation.cases* *intval-not.simps*(*1*) *intval-not.simps*(*2*) *intval-not.simps*(*3*) *intval-not.simps*(*4*) *new-int.simps*)

Optimisations

optimization *NotCancel*: *exp*$[\sim(\sim a)] \longmapsto a$
by (*metis* *exp-not-cancel*)

end

end

9.7 OrNode Phase

```
theory OrPhase
  imports
    Common
begin

context stamp-mask
begin
```

Taking advantage of the truth table of or operations.

#	x	y	$x\|y$
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	1

If row 2 never applies, that is, canBeZero x & canBeOne y = 0, then $(x|y) = x$.

Likewise, if row 3 never applies, canBeZero y & canBeOne x = 0, then $(x|y) = y$.

```
lemma OrLeftFallthrough:
  assumes (and (not (↓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
      using eval
      by (metis BinaryExprE bin-eval-new-int new-int.simps)
    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 = intval-or (IntVal b xv) (IntVal b yv)
      using e xv yv
      by (metis bin-eval.simps(5) eval(2) evalDet unfold-binary)
    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 = intval-or (IntVal b xv) (IntVal b yv)
    by (smt (verit, ccfv-threshold) and.idem assms bit.conj-disj-distrib eval-unused-bits-zero
intval-or.simps(1) new-int.simps new-int-bin.simps not-down-up-mask-and-zero-implies-zero
word-ao-absorbs(3) xv yv)
    then show ?thesis
      using vdef
      using xv 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
      using eval
      by (metis BinaryExprE bin-eval-new-int new-int.simps)
    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 = intval-or (IntVal b xv) (IntVal b yv)
      using e xv yv
      by (metis bin-eval.simps(5) eval(2) evalDet unfold-binary)
    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 = intval-or (IntVal b xv) (IntVal b yv)
      by (metis (no-types, lifting) assms eval-unused-bits-zero intval-or.simps(1)
new-int.elims new-int-bin.elims stamp-mask.not-down-up-mask-and-zero-implies-zero
stamp-mask-axioms word-ao-absorbs(8) xv yv)
    then show ?thesis
      using vdef
      using 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]
   apply (cases x; auto) using bin-or-equal assms
  by auto+

lemma val-elim-redundant-false:
  assumes x = new-int b v
  and      val[x | false] ≠ UndefVal
  shows   val[x | false] = val[x]
   using assms apply (cases x; auto) by presburger

lemma val-shift-const-right-helper:
   val[x | y] = val[y | x]
   apply (cases x; cases y; auto)
  by (simp add: or.commute)+

lemma val-or-not-operands:
 val[~x | ~y] = val[~(x & y)]
  apply (cases x; cases y; auto)
  by (simp add: take-bit-not-take-bit)


lemma exp-or-equal:
  exp[x | x] ≥ exp[x]
   using val-or-equal apply auto
    by (smt (verit, ccfv-SIG) evalDet eval-unused-bits-zero intval-negate.elims int-
val-or.simps(2)
       intval-or.simps(6) intval-or.simps(7) new-int.simps val-or-equal)

lemma exp-elim-redundant-false:
 exp[x | false] ≥ exp[x]
   using val-elim-redundant-false apply auto
   by (smt (verit) Value.sel(1) eval-unused-bits-zero intval-or.elims new-int.simps
       new-int-bin.simps val-elim-redundant-false)
```

Optimisations

optimization *OrEqual*: $x \mid x \longmapsto x$
  by (*meson exp-or-equal le-expr-def*)

optimization *OrShiftConstantRight*: $((const\, x) \mid y) \longmapsto y \mid (const\, x)$ *when* $\neg$(*is-ConstantExpr* *y*)
  using *size-flip-binary* **apply** *force*
  apply *auto*
  by (*simp add*: *BinaryExpr unfold-const val-shift-const-right-helper*)

optimization *EliminateRedundantFalse*: $x \mid false \longmapsto x$
  by (*meson exp-elim-redundant-false le-expr-def*)

optimization *OrNotOperands*: $({\sim}x \mid {\sim}y) \longmapsto {\sim}(x \,\&\, y)$
  apply (*metis add-2-eq-Suc′ less-SucI not-add-less1 not-less-eq size-binary-const size-non-add*)
   apply *auto* **using** *val-or-not-operands*
 by (*metis BinaryExpr UnaryExpr bin-eval.simps*(*4*) *intval-not.simps*(*2*) *unary-eval.simps*(*3*))

optimization *OrLeftFallthrough*:
  $x \mid y \longmapsto x$ *when* ((*and* (*not* (*IRExpr-down* *x*)) (*IRExpr-up* *y*)) = *0*)
  using *simple-mask.OrLeftFallthrough* **by** *blast*

optimization *OrRightFallthrough*:
  $x \mid y \longmapsto y$ *when* ((*and* (*not* (*IRExpr-down* *y*)) (*IRExpr-up* *x*)) = *0*)
  using *simple-mask.OrRightFallthrough* **by** *blast*

end

end

9.8 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-int b v
  assumes val[(x + y) − y] ≠ UndefVal
  shows   val[(x + y) − y] = val[x]
  using bin-sub-after-right-add
  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] ≠ UndefVal
  shows   val[(x − y) − x] = val[−y]
  using assms apply (cases x; cases y; auto)
  using intval-sub.elims by fastforce

lemma val-sub-then-left-sub:
  assumes y = new-int b v
  assumes val[x − (x − y)] ≠ UndefVal
  shows   val[x − (x − y)] = val[y]
  using assms apply (cases x; cases y; auto)
  by (metis (mono-tags) intval-sub.simps(5))
```

lemma *val-subtract-zero*:
  assumes $x = \textit{new-int}\ b\ v$
  assumes $\textit{intval-sub}\ x\ (\textit{IntVal}\ b\ 0) \neq \textit{UndefVal}$
  shows $\textit{intval-sub}\ x\ (\textit{IntVal}\ b\ 0) = \textit{val}[x]$
  using *assms* **by** (*induction x*; *simp*)

lemma *val-zero-subtract-value*:
  assumes $x = \textit{new-int}\ b\ v$
  assumes $\textit{intval-sub}\ (\textit{IntVal}\ b\ 0)\ x \neq \textit{UndefVal}$
  shows $\textit{intval-sub}\ (\textit{IntVal}\ b\ 0)\ x = \textit{val}[-x]$
  using *assms* **by** (*induction x*; *simp*)

lemma *val-sub-then-left-add*:
  assumes $\textit{val}[x - (x + y)] \neq \textit{UndefVal}$
  shows $\textit{val}[x - (x + y)] = \textit{val}[-y]$
  using *assms* **apply** (*cases x*; *cases y*; *auto*)
  by (*metis* (*mono-tags*, *lifting*) *intval-sub.simps*(*5*))

lemma *val-sub-negative-value*:
  assumes $\textit{val}[x - (-y)] \neq \textit{UndefVal}$
  shows $\textit{val}[x - (-y)] = \textit{val}[x + y]$
  using *assms* **by** (*cases x*; *cases y*; *auto*)

lemma *val-sub-self-is-zero*:
  assumes $x = \textit{new-int}\ b\ v \wedge \textit{val}[x - x] \neq \textit{UndefVal}$
  shows $\textit{val}[x - x] = \textit{new-int}\ b\ 0$
  using *assms* **by** (*cases x*; *auto*)

lemma *val-sub-negative-const*:
  assumes $y = \textit{new-int}\ b\ v \wedge \textit{val}[x - (-y)] \neq \textit{UndefVal}$
  shows $\textit{val}[x - (-y)] = \textit{val}[x + y]$
  using *assms* **by** (*cases x*; *cases y*; *auto*)

lemma *exp-sub-after-right-add*:
  shows $\textit{exp}[(x + y) - y] \geq \textit{exp}[x]$
  apply *auto* **using** *val-sub-after-right-add-2*
  using *evalDet eval-unused-bits-zero intval-add.elims new-int.simps*
  by (*smt* (*verit*))

lemma *exp-sub-after-right-add2*:
  shows $\textit{exp}[(x + y) - x] \geq \textit{exp}[y]$
  using *exp-sub-after-right-add* **apply** *auto*
  using *bin-eval.simps*(*1*) *bin-eval.simps*(*3*) *intval-add-sym unfold-binary*
  by (*smt* (*z3*) *Value.inject*(*1*) *diff-eq-eq evalDet eval-unused-bits-zero intval-add.elims*

    intval-sub.elims new-int.simps new-int-bin.simps take-bit-dist-subL)

lemma *exp-sub-negative-value*:

$exp[x - (-y)] \geq exp[x + y]$
apply *simp* **using** *val-sub-negative-value*
by (*smt* (*verit*) *bin-eval.simps*(*1*) *bin-eval.simps*(*3*) *evaltree-not-undef* *unary-eval.simps*(*2*) *unfold-binary unfold-unary*)

lemma *exp-sub-then-left-sub*:
shows $exp[x - (x - y)] \geq exp[y]$
using *val-sub-then-left-sub* **apply** *auto*
subgoal premises *p* **for** *m p xa xaa ya*
proof–
obtain *xa* **where** *xa*: $[m, p] \vdash x \mapsto xa$
using *p*(*2*) **by** *blast*
obtain *ya* **where** *ya*: $[m, p] \vdash y \mapsto ya$
using *p*(*5*) **by** *auto*
obtain *xaa* **where** *xaa*: $[m, p] \vdash x \mapsto xaa$
using *p*(*2*) **by** *blast*
have *1*: $val[xa - (xaa - ya)] \neq UndefVal$
by (*metis evalDet p*(*2*) *p*(*3*) *p*(*4*) *p*(*5*) *xa xaa ya*)
then have $val[xaa - ya] \neq UndefVal$
by *auto*
then have $[m,p] \vdash y \mapsto val[xa - (xaa - ya)]$
by (*metis 1 Value.exhaust evalDet eval-unused-bits-zero evaltree-not-undef intval-sub.simps*(*6*) *intval-sub.simps*(*7*) *new-int.simps p*(*5*) *val-sub-then-left-sub xa xaa ya*)
then show *?thesis*
by (*metis evalDet p*(*2*) *p*(*4*) *p*(*5*) *xa xaa ya*)
qed
done

thm-oracles *exp-sub-then-left-sub*

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*: $((x - y) - x) \longmapsto -y$
apply (*metis Suc-lessI add-2-eq-Suc' add-less-cancel-right less-trans-Suc not-add-less1 size-binary-const size-binary-lhs size-binary-rhs size-non-add*)
apply *auto*
by (*metis evalDet 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* **apply** *simp*
 using *exp-sub-then-left-sub* **by** *blast*

optimization *SubtractZero*: $(x - (const\ IntVal\ b\ 0)) \longmapsto x$
 apply *auto*
 by (*smt* (*verit*) *add.right-neutral diff-add-cancel eval-unused-bits-zero intval-sub.elims*

   intval-word.simps new-int.simps new-int-bin.simps)

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** *simp*

thm-oracles *SubNegativeValue*

lemma *negate-idempotent*:
 assumes $x = IntVal\ b\ v \wedge take\text{-}bit\ b\ v = v$
 shows $x = val[-(-x)]$
 using *assms*
 using *is-IntVal-def* **by** *force*

optimization *ZeroSubtractValue*: $((const\ IntVal\ b\ 0) - x) \longmapsto (-x)$
        when (*wf-stamp x* ∧ *stamp-expr x* = *IntegerStamp b lo hi* ∧ ¬(*is-ConstantExpr x*))
  defer
 apply *auto* **unfolding** *wf-stamp-def*
 apply (*smt* (*verit*) *diff-0 intval-negate.simps*(*1*) *intval-sub.elims intval-word.simps*

   new-int-bin.simps unary-eval.simps(*2*) *unfold-unary*)
 using *add-2-eq-Suc′ size.simps*(*2*) *size-flip-binary* **by** *presburger*

optimization *SubSelfIsZero*: $(x - x) \longmapsto$ *const IntVal b 0 when*
$(wf\text{-}stamp\ x \wedge stamp\text{-}expr\ x = IntegerStamp\ b\ lo\ hi)$
 apply *simp-all*
  apply *auto*
 using *IRExpr.disc(42) One-nat-def size-non-const* **apply** *presburger*
 by (*smt* (*verit*, *best*) *wf-value-def ConstantExpr evalDet 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*)

end

end

9.9 XorNode Phase

theory *XorPhase*
 imports
  Common
  Proofs.StampEvalThms
begin

phase *XorNode*
 terminating *size*
begin

lemma *bin-xor-self-is-false*:
bin$[x \oplus x] = 0$
 by *simp*

lemma *bin-xor-commute*:
bin$[x \oplus y] =$ *bin*$[y \oplus x]$
 by (*simp add*: *xor.commute*)

lemma *bin-eliminate-redundant-false*:
bin$[x \oplus 0] =$ *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*
 using *assms* **by** (*cases x*; *auto*)

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*
 using *assms* **by** (*cases x*; *auto*)

lemma *val-xor-self-is-false-3*:
  assumes $val[x \oplus x] \neq UndefVal \wedge x = IntVal\ 64\ v$
  shows $val[x \oplus x] = IntVal\ 64\ 0$
  using *assms* **by** (*cases x*; *auto*)

lemma *val-xor-commute*:
   $val[x \oplus y] = val[y \oplus x]$
   apply (*cases x*; *cases y*; *auto*)
  by (*simp add*: *xor.commute*)+

lemma *val-eliminate-redundant-false*:
  assumes $x = new\text{-}int\ b\ v$
  assumes $val[x \oplus (bool\text{-}to\text{-}val\ False)] \neq UndefVal$
  shows $val[x \oplus (bool\text{-}to\text{-}val\ False)] = x$
  using *assms* **apply** (*cases x*; *auto*)
  by *meson*

lemma *exp-xor-self-is-false*:
 assumes $wf\text{-}stamp\ x \wedge stamp\text{-}expr\ x = default\text{-}stamp$
 shows $exp[x \oplus x] \geq exp[false]$
  using *assms* **apply** *auto* **unfolding** *wf-stamp-def*
 using *IntVal0 Value.inject*(*1*) *bool-to-val.simps*(*2*) *constantAsStamp.simps*(*1*) *evalDet*

     int-signed-value-bounds new-int.simps unfold-const val-xor-self-is-false-2
valid-int
    valid-stamp.simps(*1*) *valid-value.simps*(*1*) *wf-value-def*
  by (*smt* (*z3*) *validDefIntConst*)

lemma *exp-eliminate-redundant-false*:
  shows $exp[x \oplus false] \geq exp[x]$
  using *val-eliminate-redundant-false* **apply** *auto*
  subgoal premises *p* **for** *m p xa*
   proof −
    obtain *xa* **where** *xa*: $[m,p] \vdash x \mapsto xa$
     using *p*(*2*) **by** *blast*
    then have $val[xa \oplus (IntVal\ 32\ 0)] \neq UndefVal$
     using *evalDet p*(*2*) *p*(*3*) **by** *blast*
    then have $[m,p] \vdash x \mapsto val[xa \oplus (IntVal\ 32\ 0)]$
     apply (*cases xa*; *auto*) **using** *eval-unused-bits-zero xa* **by** *auto*
    then show *?thesis*
     using *evalDet p*(*2*) *xa* **by** *blast*
   qed
  done

Optimisations

optimization *XorSelfIsFalse*: $(x \oplus x) \longmapsto false$ *when*
      $(wf\text{-}stamp\ x \wedge stamp\text{-}expr\ x = default\text{-}stamp)$

  using *size-non-const* **apply** *force*
  using *exp-xor-self-is-false* **by** *auto*

optimization *XorShiftConstantRight*: $((const\ x) \oplus y) \longmapsto y \oplus (const\ x)$ *when* $\neg(is\text{-}ConstantExpr\ y)$
  using *size-flip-binary* **apply** *force*
  unfolding *le-expr-def* **using** *val-xor-commute*
  by *auto*

optimization *EliminateRedundantFalse*: $(x \oplus false) \longmapsto x$
    using *exp-eliminate-redundant-false* **by** *blast*

end

end

10 Verifying term graph optimizations using Isabelle/HOL

theory *TreeSnippets*
  imports
    Canonicalizations.BinaryNode
    Canonicalizations.ConditionalPhase
    Canonicalizations.AddPhase
    Semantics.TreeToGraphThms
    Snippets.Snipping
    HOL−Library.OptionalSugar
begin

— First, we disable undesirable markup.
declare [[*show-types=false*,*show-sorts=false*]]
no-notation *ConditionalExpr* (- *?* - : -)

— We want to disable and reduce how aggressive automated tactics are as obligations are generated in the paper
method *unfold-size* = −
method *unfold-optimization* =
  (*unfold rewrite-preservation.simps*, *unfold rewrite-termination.simps*,
    rule conjE, *simp*, *simp del*: *le-expr-def*)

10.1 Markup syntax for common operations

notation (*latex*)
  kind (-⟪-⟫)

notation (*latex*)
valid-value (- ∈ -)

notation (*latex*)
val-to-bool (*bool-of* -)

notation (*latex*)
constantAsStamp (*stamp-from-value* -)

notation (*latex*)
size (*trm(-)*)

10.2 Representing canonicalization optimizations

We wish to provide an example of the semantics layers at which optimizations can be expressed.

lemma *diff-self*:
  fixes $x :: int$
  shows $x - x = 0$
  by *simp*
lemma *diff-diff-cancel*:
  fixes $x\ y :: int$
  shows $x - (x - y) = y$
  by *simp*
thm *diff-self*
thm *diff-diff-cancel*

algebraic-laws

$$x - x = 0 \tag{1}$$
$$x - (x - y) = y \tag{2}$$

lemma *diff-self-value*: $\forall v::'a::len\ word.\ v - v = 0$
  by *simp*
lemma *diff-diff-cancel-value*:
  $\forall\ v_1\ v_2::'a::len\ word\ .\ v_1 - (v_1 - v_2) = v_2$
  by *simp*

algebraic-laws-values

$$\forall v :: 'a\ word.\ v - v = (0 :: 'a\ word) \tag{3}$$
$$\forall (v_1::'a\ word)\ v_2 :: 'a\ word.\ v_1 - (v_1 - v_2) = v_2 \tag{4}$$

translations

```
  n <= CONST ConstantExpr (CONST IntVal b n)
  x − y <= CONST BinaryExpr (CONST BinSub) x y
notation (ExprRule output)
  Refines (- ⟼ -)
lemma diff-self-expr:
  assumes ∀ m p v. [m,p] ⊢ exp[e − e] ↦ IntVal b v
  shows exp[e − e] ≥ exp[const (IntVal b 0)]
  using assms apply simp
  by (metis(full-types) evalDet val-to-bool.simps(1) zero-neq-one)

method open-eval = (simp; (rule impI)?; (rule allI)+; rule impI)

lemma diff-diff-cancel-expr:
  shows exp[e₁ − (e₁ − e₂)] ≥ exp[e₂]
  apply open-eval
  subgoal premises eval for m p v
  proof −
    obtain v1 where v1: [m, p] ⊢ e₁ ↦ v1
      using eval by blast
    obtain v2 where v2: [m, p] ⊢ e₂ ↦ v2
      using eval by blast
    then have e: [m, p] ⊢ exp[e₁ − (e₁ − e₂)] ↦ val[v1 − (v1 − v2)]
      using v1 v2 eval
      by (smt (verit, ccfv-SIG) bin-eval.simps(3) evalDet unfold-binary)
    then have notUn: val[v1 − (v1 − v2)] ≠ UndefVal
      using evaltree-not-undef by auto
    then have val[v1 − (v1 − v2)] = v2
      apply (cases v1; cases v2; auto simp: notUn)
      using eval-unused-bits-zero v2 apply blast
      by (metis(full-types) intval-sub.simps(5))
    then show ?thesis
      by (metis e eval evalDet v2)
  qed
  done

thm-oracles diff-diff-cancel-expr
```

algebraic-laws-expressions

$$e - e \longmapsto 0 \tag{5}$$

$$e_1 - (e_1 - e_2) \longmapsto e_2 \tag{6}$$

```
no-translations
  n <= CONST ConstantExpr (CONST IntVal b n)
  x − y <= CONST BinaryExpr (CONST BinSub) x y


definition wf-stamp :: IRExpr ⇒ bool where
```

wf-stamp e = (∀ *m p v*. ([*m*, *p*] ⊢ *e* ↦ *v*) ⟶ *valid-value v* (*stamp-expr e*))

lemma *wf-stamp-eval*:
  assumes *wf-stamp e*
  assumes *stamp-expr e* = *IntegerStamp b lo hi*
  shows ∀ *m p v*. ([*m*, *p*] ⊢ *e* ↦ *v*) ⟶ (∃ *vv*. *v* = *IntVal b vv*)
  using *assms* **unfolding** *wf-stamp-def*
  using *valid-int-same-bits valid-int*
  by *metis*

phase *SnipPhase*
  terminating *size*
begin
lemma *sub-same-val*:
  assumes *val*[*e* − *e*] = *IntVal b v*
  shows *val*[*e* − *e*] = *val*[*IntVal b 0*]
  using *assms* **by** (*cases e*; *auto*)

sub-same-32

optimization *SubIdentity*:
  e − *e* ⟼ *ConstantExpr* (*IntVal b 0*)
    when ((*stamp-expr exp*[*e* − *e*] = *IntegerStamp b lo hi*) ∧ *wf-stamp exp*[*e* − *e*])

   using *IRExpr.disc*(*42*) *size.simps*(*4*) *size-non-const*
   apply *simp*
  apply (*rule impI*) **apply** *simp*
proof −
   assume *assms*: *stamp-binary BinSub* (*stamp-expr e*) (*stamp-expr e*) = *IntegerStamp b lo hi* ∧ *wf-stamp exp*[*e* − *e*]
  have ∀ *m p v* . ([*m*, *p*] ⊢ *exp*[*e* − *e*] ↦ *v*) ⟶ (∃ *vv*. *v* = *IntVal b vv*)
    using *assms wf-stamp-eval*
    by (*metis stamp-expr.simps*(*2*))
  then show ∀ *m p v*. ([*m*,*p*] ⊢ *BinaryExpr BinSub e e* ↦ *v*) ⟶ ([*m*,*p*] ⊢ *ConstantExpr* (*IntVal b 0*) ↦ *v*)
    using *wf-value-def*
   by (*smt* (*verit*, *best*) *BinaryExprE TreeSnippets.wf-stamp-def assms bin-eval.simps*(*3*) *constantAsStamp.simps*(*1*) *evalDet stamp-expr.simps*(*2*) *sub-same-val unfold-const valid-stamp.simps*(*1*) *valid-value.simps*(*1*))
qed
thm-oracles *SubIdentity*

RedundantSubtract

optimization *RedundantSubtract*:
  e_1 − (e_1 − e_2) ⟼ e_2

  using *size-simps* **apply** *simp*
  using *diff-diff-cancel-expr* **by** *presburger*
end

10.3 Representing terms

We wish to show a simple example of expressions represented as terms.

ast-example

```
BinaryExpr BinAdd
 (BinaryExpr BinMul x x)
 (BinaryExpr BinMul x x)
```

Then we need to show the datatypes that compose the example expression.

abstract-syntax-tree

```
datatype IRExpr =
  UnaryExpr IRUnaryOp IRExpr
  | BinaryExpr IRBinaryOp IRExpr IRExpr
  | ConditionalExpr IRExpr IRExpr IRExpr
  | ParameterExpr nat Stamp
  | LeafExpr nat Stamp
  | ConstantExpr Value
  | ConstantVar (char list)
  | VariableExpr (char list) Stamp
```

value

```
datatype Value = UndefVal
  | IntVal nat (64 word)
  | ObjRef (nat option)
  | ObjStr (char list)
```

10.4 Term semantics

The core expression evaluation functions need to be introduced.

eval

```
unary-eval :: IRUnaryOp ⇒ Value ⇒ Value
bin-eval :: IRBinaryOp ⇒ Value ⇒ Value ⇒ Value
```

We then provide the full semantics of IR expressions.

no-translations

$(prop)\ P \wedge Q \Longrightarrow R <= (prop)\ P \Longrightarrow Q \Longrightarrow R$
translations
$(prop)\ P \Longrightarrow Q \Longrightarrow R <= (prop)\ P \wedge Q \Longrightarrow R$

tree-semantics

$$\frac{[m,p] \vdash xe \mapsto x \quad result = unary\text{-}eval\ op\ x \quad result \neq UndefVal}{[m,p] \vdash UnaryExpr\ op\ xe \mapsto result}$$

$$\frac{[m,p] \vdash xe \mapsto x \quad [m,p] \vdash ye \mapsto y \quad result = bin\text{-}eval\ op\ x\ y \quad result \neq UndefVal}{[m,p] \vdash BinaryExpr\ op\ xe\ ye \mapsto result}$$

$$\frac{[m,p] \vdash ce \mapsto cond \quad branch = (\mathsf{if}\ bool\text{-}of\ cond\ \mathsf{then}\ te\ \mathsf{else}\ fe) \quad [m,p] \vdash branch \mapsto result \quad result \neq UndefVal}{[m,p] \vdash ConditionalExpr\ ce\ te\ fe \mapsto result}$$

$$\frac{wf\text{-}value\ c}{[m,p] \vdash ConstantExpr\ c \mapsto c} \qquad \frac{i < |p| \quad p_{[i]} \in s}{[m,p] \vdash ParameterExpr\ i\ s \mapsto p_{[i]}}$$

$$\frac{val = m\ n \quad val \in s}{[m,p] \vdash LeafExpr\ n\ s \mapsto val}$$

no-translations
$(prop)\ P \Longrightarrow Q \Longrightarrow R <= (prop)\ P \wedge Q \Longrightarrow R$
translations
$(prop)\ P \wedge Q \Longrightarrow R <= (prop)\ P \Longrightarrow Q \Longrightarrow R$

And show that expression evaluation is deterministic.

tree-evaluation-deterministic

$[m,p] \vdash e \mapsto v_1 \wedge [m,p] \vdash e \mapsto v_2 \Longrightarrow v_1 = v_2$

We then want to start demonstrating the obligations for optimizations. For this we define refinement over terms.

expression-refinement

$e_1 \sqsupseteq e_2 = (\forall m\ p\ v.\ [m,p] \vdash e_1 \mapsto v \longrightarrow [m,p] \vdash e_2 \mapsto v)$

To motivate this definition we show the obligations generated by optimization definitions.

phase *SnipPhase*
  terminating *size*
begin

InverseLeftSub

optimization *InverseLeftSub*:
$(e_1 - e_2) + e_2 \longmapsto e_1$

InverseLeftSubObligation

1. $trm(e_1) < trm(BinaryExpr\ BinAdd\ (BinaryExpr\ BinSub\ e_1\ e_2)\ e_2)$
2. $BinaryExpr\ BinAdd\ (BinaryExpr\ BinSub\ e_1\ e_2)\ e_2 \sqsupseteq e_1$

using *RedundantSubAdd* **by** *auto*

InverseRightSub

optimization *InverseRightSub*: $e_2 + (e_1 - e_2) \longmapsto e_1$

InverseRightSubObligation

1. $trm(e_1) < trm(BinaryExpr\ BinAdd\ e_2\ (BinaryExpr\ BinSub\ e_1\ e_2))$
2. $BinaryExpr\ BinAdd\ e_2\ (BinaryExpr\ BinSub\ e_1\ e_2) \sqsupseteq e_1$

using *RedundantSubAdd2*(*2*) *rewrite-termination.simps*(*1*) **apply** *blast*
using *RedundantSubAdd2*(*1*) *rewrite-preservation.simps*(*1*) **by** *blast*
end

expression-refinement-monotone

$e \sqsupseteq e' \Longrightarrow UnaryExpr\ op\ e \sqsupseteq UnaryExpr\ op\ e'$

$x \sqsupseteq x' \wedge y \sqsupseteq y' \Longrightarrow BinaryExpr\ op\ x\ y \sqsupseteq BinaryExpr\ op\ x'\ y'$

$ce \sqsupseteq ce' \wedge te \sqsupseteq te' \wedge fe \sqsupseteq fe' \Longrightarrow$
$ConditionalExpr\ ce\ te\ fe \sqsupseteq ConditionalExpr\ ce'\ te'\ fe'$

phase *SnipPhase*
terminating *size*
begin

BinaryFoldConstant

optimization *BinaryFoldConstant*: *BinaryExpr op* (*const v1*) (*const v2*)
$\longmapsto$ *ConstantExpr* (*bin-eval op v1 v2*)

BinaryFoldConstantObligation

1. $trm(ConstantExpr\ (bin\text{-}eval\ op\ v1\ v2))$
   $< trm(BinaryExpr\ op\ (ConstantExpr\ v1)\ (ConstantExpr\ v2))$
2. $BinaryExpr\ op\ (ConstantExpr\ v1)\ (ConstantExpr\ v2) \sqsupseteq$
   $ConstantExpr\ (bin\text{-}eval\ op\ v1\ v2)$

using *BinaryFoldConstant*(*1*) **by** *auto*

AddCommuteConstantRight

optimization *AddCommuteConstantRight*:
$(const\ v) + y \longmapsto y + (const\ v)\ when\ \neg(is\text{-}ConstantExpr\ y)$

AddCommuteConstantRightObligation

1. $\neg\ is\text{-}ConstantExpr\ y \longrightarrow$
   $trm(BinaryExpr\ BinAdd\ y\ (ConstantExpr\ v))$
   $< trm(BinaryExpr\ BinAdd\ (ConstantExpr\ v)\ y)$
2. $\neg\ is\text{-}ConstantExpr\ y \longrightarrow$
   $BinaryExpr\ BinAdd\ (ConstantExpr\ v)\ y \sqsupseteq$
   $BinaryExpr\ BinAdd\ y\ (ConstantExpr\ v)$

using *AddShiftConstantRight* **by** *auto*

AddNeutral

optimization *AddNeutral*: $e + (const\ (IntVal\ 32\ 0)) \longmapsto e$

AddNeutralObligation

1. $trm(e) < trm(BinaryExpr\ BinAdd\ e\ (ConstantExpr\ (IntVal\ 32\ 0)))$
2. $BinaryExpr\ BinAdd\ e\ (ConstantExpr\ (IntVal\ 32\ 0)) \sqsupseteq e$

apply *auto*
using *AddNeutral*(*1*) *rewrite-preservation.simps*(*1*) **by** *force*

AddToSub

optimization *AddToSub*: $-e + y \longmapsto y - e$

AddToSubObligation

1. *trm(BinaryExpr BinSub y e)* $<$ *trm(BinaryExpr BinAdd (UnaryExpr UnaryNeg e) y)*
2. *BinaryExpr BinAdd (UnaryExpr UnaryNeg e) y* $\sqsupseteq$ *BinaryExpr BinSub y e*

using *AddLeftNegateToSub* **by** *auto*

end

definition *trm* **where** *trm = size*

lemma *trm-defn*[*size-simps*]:
trm x = size x
by (*simp add*: *trm-def*)

phase

phase *AddCanonicalizations*
terminating *trm*
begin...**end**

hide-const (**open**) *Form.wf-stamp*

phase-example

phase *Conditional*
terminating *trm*
begin

phase-example-1

optimization *NegateCond*: ((!*e*) *?* *x* : *y*) $\longmapsto$ (*e* *?* *y* : *x*)

apply (*simp add*: *size-simps*)
using *ConditionalPhase.NegateConditionFlipBranches*(*1*) **by** *simp*

phase-example-2

optimization *TrueCond*: (*true* *?* *x* : *y*) $\longmapsto$ *x*

by (*auto simp*: *trm-def*)

phase-example-3

optimization *FalseCond*: (*false* *?* *x* : *y*) $\longmapsto$ *y*

by (*auto simp*: *trm-def*)

phase-example-4

optimization *BranchEqual*: $(e\ ?\ x : x) \longmapsto x$

by (*auto simp*: *trm-def*)

phase-example-5

optimization *LessCond*: $((u < v)\ ?\ x : y) \longmapsto x$
when (*stamp-under* (*stamp-expr u*) (*stamp-expr v*)
$\wedge$ *wf-stamp u* $\wedge$ *wf-stamp v*)

apply (*auto simp*: *trm-def*)
using *ConditionalPhase.condition-bounds-x*(*1*)
by (*metis*(*full-types*) *StampEvalThms.wf-stamp-def TreeSnippets.wf-stamp-def bin-eval.simps*(*12*) *stamp-under-defn*)

phase-example-6

optimization *condition-bounds-y*: $((x < y)\ ?\ x : y) \longmapsto y$
when (*stamp-under* (*stamp-expr y*) (*stamp-expr x*) $\wedge$ *wf-stamp x* $\wedge$ *wf-stamp y*)

apply (*auto simp*: *trm-def*)
using *ConditionalPhase.condition-bounds-y*(*1*)
by (*metis*(*full-types*) *StampEvalThms.wf-stamp-def TreeSnippets.wf-stamp-def bin-eval.simps*(*12*) *stamp-under-defn-inverse*)

phase-example-7

end

lemma *simplified-binary*: $\neg$(*is-ConstantExpr b*) $\Longrightarrow$ *size* (*BinaryExpr op a b*) = *size a* + *size b* + *2*
by (*induction b*; *induction op*; *auto simp*: *is-ConstantExpr-def*)

thm *bin-size*
thm *bin-const-size*
thm *unary-size*
thm *size-non-add*

termination

trm(UnaryExpr op e) = trm(e) + 2

trm(BinaryExpr op x (ConstantExpr cy)) = trm(x) + 2

trm(BinaryExpr op a b) = trm(a) + trm(b) + 2

trm(ConditionalExpr cond t f) = trm(cond) + trm(t) + trm(f) + 2

trm(ConstantExpr c) = 1

trm(ParameterExpr ind s) = 2

trm(LeafExpr nid s) = 2

graph-representation

typedef IRGraph =
$\{g :: \mathit{ID} \rightharpoonup (\mathit{IRNode} \times \mathit{Stamp}) \;.\; \mathit{finite}\ (\mathit{dom}\ g)\}$

no-translations
$(\mathit{prop})\ P \wedge Q \Longrightarrow R <= (\mathit{prop})\ P \Longrightarrow Q \Longrightarrow R$
translations
$(\mathit{prop})\ P \Longrightarrow Q \Longrightarrow R <= (\mathit{prop})\ P \wedge Q \Longrightarrow R$

graph2tree

$$\frac{g\langle\!\langle n\rangle\!\rangle = \mathit{ConstantNode}\ c}{g \vdash n \simeq \mathit{ConstantExpr}\ c} \quad \frac{g\langle\!\langle n\rangle\!\rangle = \mathit{ParameterNode}\ i \qquad \mathit{stamp}\ g\ n = s}{g \vdash n \simeq \mathit{ParameterExpr}\ i\ s}$$

$$\frac{\begin{array}{c} g\langle\!\langle n\rangle\!\rangle = \mathit{ConditionalNode}\ c\ t\ f \\ g \vdash c \simeq ce \qquad g \vdash t \simeq te \qquad g \vdash f \simeq fe \end{array}}{g \vdash n \simeq \mathit{ConditionalExpr}\ ce\ te\ fe}$$

$$\frac{g\langle\!\langle n\rangle\!\rangle = \mathit{AbsNode}\ x \qquad g \vdash x \simeq xe}{g \vdash n \simeq \mathit{UnaryExpr}\ \mathit{UnaryAbs}\ xe} \quad \frac{g\langle\!\langle n\rangle\!\rangle = \mathit{SignExtendNode}\ \mathit{inputBits}\ \mathit{resultBits}\ x \qquad g \vdash x \simeq}{g \vdash n \simeq \mathit{UnaryExpr}\ (\mathit{UnarySignExtend}\ \mathit{inputBits}\ \mathit{resultBits})}$$

$$\frac{g\langle\!\langle n\rangle\!\rangle = \mathit{AddNode}\ x\ y \qquad g \vdash x \simeq xe \qquad g \vdash y \simeq ye}{g \vdash n \simeq \mathit{BinaryExpr}\ \mathit{BinAdd}\ xe\ ye}$$

$$\frac{\mathit{is\text{-}preevaluated}\ g\langle\!\langle n\rangle\!\rangle \qquad \mathit{stamp}\ g\ n = s}{g \vdash n \simeq \mathit{LeafExpr}\ n\ s} \quad \frac{g\langle\!\langle n\rangle\!\rangle = \mathit{RefNode}\ n' \qquad g \vdash n' \simeq e}{g \vdash n \simeq e}$$

no-translations
$(\mathit{prop})\ P \Longrightarrow Q \Longrightarrow R <= (\mathit{prop})\ P \wedge Q \Longrightarrow R$

translations
$(prop)\ P \wedge Q \Longrightarrow R <= (prop)\ P \Longrightarrow Q \Longrightarrow R$

preeval

is-preevaluated (*InvokeNode n uu uv uw ux uy*) = *True*
is-preevaluated (*InvokeWithExceptionNode n uz va vb vc vd ve*) = *True*
is-preevaluated (*NewInstanceNode n vf vg vh*) = *True*
is-preevaluated (*LoadFieldNode n vi vj vk*) = *True*
is-preevaluated (*SignedDivNode n vl vm vn vo vp*) = *True*
is-preevaluated (*SignedRemNode n vq vr vs vt vu*) = *True*
is-preevaluated (*ValuePhiNode n vv vw*) = *True*
is-preevaluated (*AbsNode v*) = *False*
is-preevaluated (*AddNode v va*) = *False*
is-preevaluated (*AndNode v va*) = *False*
is-preevaluated (*BeginNode v*) = *False*
is-preevaluated (*BytecodeExceptionNode v va vb*) = *False*
is-preevaluated (*ConditionalNode v va vb*) = *False*
is-preevaluated (*ConstantNode v*) = *False*
is-preevaluated (*DynamicNewArrayNode v va vb vc vd*) = *False*
is-preevaluated EndNode = *False*
is-preevaluated (*ExceptionObjectNode v va*) = *False*
is-preevaluated (*FrameState v va vb vc*) = *False*
is-preevaluated (*IfNode v va vb*) = *False*
is-preevaluated (*IntegerBelowNode v va*) = *False*
is-preevaluated (*IntegerEqualsNode v va*) = *False*
is-preevaluated (*IntegerLessThanNode v va*) = *False*
is-preevaluated (*IsNullNode v*) = *False*
is-preevaluated (*KillingBeginNode v*) = *False*
is-preevaluated (*LeftShiftNode v va*) = *False*
is-preevaluated (*LogicNegationNode v*) = *False*
is-preevaluated (*LoopBeginNode v va vb vc*) = *False*
is-preevaluated (*LoopEndNode v*) = *False*
is-preevaluated (*LoopExitNode v va vb*) = *False*
is-preevaluated (*MergeNode v va vb*) = *False*
is-preevaluated (*MethodCallTargetNode v va*) = *False*
is-preevaluated (*MulNode v va*) = *False*
is-preevaluated (*NarrowNode v va vb*) = *False*
is-preevaluated (*NegateNode v*) = *False*
is-preevaluated (*NewArrayNode v va vb*) = *False*
is-preevaluated (*NotNode v*) = *False*
is-preevaluated (*OrNode v va*) = *False*
is-preevaluated (*ParameterNode v*) = *False*
is-preevaluated (*PiNode v va*) = *False*
is-preevaluated (*ReturnNode v va*) = *False*
is-preevaluated (*RightShiftNode v va*) = *False*
is-preevaluated (*ShortCircuitOrNode v va*) = *False*
is-preevaluated (*SignExtendNode v va vb*) = *False*

deterministic-representation

$g \vdash n \simeq e_1 \wedge g \vdash n \simeq e_2 \Longrightarrow e_1 = e_2$

thm-oracles *repDet*

well-formed-term-graph

$\exists\, e.\ g \vdash n \simeq e \wedge (\exists\, v.\ [m,p] \vdash e \mapsto v)$

graph-semantics

$([g,m,p] \vdash n \mapsto v) = (\exists\, e.\ g \vdash n \simeq e \wedge [m,p] \vdash e \mapsto v)$

graph-semantics-deterministic

$[g,m,p] \vdash n \mapsto v_1 \wedge [g,m,p] \vdash n \mapsto v_2 \Longrightarrow v_1 = v_2$

thm-oracles *graphDet*

notation (*latex*)
graph-refinement (*term-graph-refinement* -)

graph-refinement

term-graph-refinement $g_1\ g_2 =$
$(ids\ g_1 \subseteq ids\ g_2 \wedge$
$(\forall\, n.\ n \in ids\ g_1 \longrightarrow (\forall\, e.\ g_1 \vdash n \simeq e \longrightarrow g_2 \vdash n \trianglelefteq e)))$

translations
n <= CONST as-set n

graph-semantics-preservation

$e_1{}' \sqsupseteq e_2{}' \wedge$
$\{n\} \lhd g_1 \subseteq g_2 \wedge$
$g_1 \vdash n \simeq e_1{}' \wedge g_2 \vdash n \simeq e_2{}' \Longrightarrow$
term-graph-refinement $g_1\ g_2$

thm-oracles *graph-semantics-preservation-subscript* *maximal-sharing*

$$\begin{array}{l} \mathit{maximal\text{-}sharing}\ g = \\ (\forall\, n_1\ n_2. \\ \quad n_1 \in \mathit{true\text{-}ids}\ g \wedge n_2 \in \mathit{true\text{-}ids}\ g \longrightarrow \\ \quad (\forall\, e.\ g \vdash n_1 \simeq e \wedge \\ \qquad g \vdash n_2 \simeq e \wedge \mathit{stamp}\ g\ n_1 = \mathit{stamp}\ g\ n_2 \longrightarrow \\ \qquad n_1 = n_2)) \end{array}$$

tree-to-graph-rewriting

$$\begin{array}{l} e_1 \sqsupseteq e_2 \wedge \\ g_1 \vdash n \simeq e_1 \wedge \\ \mathit{maximal\text{-}sharing}\ g_1 \wedge \\ \{n\} \lhd g_1 \subseteq g_2 \wedge \\ g_2 \vdash n \simeq e_2 \wedge \\ \mathit{maximal\text{-}sharing}\ g_2 \Longrightarrow \\ \mathit{term\text{-}graph\text{-}refinement}\ g_1\ g_2 \end{array}$$

thm-oracles *tree-to-graph-rewriting*

term-graph-refines-term

$$(g \vdash n \unlhd e) = (\exists\, e'.\ g \vdash n \simeq e' \wedge e \sqsupseteq e')$$

term-graph-evaluation

$$g \vdash n \unlhd e \Longrightarrow \forall\, m\ p\ v.\ [m,p] \vdash e \mapsto v \longrightarrow [g,m,p] \vdash n \mapsto v$$

graph-construction

$$\begin{array}{l} e_1 \sqsupseteq e_2 \wedge g_1 \subseteq g_2 \wedge g_2 \vdash n \simeq e_2 \Longrightarrow \\ g_2 \vdash n \unlhd e_1 \wedge \mathit{term\text{-}graph\text{-}refinement}\ g_1\ g_2 \end{array}$$

thm-oracles *graph-construction*

term-graph-reconstruction

$$g \oplus e \leadsto (g', n) \Longrightarrow g' \vdash n \simeq e \wedge g \subseteq g'$$

refined-insert

$e_1 \sqsupseteq e_2 \wedge g_1 \oplus e_2 \rightsquigarrow (g_2,\ n') \Longrightarrow$
$g_2 \vdash n' \trianglelefteq e_1 \wedge \textit{term-graph-refinement}\ g_1\ g_2$

end